Memory Allocation under Hardware Compression

Muhammad Laghari, Yuqing Liu*, Gagandeep Panwar, David Bears*,
Chandler Jearls*, Raghavendra Srinivas*, Esha Choukse§, Kirk W. Cameron, Ali R. Butt, Xun Jian
Virginia Tech §*Microsoft Research*

***Abstract*— As the scaling of memory density slows *physically*, a promising solution is to scale memory *logically* by enhancing the CPU's memory controller to encode and store data more densely in memory. This is known as hardware memory compression. Hardware memory compression decouples OS-managed physical memory from *actual memory (i.e., DRAM)*; the memory controller spends a dynamically varying amount of DRAM on each physical page, depending on the compressibility of the page's content.**

The newly-decoupled actual memory effectively forms a new layer of memory beyond the traditional layers of virtual, pseudo-physical, and physical memory. We note unlike these traditional memory layers, each with its own specialized allocation interface (e.g., malloc/mmap for virtual memory, page tables+MMU for physical memory), this new layer of memory introduced by hardware memory compression still awaits its own unique memory allocation interface; its absence makes the allocation of actual memory imprecise and, sometimes, even impossible.

Imprecisely allocating less actual memory, and/or unable to allocate more, can harm performance. *Even* imprecisely allocating *more* actual memory to some jobs can be harmful as it can result in allocating less actual memory to other jobs in highly-occupied memory systems, where compression is useful.

To restore precise memory allocation, we design a new memory allocation specialized for this new layer of memory and, subsequently, architect a new MMU-like component in the memory controller and tackle the corresponding design challenges.

We create a full-system FPGA prototype of a hardware-compressed memory system with precise memory allocation. Our evaluations using the prototype show that jobs perform stably under colocation. The performance variation is only 1%-2%; in comparison, it is 19%-89% under the prior art.

***Index Terms*—Hardware compression; memory management.**

I. Introduction

DRAM density scaling increasingly lags behind other components (see Figure 1). Unlike Flash, DRAM only supports one layer of cells per die due to the tall aspect ratio of DRAM capacitors. Unlike CPU scaling, DRAM scaling faces the challenge of not only scaling transistors, but also capacitors, which is difficult as smaller capacitors hold less charge.

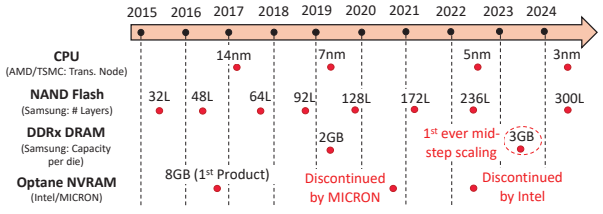

Figure 1: Memory, CPU, and NAND, scaling over the past 9 years [8], [14], [15], [18], [22], [28], [43]–[48]. The chart shows dates of first availability.

* All work done at Virginia Tech.

As DRAM scaling slows ***physically***, memory compression is a promising solution to scale up DRAM density ***logically***. Meta [55] data centers report their workloads have a high average memory compression ratio of 3x; compression ratio refers to memory footprint before compression divided by memory footprint after compression (assuming every compressible page is compressed). As such, hyperscale data centers (e.g., Meta [55], Google [25]) are using OS memory compression.

Unfortunately, OS memory compression incurs costly OS overheads; for example, whenever a process accesses an OS-compressed virtual page, the MMU incurs a costly page fault to wake up the OS to expand the virtual page to a full physical page [32]. As such, data centers can only compress a small fraction of the total pages – only the ***extremely cold*** pages and save little (e.g., 5%-20% of) memory [25], [55]; this is a far cry from what can be theoretically saved given the high memory compression ratio of typical workloads.

To affordably compress more memory data (e.g., lukewarm or even hot data) to save more memory, prior works have explored hardware memory compression [13], [17], [24], [33], [34], [37], [40], [52], [59], where the memory controller in the CPU transparently compresses and decompresses memory values. Unlike traditional systems, where physical memory is actual memory (i.e., DRAM), hardware-compressed memory decouples physical memory from actual memory (see Figure 2); the memory controller spends a varying amount of DRAM on each physical page according to the compression ratio of its content. To distinguish from physical memory, prior works call DRAM ***machine-physical memory*** [13], [35].

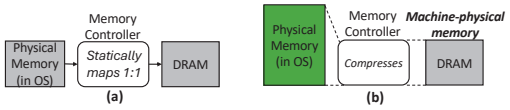

Figure 2: (a) A traditional system, where physical memory is DRAM. (b) Hardware memory compression decouples physical memory from DRAM.

This decoupling complicates memory management, however. The seminal work – MXT [52] – identifies that machine-physical memory can run out when physical memory is still abundant (see Section II-A). To prevent the system from running so low on free machine-physical memory that not even OS threads (e.g., swap daemons) can run and result in system-wide deadlock, MXT provides hardware support to enable the OS to cap the total/system-wide machine-physical memory usage (see Section II-A).

Problem: We identify the decoupling of physical and machine-physical memory also complicates memory alloca-

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00075

tion. Memory allocation – giving memory to a process or a group of related processes (a.k.a a job in Unix [12]) – is required to ensure stable/predictable performance and even correctness. In traditional systems (i.e., ones without hardware compression), memory allocation is precise. For example, after a cloud user specifies and pays for N GB of memory for his/her VM, the host can and – typically will – precisely allocate N GB of actual memory to the VM, regardless of whether the VM's guest OS is compressing memory internally. By decoupling physical memory and machine-physical memory, however, hardware-memory compression makes memory allocation 1) ***imprecise*** and 2) ***sometimes even impossible*** under the existing memory allocation interface.

Why Imprecise? Since the actual size of each physical page can vary dynamically according to the compression ratio of the page, to precisely allocate the specified S amount of actual memory (i.e., machine-physical memory) to a process/job, OS cannot simply allocate to it S amount of physical memory, like in traditional systems. A plausible option is to allocate $S \cdot C$ amount of physical memory, where C is the job's compression ratio; but since compression ratio is an application-level characteristic that is uncontrollable by and often unknown to the OS, the OS does not know how much physical memory to allocate. Overestimating or underestimating a job's compression ratio can make the allocated machine-physical memory several times more or less than specified and, therefore, imprecise.

Why Sometimes Impossible? The OS allocates physical pages to a process by pairing them with the virtual pages that the process is currently using. When every in-use virtual page in a process already has a physical page (e.g., when the process is fully in memory, without anything swapped out), the OS cannot allocate meaningfully more physical pages to the process and, thus, cannot allocate to it more machine-physical memory (see Section III-C Figure 8). If such processes (i.e., processes that are fully in memory) could still be allocated more machine-physical memory, they could still benefit from having more of their data decompressed and faster to access.

Consequences: Allocating less memory to a job, either due to allocating imprecisely or not being able to allocate, means more of the job's physical pages must be compressed and more of its accesses will suffer from decompression and additional translation overheads. Getting significantly less memory than specified can even cause a job to spill out to swap and slow down even more. In a highly-consolidated memory system, where compression is useful, ***even*** imprecisely allocating ***more*** memory to a job is harmful as it leads to allocating less memory to other jobs and harming their performance.

Observation: We observe every layer of memory (e.g., virtual, pseudo-physical, physical) has its own specialized memory allocation interface (e.g., malloc/mmap for virtual memory, page tables+MMU for physical memory); the ***only exception*** is the ***new layer*** of machine-physical memory that hardware memory compression decouples from physical memory. Trying to make do without a specialized memory allocation interface for this new layer of memory naturally gives rise to various memory allocation problems.

Proposal: We architect a new MMU-like component to enable the OS to ***directly*** allocate machine-physical memory and, thus, avoid all problems due to allocating it ***indirectly through*** allocating physical memory. We call our proposal the *Direct Machine-physical Memory Allocation Unit (DMU)*. Figure 3 gives an example of what DMU can enable.

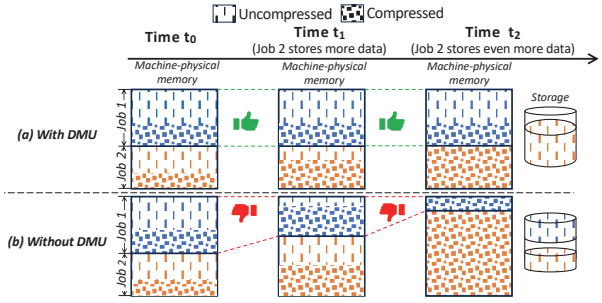

Figure 3: An example difference between hardware-compressed memory systems with and without DMU. (a) With DMU, each job can be precisely allocated its own dedicated amount of machine-physical memory, regardless of the memory needs of another job. (b) Without DMU, how much machine-physical memory a job uses is affected by another job. When a job's memory needs increases (e.g., Job 2 at t_1 and t_2), the needed space can be freed up from another job (e.g., Job 1) by compressing more of its data (see Job 1 at t_1) or even spilling out some of its data to storage (see Job 1 at t_2.)

Challenges: There are several design challenges. As ***one*** example, when the memory controller frees up memory ***transparently*** via compression, the OS does not know ***which*** machine-physical pages are freed and, thus, can be (re)-allocated; this complicates the traditional page-based allocation, which requires the OS to express ***which*** pages to allocate.

Idea: We create a specialized interface for machine-physical memory – an ***objective-based*** allocation that allows the OS to directly express ***how much*** machine-physical memory to allocate to individual jobs to precisely satisfy user-specified memory needs; exposing to the OS ***how much*** memory hardware has freed from a job, so that the OS can (re-)allocate them, is simpler than exposing ***which*** pages the memory controller has freed. DMU meets the memory allocation objective for a job by guiding the memory controller to compress the job ***precisely*** down to its machine-physical memory allocation (i.e., no more, no less); if it is not sufficiently compressible, DMU raises a fault (like a page fault) to assist the OS with spilling out parts of the job to keep it within its allocation.

We make the following contributions in this paper:

1) We identify hardware memory compression causes a new memory management problem: it makes memory allocation imprecise and, sometimes, even impossible under the existing physical memory allocation interface. Our evaluations of the prior art [33] show this problem can cause 19% to 89% variation in performance.
2) We are the first to explore a dedicated memory allocation interface specialized for machine-physical memory.
3) We architect the Direct Machine-physical Memory Allocation Unit (DMU) and prototype it on the FPGA.
4) In our FPGA evaluations, jobs perform stably when collocated with jobs of different sizes, with only 1%-2% average variation (down from the 19%-89%).

II. Related Work

Hardware memory compression enhances the memory controller to transparently compress memory values and move/pack them more densely to fit more data in memory.

