MODELING THE LINUX PAGE CACHE FOR ACCURATE SIMULATION OF DATA-INTENSIVE APPLICATIONS

Hoang-Dung Do

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

February 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Hoang-Dung Do**

Entitled: **Modeling the Linux page cache for accurate simulation of data-intensive applications**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____
Chair of Department or Graduate Program Director

_____ 20 ____ _____

Rama Bhat, Ph.D.,ing., FEIC, FCSME, FASME, Interim Dean
Faculty of Engineering and Computer Science

Abstract

Modeling the Linux page cache for accurate simulation of data-intensive applications

Hoang-Dung Do

The emergence of Big Data in various fields in recent years has led to a growing need in data processing and an increasing number of data-intensive applications. These massive amount of data and applications must be executed on large-scale infrastructures such as cloud or High Performance Computing (HPC) clusters. However, some relevant challenges remain in resource management , performance, scheduling, scalability, etc. As a results, there is an increasing demand for performance optimization solutions, including data processing. While infrastructures with sufficient compute power and storage capacity are available, the I/O performance on disks remains as a bottleneck. Apart from hardware improvements, the Linux page cache is an efficient approach to reduce I/O overheads, but few experimental studies of its interactions with Big Data applications exist, partly due to limitations of real-world experiments. Simulation is a popular approach to address these issues, however, existing simulation frameworks do not simulate page caching fully, or even at all. As a result, simulation-based performance studies of data-intensive applications lead to inaccurate results.

This thesis proposes an I/O simulation model that includes the key features of the Linux page cache. We have implemented this model as part of the WRENCH workflow simulation framework, which itself builds on the popular SimGrid distributed systems simulation framework. Our model and its implementation enable the simulation of both single-threaded and multithreaded applications, and of both writeback and writethrough caches for local or network-based filesystems. We evaluate the accuracy of our model in different conditions, including sequential and concurrent applications, as well as local and remote I/Os. We find that our page cache model reduces the simulation error by up to an order of magnitude when compared to state-of-the-art, cacheless simulations.

Acknowledgments

First and foremost, I would like to express my gratitude to my supervisor, Prof. Tristan Glatard, from the Department of Computer Science and Software Engineering at Concordia University, for the patient guidance, useful comments, valuable feedback and advice he has provided during my master program. I am glad and consider myself lucky to have a chance to work with a knowledgeable, motivated professor who really cared about his students' research.

I also would like to extend my gratitude to Valérie Hayot-Sasson, a very kind and helpful colleague, for her support throughout this research. This thesis could have not been done without her knowledge and contributions.

It is also my great pleasure to work with all the members in Big Data Infrastructure for Neuroinformatics Laboratory at Concordia University, especially Martin, Mathieu and Bhupinder for being nice and helpful colleagues, who helped me to quickly adapt to the environment in the laboratory as well as the university.

I also thank to Concordia University and, again, Prof. Tristan Glatard for giving me the opportunity to do my master and have a valuable studying time in the university.

This thesis is partially supported by NSF contracts #1923539 and #1923621.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Data-intensive applications and HPC

In the last decades, information technology has been significantly changing the world we are living in. Thanks to both software and hardware advancements, the application development and operational costs have become more reasonable. As a result, applications has been widely used in various fields by companies, organizations and individuals. Simultaneously, they have been collecting a massive amount of data at increasing generating speed. In addition, the last decade witnessed the rise of Big Data and IoT with the rapid development in mobile devices, wearable devices, smart appliances and connectivity technologies. Data has increasingly been collected from these sources and transferred through network day by day. The ever-increase in the number and scale of open-data and data sharing initiatives have resulted in the evolution in Big Data and data-intensive applications, which have been increasingly playing an important role in many fields such as science, medical, military, finance, commerce, etc.

Big Data is usually defined with "three Vs" corresponding to *volume*, *velocity* and *variety* [10]. For *volume*, the vast amounts of data require scalable and distributed approaches for storage solutions as well as querying of information and insights. *Velocity* emphasizes on the speed that data is generated from large data streams. Real-time analytics of streaming data requires immediate responses from the data ingested by systems. *variety* refers to the diversity of data types including both structured and unstructured data such as text, audio, images, videos from various data sources. This

data diversity requires not only elastic storage, transfer, processing but also compute power to obtain deeper and valuable insights. Due to the above challenges in handling the sheer size of the data, data-intensive applications must be executed on large-scale infrastructures such as High Performance Computing (HPC) clusters or the cloud, which provides a large scale solution and parallelism at micro as well as macro level to boost data processing.

Data-intensive applications consist of complex long tasks and workflows with a significant amount of data. A modification can help to reduce the execution time by hours of even days. It is thus crucial to quantify and optimize the performance of these applications on HPC platforms. However, these systems have heterogeneous and complex architectures since they are built by and for different organizations, used for specific purposes, thus must be optimized in different ways. The goals of optimization of these platforms include determining which type of hardware/software stacks are best suited to different application classes, as well as understanding the limitations of current algorithms, designs and technologies.

