The Need for Precise and Efficient Memory Capacity Budgeting

Shaleen Garg
Rutgers University
New Brunswick, New Jersey, USA
shaleen.garg@rutgers.edu

Manish Parashar
Rutgers University
New Brunswick, New Jersey, USA
parashar@ored.rutgers.edu

Sudarsun Kannan
Rutgers University
New Brunswick, New Jersey, USA
sudarsun.kannan@rutgers.edu

ABSTRACT

Modern high performance computing (HPC) systems pack hundreds of CPU cores to enable extreme parallelism. However, with increasing core counts, the effective per-core memory capacity is reducing. Reducing performance bottlenecks require precise monitoring and budgeting of application memory capacity requirements for attaining high performance, maximum resource efficiency, and low performance variability. Unfortunately, current operating systems (OS) and their toolsets are inaccurate, lack the capability to precisely measure the memory requirements of applications, forcing system administrators to either underestimate or over-provision memory, consequently compromising performance or resource efficiency, respectively.

In this paper, we decipher the memory budgeting limitations in current OSes and their impact on both homogeneous and heterogeneous memory systems (e.g., nonvolatile memory). The limitations mainly stem from the mismatch between application-level and global memory accounting in the OS memory manager, fixing which can be prohibitively expensive at runtime. Our analysis of popular HPC workloads using widely-used memory budgeting strategies and deep instrumentation of the memory management layer reveals that imprecise budgeting can reduce performance by more than 1.65x and 2.05x in homogeneous and heterogeneous memory systems respectively. The program's memory requirement increases by up to 25x without significant performance gains. We also briefly describe our ongoing research approach to redesign the budgeting mechanisms in the OS.

CCS CONCEPTS

• **Software and its engineering → Memory management**; **Virtual memory**.

KEYWORDS

HPC, Memory, Resource Efficiency, Linux, Performance

ACM Reference Format:
Shaleen Garg, Manish Parashar, and Sudarsun Kannan. 2020. The Need for Precise and Efficient Memory Capacity Budgeting. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3422575.3422791

1 INTRODUCTION

We have entered an era of computing where three trends dominate the modern high-performance computing systems. The first is an application trend, where applications with lots of parallelism must quickly access large amounts of data. Modern HPC applications that run across hundreds and thousands of nodes require huge amounts of memory resources. Memory is not only used by applications but also used by other OS subsystems that include the storage stack (e.g., file systems), the network stack (e.g., network subsystem), accelerator drivers (e.g., GPU, FPGA), OS schedulers, and even the virtual memory subsystem (e.g., page table management). However, with increasing core count, the effective per-core memory capacity is reducing, which can significantly impact performance [25, 42].

The second trend is the hardware trend, where adding new heterogeneous memory technologies with different capacity, bandwidth, and latency characteristics provide performance and capacity benefits but complicate software management. This includes recently released Intel DC Optane memory [2], promising die-stacked high bandwidth memory technologies with significantly lower capacity than DRAM [4, 5], and integrated CPU-GPU memory [16].

The third trend is the software trend, in which the complexity of software memory management, such as memory virtualization design, is ever increasing. Unfortunately, today's OS memory managers do not satisfy expected *fundamental properties*, such as resource efficiency, high-performance, and non-variable performance. Designing a generic solution to satisfy these fundamental properties for both homogeneous and heterogeneous memory is a complex grand challenge.

Problem Focus: The first step towards solving this grand challenge requires an understanding of the memory resource requirements of applications in both homogeneous and heterogeneous settings. Surprisingly, despite years of design, optimization, and development, current operating systems (OSes), memory management subsystems, and budgeting tools are ill-equipped and imprecise. Lack of precise memory resource estimation capabilities often force administrators and application developers to over-provision or under-provision memory, which could impact performance, resource efficiency, or performance predictability [1]. Our analysis of several commonly used Linux tools, system call APIs, and our deep OS instrumentation reveals that current memory accounting mechanisms are inaccurate. Underestimating memory usage leads to undesired effects such as applications swapping to secondary storage, or even application and system termination. Overprovisioning memory potentially increases operational cost forcing administrators to add memory hardware or spill application to additional physical nodes, consequently increasing communication costs.

Factors for Imprecise Memory Budgeting. The inaccuracies in current Linux tools and OSes stem from several factors. First, several widely used user-level tools and OS system calls that estimate the resident set size (i.e., maximum active pages) localize accounting to application-level pages and do not precisely account memory usage by other subsystems. This includes the file system, network subsystem, device drivers, accelerator modules, OS data structures, and swap managers that facilitate application access to hardware and other services, resulting in resource contention across applications and subsystems. Consequently, using these tools to budget memory impacts application performance.

Second, in addition to application-level counters, OSes also maintain a global list and counters for system-wide tracking of active and inactive memory and expose this information to users. These counters are primarily updated by the OS page reclamation engines to reclaim inactive memory when the free memory availability is scarce. Updating these counters requires a global walk across the active and inactive list and other bookkeeping operations. Due to forbiddingly high overhead, OSes such as Linux reduce the frequency at which pages are scanned, active, inactive lists and the related global counters are updated. The global counters are imprecise and, as we show, can significantly overestimate the memory requirements of applications, impacting resource efficiency.

Finally, OSes use rigid policies that always prioritize the allocation of heap memory by forcing active pages mapped to files to be cleaned and reused when memory is scarce. Consequently, this impacts application performance due to reduced I/O performance. This combined with a lack of precise and efficient monitoring of application resident sets and global counters, impedes I/O performance and the overall application performance.