To move data after compressing them, the seminal work MXT [52] and many follow-on works [13], [17], [24], [33], [37], [40], [59] enhance the memory controller to transparently manage a new address translation table – a linear table where entry i (or super entry i) is statically dedicated to mapping 4KB physical page i to its current machine-physical address(es). The table has enough entries for 2X (MXT [52]), 4X (TMCC [33]), or 8X (Compresso [13]) as many physical pages as machine-physical pages. The table entries are stored in memory and cached in the memory controller.

After compressing and packing the physical pages more densely to free up machine-physical memory, the memory controller tracks the freed memory in hardware-managed free list(s) to use them in the future when the applications write to more physical pages.

Prior works have addressed several key challenges.

A. Capping System-level Machine-Physical Memory Usage

Decoupling physical memory from machine-physical memory complicates memory management. Depending on the current compression ratio, machine-physical memory can run out before physical memory. Consider for example a system with $2GB$ of machine-physical memory and $8GB$ of physical memory; after the system uses just $2GB$ of physical memory – with $8GB - 2GB = 6GB$ of physical memory still free – the system can completely run out of the $2GB$ machine-physical memory if the memory values are all incompressible. When a system entirely runs out of machine-physical memory, even critical OS routines like the swap daemon do not have the memory they need to run, causing the system to deadlock and require a hard reboot.

To address this problem, when the hardware free list tracking free machine-physical locations runs low, MXT raises a hardware interrupt to alert the OS to evict data from memory and deallocate physical pages [1]. The OS is enhanced to proactively zero deallocated physical pages so that the memory controller can compress them perfectly to free up machine-physical memory to replenish the free list. Later works all inherit this approach, albeit sometimes with minor adaptations; for example, Compresso [13] calls a host-side balloon driver to spill out data from the host.

B. Reducing Penalty of Accessing Compressed Data

While accesses to compressed data are faster than under OS compression, they are still slower than uncompressed data due to needing decompression. As such, MXT compresses data in a recency-aware manner by managing a part of DRAM as a cache to store hot data uncompressed.

MXT, however, still frequently accesses compressed data; its uncompressed data cache in DRAM has a small fixed size of 100MB and, thus, can suffer from significant miss rates.

To further reduce decompression latency, many follow-on works (e.g., RMC [17], LCP [37], Compresso [13]) explore lightweight memory-block-level compression algorithms. Unlike MXT, which uses aggressive 1KB-granularity compression, these later works individually compress and compact 64B blocks to enable many times faster decompression.

Block-level compression, however, requires fine-grained address translation; in general, the finer the translations, the higher the translation miss rates. Block-level compression also needs costly page recompression: writes to a block can make the block less compressible and, thus, bigger so that it no longer fits in its current location; migrating the block elsewhere leaves behind an unused location/space fragment, which is later reclaimed via a costly page recompression [13].

These issues have renewed interest in recency-aware compression [33], [34]. Leaving hot pages uncompressed, so that hot pages continue to use coarse translation, can reduce overall translation overheads. Selectively recompressing LRU pages can also make page recompression less frequent.

Later recency-aware compression works (TMCC [33] and its extension DyLeCT [34]) improve uncompressed data hit rate over MXT by scaling up uncompressed memory to gigabytes, up to the entire memory. They take an OS-inspired approach – adaptively compress only the minimum pages needed to meet the system's current memory demand. They compress more pages ***only*** if the hardware free list drops below a threshold (see Figure 4). They also make uncompressed memory exclusive with respect to compressed memory, not inclusive like MXT.

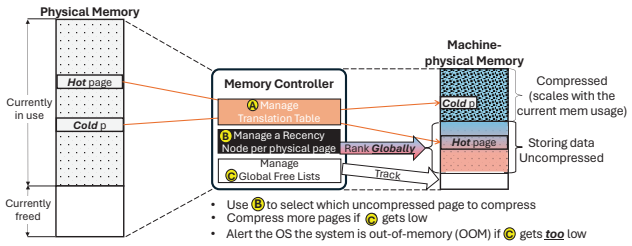

Figure 4: Recent prior works [33], [34] on recency-aware compression adapts to the current memory demand. Memories shown contiguously for clarity ***only***; when assigning machine-physical memory to individual physical pages, the memory controller uses arbitrary machine-physical pages and sub-pages at the top of the free lists.

To further improve hit rate, TMCC globally ranks uncompressed pages fully-associatively via a doubly-linked Recency List. The list's head tracks the most-recently used (MRU) physical page; the list's tail tracks the least-recently used (LRU) uncompressed physical page. TMCC statically reserves memory to store a statically-dedicated linked list node for each physical page and call these linked list nodes *recency nodes*.

While MXT, TMCC, and DyLeCT evaluate 1KB subpage-level and 4KB page-level compression algorithms, recency-aware compression can work complementarily with block-level compression; for example, lightweight compression can be used on cold pages while leaving hot pages uncompressed.

The hot pages of recency-aware compression can also benefit from *bandwidth-only* lightweight block-level compression [2], [6], [23], [36], [41], [53], [57], [58], which only improves the effective bandwidth of a page, without shrinking the size of the page. Instead of packing compressed blocks densely to store more data, bandwidth-only compression stores each compressed block in-place [42] or adjacent to its original location [58] and, thus, requires little or no overhead translation.

III. Allocation of Actual Memory: Background, Importance, and Newly-induced Problems

This section describes the importance of precise memory allocation and how hardware compression interferes with it.

A. Background on Allocation of Actual Memory

Unlike the various layers of logical (e.g., virtual, pseudo-physical) memory, which are ***needed for correctness***[1], actual memory (i.e., DRAM) is only ***needed for performance***; programs can always run correctly on swap alone, with little or no actual memory. The more actual memory, the less frequent are costly events such as swapping in/out, memory compression/decompression, OS file cache misses for storage-intensive applications, and garbage collection for Java and other managed programs.

We focus on the allocation of actual memory; as such, '***memory***' always refers to ***actual memory***, unless stated otherwise.

In multi-tenant systems (e.g., cloud, cluster), consolidating more jobs per server requires allocating to each job the minimum memory it needs to meet its performance needs. Special cases aside, the host does not know how much memory each job needs for performance; this depends on complex factors such as the current input data size, what are the important processes in the VM, the execution times of these processes, and how they vary with memory. The host is unaware of these aforementioned factors that determine each job's individual memory needs; requiring users to expose some of these factors (e.g., what are the current inputs) also raises privacy concerns that go against the emerging trend of confidential computing.

As such, multi-tenant systems universally require users to specify the actual memory they need for performance; the host then precisely allocates the specified actual memory, so that users need not worry about the host being a potential cause when their jobs' memory-related performance is poor. For example in cloud, after a user selects and pays for a VM or function [3] of a certain memory size (e.g., *M GB*), the host launches a VM or container with the specified memory size (e.g., *M GB*); when *M GB* is allocated to the VM, no matter how much data the VM accesses or whether its guest OS internally compresses memory, everything stays in the *M GB* of allocated physical memory, which is *traditionally* the actual memory. In on-site clusters, which mostly use Slurm [56], the "mem" limit parameter in the job submission script is used by Slurm to set the *memory.limit_in_bytes* parameter in the job's Cgroup, which is the Linux feature to control the resource usage of containers; if the job script does not explicitly specify "mem" limit, Slurm implicitly sets a default Cgroup *memory.limit* that is proportional to the number of requested cores [29].

[1]For example, a program calling malloc can crash if the virtual memory size is smaller than the malloc size (e.g., mallocing 5GB on a 32-bit machine, which only supports 4GB of virtual memory); similarly, a program calling malloc can crash if the pseudo-physical memory size of a VM is smaller than the size of a ***single*** malloc request.

B. Consequences of Losing Precise Memory Allocation

Imprecisely allocating less memory to a job than how much its user has carefully specified can obviously harm its performance. Below, we describe other negative consequences.

Interfering with profiling: To discourage users from over-specifying memory, cloud/clusters charges/queue jobs according to their specified memory. Determining how much to specify to minimize cost or wait time while meeting performance needs often requires users to profile[2] their jobs under different memory sizes that they specify. For example, since OS memory compression gives a graceful tradeoff between performance and memory size, a user deploying it in a VM can specify various smaller (e.g., 3/4, 1/2) VM sizes across different profiling runs to find the smallest memory size that meets performance needs for later production runs.

Imprecisely allocating different amounts of memory from what the user specifies during different runs can make performance unpredictable across runs and, therefore, render profiling ineffective, or even useless.

Causing system out-of-memory: Even imprecisely allocating more memory to a job can be harmful. It can cause a mostly-occupied memory system to run out of memory, preventing other jobs from getting their needed memory and/or slowing down their memory allocation.

To estimate the magnitude of the slowdown, we compare performance with and without precise allocation on a real-system – a traditional server with 24-cores, Ubuntu 22.04, and an NVMe storage for swap. We collocate two jobs per experiment; Job 1 runs a GraphBig benchmark, while Job 2 runs a file-processing program that accesses a 140GB file via mmap().

To evaluate precise memory allocation, we set the Cgroup *memory.limit* of Job 1 to the memory footprint of GraphBig (i.e., 98GB) and set Job 2's Cgroup to use most of the remaining memory in the system (i.e., 90GB, as the system has a total of 190GB). We turn off OS memory compression as it currently does not work well with Cgroups.[3]

[2]User profiling may not be always perfect and sometimes cause over-specification of memory. As such, a provider may 'steal' a little bit of memory that the user has specified/purchased (even if no provider admits they do and many publicly announce they do not [4], [31], [54]). 'Stealing' memory still requires ***precisely deallocating*** memory in a controlled manner, according to the prices and priorities of different classes of VMs/services.

[3]Unlike VMs, Cgroups cannot enforce precise memory allocation when using OS memory compression; Cgroups only limit each job's uncompressed memory size, but not its total (i.e., uncompressed + compressed) memory size. As such, a program controlled by a Cgroup can still use up to all the memory in the system. We verify this empirically using the latest Linux kernel.