1.2 Quantifying performance

Quantifying performance of the platforms basically relies on empirical methods, which require running experiments on those platforms. There are two common approaches to conduct experiments platforms, which are running real experiments on real platforms and using simulation tools. In the first approach, real-world tasks, applications, workflows or pipelines are executed directly on a real platform and the information about the performance of the system is recorded for later analysis. This method can guarantee that the results are realistic since it uses the records obtained from real systems with corresponding configurations. Unfortunately, performance studies relying on real-world experiments on compute platforms face many difficulties. The first challenge is that platforms are normally shared with dynamics loads, the randomness in the status of the systems that hinder the reproducibility of the experiment results. The next difficulty can be the labor-intensive experimental setups and time-consuming applications, since experiments must be re-executed multiple times with multiple settings. Moreover, because real-platforms are not built for experimental purposes, users may not have the freedom to choose platform architectures,

software and system configurations since some settings may not be permitted and the operational cost of the platforms are usually high. These shortcomings apparently preclude the exploration of hypothetical scenarios.

Simulations address these concerns by providing simulation models and abstractions for the performance of computer hardware, such as CPU, network and storage. The simulation models describe the interactions between simulated applications and simulated platforms by evolving application activities (such as computation, data transfer) which consume simulated resources (CPU, network, storage) throughout simulated application run time [8]. As a result, simulations provide a cost-effective, fast, easy and reproducible way to evaluate application performance on arbitrary platform configurations. It thus comes as no surprise that a large number of simulation frameworks have been developed and used for research and development [2, 4, 28, 5, 26, 27, 23, 20, 25, 30, 8, 6, 7].

1.3 Page cache and simulation

Due to hardware limitations, the performance of platforms can be hampered by bottlenecks such as data transfer through network links, accessing data on disks. When it comes to disk I/O, the bottleneck remains as a result of limited disk read and write bandwidths, seek time involved in disk accesses, and shared bandwidth between processes. In data-intensive applications, a significant proportion of execution time is spent on data read and write. Thus, the detrimental effects of I/O bottleneck can be more severe on this type of tasks. Failing to effectively access data can hinder the performance of the whole application, and can lead to wasteful idle time of other resources. Although efforts to increase the bandwidth of storage devices with advancements in technology, such as solid-state drive (SSD), the I/O bottleneck still exists.

Caching is an ubiquitous technique to minimize data transfer from low-speed storage by using a high-speed storage layer to store data temporarily. Caching reuses previously accessed data for future requests, so that it helps reducing data transfer times. Following the idea of caching, page cache is an architectural approach implemented in Linux that leverages main memory to improve the effectiveness in accessing data on disks, reduce filesystem data transfer times. As main memory is known to

have significant faster read and write bandwidths compared to those of disks, the idea behind page cache is to make I/O operations occur in memory instead of disks. With the page cache, previously read data can be kept in cache and then re-read directly from memory with memory read bandwidth. In writing, written data can be written to memory with memory write bandwidth before being asynchronously flushed to disk, resulting in improved I/O performance on slower storage devices. In case the amount of cache available is not sufficient for caching new data, data previously read or written to page cache can be flushed to disks or evicted from cache with data flushing and cache eviction mechanisms. These mechanisms are implemented in the Linux kernel and triggered base on particular conditions of total amount of memory, the amount of data being written (i.e., dirty data), the amount of memory available for written data, configurations and parameters defined in the kernel, etc. These parameters combined with the above mentioned flushing and eviction mechanisms are the key features of the page cache. Hence, these factors should be taken into account when determining the impact of I/O on application performance, particularly in data-intensive applications.

Aside from hardware resources, the existence of the page cache in systems has considerable impacts on the performance of platforms when it mitigates the I/O bottleneck. As such, it is necessary to have a simulation model of the page cache when simulating data-intensive applications in order to achieve higher accuracy in simulation results. While existing simulation frameworks of parallel and distributed computing systems capture many relevant features of hardware as well as software stacks, they lack the ability to simulate page cache with enough details to capture key features such as dirty data, data flushing mechanisms and cache eviction policies [26, 27]. Some simulators, such as the one in [36], do capture such features, but are domain-specific.

1.4 Contributions

This thesis presents a page cache simulation model with important features including data model representing cached data, data flushing mechanism and cache eviction policy, which are not fully captured in most of other simulation frameworks. The thesis also presents the simulation algorithms of file read and file write, which are in a

chunk-by-chunk manner. These functions play the role as the read and write functions in the kernel, simulate file read and file write by interacting with the simulated page cache. We also provide the implementation of the the page cache model, file I/O algorithms in WRENCH [7], a workflow simulation framework based on the popular SimGrid distributed simulation toolkit [8]. In order to validate, evaluate the accuracy and scalability of the page cache simulation model, we implement a simulator in pure Python as well as in C with WRENCH framework and compare it to a simulator with the original WRENCH version, which is not integrated the page cache model. The simulator is evaluated in different experiments, which are common scenarios in data-intensive applications on HPC. The experiments range from single-threaded, multi-threaded applications, remote storage application using synthetic data, as well as a real neuroimaging pipeline with real data. The experimental results show that our page cache model significantly improves the simulation accuracy by reducing the relative simulation errors by up to 9 times, though there are some limitations in the current version of SimGrid framework. In addition, our simulator not only achieves more accurate simulation results but also simulates correctly the behaviors of the page cache, proves the correctness of the model and opens the opportunities to be further enhanced.