Our Analysis. In this paper, we systematically evaluate and characterize the inaccuracies of existing memory budgeting mechanisms to understand the implications of the limitations mentioned above. We study widely used user-space tools, OS-level counters, and perform deep kernel instrumentation. For our analysis, we use well-known HPC benchmarks and applications that are compute and I/O intensive. To understand the impact, we conduct experiments on a single and multi-node homogeneous memory system, as well as a heterogeneous system equipped with Intel DC Optane persistent memory. Our results show that imprecise estimation of memory with current tools can increase memory consumption by up to 25x, whereas the runtime increases by up to 1.65x and 2.05x in homogeneous and heterogeneous memory systems, respectively. The performance variability increases by 1.21x.

In this position paper, we mainly focus on the imprecise memory budgeting in current OSes. Our on-going research is tackling solution to these issues by redesigning and unifying OS virtual memory management budgeting mechanisms across different layers, and also extending the LRU management [18].

2 BACKGROUND AND RELATED WORK

We first provide a brief background on the working set and resident set size estimation in current OSes such as Linux, different types of memory usage, and how they integrate with Linux virtual memory management. We briefly describe the memory heterogeneity trend to drive home the point that efficient and accurate memory budgeting is critical for future systems.

2.1 Working Set

One of the first models to identify a program's working set was from the seminal work done by Peter Denning [13]. The working set model has been adopted in different forms across most OSes today [14, 23]. A more straightforward interpretation of the working set model is the following. Applications are assigned memory in units of pages. Not all pages can fit in memory, and those not available in memory are fetched from disk. Moving pages from memory to disk incurs a cost and could potentially stall the program. To minimize the stalls, Denning proposed the concept of the working set, which encompasses a set of pages required by an application to continue execution. More formally, Denning's working set model assumes that a page used in the past T time units is expected to be referenced again within α time units ($T >> \alpha$). An ideal working set prediction model could predict all pages referenced in the next α time units without requiring the program to stall for the disk.

2.2 Realization of Working Set in Linux

Practical realization of the working set model is difficult; specifically, predicting pages required for future memory accesses requires an extensive static analysis of a workload. Therefore, most OSes implement a simpler working set model to realize Denning's original design. For example, Linux (and FreeBSD) implements a simplified LRU 2Q design originally proposed by Theodore et al. [23]. The goal of OS LRU 2Q mechanism, at a high level, is to separate a set of active and inactive pages, such that the inactive pages can be swapped or reclaimed. Linux maintains an active LRU list (pages that cannot be swapped and are in active use), and an inactive LRU list of pages.

In addition, the mechanism also accounts for pages that are potentially reclaimable ie., they can be swapped to disk when freely available memory capacity is low. When pages in the inactive list are referenced, they are promoted to the active LRU list, whereas those pages in the active list, which are infrequently accessed, are demoted to the inactive-list. The pages in these lists are reclaimable, which indicates that pages can be moved to disk when memory pressure is high. Because these reclaimable pages are contained in just two global lists, any page belonging to any process may be reclaimed, rather than only those belonging to a process that increased the overall system memory pressure. Further, OSes maintain a third list of non-reclaimable pages (non-LRU active list). Non-reclaimable pages include pages currently being modified by the application, the pages that are explicitly locked by applications, or the pages used for maintaining OS data structures.

Policies. The policies surrounding the LRU mechanism, such as what fraction of pages are marked as active or inactive and when they must be reclaimed, are rather empirical and static. The policies have shown to work well in practice, and adjustments have been made based on user feedback and benchmarks [19]. While the overall LRU mechanism is generic, over the years, these mechanisms have swayed towards swap management. The swap manager moves inactive pages to slower memory when the memory is scarce or overcommitted. For example, several real-world applications oscillate between periods of high allocation (before computation), followed by periods of infrequent allocations. To satisfy allocation requests quickly, OSes such as Linux maintain the number of pages in the active LRU list about two-thirds the size of the inactive LRU list. During high page allocation intervals, the pages in the inactive LRU list can be quickly reclaimed (by swapping the pages). While this two-third policy has worked well for many applications, OSes also provide flexibility for users (or administrators) to control active to inactive page ratios and tune them to a specific application.

2.3 Memory Provisioning

The first step towards attaining high performance without compromising resource efficiency and performance variability is understanding and precisely measuring the direct and indirect memory costs of an application. Direct memory refers to memory directly allocated by the application, whereas indirect memory needs refer to memory allocated by different libraries and OS subsystems required for application execution and performance. While there have been prior work on containers and Linux cgroups [21] and VMs [35], to the best of our knowledge, other research directions have not explored the fallacies of the core OS accounting system (which is assumed to be correct). Recent industrial efforts such as as [1] have recently proposed indirect metrics such as memory pressure. While definitely valuable, in this paper, we describe the main issues with current budgeting and accounting mechanisms.

Capturing memory requirements is critical because most modern HPC applications are not only compute-intensive but also I/O intensive (storage and network) [26, 27, 38, 39]. Applications are increasingly becoming accelerator-intensive [37], requiring frequent movement of huge amounts of data between host and accelerators. Applications, not only consume memory from heap allocations (commonly referred to as anonymous memory), but also across userspace libraries, storage stack (e.g., page cache), network stack (network buffers, RDMA buffers), DMA buffers, and memory buffers for kernel data structures. Memory needs of applications can substantially change depending on the number of threads, input data, and execution paths adopted by the application, and the intensity with which storage, network, and accelerator devices are accessed. These factors complicate precise memory budgeting.