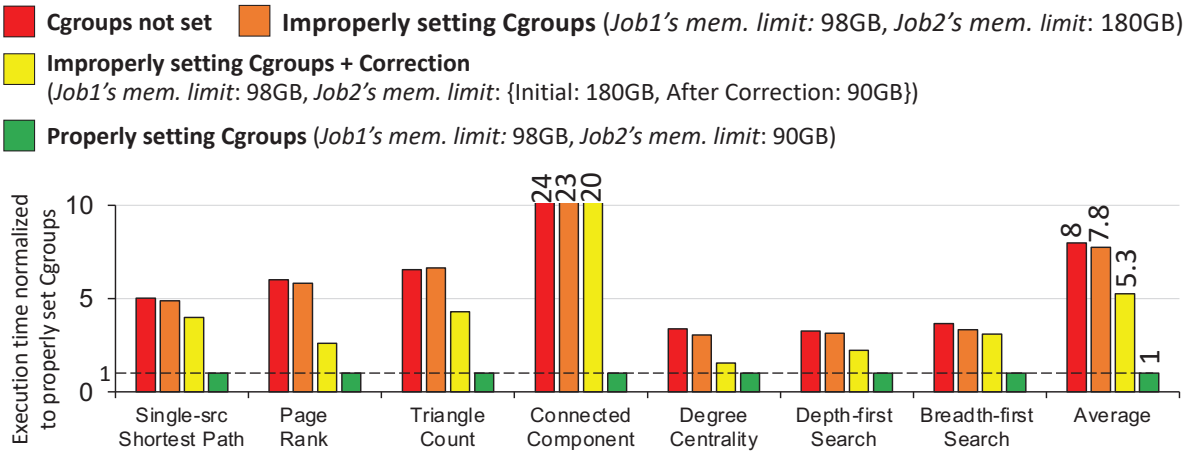

Figure 5: Execution time of the same GraphBig [30] benchmarks and dataset (*datagen-8_5-fb* [19]) used in prior works [33], [34], when collocated with a 2^{nd} job. Results remain similar under the full range of *'Swappiness'* settings.

We evaluate imprecise memory allocation by repeating the above without Cgroups. Job 1 slows down by 8X, on average (see Figure 5). Figure 6 shows what happened during the precise and imprecise memory allocation experiments. The performance of Job 2, which runs the file-processing program, improves only slightly (i.e., by 7%) because the additional memory only marginally improves Job 2's file cache hit rate.

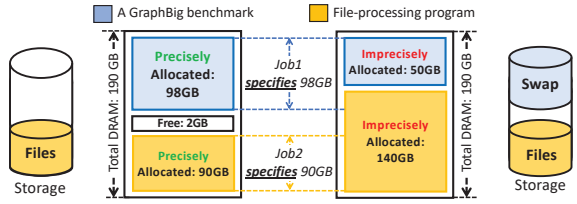

Figure 6: **Left:** Precisely allocating machine-physical memory. **Right:** Imprecisely allocating more machine-physical memory to a job.

As sensitivity analysis, we re-evaluate imprecise memory allocation by (i) setting wrong *memory.limit* so that the sum of the two limits exceeds system memory size and (ii) setting wrong *memory.limit* initially and correcting it when system memory is low. The slowdown remains similar (see Figure 5).

In summary, even imprecisely allocating more memory to a job can be harmful as it can lead to imprecisely allocating less memory to other jobs.

C. Memory Allocation under Hardware-compressed Memory

Hardware memory compression reduces physical memory to a logical memory layer. As such, when specifying how much actual memory their jobs need for performance, users will specify machine-physical (instead of physical) memory. Meanwhile, for service providers, reliably meeting user requests for machine-physical memory is also easier than meeting user requests for physical memory (see Figure 7).

To precisely allocate a specified S amount of actual memory (i.e., machine-physical memory) to a process/job, the OS

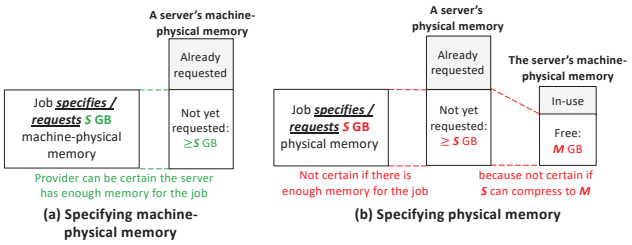

Figure 7: What type of memory users should specify and the impact on service providers. **(a)** When jobs specify machine-physical memory, if a server has more unrequested machine-physical memory than the specified amount, the provider is certain the server can meet the memory request; this assumes the system can also ensure every job uses only up to its requested amount, which is enabled by our work. **(b)** When jobs specify physical memory, the provider cannot be certain whether a server can meet the request even if the server has more unrequested physical memory than the specified amount; while the provider can be certain the request can be met if the server has more free machine-physical memory than the specified physical memory, reserving for each job the same amount of machine-physical memory as the physical memory the job specifies defeats the purpose of memory compression (i.e, to use more physical memory than the system's machine-physical memory).

cannot simply allocate S amount of physical memory, like previously in traditional systems. This is because hardware-compressed memory spends a dynamically varying amount of machine-physical memory on each physical page, depending on how compressible are its values. A plausible option is to allocate $S \cdot C$ amount of physical memory, where C is the job's compression ratio. But a program's compression ratio is uncontrollable by and often unknown to the OS; so, the OS does not know how much physical memory to allocate.

The OS can perhaps ***pessimistically assume*** a low compression ratio of 1 (i.e., assume nothing is compressible). This means allocating only as many physical pages as the machine-physical pages in the system (and not more). This yields no benefit (i.e., no increase in effective capacity) and only loss (i.e., compressed data are slower to access).

To get strong benefit (i.e., much more than OS compression), the OS can perhaps ***optimistically assume*** a high

(e.g., 4X) compression ratio. When assuming 4X, allocating a specified S amount of machine-physical memory means allocating $4S$ physical memory. For jobs with $< 4X$ compression ratios (e.g., 2X), this means allocating more machine-physical memory than specified (e.g., by $4X/2 = 2X$).

Imprecisely allocating a job 2X the machine-physical memory can have similar effect as improperly setting 2X the Cgroup *memory.limit* (see orange, yellow bars in Figure 5). Concretely, when prior works (e.g., MXT) run low on free machine-physical memory (e.g., due to imprecisely allocating too much memory), many memory accesses from user jobs must be blocked to make time to slowly spill out data and free up enough machine-physical memory to safely avoid deadlock. Meanwhile, the OS neither knows nor controls the compression ratio of each job; as such, the OS knows not which jobs are using too much machine-physical memory and, therefore, cannot surgically block and spill out data only from offending jobs (e.g., by inflating the memory balloons in their VMs). As such, all jobs can slow down significantly.

Despite making allocation imprecise, hardware memory compression needs precise allocation more frequently than traditional systems. Hardware memory compression is most useful for memory systems that are mostly-occupied (if not, why compress?); intuitively, the more occupied the memory, the higher likelihood that imprecisely allocating more machine-physical memory to a job than specified can cause the system to run out of memory.

Even when all processes are fully in-memory, with nothing spilled out, imprecise memory allocation is still harmful. Imprecisely allocating more memory to *Job A* (e.g., because it is less compressible than estimated) may mean needing to compress another *Job B* more aggressively; this slows down *Job B* in an unpredictable manner that depends on the compression ratio of *Job A*. The problem gets worse under recency-aware compression, which selectively compresses colder data; jobs that access memory less often than other jobs can get over-compressed.

Ideally, each job should be compressed into its specified memory (e.g., into 100GB if it specifies 100GB) and not get over-compressed (e.g., down to 20GB) when a collocated job is less compressible and/or more memory-intensive than it.

The OS, however, has no means of asking hardware to spend more memory on the 'over-compressed' victim job (e.g., the *Job B*), so that more of its pages can become uncompressed. When allocating machine-physical memory indirectly through allocating physical memory, the OS cannot allocate more machine-physical memory to a process that cannot be allocated more physical memory. As shown in Figure 8, a process cannot be allocated more physical pages when every virtual page used by the process already has a physical page (e.g., when the process is fully in memory, with nothing swapped out).

Conversely, for a job that is taking up too much machine-physical memory due to being more memory-intensive and thus evading compression under recency-aware compression, the OS has no way of instructing the memory controller to spend less machine-physical memory on it (i.e., to compress it more).

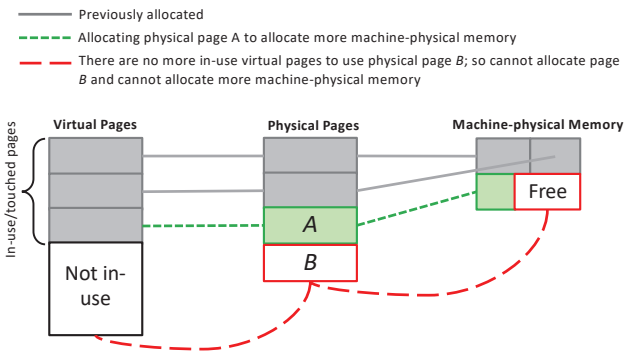

Figure 8: OS cannot meaningfully allocate more machine-physical memory when every virtual page used by the job already has a physical page.

D. No Simple Solution

To address the problem of imprecise allocation, a plausible solution is to modify the OS to periodically sample the compression ratios of allocated physical pages (e.g., by reading their content) and, in turn, estimate each job's compression ratios. Periodic sampling raises the question of precision, especially for short-lived processes like Function-as-a-service (FaaS) and micro-services. It also introduces a new continuous OS overhead that is not even in OS memory compression and, thus, contradicts the goal of hardware memory compression – reducing OS overheads for compression. The alternative of users sampling compression ratio, instead of the host OS, and then reporting them to the host OS can burden the users and raise new trust concerns for the service providers; furthermore, a faulty sampling can cause system-level problems (e.g., system running out of memory), unlike the various types of user-level sampling being done today, where faulty sampling only affects that user's program.

In comparison, when the guest OS performs memory compression in today's systems, neither the system nor the users sample compression ratio; the VMs can only use up to the memory that they booted up with, regardless of the compression ratio of their workloads. In other words, memory allocation remains precise under OS memory compression. This is because the host OS ***directly*** allocates machine-physical memory (as physical memory ***is*** machine-physical memory in traditional systems) and need not use compression ratios to reverse engineer how much physical memory to approximate the desired amount of machine-physical memory to allocate.

Meanwhile, there can be no simple solution to the second problem where the OS cannot allocate more machine-physical memory to jobs to which no more physical pages can be allocated (e.g., jobs that are fully in memory).

IV. Restoring Precise Allocation via DMU

We note every layer of memory – virtual memory, pseudo-physical memory, and physical memory – has its own allocation interface, except for the new layer of machine-

physical memory that hardware compression decouples from physical memory. Trying to make do without its own allocation interface for this new layer of memory naturally gives rise to various memory allocation problems.

As such, we explore a direct hardware interface to enable the OS to directly allocate machine-physical memory. Directly allocating machine-physical memory eliminates the need for sampling compression ratios (either at system- or user-level), just like OS memory compression (as discussed in Section III-D). Furthermore, allocating machine-physical memory directly enables the OS to allocate more of it to processes to which no more physical memory can be allocated.

To support the new memory allocation interface, we architect a new MMU-like component that we call the *Direct Machine-physical Memory Allocation Unit (DMU)*. DMU resides in the memory controller (MC), as only the MC is aware of machine-physical memory (see Figure 9).