1.5 Thesis organization

This thesis consists of five main parts divided into chapters. Chapter 2 provides the background knowledge of Linux page cache with key features and mechanisms. It also summarizes the simulation approach in studies in HPC performance with existing simulation frameworks and models. The reasons for the choice of simulation tools is this thesis are explained in this chapter as well. Chapter 3 presents the page cache simulation model with the key features in three main sections. The first section describes simulation of the page cache components including data presentation, page cache LRU lists and, flushing and eviction mechanisms. The second section interprets the algorithms simulating file read and write which work in a chunk-by-chunk style. The last section in this chapter briefly presents the implementation plan to integrate the page cache model into WRENCH framework. Chapter 4 proposes four experiment scenarios to validate the model including a single-threaded application, a concurrent

applications scenario, an experiment with remote storage, and a real neuroimaging application. The experiment results and evaluation are also presented in this chapter. Finally, Chapter 5 concludes this thesis and discusses the future work.

Chapter 2

Related work

2.1 Page cache

2.1.1 Page cache reduces I/O cost

The Linux page cache

To offsets the cost of disk I/O, the Linux kernel implements a disk cache, called *page cache*, by storing in memory data that requires disk accesses. There are two reasons that make the disk caches important to operating systems. First, disk accesses are several orders of magnitude slower than memory accesses. Second, there is a likelihood data accessed before will be accessed again in near future [24]. With memory bandwidth, which is much faster than disk bandwidth, combined with the situations that data can be accessed in memory instead of on disk, the I/O performance can be largely improved. In Linux, the page cache is a part of RAM, which includes physical pages referring to pages on disk. The size of the page cache is dynamic when it can grow when there is enough free memory, and can be shrink to release memory if needed.

When the kernel starts a read operation, it checks if the required data is in memory. If yes, called a *cache hit*, data is then read directly from memory, with memory bandwidth, instead of from disk. If not, called a *cache miss*, data is read from disk and the kernel places a new entry representing this data in the page cache for later reads [24]. Data cached in the the page cache is pages, which means files must not be cached entirely, the page cache can keep the whole file, or only part a file.

Writeback and writethrough page cache

When use of page cache is enabled for a given filesystem, all written pages are written first to page cache, prior to being written to disk. Accessing of these written pages may result in cache hits, should the pages remain in memory. Generally speaking, the page cache can implement one in three different write strategies.

In the first strategy, which is *no-write*, the page cache simply does not involve in write operations. In *no-write*, data is written directly to disk, the cache is invalidated and read from disk for any subsequent requests. This strategy is rarely implemented since it not only fails to cache data, but also costly invalidates the page cache [24].

The second strategy is *writethrough*, in which the kernel updates both disk and memory cache in write operations. The name *writethrough* itself suggests that data is written *through* the page cache to disk with disk write bandwidth [24]. This is a simple solution that can keep data in cache, synchronized between page cache and disk, but it does not make the write operations benefit from fast memory write bandwidth.

The third strategy is the approach implemented in Linux kernel, called *writeback*. With writeback cache, the kernel perform write operations by writing data directly into the page cache. However, unlike writethrough, the storage is not immediately updated. Instead, the pages that have been written to page cache are marked as *dirty* data. These dirty pages are periodically written back to disk by a flusher process in predefined intervals. In addition, if the kernel needs to reclaim some free memory, it can immediately trigger data flushing to write back dirty data to disk. After being written back to backing store, these pages are no longer dirty and can be removed from page cache when the free memory is insufficient. The writeback strategy is considered to outperform writethrough as well as direct I/O (page cache bypassed for I/O) as it delays disk writes to perform a bulk write at a later time [24].

Caching on NFS

Data caching requires keeping data close to where it is requested. In network filesystem (NFS), data caching means not sending requests to server over the network. Thus, data is cached on NFS client cache instead of a remote disk [12]. In addition, some cache schemes are restricted to ensure data integrity and consistency depending on the structure of the filesystem. In reading on NFS, if data is cached on client, data is read from client cache as with local filesystem. If required data is not cached on

the client, it will be read from server. On server side, the kernel also checks for the availability of data in server cache to decide whether data is read from cache or disk. However, there is no server cache in writing since the data written to NFS server cannot be cached and must be written to disk before the write call on the NFS client finishes to ensure data integrity. If server cache is enable for writing, if the server crashes during a cache write could result in a problem since the client could not be aware of if data has been written successfully. On the other hand, given a scenario where multiple write operations queue up on client side, a client failure before data is written could leave the NFS server with an old version of file begin written. Thus, only writethrough strategy can be implemented for writing on NFS.