User-level Tools. Several user-level tools such as Valgrind [30], `sacct` in SLURM [41], memusage [29], and gnu time [17] are widely used to intercept allocations of a program and capture memory usage. While these tools have become sophisticated to capture the memory allocation needs of supporting libraries (e.g., MPI libraries for HPC applications), unfortunately, they lack the capability to capture the memory allocations outside the application's bounds, which includes OS subsystems, drivers, and shared data across applications.

Capturing Resident Set Size. To overcome the problem of the lack of information about memory use, the Linux kernel community [3, 20] and industry [1] has relentlessly aimed to develop tools to capture the real memory usage of applications. Because the prediction of a working set is hard, OSes such as Linux and FreeBSD moved away from the working set size model to a process resident set size (RSS) model. A process's RSS is the number of physical memory pages currently owned by that process. The process resident set size provides a high-level estimate of pages referenced by a process that includes heap , file-backed pages, and shared memory. While seemingly simple, a significant difference exists between a process's RSS and the amount of memory actually in use. The reasons stem from the fact that allocated pages are not immediately referenced by an application or pages released by an application are not immediately reclaimed by the OS. Consequently (as we will show), the overall memory usage of applications can be significantly higher compared to OS reported resident set sizes. To overcome these issues, the OS community has been developing several fine-grained system-level memory usage tools and process-level tools. For example, recent kernels export process-level information on the pages actually referenced by an application (via. smaps [3]).

2.4 Memory Hardware Trends

As systems continue to embrace memory heterogeneity, accurate memory provisioning is critical given the substantial capacity and performance differences across heterogeneous memory technologies. For example, technologies such as die-stacked 3D-DRAM, Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM), and byte- addressable NVMs showing early promise in addressing the big-data needs of modern applications [16, 32, 34, 36]. While offering research promise, these devices pose a myriad of complex performance and capacity tradeoffs. For example, technologies like 3D-DRAM, HMC, and HBM provide 10× higher bandwidth and 1.5× lower latency than conventional DRAM, but suffer 8-16× [4, 6, 10, 31] lower capacity. Meanwhile, byte-addressable NVMs offer 4-8× higher capacity than DRAM but suffer 2-3× higher read latency, 5× higher write latency, and 3-5× reductions in access bandwidth. For heterogeneous memory systems, underestimation or over-provisioning of application memory can substantially hurt application performance and lead to overall system inefficiency.

2.5 Working Set Detection in Heterogeneous Memory Systems.

Working set size detection has been explored in other contexts. For example, more recently, working set detection for page placement in a heterogeneous memory system has been actively explored [5, 40]. The working set is used for placing hot pages to fast memory and cold pages to slower memory. Working set detection involves forcing TLB faults [15] on application (to force them to access the page table), and scanning through the page table for many reference pages. Unfortunately, inducing TLB faults and scanning the page table (for large workloads) are even more expensive than LRU-based tracking. To reduce overheads, prior research such as Thermostat [5] uses a 30-second interval between scans. Other systems such as Nimble [40] also depend on Linux LRU for page migration in heterogeneous memory, thereby inheriting the budgeting issues.

3 MOTIVATION

Next, we discuss the limitations of current memory budgeting tools, motivating the need for precise and efficient memory budgeting mechanisms. We observe that there is a lack of research focus on such practical issues, which is critical for achieving high performance and resource efficiency.

3.1 Lack of Preciseness

Current system software tools and OSes lack the capability to precisely measure memory usage for a given workload. The lack of precise estimation of memory usage impacts performance, overall system resource efficiency, and introduces performance variability. Performance impact and variability are introduced due to underestimating the memory requirements, whereas resource efficiency stems from overestimating memory requirements. For example, the maximum resident set size (**MaxRSS**) reported by the OS for an application is generally imprecise and often conservative. This could lead to a contention between the program's performance-critical active heap and I/O memory pages. Under extreme memory pressure, the application could be eventually terminated, at times, freezing the entire system.

Similarly, when memory use is unbounded, applications could run with the maximum performance, but amplify memory capacity use. *Memory capacity amplification* can be defined as the ratio between the memory consumed by an application to actual memory required to run the application to completion without swapping. Generally, administrators and users resolve memory capacity issues by provisioning additional memory hardware or spilling the application to additional physical nodes [12]. Consequently, when capacity amplification is high, resource efficiency is compromised. To understand the reasons for the lack of precise memory requirement information, we briefly discuss how the program's resident set size is estimated, followed by the limitations in the current OSes.

Memory Estimation. Current OSes track memory pages and their references using 2Q (queue) LRU design, which orders pages based on their access frequency. The first LRU queue, called the `active-list`, represents the most recently used pages, representing the union of all program's working sets. The second queue, called the `inactive-list`, maintains pages that were used and allocated at some point but have not been used for a long duration. Because pages go through intervals of active and inactive use, promoting recently referenced pages from `inactive-list` to the `active-list` and vice-versa happens periodically. To conserve memory, during cleanup operations, first, the clean file-backed pages are released. Second, file-mapped pages in the active-list that are not referenced recently are moved to the inactive-list. Finally, file-backed pages that are dirty (with uncommitted data) are flushed to disk, marked clean, but are not reclaimed immediately. While seemingly simple, the active and inactive lists are globally maintained and shared across the entire system. Maintaining a global list, walking through, and identifying active and inactive pages is expensive and incurs substantially higher CPU cycles. We measured the page walk cycles using in-kernel instrumentation. Our analysis shows that the cost of walking through the active and inactive lists in a 5-second interval incurs 22% performance slowdown for a graph benchmark [7].