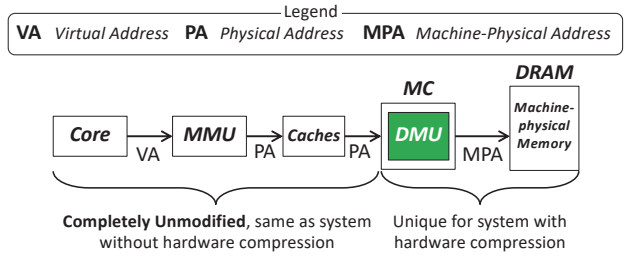

Figure 9: DMU's placement in the processor. The caches, the MMU, and the contents of the MMU (e.g., TLB entries and their permission bits etc.) remain completely unchanged because DMU manages machine-physical memory as a new layer that is independent from the virtual and physical memory layers.

DMU enables the OS to directly allocate machine-physical memory to specific processes/jobs that need precise allocation. For example, when jobs A and B specify *A GB* and *B GB*, the OS can allocate via DMU *A GB* and *B GB* of machine-physical memory to them, respectively. For processes/jobs that do not need precise memory allocation (for example, single-user systems like desktops typically do not specify memory requirements for any process), DMU treats them collectively as one logical job (e.g., compress them together); in hardware, DMU ***implicitly*** allocates to this logical job all the remaining machine-physical memory in the system (see Figure 10 (a)).

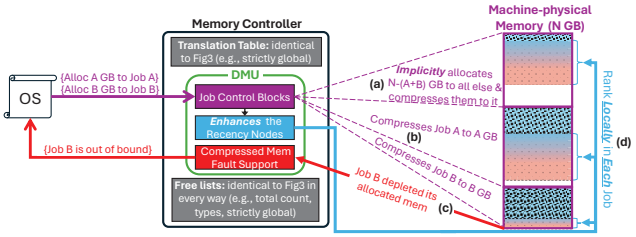

Figure 10: An overview of DMU and how it interacts with software. Contiguity in machine-physical memory is shown for clarity ***only***.

Like how different layers of memory are allocated (mostly) independently, machine-physical memory allocation is (mostly) independent from physical memory allocation (e.g., it cares not if 4KB physical pages or huge pages are allocated). When the OS allocates more physical pages to a process as it touches more virtual pages, DMU guides the MC to compress the allocated physical pages into the allocated machine-physical memory (see Figure 10 (b)).

Physical memory allocation is only affected when the allocated physical memory cannot fit in the allocated machine-physical memory. DMU raises a *compressed memory fault*, like a page fault, to alert the OS (see Figure 10 (c)) to deallocate some of the process' physical pages (e.g., by spilling out some values) and *cap* how many physical pages to allocate to the process (i.e., *allocate more only after deallocating more*).

Challenges: Architecting a new MMU-like component to allocate machine-physical memory faces several challenges:

(1) MMU exposes a page-based allocation interface where the OS expresses which physical pages to allocate to a process by recording them in a page table and exposing the table to the MMU. Specifying which pages to allocate requires knowing which pages are free; but when the memory controller transparently compresses physical pages to free up machine-physical memory, the OS does not know which machine-physical pages are free. The freed machine-physical pages can also soon be no longer free as the compression ratio fluctuates; correctly cleaning up out-of-date OS records of pages previously exposed as free can be complex due to needing to handle various software-hardware race conditions.

(2) Specifying which machine-physical pages to allocate to each job restricts which machine-physical pages to use for the job. In comparison, prior works without precise allocation can store any data in any free location. Finding/tracking individually for each job the specific machine-physical locations the job is allowed to use can require complex changes to the MC. For example, hardware memory compression maintains many (e.g., 64 [33]) free lists, each to track free spaces of a different size to later use them to store compressed data of matching sizes [33], [35], [40]; maintaining for each job its own full collection of free lists to track free spaces within the specific machine-physical pages allocated to the job is complex.

A. Objective-based Memory Allocation

We note page-based allocation expresses to hardware the higher-level objective of how much memory to allocate ***in an indirect manner***; collectively, the specified set of physical pages indirectly convey to hardware the total physical memory to allocate. Although indirect, specifying ***which physical page*** to allocate allows the OS to also specify ***which virtual page*** to use the page. Traditionally, this leads to a key benefit of the page-based allocation – relieving hardware from making decisions on virtual-to-physical address mappings, which helps keep hardware 'dumb' and simple (see Figure 11.a).

We observe in the context of hardware memory compression, which intelligently manages machine-physical memory, this key benefit of page-based allocation simply disappears; ***hardware*** transparently compressing and packing data more

densely requires ***hardware to actively decide*** the machine-physical address(es) to use for each physical page. Rather than ***simplifying*** hardware, a page-based allocation would ***complicate*** hardware (i.e., causing the two design challenges).

As such, instead of allocating the machine-physical memory ***indirectly by*** specifying individual machine-physical pages, ***we propose*** and architect a more direct ***Objective-based allocation*** to enable OS to directly express the high-level objective of ***how much*** machine-physical memory to allocate (see Figure 11.b).

Specifying ***how much*** machine-physical memory to allocate only requires knowing ***how much*** machine-physical memory is free; exposing to the OS ***how much*** machine-physical memory is free is less complex and much faster than individually exposing ***which*** machine-physical pages are free. Furthermore, the OS specifying only high-level objectives, instead of micro-managing ***which*** machine-physical pages to allocate, gives hardware the freedom to store any data anywhere like before (see Figure 11.b); as such, DMU can keep the same number of free lists as before.

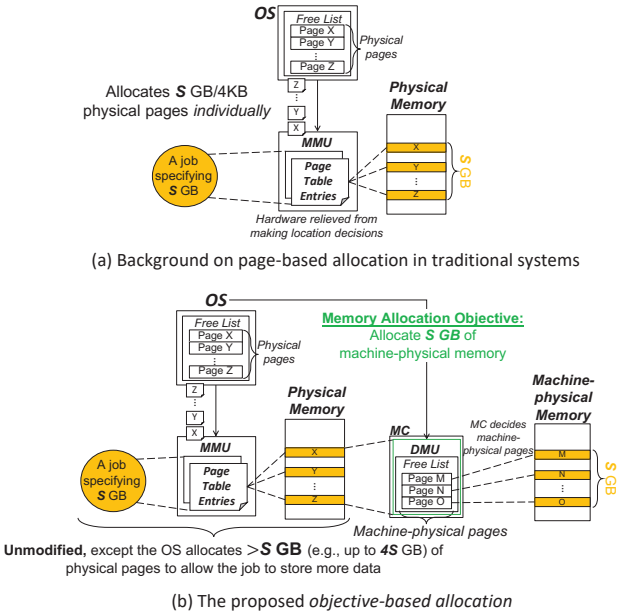

Figure 11: Contrasting different approaches of memory allocation. **(a) Traditional page-based allocation:** OS allocates physical memory by allocating specific physical pages to a job. **(b) Objective-based Allocation:** OS conveys to DMU the high-level objective of how much machine-physical memory to allocate; DMU guides the MC to meet the memory allocation objective.

Figure 12 summarizes the benefits of objective-based allocation over reusing page-based allocation.

1) ***Expressing How Much*** *to Allocate:* Unlike a page table, which has many entries to record the set of allocated physical pages, the OS records the total machine-physical memory to allocate to a process in a single 64B memory block; we call it the *Compression-Objective Control Block* or, simply, *control block*. Each control block contains an 8B field called the *Total Allocation Objective*; it records a single value (e.g., 19 GB)

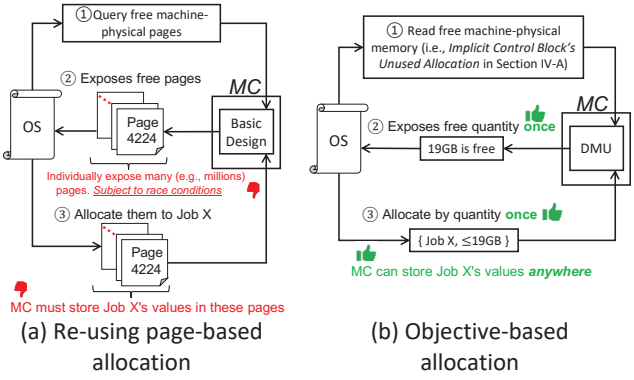

Figure 12: Benefits of using objective-based allocation.

that can be increased or decreased at ***any granularity*** (e.g, 4 KB or 3 MB) through a ***single*** memory allocation.

Like a page table, which records the ***physical memory*** *allocated to* ***the virtual pages*** *used by a process*, each control block records the ***machine-physical memory*** *allocated to* ***the physical pages*** *used by a process*. Since a control block is only 64B, the individual physical pages to be managed by the control block must be recorded elsewhere; instead of adding more hardware data structures, DMU reuses the recency node (see Section 4) of each physical page by adding an OS-writable *control block ID field* (see Figure 13). The recency nodes were chosen because having a control block ID field also enhances them to rank recency locally within each job (see Figure 10.d); Section IV-B will describe in detail how to rank recency within each job.

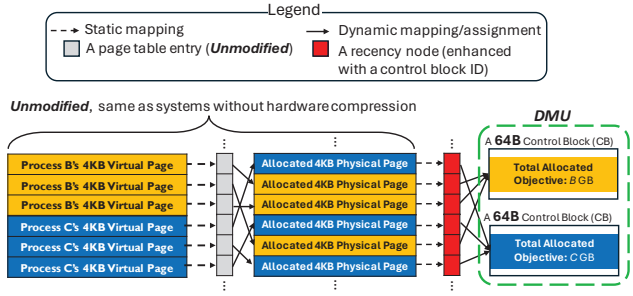

Figure 13: The physical pages of each process share the Total Allocation Objective recorded in a control block. While the OS allocates a physical page to a process, the OS can map the physical page to the process' control block by writing the control block's ID to the physical page's recency node. Core OS structures (e.g., virtual and physical memory allocators, page tables) remain completely intact because DMU manages machine-physical memory as a new layer that is completely independent from the prior virtual and physical memory layers.

The physical pages mapped to a control block can belong to a single process or belong to multiple processes or jobs; as such, a control block serves to enforce ***either an individual allocation objective*** of a single process/job or ***a joint objective*** across multiple jobs. We concisely call all process(es)/job(s)

using the ***same*** control block collectively as one *Collectively-compressed Job (C-job)*.

To expose the recency nodes and control blocks to the OS, DMU maps them to a reserved ***physical memory*** range (see Figure 14). The OS can use existing software APIs to make the address range uncacheable so that when the OS writes to them, the stores go to memory and immediately affect DMU's operations. Like how in a page table entry, some fields are updated by the OS while others (e.g., the accessed and dirty bits) are updated by the MMU, the OS and DMU update different fields within each control block and recency node.