2.1.2 Cache eviction

Cache eviction mechanism is one of the key features in the operations of page cache. It is responsible for deciding removing pages from the page cache to make memory space available for new entries as well as free RAM for other uses. Whenever space in memory becomes limited, either as a result of application memory or page cache use, page cache data may be evicted. Only clean data, which is not marked as dirty and persisted to storage, can be flagged for eviction and removed from memory. If clean pages are insufficient, written data that has not yet been persisted to disk (dirty data) must first be copied (flushed) to storage and marked as clean to make more pages available for eviction [24, 3]. The crucial part in the cache eviction mechanism is to decide which pages to evict. According to the idea of the page cache, data that is more likely to be accessed in the future should be kept in page cache, and the data that is least likely to be used should be evicted. Different cache eviction algorithms have also been proposed [9].

Page cache replacement policies

We have mentioned about the page cache, adding pages to the page cache, cache eviction. But how the page cache is exactly structured, how data is placed and removed from the page cache?

The idea of the page cache is to keep data that is likely to be accessed again in the future, but an algorithm that knows the future in advance, often referred as *clairvoyant algorithm*, is impossible to implement. Many algorithms have been

designed and proposed to approximate the *clairvoyant algorithm*.

One of the most commonly used strategy is *Least Recently Used (LRU)*. This page replacement policy is based on the principle of locality that data references of a process tend to cluster. Thus, it selects pages that has not been referred for a longest time to remove [9]. However, one of the drawbacks of LRU is not considering data access frequency.

The *CLOCK* algorithm improves this shortcoming of LRU by structuring the page cache as a circular list with a hand pointing to the tail of the list. Each page has an reference bit, which is turned on if the page is referenced. Only the oldest pages with the reference bit set to zero can be removed [9]. This algorithm is improved with *Dueling CLOCK*, which uses interchangeably and adaptively two variants of CLOCK for better performance than LRU and CLOCK [9].

Another variant of LRU is *LRU-K*, which takes page frequency information into account while replacing pages. It looks backward on the LRU list for the k^{th} most recent reference of a candidate page and replace the page with the oldest k^{th}reference. Experimental results indicate that LRU-2 can increase performance compared to LRU [9].

Low Inter-reference Recency Set (LIRS) is another algorithm which takes Inter-Reference-Recency into account instead of the recency of a single page as in LRU-K. In LIRS, Inter-Reference-Recency (IRR) of a page refers to the number of pages accessed between two consecutive references of that page. The idea of the algorithm is to keep only a small number of pages with high IRR since they are not accessed frequently (normally around 1%) [9].

CLOCK-Pro is another variant of CLOCK that attempts to approximate LIRS by using page reuse distance, which is similar to IRR in LIRS. A page with small reuse distance is categorized as *hot* page, and a page with large reuse distance is a *cold* page. It also keep historical metadata of previously accessed pages. Simulation studies showed that the performance of CLOCK-Pro can approximate LIRS [9].

Adaptive Replacement Cache (ARC) is an algorithm that keeps track of both frequently used and recently used pages by maintaining two lists: L1 for pages that are accessed only once recently, and L2 for the pages that are accessed more than once recently. Each list is then split into the top cache entries (real pages) and bottom ghost entries (metadata of evicted pages). The pages and metadate entries

are continually moved between these lists, added to or removed from these lists to adaptively adjust the size of frequently and recently used lists based on particular workloads [9].

CLOCK with Adaptive Replacement (CAR) was proposed to inherit the adaptivity of ARC and the implementation efficiency of CLOCK. It implements two circular lists of cache and ghost entries as in ARC [9].

Page cache LRU lists

Among page replacement policies, LRU is considered one of the most commonly used algorithms that is successful for general purpose page cache. It approximates well the future use of pages but fails when putting on the top of the list a file that is accessed only once. Therefore, Linux kernel implements two-list strategy, which is based on LRU, due to its efficiency in both implementation and performance.

The two-list strategy in Linux kernel maintains two LRU lists: *active list* and *inactive list* [24, 3]. The lists work as queues, in which pages are added to the head and removed from the tail. When a page is referenced, if it is not in the page cache, it then be added to the inactive list. Should pages located on the inactive list be accessed, they will be moved from the inactive to the active list. They are also kept balanced by moving pages from the active list to the inactive list when the active list grows significantly larger than the inactive list. As a result, the active list only contains pages that are accessed more than once, while the inactive list basically contains pages that are accessed once only, or pages that have been accessed more than once but moved from the active list. Since the pages in the active list are more frequently accessed, they are consider "hot" and not available for eviction. In contrast, the pages in the inactive list, which are less frequently accessed, are considered "cold" and available for eviction. Both lists operate using LRU eviction policies, meaning that data that has not be accessed recently will be moved first.

This two-list strategy, known as *LRU/2*, not only solves the problem of frequently accessed data in LRU, but also allows better performance with simple implementation. Now assume that in a particular scenario, when a user is working in a workspace editing multiple smalle files. When the files are loaded, they are read from disk for the first time and added to the page cache. When the files are edited and then saved, new versions of the files are written to the page cache. After that, if a saved file is

opened again, the file is read directly from the page cache instead disk, which makes reading time way faster.