To combat such high overheads, OSes reduce the frequency of 2Q walk cycles, consequently impacting the preciseness of active (in-use) and inactive page usage information. This impreciseness leads to two effects. First, the application-level MaxRSS calculation is restricted to an application's heap, file-backed, and shared memory pages, failing to account in the several OS-level pages allocated between the active-inactive scan intervals and under-estimate the memory requirements of the application. Second, system-wide memory usage tools overestimate memory requirements by accounting for infrequently referenced pages that are not identified as inactive due to infrequent scans.

3.2 Heap and I/O Subsystem Contention

For I/O intensive applications, when the memory is scarce, there is a constant contention for allocating free pages between application's heap and I/O (e.g., file-mapped) pages. In current memory managers, when memory is scarce, the heap page allocation requests are prioritized over allocation requests from the I/O subsystem [24] such as I/O page-cache pages. Similarly, I/O pages are persisted and evicted first before any heap pages are swapped irrespective of their inactive status. Imprecise and infrequent identification of active and inactive pages combined with prioritizing heap pages over I/O pages leads to consistent victimization of I/O pages being flushed or reclaimed. As a result, for I/O-intensive applications, the time spent waiting for I/O to complete increases, reducing the application perceived I/O bandwidth significantly.

4 ANALYSIS

For optimal application performance without compromising memory efficiency or introducing performance variability in traditional homogeneous and heterogeneous memory systems, precise estimation of memory usage is critical. We perform a detailed workload characterization of widely used HPC benchmarks & applications and decipher the limitations of current memory capacity budgeting techniques. Specifically, we evaluate their impact on memory consumption, performance, and variability for homogeneous and heterogeneous memory systems.

4.1 Goals

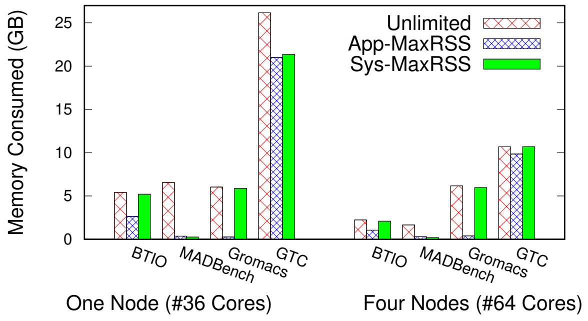
Our analysis aims to answer the following questions.

- How does memory consumption vary across different budgeting approaches?
- What is the performance impact of inaccurate capacity budgeting techniques across one or more physical machines?
- How imprecise capacity budgeting impacts performance in heterogeneous memory systems, such as non-volatile memory (NVM)?

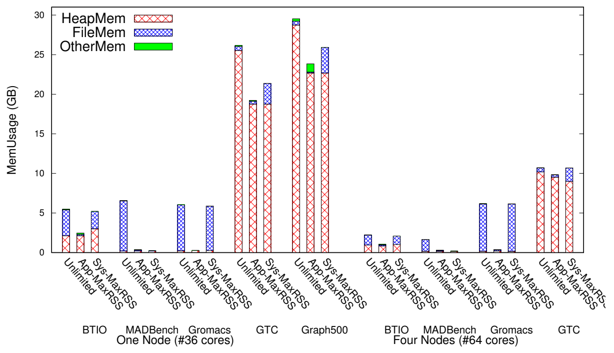
4.2 Experimental Setup

We use different system configurations for the single node, multi-node, and heterogeneous memory analysis. For the single node analysis, we use a 40-core Intel Xeon 2.67 GHz dual-socket system, with 80GB memory per socket and a 512 GB Intel SSD. For the evaluation across multiple nodes, we use four instances of 16-core Intel Xeon 2.0 GHz single-socket system with 64GB ECC memory and 256 GB NVMe flash drive. Our analysis also involves deep kernel-level instrumentation via. custom system calls, limiting our current scale of analysis. In both single and multi-node analysis, to vary the memory capacity of the systems and restrict the available memory to a process, we block the memory beyond the allocated budget by mounting a RAM-based file system and hiding the additional memory from the virtual memory's view. We describe the method of deriving memory budgets shortly.

Heterogeneous Memory Setup. To understand the impact of memory capacity on heterogeneous memory systems, we use a dual-socket, 64-core, 2.7GHz Intel(R) Xeon(R) Gold platform with 32GB DRAM, 512GB (4x128GB) DC Optane persistent memory with



(a) Capacity Usage Across Budgeting Strategies.



(b) Capacity Usage - Page-type Breakdown.

Figure 1: Memory Capacity Usage Across Budgeting Mechanisms. Memory Consumption and their breakdowns when using different techniques are shown in (Fig 1a) and (Fig 1b) respectively. Two types of settings, one node (#36 cores) and four nodes (#64 cores) are used. Unlimited is an unbounded run where there is abundant memory for programs to expand as desired. For, App-MaxRSS, the memory budget represents application-level accounting. Sys-MaxRSS represents the memory budget derived from global heap, file, and other process relevant pages through our deep kernel instrumentation.

8GB/sec read and 3.8GB/sec rand-write bandwidth [22], and 512GB SSD.

4.3 Analysis Workloads

For our analysis, we use the following five widely-used workloads known to be compute and I/O-intensive.