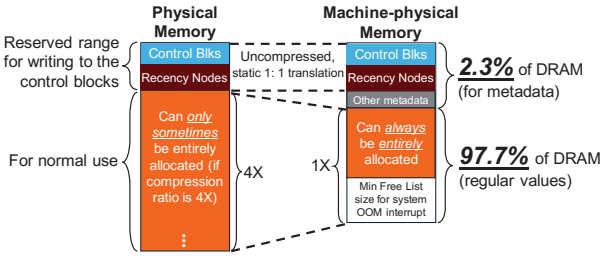

Figure 14: Memory layout under DMU. To support many (e.g., 16384) jobs, the control blocks only statically consume little (e.g., 16384 · 64B = 1MB) DRAM. "Other metadata" refers to other hardware data structures not touched by DMU (e.g., the translation table).

On power up, DMU maps all physical pages to control block 0, which we call the ***implicit control block***. Unlike other control blocks, to which the OS allocates machine-physical memory by writing to their Total Allocation Objectives, DMU implicitly allocates machine-physical memory to the implicit control block. The OS writes to the Total Allocation Objective field in implicit control block ***only ever once*** – the total DRAM in the system discovered from the BIOS to initialize DMU after the OS boots up.

*2) **Exposing How Much** Memory is Free:* A key benefit of specifying ***how much*** machine-physical memory (instead of ***which*** machine-physical pages) to allocate is only needing to expose to the OS ***how much*** machine-physical memory (instead of ***which machine-physical pages***) are free. Exposing ***how much is free*** is fast. When free memory can be exposed very quickly and efficiently, the OS can simply ask DMU (see Figure 12.b ①, ②) how much machine-physical memory is currently free to allocate ***right before*** each memory allocation (see Figure 12.b ③), ***without ever needing to record any previously-exposed free memory***; this avoids having old OS records to clean up when the free memory exposed previously is no longer free (e.g., as compression ratios fluctuate).

Each control block has an Unused Allocation field to dynamically track how much machine-physical memory allocated to a control block is currently unused. This field is ***READ-ONLY to software*** and updated by DMU. For example, when the MC compresses a physical page and frees up Z bytes of machine-physical memory, DMU arithmetically adds Z to the Unused Allocation field of the control block to which the physical page is currently mapped. Table 1 describes how DMU updates Unused Allocation in all the ways that are common across all control blocks (i.e., implicit or not).

Table 1: When and how DMU updates the Unused Allocation in a control block in all the general cases that are ***common across all*** control blocks.

MC and OS Actions	Machine-physical Mem	Unused Allocation
MC compresses a physical page.	Z bytes freed	+= Z bytes
MC spends more machine-physical memory on a physical page (e.g., to make a hot page uncompressed).	X bytes used	–= X bytes
While allocating a physical page, OS maps the page to the control block.	Y bytes used	–= Y bytes
OS deallocates a physical page that is currently mapped to the control block.	Y bytes freed	+= Y bytes

The Unused Allocation in the implicit control block exposes to the OS how much machine-physical memory is currently ready to be allocated (see Figure 12.b ①). When the OS allocates m more bytes to a control block i (i.e., by writing ***T+m*** to its Total Allocation Objective, where T is the current value in this field), DMU subtracts m from the implicit control block's Unused Allocation and adds m to the Unused Allocation of control block i. These simple arithmetic-based memory allocation operations allow the OS to allocate ***in O(1)*** up to all of the Unused Allocation in the implicit control block.

If the host wishes to allocate to a job more machine-physical memory than there is currently available under the implicit control block's Unused Allocation, system software can ask DMU to compress more pages to free up more memory to increase the Unused Allocation. Each control block has an *Unused Allocation Objective* field; DMU asynchronously compresses each block's C-job to increase the block's Unused Allocation to match this objective. This second objective is a best-effort target, rather than a rigorous 'military' objective like Total Allocation Objective; a compressed memory fault is raised only if the latter is unmet, but not if the former is unmet.

Instead of increasing the implicit control block's Unused Allocation Objective, the OS can also increase other control blocks' Unused Allocation Objectives and deallocate from them the freed machine-physical memory. The Unused Allocation in a regular control block exposes to the OS how much machine-physical memory can be deallocated from the block. After the OS deallocates m bytes from a block (i.e., by writing ***T-m*** to its Total Allocation Objective), DMU subtracts m from its Unused Allocation and adds m to that of the implicit block.

Deallocating machine-physical memory from a user job's control block corresponds to a ***potential*** deployment scenario where the host precisely 'steals' from user jobs that have over-specified their memory needs (see Footnote 2 in Section III-B). To support the host with determining how much to 'steal' from a user job without noticeably harming its performance, each control block contains a *#Accesses to Compressed Pages* field to record how many of the accesses to the control block's physical pages are to compressed physical pages; the host may read this field to estimate the potential performance overhead on the control block's C-job due to increasing the block's

Unused Allocation Objective. The host may use the 'stolen' memory to cache more file pages for its own jobs; if a user job later needs its 'stolen' machine-physical memory, the host may evict the file pages to free up machine-physical memory to re-allocate back to the user job.

*3) Allocating **Minimum** Uncompressed Memory:* When a job runs low on uncompressed physical pages, the job can slow down significantly as most accesses will be to compressed pages. In this case, leaving more of the recently-used physical pages uncompressed may be better even if this requires spilling more virtual pages or file pages to storage.

As such, each control block also supports an objective of how many recently-accessed pages to leave uncompressed ***at the very minimum***. Leaving recently-accessed pages uncompressed essentially creates a fast cache; as such, we call the new objective the *Min Uncompressed Cache Objective*. Setting it to 100MB in a control block functionally creates for the block a ***private*** L4 cache with a ***minimum*** of 100MB. Only pages that are deliberately left uncompressed after recent accesses to them (as opposed to incompressible pages) count towards meeting this third and final objective.

B. DMU Backend to Enforce the Allocation Objectives

Today, after the OS allocates physical memory to a process through the MMU, the MMU stores the process' virtual pages into the allocated physical pages.

Similarly, after the OS allocates machine-physical memory to a C-job through DMU, DMU guides the MC to compress its physical pages into its allocated machine-physical memory. The select physical pages to compress in each C-job should be the C-job's coldest pages. Traditionally, each VM or Cgroup has its own thread (e.g., swap daemon) to rank the recency of the virtual pages in the VM or Cgroup and use it to select victim pages [26]. Ranking recency locally within individual VMs or Cgroups (as opposed to globally across all VMs or Cgroups) prevents the swap daemon from excessively swapping out from a VM/Cgroup that is less memory-intensive than another co-located VM/Cgroup. But giving each control block its own compression scheduling hardware, like having its own LRU/swap thread in each VM/Cgroup, can incur costly hardware overhead.

As such, a ***key design challenge*** is how to share the same compression scheduling logic across all control blocks.

To address the challenge, DMU combines all the control blocks and recency nodes in a ***single cohesive*** fan-like structure in Figure 15; DMU asynchronously walks the fan to schedule compression to ensure that within each C-job, compress only as many colder pages as needed, and across C-jobs, the ASIC compressor is used fairly.

To select the coldest page in a C-job, DMU adds to each control block an LRU pointer and an MRU pointer to point to the recency node of the least-recently-used page and the most-recently-used page, respectively, among all uncompressed physical pages currently mapped to the block. Each control block uses these two pointers to connect ***transitively*** to all recency nodes of all the uncompressed physical pages

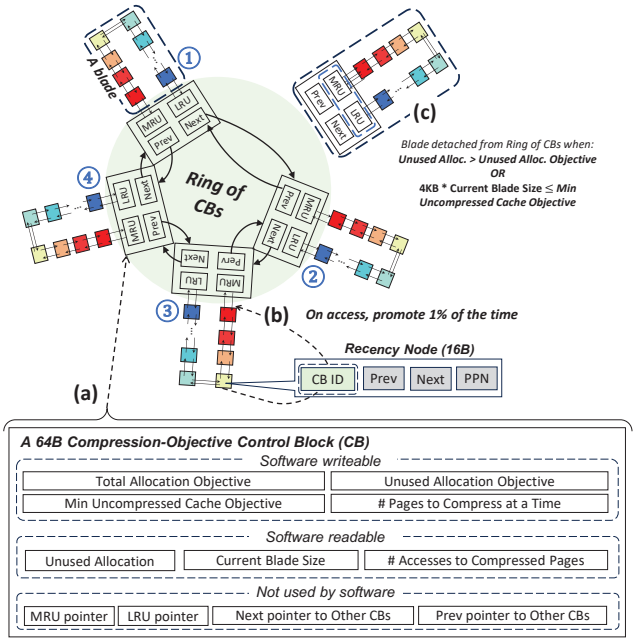

Figure 15: Compression scheduling in a small example with five compressed-objective Control Blocks (CBs). DMU finds the next physical page to compress after chasing at most two pointers. While multiple fields are read to schedule each compression, they are all in one 64B control block, see (a). Combining all OS-settable fields for one C-job into a single 64B control block allows the control block to cache perfectly while the C-job is running.

that are currently mapped to the block; these recency nodes together form a blade in the 'fan'. Unlike prior works (see Figure 4), which have a single global linked list containing the recency nodes of all uncompressed pages, each blade is a smaller linked list that only contains the recency nodes of the uncompressed physical pages mapped to one control block. When the OS writes a new control block ID in a recency node (see Figure 13), DMU joins the recency node to the control block's blade if the physical page is currently uncompressed.

To rank recency locally within a blade, for every 100^{th} normal memory request, DMU logically moves the recency node of the accessed page to the head (MRU end) of a blade (see Figure 15.b) and, thus, logically 'shifts' all other recency nodes towards the tail (LRU) end. If the accessed page is compressed, DMU only joins the recency node of the page to the blade after the page is reverted to uncompressed format because each blade only tracks uncompressed pages.

To obey allocation objectives, DMU adds pointers to each control block to connect to other control blocks in a ring that forms the 'wheel' of the fan in Figure 15. DMU only selects for compression physical pages that are currently mapped to control blocks in the ring. DMU dynamically detaches a control block from the ring according to the block's current objectives (see Figure 15.c).

To schedule compression, DMU fairly round-robins through all control blocks in the ring (see Figure 15 ① to ④) continuously in the background. When visiting a control block, DMU directs the MC to compress an OS-configurable number

of physical pages recorded in the recency nodes at the LRU end of the block's blade. This configurable number – *Pages to Compress During a Visit* – is recorded in each control block. After compressing a physical page, DMU removes the page's recency node from its current blade. If the page turns out to have a low compression ratio (e.g., $< 1.15X$), DMU leaves it uncompressed, but still removes its recency node from the blade to avoid uselessly compressing it again soon.

When a C-job cannot be compressed and stored into its allocated machine-physical memory (i.e., when its Unused Allocation drops to negative), DMU raises a *compressed memory fault*. This is like how when the MMU cannot store a process' values into the process' allocated physical memory (e.g., when the process writes to a virtual page without a physical page), the MMU raises a *page fault* to prevent the store from using more physical memory and to alert the OS.