2.1.3 Flushing and periodical flushing

Unix systems allow write operations of dirty pages to be deferred and perform a bigger physical write to disk to improve performance, which is called flushing mechanisms. Beside cache eviction, flushing strategies are integral to proper page cache functioning. Basically, dirty data flushing can be triggered under following conditions:

- When the amount of free memory is below a specific threshold, or the number of dirty pages has reached its limit because only clean pages are available for eviction.
- A page has remained as dirty in the page cache for too long.

In the first case, a synchronous flusher thread is called to flushed dirty pages to disk when the amount of available memory is low (available memory includes free memory and claimable memory). Besides, the Linux kernel has the variable `/proc/sys/vm/dirty_ratio`, which defines a percentage of total available memory. When the amount of dirty data surpasses this level, flusher thread is awaken to writeback dirty data to disk. In addition, there is another important variable, which is `/proc/sys/vm/dirty_background_ratio`, also defining a percentage of total available memory. If the amount of free memory drops below this level, a flusher thread is triggered by the kernel to start flushing dirty data.

In the second case, a kernel thread (`pdflush`) is called to periodically scan for pages that remains as dirty in page cache for an amount of time longer than a predefined *expired time*, and then to explicitly write the content of these pages to disk. This mechanism, called *periodical flushing*, is to ensure that no page can remain in page cache infinitely, and to keep data synchronized between memory and storage. The Linux kernel awakes a flusher thread to writeback expired dirty page in intervals defined by `/proc/sys/vm/dirty_writeback_interval` variable, in milliseconds, with the default value usually set to 5000 milliseconds (5 seconds). The expiration time can be set with the `/proc/sys/vm/dirty_expire_interval` variable, the default value is 30000 milliseconds (30 seconds) [24].

2.2 Simulation vs experimentation

2.3 Simulation frameworks

2.3.1 Models and concerns

Many simulation frameworks have been developed to enable the simulation of parallel and distributed applications [2, 4, 28, 5, 26, 27, 23, 20, 25, 30, 8, 6, 7]. These frameworks implement simulation models and abstractions to aid the development of simulators for studying the functional and performance behaviors of application workloads executed on various hardware/software infrastructures.

The two main concerns for simulation are accuracy, the ability to faithfully reproduce real-world executions, and scalability, the ability to simulate large/long real-world executions quickly and with low RAM footprint. The above frameworks achieve different compromises between the two. At one extreme are discrete-event models that capture "microscopic" behaviors of hardware/software systems (e.g., packet-level network simulation, block-level disk simulation, cycle-accurate CPU simulation), which favor accuracy over speed. At the other extreme are analytical models that capture "macroscopic" behaviors via mathematical models. While these models lead to fast simulation, they must be developed carefully if high levels of accuracy are to be achieved [35].

In this work, we use the SimGrid and WRENCH simulation frameworks. The years of research and development invested in the popular SimGrid simulation framework [8], have culminated in a set of state-of-the-art macroscopic simulation models that yield high accuracy, as demonstrated by (in)validation studies and comparisons to competing frameworks [1, 35, 34, 14, 22, 29, 11, 31, 17, 33]. But one significant drawback of SimGrid is that its simulation abstractions are low-level, meaning that implementing a simulator of complex systems can be labor-intensive [21]. To remedy this problem, the WRENCH simulation framework [7] builds on top of SimGrid to provide higher-level simulation abstractions, so that simulators of complex applications and systems can be implemented with a few hundred lines.

2.3.2 Existing data caching simulation

Although the Linux page cache has a large impact on I/O performance, and thus on the execution of data-intensive applications, its simulation is rarely considered in the above frameworks. Most frameworks merely simulate I/O operations based on storage bandwidths and capacities. The SIMCAN framework does models page caching by storing data accessed on disk in a block cache [26]. Page cache is also modeled in iCanCloud through a component that manages memory accesses and cached data [27]. However, the scalability of the iCanCloud simulator is limited as it uses microscopic models. Besides, none of these simulators provide any writeback cache simulator nor cache eviction policies through LRU lists. Although cache replacement policies are applied in [36] to simulate in-memory caching, this simulator is specific to energy consumption of multi-tier heterogeneous networks.

In this study, we implement a page cache simulation model in the WRENCH framework. We targeted WRENCH because it is a recent, actively developed framework that provides convenient simulation abstractions, because it is extensible, and because it reuses SimGrid's scalable and accurate models.

2.3.3 SimGrid and WRENCH

Chapter 3

Methods

We separate our simulation model in two components, the I/O Controller and the Memory Manager, which together simulate file reads and writes (Figure 1). To read or write a file chunk, a simulated application sends a request to the I/O Controller. The I/O Controller interacts as needed with the Memory Manager to free memory through flushing or eviction, and to read or write cached data. The Memory Manager implements these operations, simulates periodical flushing and eviction, and reads or writes to disk when necessary. In case the writethrough strategy is used, the I/O Controller directly writes to disk, cache is flushed if needed and written data is added to page cache.