BTIO Benchmark. The NAS Parallel Benchmarks (NPB) [8] are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications. The problem sizes in NPB are predefined and indicated as different classes. We specifically use the BTIO benchmark, a Block Tri-diagonal solver that stores data to storage and also tests different parallel I/O techniques. The Block Tridiagonal (BT) benchmark solves block tridiagonal equations with 5x5 block size. We vary the workload sizes across single, multi-node, and heterogeneous memory systems.

MADbench. MADbench [11], derived from a large-scale Cosmic Microwave Background (CMB) data analysis package, represents a comprehensive I/O analysis of modern parallel file systems. MADbench examines a broad range of system architectures and I/O configurations, including Lustre, GPFS on IBM Power5, and AMD Opteron platforms; We mainly use MADbench with extensively used synchronous I/O, and each thread updates to unique files using POSIX APIs. The workload generates close to 50GB of I/O data across its run.

Gromacs. Gromacs [9] is a widely used open-source framework used primarily for dynamic simulations of biomolecules like proteins, lipids, and nucleic acids that have a lot of complicated bonded interactions. Gromacs is extremely fast at calculating the non-bonded interactions (that usually dominate simulations) many groups are also using it for research on non-biological systems, e.g., polymers. Gromacs persist a lot of data such as energy logs, coordinates, velocity, and forcefields of the molecules in the system. The application uses only a fraction of the heap but generates a huge amount of I/O data constantly checkpointing the intermediate state to the disk.

Approach	Program Performance	Resource Efficiency	Performance Variability
Unlimited	High	Low	Low
App-MaxRSS	Low	High	High
Sys-MaxRSS	Medium	Medium	Low

Table 1: Memory Budgeting Techniques and Properties. App-MaxRSS represent maximum resident set size at application level, reported through the `waitpid()` system call. Sys-MaxRSS represents the memory budget identified using our deep kernel instrumentation.

GTC. Gyrokinetic Toroidal Code (GTC) [28] is a 3-Dimensional Particle-In-Cell code used to study microturbulence in magnetic confinement fusion from the first principles of plasma theory. The checkpoint data primarily have 2D arrays representing electrons and ions. The application is highly scalable, and each core can output two million particles roughly every 120 seconds resulting in 260GB of checkpoint data. To restrict the scale of the experimentation for single node analysis in homogeneous and heterogeneous system, we limit the overall checkpoint data size to 60 GB.

Graph500. Graph500 [7] is a large scale data-intensive graph benchmark which multiple kernels including Kronecker graph generator, Breadth first search(BFS) and Single source shortest path(SSSP). Due to extreme scalability and data intensity, this benchmark is used to rank the world's supercomputers. In our experiments, we keep the scale to be 25 and edge factor to be 20.

4.4 Memory Budget Estimation Approaches

We consider the following three memory budgeting approaches.

Unlimited represents a simple approach that estimates the memory use of an application for a given dataset and thread configuration by running the application and measuring the system-wide increase in memory consumption from the time the application was run. Administrators commonly use these system-wide counters and users [33, 42] exported by the OS. When measuring the Unlimited approach, we do not impose any restrictions on the memory allocation to application. For example, Linux commands such as `free -m` track and report the system-level use of heap pages, file pages, and shared pages. Table 1 summarizes the the memory budgeting approaches.

App-MaxRSS. To calculate the RSS of a process, the OS maintains task-level accounting counters for page ownership. These counters track heap, file, and shared pages. Because applications or processes can have one or more child processes, the OS also calculates the maximum resident size (MaxRSS) across all threads and processors. The MaxRSS is estimated as the maximum sum of heap pages, file pages, and shared pages across all its children. The average and MaxRSS are exposed to applications using system calls such as `get_rusage()` and `waitpid` and in other commonly used Linux commands such as `time -v`. To reduce accounting overheads, OSes update accounting counters infrequently or when pages or regions of memory are unmapped or reclaimed from the application's address space.

Sys-MaxRSS. In this approach, we perform kernel-level instrumentation of the OS active and inactive lists, and the related LRU functions that maintain the lists. In addition to process-level virtual memory counters used in MaxRSS estimation, the Linux LRU 2Q implementation [23] maintains global counters from the system-wide active and inactive lists. The pages from these lists are periodically scanned, and based on the reference activity on pages, frequently referenced pages in the inactive list are promoted to the `active-list`, and infrequently referenced pages from the active list are demoted to the `inactive-list`. As discussed earlier, high scanning cost impedes frequent updates to global counters. In contrast to Sys-MaxRSS, Unlimited approach also accounts for locked pages (e.g., pages locked by applications or driver to avoid being reclaimed), kernel buffers (used for kernel-level data structures), unevictable pages which are mapped to files and are dirty (yet to be written to storage), and swap cache pages.

4.5 Memory Capacity Use

We first start the analysis by measuring the memory consumption of Unlimited, App-MaxRSS, and the Sys-MaxRSS budgeting strategies.

Figure 1a shows the memory used for each of the budgeting approaches for the four workloads for both single-node and multi-node (4 nodes) experiments. The y-axis represents the memory consumption in GBs. Figure 1b shows the breakdown of memory page types for each of the budget strategy. In this figure, the heap pages ("HeapMem") are pages explicitly allocated by an application, whereas the kernel allocates file pages ("FileMem") in order to buffer and reduce synchronous block writes to disk in the application's critical path. The "OtherMem" pages represent kernel-level buffers that are comparatively smaller in size.