But unlike page faults in MMUs, which prevent faulting stores from using more memory by aborting them (i.e., deleting their values) and re-executing them later, ***writebacks cannot be re-executed*** as they can take place arbitrarily long after their original stores. As such, DMU serves faulting writebacks and all following writebacks; this can make the control block's Unused Allocation more negative by using more memory. DMU implicitly 'borrows' memory from the implicit control block by reducing its Unused Allocation by the same amount. Conversely, whenever a negative Unused Allocation increases, DMU increases the implicit block's Unused Allocation by the same amount to 'return' the 'borrowed' memory.

The compressed memory fault is an asynchronous interrupt. To avoid interrupt storms, DMU raises an interrupt ***once*** when an Unused Allocation ***flips*** negative, instead of ***continuously*** while it ***remains*** negative. The compressed fault handler routine then spills out some of the faulting C-job's values and also *caps* (e.g., via Cgroups) how many physical pages to allocate to the C-job (i.e., *allocate more physical pages to the C-job only after deallocating more from it*).

The handler need not pause the C-job if the handler can ensure the C-job will not keep growing in an unbounded manner when the C-job keeps running. To ensure this, the handler can first allocate a grace amount (e.g., 10MB) of machine-physical memory to the control block to make its Unused Allocation positive; if the handler receives another compressed memory fault due to the Unused Allocation flipping negative ***again***, only then will the handler pause the control block's C-job. Later, when the spilling of the C-job's values causes the Unused Allocation to rise above 2X the grace amount, the handler deallocates 1X the grace amount to restore the original machine-physical allocation.

The alternative of page-based allocation, which slowly allocates one page at a time, would require pausing the C-job after the very first fault. Otherwise, there is the risk that the C-job may grow faster than the slow memory allocation and make Unused Allocation stay negative ***constantly***, which prevents it from flipping negative (note that flipping negative require first turning positive). Preventing Unused Allocation from flipping negative will prevent a second fault from ever getting raised.

Figure 16 summarizes how DMU enforces all three memory allocation objectives.

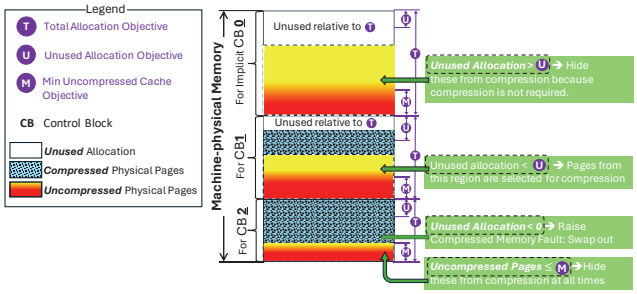

Figure 16: Enforcing the three objectives. Contiguity shown for clarity ***only***.

Figure 17 shows a comprehensive summary of hardware and software interactions under DMU; it also shows how DMU interacts with the underlying hardware memory compression.

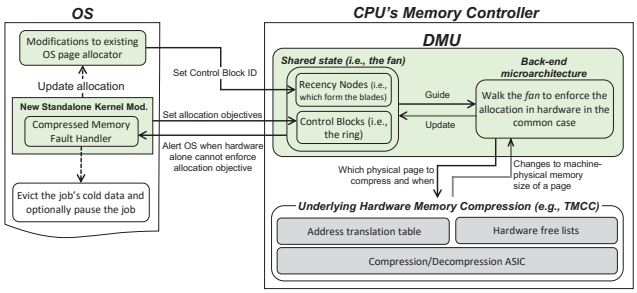

Figure 17: A comprehensive summary of DMU.

C. Discussions

Multiple Memory Controllers: Intel Xeon CPUs [20] have two MCs, each controlling multiple channels. Prior works typically interleave different 4KB physical pages across MCs and interleave an individual 4KB page across all channels within the same MC [33], [34]. To allocate N bytes of machine-physical memory, OS can write the Total Allocation Objective twice, each to a different MC's DMU to allocate $N/2$ bytes of machine-physical memory.

Shared Pages: Different jobs can share the same physical page (e.g., a C library page). But each physical page can only be mapped to one control block at a time because each recency node records only one control block ID; as such, each shared physical page is only 'charged' to one control block, like how Linux 'charges' a shared physical memory only to one Cgroup [16]. Alternatively, the OS may map all shared physical pages used by different jobs to a common control block and allocate to the block enough machine-physical memory so all shared pages stay uncompressed; note shared pages need not be compressed because any degree of sharing already equates to high compression. The OS can then decrease the Total Allocation Objective in each job's control block by the number of shared physical pages the job is using (i.e., decrease the objective by the same number of machine-physical pages).

Virtual Machines (VMs): While Section IV mostly centers on native execution and Cgroups, DMU works the same for VMs except when the VM runs out of memory. The compressed memory fault handler calls the hypervisor to invoke the balloon driver inside the VM; balloon drivers are extensively used by hypervisors today to reclaim memory from VMs. The balloon driver inflates a memory balloon, which uses up the pseudo-physical memory inside the VM and spills the VM's data to the VM's file system or swap space.

D. Future Work

This is the first work that explores precise memory allocation in a system with hardware memory compression. This paper focuses on designing the architecture support; many software optimizations are left to be explored.

Determining the Optimal Objectives for Workloads: Finding the optimal *Total Allocation Objective* for a workload under DMU can rely on the same profiling approach as finding the optimum VM size when using OS memory compression in a VM; when using OS memory compression in a VM, users re-run their workloads in VMs of different memory sizes to find the size that meets the performance requirements and use it for production runs (see Section III-B). For example, we ran $radiosity$ [39] in a 1GB AWS VM with OS memory compression and in a 2GB AWS VM without OS memory compression. Running $radiosity$ in the $1GB$ VM costs only half as much as running it in the $2GB$ VM, while only taking 3% longer. Through this simple profiling, one can conclude that 1GB is a highly cost-effective memory size for $radiosity$.

Under DMU, users can run their workloads under different total allocation objective values to find the optimal value that fulfills their performance needs. Today, profiling is often done manually and can burden users. A potential future work is to explore new software techniques to automate this profiling.

Alternative Way to Use DMU: We focus on the use case of giving users control of how much memory to save via compression; for example, specifying a machine-physical memory size of 2GB for a job that typically uses 6GB of physical memory allows the user to save 4GB. However, giving users this new control can also be a new burden to the users. An alternative use case is let users specify physical memory, instead of machine-physical memory, to express how much logical data to store in memory and let the service providers decide how much machine-physical memory to save; the providers can use the saved space to consolidate/pack more VMs to make the VMs cheaper. However, as compression ratio can fluctuate, saving space and packing VMs too aggressively can necessitate costly VM migrations that degrade QoS. As such, to practically enable this second use case, a potential future work is to explore scheduler-level enhancements to maximize memory savings while maintaining QoS.

V. Results

To evaluate DMU, we create a full-system (i.e., software+hardware) prototype on the FPGA. The prototype boots up Linux with two softcores. A 5-minute video demo of the prototype is available at https://youtu.be/-1JG3JnIY3U. Below, we describe the internal details of the prototype.

Software Side: To utilize DMU, we implement a Machine-physical Memory Module (MPM) and install it in Linux as a new loadable kernel module. MPM assigns control blocks to specified processes and then reads and writes the control blocks to allocate machine-physical memory to the specified processes. We also add 10 lines of code in the existing kernel code (i.e., *mm/page_alloc.c:get_page_from_freelist*) so that whenever a physical page is allocated to a process, the added code gets from MPM the ID of the control block assigned to the process and records the ID in the page's recency node (see Figure 14). Conversely, we add 7 lines of code to *mm/page_alloc.c:__free_one_page* so that whenever the kernel deallocates a physical page, the added code zeroes the page and zeroes the control block ID field in the page's recency node. To accommodate ***large pages*** (e.g., ***huge pages***, ***higher-order pages***), when the kernel allocates or deallocates a large page to or from to a process, the added code runs in a loop to individually ***map/unmap*** every 4KB physical page that is part of the large page ***to/from*** the process' control block.

MPM also handles the compressed memory fault. The handler spills out a job's data by reducing the job's Cgroup's $memory.limit$ to below the job's current physical memory size. Note a job can be a part of multiple Cgroups (e.g., due to hierarchical Cgroups [38]); as such, adding a job to a new Cgroup need not conflict with the job's pre-existing Cgroups.

Hardware Side: We implement DMU on an FPGA (see Figure 18) by enhancing the memory controller RTL of an open-source RISC-V softcore [9]. The DMU prototype supports all three objectives per control block (see Figure 16).

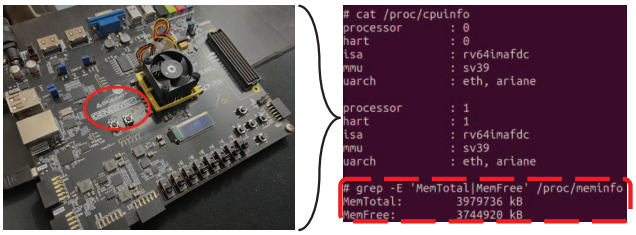

Figure 18: Our full-system prototype on a Genesys 2 Kintex-7™ FPGA. Linux boots up with two cores and ***3979736 KB – 4X the 1GB DRAM on board***.

As for the underlying compression-capable MC that DMU guides, we implement the address translation table of a recency-aware design – TMCC [33], which has a single 8B translation per page, and also TMCC's free lists. [4] Whenever a compressed page is accessed, our prototype decompresses the page in memory (i.e., store the page in memory uncompressed) like TMCC and OS memory compression; after decompressing a page, it joins the page's recency node at the MRU end of the blade. For compression, we integrate only the LZ portion

[4]TMCC's free lists of subpages follow after Linux ZSMalloc [27], [51] algorithm to practically eliminate all fragmentation for storing compressed pages. Our FPGA prototype helps confirm ZSMalloc is feasible in hardware.

of the ASIC Deflate Verilog from [33] as our FPGA is small.

A. Measuring Performance Variation under Collocation

On our FPGA prototype, we measure performance variation under collocation as the memory needs of one of the jobs increases. We run three experiments, where each experiment collocates two jobs; different experiments always keep Job 1 the same but vary how much data Job 2 stores in memory. We measure the change in the performance of Job 1 across the three experiments. Job 1 runs Single-source Shortest Path (SSSP) from GAP[5] [10] with *datagen-8_0-fb* [19] dataset. With this dataset, SSSP has a maximum resident physical memory size (RSS) of 70MB[6]; through profiling, we determine Job 1 performs well when allocated 40MB of machine-physical memory (i.e., by setting the Total Allocation Objectives of control block 1 to 40MB). Job 2 runs the same file-processing program from Section III-B; we allocate 400MB to Job 2.