3.1 Memory Manager

The Memory Manager simulates two parallel threads: the main one implements flushing, eviction, and cached I/Os synchronously, whereas the second one, which operates in the background, periodically searches for expired dirty data in LRU lists and flushes this data to disk. We use existing storage simulation models [22] to simulate disk and memory, characterized by their storage capacity, read and write bandwidths, and latency. These models account for bandwidth sharing between concurrent memory or disk accesses.

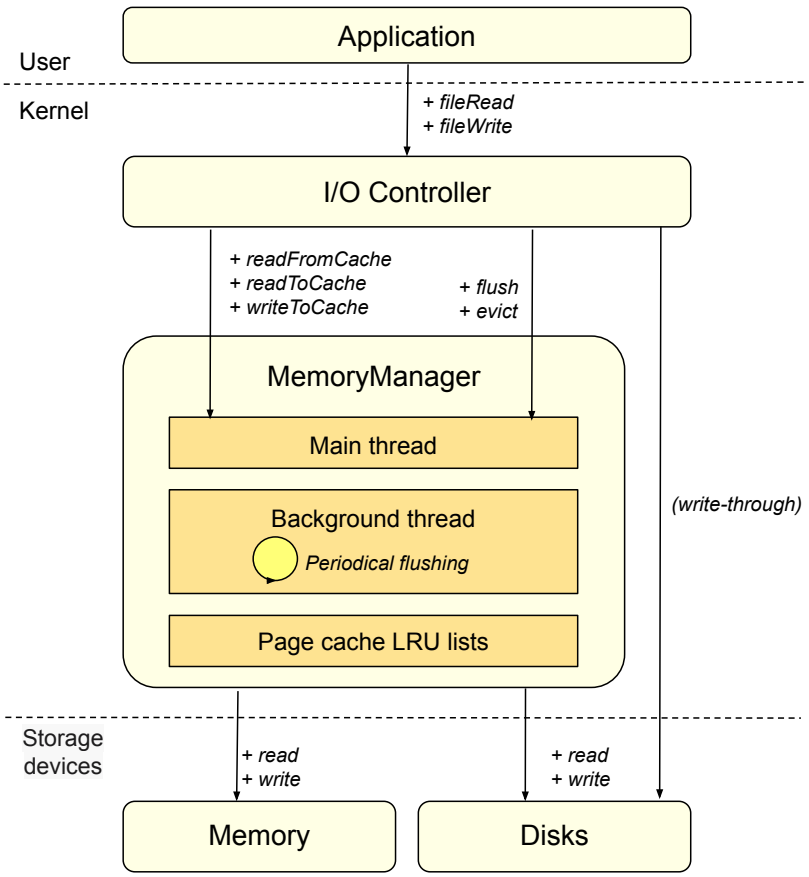

Figure 1: Overview of the page cache simulator. Applications send file read or write requests to the I/O Controller that orchestrates flushing, eviction, cache and disk accesses with the Memory Manager. Concurrent accesses to storage devices (memory and disk) are simulated using existing models.

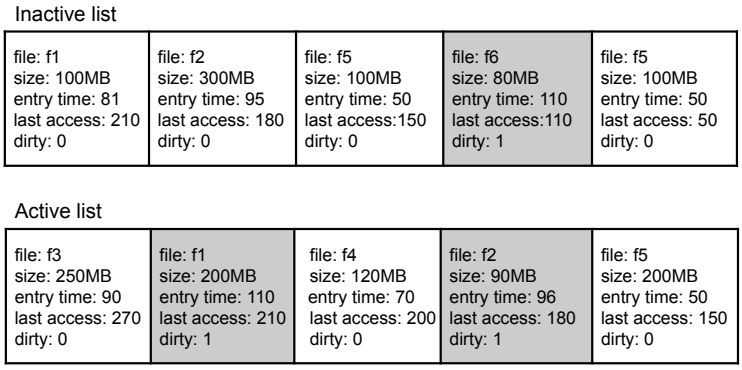

Figure 2: Model of page cache LRU lists with data blocks.

3.1.1 Page cache LRU lists

In the Linux kernel, page cache LRU lists contain file pages. However, due to the large number of file pages, simulating lists of pages induces substantial overhead. Therefore, we introduce the concept of a data block as a unit to represent data cached in memory. A data block is a subset of file pages stored in page cache that were accessed in the same I/O operation. A data block stores the file name, block size, last access time, a dirty flag that represents whether the data is clean (0) or dirty (1), and an entry (creation) time. Blocks can have different sizes and a given file can have multiple data blocks in page cache. In addition, a data block can be split into an arbitrary number of smaller blocks.

We model page cache LRU lists as two lists of data blocks, an active list and an inactive list, both ordered by last access time (earliest first, Figure 2). As in the kernel, our simulator limits the size of the active list to twice the size of the inactive list, by moving least recently used data blocks from the active list to the inactive list [15, 24].

At any given time, a file can be partially cached, completely cached, or not cached at all. A cached data block can only reside in one of two LRU lists. The first time they are accessed, blocks are added to the inactive list. On subsequent accesses, blocks of the inactive list are moved to the top of the active list. Blocks written to cache are marked dirty until flushed.