First, we observe a substantial difference in memory consumption across the three memory budgeting strategies. The Unlimited approach increases memory usage compared to App-MaxRSS for all workloads in a single node and multi-node configuration. For Gromacs, the Unlimited approach increases memory consumption by 25x over App-MaxRSS. The increase in memory use is attributed to the following reasons. First, OSes such as Linux maintain a fast path and slow path for memory allocation. The fast path is designed to optimize memory allocation when the memory availability is not scarce. Consequently, allocation only entails a quick allocation of one or more pages from the virtual memory free list. In contrast, the slow path is designed to be used when memory resources are scarce. In the slow path, the OS must also attempt to reclaim unused or inactive pages, and when reclamation fails, I/O file-backed pages in the dirty state (with uncommitted updates to disk) are flushed before releasing them for subsequent use in other I/O requests. Similarly, even after an application releases memory, the OSes do not immediately release the pages to free list unless the memory resource are scarce. Consequently, most allocations in the Unlimited approach use the fast path. As expected, memory allocation, garbage collection, and other management costs reduce, but the memory usage increases. Therefore, using Unlimited approach for memory budgeting might not provide the best space efficiency. As our results show, the memory use trend is not restricted to an application deployed in a single node, but even across multiple nodes.

Program	Unlimited (%)	AppRSSMax (%)	SysRSSMax (%)
BTIO	0.1	14.34	1.20
MADBench	0.31	3.91	3.75
Gromacs	3.20	1.24	1.49
GTC	41.39	42.95	51.67

Table 2: I/O Time Percentage.

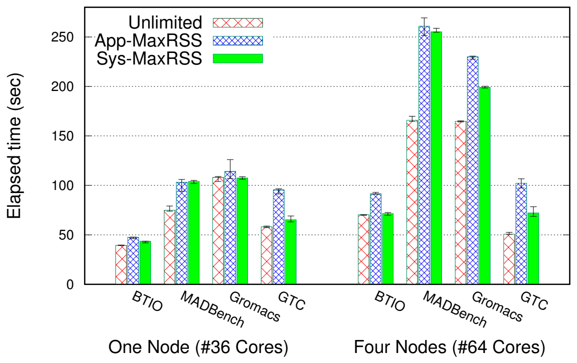
As shown in Fig 1b, a significant fraction of allocated memory is used for file-backed pages in all applications except GTC, which has a large application memory (heap) footprint. Reducing the file pages forces the program to persist its file data in the critical path and wait for disk reads. Additionally, the performance predictability reduces due to contention between heap and file pages.

Next, the App-MaxRSS only accounts for the application-level heap, file, and shared memory pages, which are only updated when pages are allocated or released by the application. We observe that App-MaxRSS mostly underestimates the memory requirement of a process, resulting in a substantially lower memory budget compared to Unlimited and Sys-MaxRSS for BTIO and Gromacs. The underestimation stems from two primary factors: the App-MaxRSS accounting in the OS virtual memory layer is independent of the system-wide accounting use [18]. A memory page that is marked for deletion when application releases a page is accounted (subtracted) from application-level counters, but the page continues to be accounted as an active page in the global `active-list` until the next LRU scan, which are run when the free memory availability reduces beyond a threshold. Note that OSes such as Linux employ a lazy garbage collection of pages released by the application.

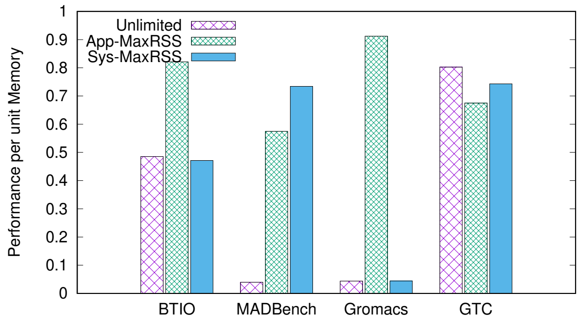
4.6 Impact of Memory Capacity on Performance

Next, we analyze the performance of applications and the "performance to capacity" ratio for the three memory budgeting strategies. Figure 2a shows the performance of each workload as a virtue of the memory budgets. Figure 2b shows performance per gigabyte of memory capacity ratio, and Figure 2 shows the average percentage of time spent waiting for I/O to complete.

First, with the Unlimited approach, applications can quickly allocate memory from the fast path, reduce garbage collection frequency, and there is no contention between applications, libraries, and OS subsystems. Consequently, the Unlimited approach provides maximum performance across all workloads. Next, with App-MaxRSS, applications are allocated with a memory budget as reported by the OS' MaxRSS calculation, which takes into account



(a) Runtime with Different Memory Budgets



(b) Performance to Memory Ratio

Figure 2: Performance Impact of Budgeting Strategies. Performance and its variability(Fig 2a), and Performance per unit memory(Fig 2b), using Eq. 3 with different memory budgets.

the heap, file, and shared memory pages attributed towards an application. However, the underestimated App-MaxRSS memory budget leads to high contention between application and file-backed paged for 3 out of 4 workloads. Further, OSes mainly prioritize heap allocation and increase the reclamation of I/O pages. Consequently, this leads to an increase in the I/O wait time and application runtime. For example, in case of BTIO, the I/O wait percentage for App-MaxRSS is more that 14% unlike other approaches with substantially lower I/O wait percentage. We observe close to 1.65x slowdown (for GTC application) for the App-MaxRSS approach compared to Unlimited.