The first experiment for SSSP uses a ***small*** 600MB file as the input to the file-processing job. We call this first experiment for SSSP the ***reference execution*** for SSSP; we compare other SSSP experiments to this reference execution to evaluate how much variation in performance SSSP can suffer.

The second experiment for SSSP uses a ***medium*** 800MB file as input to the file-processing job. The performance of SSSP remains within 99% of the reference execution (see Figure 19 first green bar). "#Accesses to Compressed Pages" in control block 1 reports 18791, while the reference execution reports 18321; they are within noise range (i.e., 2.5%) of each other. As such, Job 1 is still compressed by the same degree as the reference execution, even though Job 2 is now bigger compared to the reference execution.

The third experiment for SSSP uses a ***large*** 2GB file as input to Job 2. After Job 2 uses 1.6GB of physical memory, it no longer fits in its machine-physical allocation via compression alone. DMU then raises a compressed memory fault to spill out Job 2. But SSSP remains fully in memory; due to precise allocation, SSSP uses the same amount of machine-physical memory regardless of changes in Job 2. As such, SSSP retains 98% of its performance as the reference execution.

We repeat the above by using other GAP benchmarks instead of SSSP. Figure 19 shows when collocated with the ***medium*** file-processing job, GAP benchmarks perform within 99%, on average, of their reference executions; when collocated with the ***large*** file-processing job, it is 98% on average.

As sensitivity analysis, we also evaluate DMU by running the entire[7] SPEC CPU2006[5] Integer suite *concurrently* as Job 1. To account for the increased memory requirement, we precisely allocate 280MB to Job 1 by increasing the Total Allocation Objective of control block 1 to 280MB. Figure 20 shows that with DMU, when collocated with the ***medium*** file-processing job, applications perform within 98%, on average, of their reference executions; when collocated with the ***large*** file-processing job, it is 97% on average.

[5]We do not use the newer GraphBig and SPEC CPU2017 benchmarks because our attempts at compiling them under RISC-V were all unsuccessful.

[6]We choose a small dataset because FPGA softcores are slow.

[7]We exclude gcc, hmmer, libquantum, and h264 as they were still running on the FPGA after 8 hrs even when running standalone using smallest inputs.

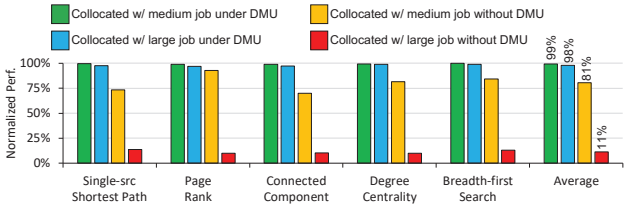

Figure 19: Performance of GAP benchmarks, ***with*** and ***without*** DMU, when collocated with a file-processing job; performance is normalized to when collocated with a small file-processing job ***with DMU***. ***With DMU***, GAP's performance varies only by $1-99\% = 1\%$ and $1-98\% = 2\%$ when collocated with a medium and large file-processing job, respectively. ***Without DMU***, GAP's performance varies by $1-81\% = 19\%$ and $1-11\% = 89\%$ when collocated with a medium and a large file-processing job, respectively. GAP benchmarks have an average ***memory compression ratio*** of 2X. The ***compression ratio*** of Job 2 fluctuates between $3.5X$ and $1X$.

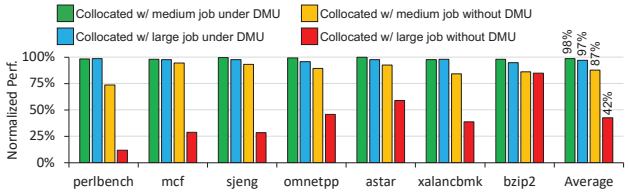

Figure 20: Performance of SPEC benchmarks, ***with*** and ***without*** DMU, when collocated with a file-processing job; performance is normalized to when collocated with a small file-processing job ***with DMU***. SPEC CPU2006 Integer has an average memory compression ratio of $3.1X$.

B. Measuring Performance Variation ***without*** DMU

We repeat the previous experiments without DMU by allocating machine-physical memory indirectly through allocating physical memory. We refer to these as "***without*** DMU" experiments; these experiments use TMCC [33] as the underlying compression and the out-of-memory interrupts proposed in IBM MXT [52] (see Section II). In these experiments, we allocate machine-physical memory ***indirectly through*** allocating physical memory. Following after the optimistic assumption in Section III-C, we allocate to each job up to 4X the physical memory as the specified machine-physical memory by setting the *memory.limit* of Job 1's Cgroup to $4 \cdot 40MB = 160MB$ and Job 2 to $4 \cdot 400MB = 1600MB$.

When collocated with the ***medium*** file-processing job, SSSP retains only $\sim 75\%$ of the performance of its reference execution in Section V-A (see Figure 19 first yellow bar). Unlike DMU, which ranks access recency locally within each job, ***without DMU***, the TMCC baseline ranks recency globally across everything in memory; Job 1 accesses memory less often than Job 2 and, thus, gets more compressed than Job 2. The GAP benchmarks in Job 1 retain, on average, only 81% of the performance of their reference executions.

The performance of SSSP becomes even worse when it is collocated with the ***large*** file-processing job; it only retains 13% of the performance of its reference execution (see Figure 19 first red bar). The large file is too big to fit in memory despite compression; the hardware free list drops very low and triggers a system-level out-of-memory interrupt. This causes the OS to spill out data to Flash, including some of SSSP's memory values. GAP benchmarks only retain, on average, 11% of the performance of their reference executions.

SPEC benchmarks show a similar trend. Figure 20 shows that ***without DMU***, SPEC collocated with the ***medium*** file-processing job only retains 87% the average performance of the reference execution; when collocated with the ***large*** file-processing job, the average performance drops to only 42%.

C. Overheads

Software: Enhancing the kernel so that whenever it allocates a physical page to a process, also map the physical page to a control block can incur a small OS overhead. To evaluate this OS overhead, we run GAP Benchmarks [10], SPEC CPU2017 Benchmarks [49], and GraphBig [30] with *datagen-8_5-fb* [19] dataset in an x86 Linux Server and measure the execution time of each application with and without our added code. The added code incurs 1.3% overhead, on average, and a worst-case overhead of 3%; they are similar with or without transparent huge pages.

Hardware: We synthesize DMU and recency-aware compression using Synopsys [50] and a 7nm process node library [7]; their combined area and frequency are $0.035mm^2$ and 3GHz, excluding the ASIC compressor and decompressor.

VI. General Hardware Compression Questions

How does it compare to OS memory compression? We compare the simulated performance of a recency-aware hardware memory compression (TMCC [33]) with OS memory compression across the same simulation window. We simulate OS compression ***highly optimistically*** (e.g., no cache pollution, compression/decompression calculations in hardware instead of software) by simulating it as TMCC with two differences – ***eliminating address translation overheads*** in the MC and ***adding page fault latency*** to each access to compressed pages. We instrument and measure Linux's ZRAM memory compression to estimate page fault latency in OS memory compression; it is 4us, on average, excluding the latency of software decompression and compression routines. We follow the same simulation methodology and benchmarks as [33]. Our Gem5 [11] results show OS compression retains only 61% the average performance of a bigger memory with no need for compression; hardware-memory compression retains 93%.

How does it interact with OS memory compression? In a system with hardware compression, OS memory compression can be disabled; this is an option in every major OS.

How to evict pages by compressibility? OS today does not swap out pages according to their compression ratios; neither does any prior work on hardware memory compression.

Hardware-encrypted Pages: Memory controllers can encrypt and decrypt memory [5], [21]. To compress an encrypted page, the MC can decrypt it to compress the plaintext, encrypt the compressed page, and store it back to memory as a compressed page.

Cross-layer effects: Hardware memory compression operates DRAM as an independent level of memory beyond physical memory, like how caches operate independently from virtual and physical memory settings. As such, hardware memory compression is agnostic of the settings and object layouts of virtual and physical memory (e.g., page alignments, page table placement). While certain performance optimizations depend on physical memory settings [33], functional correctness does not.

VII. Conclusion

This paper explores and addresses memory allocation problems under hardware memory compression. We architect DMU, an MMU-like component to enable the OS to directly allocate machine-physical memory. Unlike MMUs, which use page-based allocation, DMU uses a new objective-based allocation tailored for this new layer of memory. Our FPGA evaluations show jobs perform stably under colocation, with 1%-2% variation, down from 19%-89% under the prior art.

Acknowledgment

We thank the National Science Foundation (NSF) for generously supporting this work through grants 1942590, 1919113, and 2312785. We also thank Google for generously supporting this work through a Research Scholar Award.

References

[1] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory expansion technology (mxt): Software support and performance," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 287–301, 2001.

[2] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 228–239.

[3] Amazon AWS Lambda, "Configuring lambda function options," 2023. Last accessed on September 10, 2024, https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html#configuration-memory-console.

[4] Amazon Web Services, "Deep dive on amazon ec2," https://www.slideshare.net/AmazonWebServices/deep-dive-on-amazon-ec2.

[5] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," January 2020, https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[6] A. Angerd, A. Arelakis, V. Spiliopoulos, E. Sintorn, and P. Stenstrom, "Gbdi: Going beyond base-delta-immediate compression with global bases," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1115–1127.

[7] Arizona State University Ira A. Fulton Schools of Engineering, "ASAP 7nm PDK," https://asap.asu.edu/.

[8] L. Armasu, "Micron begins mass production of 16gb 1z-class ddr4 ram," 2019. Last accessed on September 10, 2024, https://www.tomshardware.com/news/micron-mass-production-1z-ram-dram,40175.html.

[9] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 217–232, mar 2016. [Online]. Available: https://doi.org/10.1145/2980024.2872414

[10] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017, https://arxiv.org/pdf/1508.03619.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[12] Broadcom, "Esp workload automation 12.0," 2024. Last accessed on September 6, 2024, https://techdocs.broadcom.com/us/en/ca-mainframe-software/automation/ca-workload-automation-esp-edition/12-0/reference/statements/unix-job-statement-start-a-unix-job-definition.html.

[13] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 546–558. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00051

[14] P. Clarke, "Intel, micron launch "bulk-switching" reram," 2015. Last accessed on September 10, 2024, https://www.eetimes.com/intel-micron-launch-bulk-switching-reram/.

[15] I. Cutress, "The amd zen and ryzen 7 review: A deep dive on 1800x, 1700x and 1700," 2017. Last accessed on September 10, 2024, https://www.anandtech.com/show/11170/the-amd-zen-and-ryzen-7-review-a-deep-dive-on-1800x-1700x-and-1700.