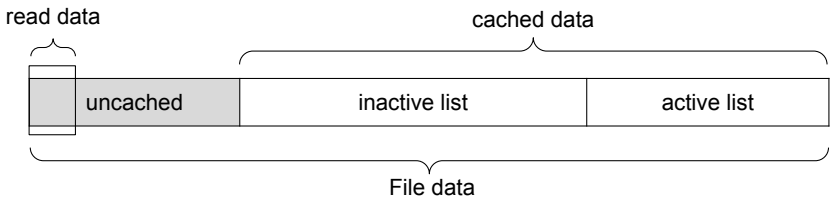

Figure 3: File data read order. Data is read from left to right: uncached data is read first, followed by data from the inactive list, and finally data from the active list.

3.1.2 Reads and writes

Our simulation model supports chunk-by-chunk file accesses with a user-defined chunk size. However, for simplicity, we assume that file pages are accessed in a round-robin fashion rather than fully randomly. Therefore, when a file is read, cached data is read only after all uncached data was read, and data from the inactive list is read before data from the active list (data reads occur from left to right in Figure 3). When a chunk of uncached data is read, a new clean block is created and appended to the inactive list. When a chunk of cached data is read, one or more existing data blocks in the LRU lists are accessed. If these blocks are clean, we merge them together, update the access time and size of the resulting block, and append it to the active list. If the blocks are dirty, we move them independently to the active list, to preserve their entry time. Because the chunk and block sizes may be different, there are situations where a block is not entirely read. In this case, the block is split in two smaller blocks and one of them is re-accessed.

For file writes, we assume that all data to be written is uncached. Thus, each time a chunk is written, we create a block of dirty data and append it to the inactive list.

3.1.3 Flushing and eviction

The main simulated thread in the Memory Manager can flush or evict data from the memory cache. The data flushing simulation function takes the amount of data to flush as parameter. While this amount is not reached and dirty blocks remain in cache, this function traverses the sorted inactive list, then the sorted active list, and writes the least recently used dirty block to disk, having set its dirty flag to 0. In case the amount of data to flush requires that a block be partially flushed, the block is

split in two blocks, one that is flushed and one that remains dirty. The time needed to flush data to disk is simulated by the storage model.

The cache eviction simulation also runs in the main thread. It frees up the page cache by traversing and deleting least recently used clean data blocks in the inactive list. The amount of data to evict is passed as a parameter and data blocks are deleted from the inactive list until the evicted data reaches the required amount, or until there is no clean block left in the list. If the last evicted block does not have to be entirely evicted, the block is split in two blocks, and only one of them is evicted. The overhead of the cache eviction algorithm is not part of the simulated time since cache eviction time is negligible in real systems.

Algorithm 1 Periodical flush simulation in Memory Manager

```
1: Input
2:     in   page cache inactive list
3:     ac   page cache active list
4:     t    predefined flushing time interval
5:     exp  predefined expiration time
6:     sm   storage simulation model
7: while host is on do
8:     blocks = expired_blocks(exp, in) + expired_blocks(exp, ac)
9:     flushing_time = 0
10:    for blk in blocks do
11:        blk.dirty = 0
12:        flushing_time = flushing_time + sm.write(blocks)
13:    end for
14:    if flushing_time < t then
15:        sleep(t - flushing_time)
16:    end if
17: end while
```

Periodical flushing is simulated in the Memory Manager background thread. As in the Linux kernel, a dirty block in our model is considered expired if the duration since its entry time is longer than a predefined expiration time. Periodical flushing is simulated as an infinite loop in which the Memory Manager searches for dirty blocks and flushes them to disk (Algorithm 1). Because periodical flushing is simulated as a

background thread, it can happen concurrently with disk I/O initiated by the main thread. This is taken into account by the storage model and reflected in simulated I/O time.

3.2 I/O Controller

As mentioned previously, our model reads and writes file chunks in a round-robin fashion. To read a file chunk, simulated applications send chunk read requests to the I/O Controller which processes them using Algorithm 2. First, we calculate the amount of uncached data that needs to be read from disk, and the remaining amount is read from cache (line 7-8). The amount of memory required to read the chunk is calculated, corresponding to a copy of the chunk in anonymous memory and a copy of the chunk in cache (line 9). If there is not enough available memory, the Memory Manager is called to flush dirty data (line 10). If necessary, flushing is complemented by eviction (line 11). Note that, when called with negative arguments, functions `flush` and `evict` simply return and do not do anything. Then, if the block requires uncached data, the memory manager is called to read data from disk and to add this data to cache (line 14). If cached data needs to be read, the Memory Manager is called to simulate a cache read and update the corresponding data blocks accordingly (line 17). Finally, the memory manager is called to deallocate the amount of anonymous memory used by the application (line 19).