Next, when using the Sys-MaxRSS approach, the runtime reduces by consuming more memory using the global-level 2Q LRU stats. However, in spite of the memory increase, the performance gains are limited. For example, for both BTIO and GTC, Sys-MaxRSS reduces runtime by 1.08x and 1.3x, respectively over App-MaxRSS approach. However, the gains for MADBench and Gromacs are limited over App-MaxRSS. The result trends are quite identical when scaling the applications across four nodes. When comparing Sys-MaxRSS and Unlimited, even with higher memory budget, Sys-MaxRSS performance does not improve. For example, for GTC, Sys-MaxRSS's slowdown is up to 1.12x over Unlimited.

$$Performance = \frac{MinRuntime}{Runtime} \tag{1}$$

$$PerformancePerGB = \frac{(MinMemoryUsed \times Performance)}{(MemoryUsed \times MaxPerformance)} \tag{2}$$

$$PerformancePerGB = \frac{(MinMemoryUsed \times MinRuntime)}{(MemoryUsed \times Runtime)} \tag{3}$$

Finally, in Figure 2b, we show the performance to memory ratio (i.e., performance per gigabyte of memory) to consider both performance and memory usage. We calculate the performance to memory ratio using equations 1, 2, and 3. The minimum memory use represents the App-MaxRSS values, and the maximum performance indicates the performance with Unlimited budgeting. As observed, none of the above approaches show uniform performance to memory gains summarizing the fallacies of all approaches.

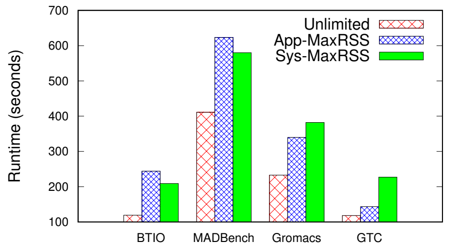


Figure 3: Performance Impact on Heterogeneous (DC Optane) Memory. Using higher core counts and faster NVMe based storage, we run applications with 64-cores and scale the workloads.

4.7 Performance on Systems with Memory Heterogenity

To understand the memory capacity budgeting impact for heterogeneous memory systems such as the recently released Intel DC Optane memory system, we use the three budgeting techniques on an Intel 3D Optane memory system with 512 GB NVM (slow memory) and 32 GB DRAM. We use the memory mode of DC Optane memory that converts the DRAM to the L4 cache and using DC Optane memory as the main memory. Figure 3 shows the performance impact. NVMs have at least 2x higher read latency and up to 5x higher write latency. Consequently, incorrect memory (NVM) provisioning shows a higher performance impact on the NVM based platform. For example, in GTC, the App-MaxRSS and Sys-MaxRSS based budgeting increases runtime by up to 1.21x and 1.92x over Unlimited, respectively. For BTIO, App-MaxRSS suffers around 2.05x overhead due to incorrect application-level memory budgeting.

4.8 Discussion and Prospective Solutions

While in this paper, we mainly focus on the fallacies of current approaches, precisely finding the memory usage of applications without compromising performance, space efficiency, or introducing performance variability requires redesigning OS memory management accounting and page tracking mechanisms. Specifically, our on-going research is focussing on three aspects. First, the redesign should enable ways to frequently update and synchronize OS global lists and counters with application-level accounting. This must be done in ways that do not increase budgeting costs (such as scanning global lists). Second, current rigid OS policies that prioritize heap pages even during high I/O activity periods, are ineffective for modern workloads that are increasingly becoming I/O-intensive. New mechanisms and policies are required to equally prioritize the I/O (storage and network) pages based on their activeness and reduce contention across page types. Finally, the solutions must also cater to the needs of heterogeneous memory systems with substantially different capacities. For example, high bandwidth die-stacked memory is expected to be of smaller capacity (16-32GB) compared to DRAMs. Inefficient capacity management can inhibit applications from exploiting the hardware benefits.

5 CONCLUSION AND FUTURE WORK

In this position paper, we highlight the need for better OS-level memory budgeting mechanisms required for high performance, resource efficiency, and lower performance variability. We decipher the reasons for inaccurate budgeting in current OSes and discuss the considerable mismatch between application-level and prohibitively expensive system-level budgeting mechanisms. Finally, we study the impact of such imprecise budgeting on HPC workloads running on homogeneous and heterogeneous (i.e., NVM) systems. Our study reveals that using current OS metrics leads to incorrect memory provisioning, leading to a substantial loss of performance and resource efficiency. Solving these problems requires redesigning OS memory management accounting and page tracking mechanisms, which is the focus of our ongoing research.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This work is partially supported by NSF CNS 1910593 award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

REFERENCES

[1] [n.d.]. Facebook PSI. https://facebookmicrosites.github.io/psi/docs/overview.

[2] [n.d.]. Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[3] [n.d.]. Linux Manual - smaps. https://man7.org/linux/man-pages/man5/proc.5.html.

[4] Inc Advanced Micro Devices. [n.d.]. AMD High Bandwidth Memory. https://www.amd.com/en/technologies/hbm.