[16] L. K. Documentation, "Memory resource controller," 2023. Last accessed on September 10, 2024, https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[17] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. USA: IEEE Computer Society, 2005, p. 74–85. [Online]. Available: https://doi.org/10.1109/ISCA.2005.6

[18] A. Frumusanu and G. Bonshor, "The amd 3rd gen ryzen deep dive review: 3700x and 3900x raising the bar," 2019. Last accessed on September 10, 2024, https://www.anandtech.com/show/14605/the-and-ryzen-3700x-3900x-review-raising-the-bar.

[19] GraphBIG, "Ldbc graphalytics," https://graphalytics.org/datasets.

[20] Intel, "Intel xeon processor scalable family overview," 2023. Last accessed on September 10, 2024, https://www.intel.com/content/www/us/en/developer/articles/news/second-generation-intel-xeon-processor-scalable-family-technical-overview.html.

[21] Intel, "Intel trust domain extensions," 2024. Last accessed on September 12, 2024, https://cdrdv2.intel.com/v1/dl/getContent/690419.

[22] D. James and J. Choe, "Techinsights memory technology update from iedm18," 2019. Last accessed on September 10, 2024, https://www.techinsights.com/blog/techinsights-memory-technology-update-iedm18.

[23] J. Kim, M. Kang, J. Hong, and S. Kim, "Exploiting inter-block entropy to enhance the compressibility of blocks with diverse data," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 1100–1114.

[24] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent dual memory compression architecture," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 206–218.

[25] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 317–330. [Online]. Available: https://doi.org/10.1145/3297858.3304053

[26] Linux Kernel Documentation, "Memory resource controller," https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt.

[27] LWN.net, "The zsmalloc allocator." [Online]. Available: https://lwn.net/Articles/477067/

[28] T. Mann, "Non-binary ddr5 is finally coming to save your wallet," 2023. Last accessed on September 10, 2024, https://www.theregister.com/2023/01/02/nonbinary_ddr5_is_finally_coming/.

[29] MIT, "Using SLURM to Submit Jobs," https://svante.mit.edu/use_slurm.html.

[30] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807626

[31] Oracle Cloud Infrastructure, "Why oracle cloud infrastructure over microsoft azure," https://www.oracle.com/cloud/oci-vs-microsoft-azure/.

[32] G. Panwar, "Utilization-adaptive memory architectures," Ph.D. dissertation, Virginia Tech, 2024. [Online]. Available: https://hdl.handle.net/10919/119460

[33] G. Panwar, M. Laghari, D. Bears, Y. Liu, C. Jearls, E. Choukse, K. W. Cameron, A. R. Butt, and X. Jian, "Translation-optimized memory compression for capacity," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 992–1011. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00073

[34] G. Panwar, M. Laghari, E. Choukse, and X. Jian, "Dylect: Achieving huge-page-like translation performance for hardware-compressed memory," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 1129–1143. [Online]. Available: https://doi.org/10.1109/ISCA59077.2024.00085

[35] S. Park, I. Kang, Y. Moon, J. H. Ahn, and G. E. Suh, "Bcd deduplication: effective memory compression using partial cache-line deduplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 52–64. [Online]. Available: https://doi.org/10.1145/3445814.3446722

[36] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 188–200.

[37] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.

[38] R. H. C. Portal, "Relationships between subsystems, hierarchies, control groups and tasks," 2023. Last accessed on September 10, 2024, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks.

[39] Princeton, "Parsec 3.0," http://parsec.cs.princeton.edu/.

[40] C. Qian, L. Huang, Q. Yu, Z. Wang, and B. Childers, "Cmh: Compression management for improving capacity in the hybrid memory cube," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 121–128. [Online]. Available: https://doi.org/10.1145/3203217.3203235

[41] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 325–334. [Online]. Available: https://doi.org/10.1145/2370816.2370864

[42] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "Memzip: Exploring unconventional benefits from memory compression," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 638–649.

[43] A. Shilov, "Samsung preps pcie 4.0 and 5.0 ssds with 176-layer v-nand," 2021. Last accessed on September 10, 2024, https://www.tomshardware.com/news/samsung-envisions-1000-layer-v-nand.

[44] A. Shilov, "Samsung preps next-gen v-nand memory: Higher capacity and performance," 2022. Last accessed on September 10, 2024, https://www.tomshardware.com/news/samsung-preps-v8-nand-memory.

[45] A. Shilov, "Amd: Zen 5-based cpus for client and server applications on-track for 2024," 2024. Last accessed on April 4, 2024, https://www.anandtech.com/show/21251/amd-zen-5-based-cpus-for-client-and-server-applications-due-in-2024.

[46] A. Shilov, "Samsung says 300+ layer v-nand is on track for 2024," 2024. Last accessed on September 10, 2024, https://www.tomshardware.com/news/samsung-says-300-layer-v-nand-is-on-track-for-2024.

[47] R. Smith, "Intel announces optane storage brand for 3d xpoint products," 2015. Last accessed on September 10, 2024, https://www.anandtech.com/show/9541/intel-announces-optane-storage-brand-for-3d-xpoint-products.

[48] R. Smith and G. Bonshor, "Amd zen 4 ryzen 9 7950x and ryzen 5 7600x review: Retaking the high-end," 2022. Last accessed on September 10, 2024, https://www.anandtech.com/show/17585/amd-zen-4-ryzen-9-7950x-and-ryzen-5-7600x-review-retaking-the-high-end.
[49] Standard Performance Evaluation Corporation, "Spec cpu 2017," https://www.spec.org/cpu2017/.
[50] Synopsis, "Synopsis design compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.
[51] the linux kernel documentation, "zsmalloc." [Online]. Available: https://www.kernel.org/doc/html/v4.19/vm/zsmalloc.html
[52] R. Tremaine, T. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: Ibm mxt in a memory controller chip," *IEEE Micro*, vol. 21, no. 2, pp. 56–68, 2001.
[53] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 41–53. [Online]. Available: https://doi.org/10.1145/2749469.2750399
[54] W. Vogels, "Tweet by amazon's chief technical officer," https://twitter.com/Werner/status/25137574680.
[55] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 609–621. [Online]. Available: https://doi.org/10.1145/3503222.3507731
[56] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.
[57] V. Young, S. Kariyappa, and M. K. Qureshi, "CRAM: efficient hardware-based memory compression for bandwidth enhancement," *CoRR*, vol. abs/1807.07685, 2018. [Online]. Available: http://arxiv.org/abs/1807.07685
[58] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 570–581.
[59] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, oct 2015. [Online]. Available: https://doi.org/10.1145/2808233

APPENDIX

A. Abstract

Our artifact demonstrates the video experiment (available at: https://youtu.be/-1JG3JnIY3U) from Section V, which shows how DMU restores precise memory allocation in a system with hardware memory compression. This artifact demonstrates the following three objectives from the paper: *Total Allocation Objective*, *Unused Allocation Objective*, and *Minimum Uncompressed Cache Allocation Objective*, just like in the video.[8]

We also release the RISC-V bitstream, the OS binary, and the kernel module binaries to reproduce the results of this experiment and to facilitate further exploration after artifact evaluation. The released artifacts are available at https://github.com/HEAP-Lab-VT/DMU and at https://doi.org/10.5281/zenodo.13753990.

[8]The Results section (i.e., Section V) of the paper presents two main results: the video demonstrating DMU successfully enforcing the objectives, and the main performance results (i.e., Figure 19). Due to limited time before the camera-ready and limited FPGA resources, our artifact evaluation only includes the first main results, instead of Figure 19, which is much more time-consuming. Since the *Results Reproduced* badge requires **all** main results to be evaluated, we were ineligible to receive it. Nonetheless, the evaluators successfully reproduced the results for the experiment outlined in the Appendix, (i.e., first main result).

B. Artifact check-list (meta-information)

- **Program:** SPEC CPU2006 *mcf*. It is included in the evaluation setup.
- **Compilation:** No compilation required.
- **Binary:** All the required binaries are present at https://doi.org/10.5281/zenodo.13753990. The binaries used are: (1) OS binary, (2) Linux kernel module binaries, (3) *mcf* and microbenchmark binaries, and (4) FPGA bitstream.
- **Data set:** Dataset (*inp.in*) for *mcf* is included in the shared DOI link.
- **Run-time environment:** Ubuntu with Linux kernel 5.1.0.
- **Hardware:** (1) Genesys 2 Kintex-7 FPGA, (2) MicroSD card reader/writer, (3) a Linux machine to connect the FPGA board.
- **Execution:** Detailed instructions are provided in `README` at https://doi.org/10.5281/zenodo.13753990.
- **Metrics:** Following metrics will be evaluated: (1) Physical memory usage statistics reported by the OS, and (2) DRAM usage statistics reported by DMU's kernel module. The commands used to query these metrics are provided inside `README` file.
- **Output:** The output is reported on the console; no output file is created at the end of the experiment.
- **Experiments:** Details of how to run the experiment (i.e., commands and scripts) are present at https://doi.org/10.5281/zenodo.13753990.
- **How much disk space required (approximately)?:** (1) 40GB for Vivado Design Suite, (2) 1GB on the MicroSD card for relevant binaries.
- **How much time is needed to prepare workflow (approximately)?:** (1) 2 hours to download and install Vivado Design Suite. And (2) 10 minutes to prepare the MicroSD card.
- **How much time is needed to complete experiments (approximately)?:** 1-2 hours
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Yes, available at: https://doi.org/10.5281/zenodo.13753990.

C. Description

- **How to access:** The binaries and a `README` file are present at https://doi.org/10.5281/zenodo.13753990.
- **Hardware dependencies:** A Linux machine, a Genesys2 Kintex-7 FPGA board, and a MicroSD card.
- **Software dependencies** Vivado Design Suite - HLx Editions 2020.2.
- **Data sets:** SPEC 2006 *mcf* with *inp.in* (binary provided at https://doi.org/10.5281/zenodo.13753990).

D. How to access

Please download the required binaries and follow the installation instructions on https://doi.org/10.5281/zenodo.13753990.

E. Experiment workflow

Detailed instructions are provided in the `README` at https://doi.org/10.5281/zenodo.13753990 on how to setup the experiment workflow.

F. Evaluation and Expected Results

- Co-located workloads receive the amount of machine-physical memory they specify in the presence of hardware memory compression, ensuring precise memory allocation.
- Workloads are compressed only when they can no longer fit within their specified machine-physical memory in plaintext (i.e., in their uncompressed format), ensuring high-performance access to the memory values of the workloads when compression is not required.
- When a workload can no longer be further compressed (i.e., when the memory values of a workload are fully compressed), the system swaps out only the memory values of that workload (i.e., all other co-located workloads remain in memory and are not swapped out), instead of crashing the workload.
- The combined resident set size (RSS) of all workloads (i.e., the aggregate amount of memory used by all workloads) is significantly larger (i.e., up to 4 times greater) than the amount of DRAM installed on the FPGA board.