[5] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 631–644. https://doi.org/10.1145/3037697.3037706

[6] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/2749469.2750397

[7] J. A. Ang, B. Barrett, K. Wheeler, and R. C. Murphy. 2010. Introducing the Graph 500.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[9] H.J.C. Berendsen, D. [van der Spoel], and R. [van Drunen]. 1995. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications* 91, 1 (1995), 43 – 56. https://doi.org/10.1016/0010-4655(95)00042-E

[10] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 469–479. https://doi.org/10.1109/MICRO.2006.18

[11] Jonathan Carter, Julian Borrill, and Leonid Oliker. 2004. Performance Characteristics of a Cosmology Package on Leading HPC Architectures. In *High Performance Computing - HiPC 2004, 11th International Conference, Bangalore, India, December 19-22, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3296)*, Luc Bougé and Viktor K. Prasanna (Eds.). Springer, 176–188. https://doi.org/10.1007/978-3-540-30474-6_23

[12] Ohio Supercomputer Center. [n.d.]. Out-of-Memory (OOM) or Excessive Memory Usage. https://www.osc.edu/documentation/knowledge_base/out_of_memory_oom_or_excessive_memory_usage.

[13] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. https://doi.org/10.1145/363095.363141

[14] FreeBSD. [n.d.]. LRU Management in FreeBSD. https://wiki.freebsd.org/Memory.

[15] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: A Tool to Instrument X86-64 TLB Misses. *SIGARCH Comput. Archit. News* 42, 2 (Sept. 2014), 20–23. https://doi.org/10.1145/2669594.2669599

[16] V. Garcıa, J. Gomez-Luna, T. Grass, A. Rico, E. Ayguade, and A. J. Pena. 2016. Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10.

[17] Assaf Gordon. [n.d.]. GNU Time. https://www.gnu.org/software/time/.

[18] Mel Gorman. [n.d.]. LRU Management in Linux. https://www.kernel.org/doc/gorman/html/understand/understand013.html.

[19] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, USA.

[20] Brendan Gregg. [n.d.]. Linux Resident Set Size. http://www.brendangregg.com/wss.html.

[21] Hang Huang, Jia Rao, Song Wu, Hai Jin, Kun Suo, and Xiaofeng Wu. 2019. Adaptive Resource Views for Containers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 243–254. https://doi.org/10.1145/3307681.3325403

[22] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

[23] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450.

[24] Crobett Jonathan. [n.d.]. Linux Swapping. https://lwn.net/Articles/495543/.

[25] Tim Kaldewey, Andrea Di Blas, Jeff Hagen, Eric Sedlar, and Scott Brandt. [n.d.]. Memory Matters. http://timkaldewey.de/pubs/Memory_Matters__RTSS08.pdf.

[26] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. 2009. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 1–12.

[27] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 2010. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, USA, 1–12. https://doi.org/10.1109/SC.2010.32

[28] Kamesh Madduri, Khaled Z. Ibrahim, Samuel Williams, Eun-Jin Im, Stephane Ethier, John Shalf, and Leonid Oliker. 2011. Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2063384.2063415

[29] Linux Manual. [n.d.]. MemUsage Tool. https://man7.org/linux/man-pages/man1/memusage.1.html.

[30] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 44 – 66. https://doi.org/10.1016/S1571-0661(04)81042-9 RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[31] Mark Oskin and Gabriel H. Loh. 2015. A Software-Managed Approach to Die-Stacked DRAM. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 188–200. https://doi.org/10.1109/PACT.2015.30

[32] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. 2019. System Evaluation of the Intel Optane Byte-Addressable NVM. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 304–315. https:

//doi.org/10.1145/3357526.3357568

[33] QUML. [n.d.]. Using Resident Set Size in HPC Systems. https://docs.hpc.qmul.ac.uk/using/memory/.

[34] Solmaz Salehian and Yonghong Yan. 2017. Evaluation of Knight Landing High Bandwidth Memory for HPC Workloads. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms* (Denver, CO, USA) *(IA3'17)*. Association for Computing Machinery, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/3149704.3149766

[35] VMWare. [n.d.]. VMWare vNUMA. https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[36] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. 2019. Evaluation of Intel 3D-Xpoint NVDIMM Technology for Memory-Intensive Genomic Workloads. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 277–287. https://doi.org/10.1145/3357526.3357528

[37] Tim Wickberg and Christopher Carothers. 2012. The RAMDISK Storage Accelerator: A Method of Accelerating I/O Performance on HPC Systems Using RAMDISKs. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers* (Venice, Italy) *(ROSS '12)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/2318916.2318922

[38] Bing Xie, Yezhou Huang, Jeffrey S. Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. 2017. Predicting Output Performance of a Petascale Supercomputer. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing* (Washington, DC, USA) *(HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 181–192. https://doi.org/10.1145/3078597.3078614

[39] Bing Xie, Sarp Oral, Christopher Zimmer, Jong Youl Choi, David Dillow, Scott Klasky, Jay Lofstead, Norbert Podhorszki, and Jeffrey S. Chase. 2020. Characterizing Output Bottlenecks of a Production Supercomputer: Analysis and Implications. *ACM Trans. Storage* 15, 4, Article 26 (Jan. 2020), 39 pages. https://doi.org/10.1145/3335205

[40] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 331–345. https://doi.org/10.1145/3297858.3304024

[41] Andy B. Yoo, Morris A. Jette, and Mark Grondona. 2003. SLURM: Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.

[42] Darko Zivanovic, Milan Pavlovic, Milan Radulovic, Hyunsung Shin, Jongpil Son, Sally A. Mckee, Paul M. Carpenter, Petar Radojković, and Eduard Ayguadé. 2017. Main Memory in HPC: Do We Need More or Could We Live with Less? *ACM Trans. Archit. Code Optim.* 14, 1, Article 3 (March 2017), 26 pages. https://doi.org/10.1145/3023362