Algorithm 2 File chunk read simulation in I/O Controller

```
1: Input
2:     cs   chunk size
3:     fn   file name
4:     fs   file size (assumed to fit in memory)
5:     mm MemoryManager object
6:     sm   storage simulation model
7: disk_read = min(cs, fs - mm.cached(fn))                    ▷ To be read from disk
8: cache_read = cs - disk_read                                ▷ To be read from cache
9: required_mem = cs + disk_read
10: mm.flush(required_mem - mm.free_mem - mm.evictable, fn)
11: mm.evict(required_mem - mm.free_mem, fn)
12: if disk_read > 0 then                                     ▷ Read uncached data
13:     sm.read(disk_read)
14:     mm.add_to_cache(disk_read, fn)
15: end if
16: if cache_read > 0 then                                    ▷ Read cached
17:     mm.cache_read(cache_read)
18: end if
19: mm.use_anonymous_mem(cs)
```

Algorithm 3 describes our simulation of chunk writes in the I/O Controller. Our algorithm initially checks the amount of dirty data that can be written given the dirty ratio (line 5). If this amount is greater than 0, the Memory Manager is requested to evict data from cache if necessary (line 7). After eviction, the amount of data that can be written to page cache is calculated (line 8), and a cache write is simulated (line 9). If the dirty threshold is reached and there is still data to write, the remaining data is written to cache in a loop where we repeatedly flush and evict from the cache (line 12-18).

Algorithm 3 File chunk write simulation in I/O Controller

```
1: Input
2:     cs    chunk size
3:     fn    file name
4:     mm MemoryManager object
5: remain_dirty = dirty_ratio * mm.avail_mem - mm.dirty
6: if remain_dirty > 0 then                                  ▷ Write to memory
7:     mm.evict(min(cs, remain_dirty) - mm.free_mem)
8:     mem_amt = min(cs, mm.free_mem)
9:     mm.write_to_cache(fn, mem_amt)
10: end if
11: remaining = cs - mem_amt
12: while remaining > 0 do                    ▷ Flush to disk, then write to cache
13:     mm.flush(cs - mem_amt)
14:     mm.evict(cs - mem_amt - mm.free_mem)
15:     to_cache = min(remaining, mm.free_mem)
16:     mm.write_to_cache(fn, to_cache)
17:     remaining = remaining - to_cache
18: end while
```

The above model describes page cache in writeback mode. Our model also includes a write function in writethrough mode, which simply simulates a disk write with the amount of data passed in, then evicts cache if needed and adds the written data to the cache.

3.3 Implementation

We first created a standalone prototype simulator to evaluate the accuracy and correctness of our model in a simple scenario before integrating it in the more complex WRENCH framework. The prototype uses the following basic storage model for both memory and disk:

$$t_r = D/b_r$$
$$t_w = D/b_w$$

where:

- t_r is the data read time
- t_w is the data write time
- D is the amount of data to read or write
- b_r is the read bandwidth of the device
- b_w is the write bandwidth of the device

This prototype does not simulate bandwidth sharing and thus does not support concurrency: it is limited to single-threaded applications running on systems with a single-core CPU. We used this prototype for a first validation of our simulation model against a real sequential application running on a real system. The Python 3.7 source code is available at https://github.com/big-data-lab-team/paper-io-simulation/tree/master/exp/pysim.

We also implemented our model as part of WRENCH, enhancing its internal implementation and APIs with a page cache abstraction, and allowing users to activate the feature via a command-line argument. We used SimGrid's locking mechanism to handle concurrent accesses to page cache LRU lists by the two Memory Manager threads. For the experiments, we used WRENCH 1.6 at commit 6718537433, which uses SimGrid 3.25, available at https://framagit.org/simgrid/simgrid. Our implementation is now part of WRENCH's master branch and will be available to users with the upcoming 1.8 release. WRENCH provides a full SimGrid-based simulation environment that supports, among other features, concurrent accesses to storage devices, applications distributed on multiple hosts, network transfers, and multi-threading.

Chapter 4

Experiments and Results

4.1 Experiments

Our experiments compared real executions with our Python prototype, with the original WRENCH simulator, and with our WRENCH-cache extension. Executions included single-threaded and multi-threaded applications, accessing data on local and network file systems. We used two applications: a synthetic one, created to evaluate the simulation model, and a real one, representative of neuroimaging data processing.

Experiments were run on a dedicated cluster at Concordia University, with one login node, 9 compute nodes, and 4 storage nodes connected with a 25 Gbps network. Eachcompute node had 2 × 16-core Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 250 GiB of RAM, 6 × SSDs of 450 GiB each with the XFS file system, 378 GiB of tmpfs, 126 GiB of devtmpfs file system, CentOS 8.1 and NFS version 4. We used the `atop` and `collectl` tools to monitor and collect memory status and disk throughput. We cleared the page cache before each application run to ensure comparable conditions.

The synthetic application, implemented in C, consisted of three single-core, sequential tasks where each task read the file produced by the previous task, incremented every byte of this file to emulate real processing, and wrote the resulting data to disk. Files were numbered by ascending access times (File 1 and File 2 were the files read and written respectively by Task 1, etc). The anonymous memory used by the application was released after each task, an this memory release is also simulated in the Python prototype and in WRENCH-cache. As our focus was on I/O rather than

compute, we measured CPU times of application tasks on a cluster node (Table 1), and used these durations in our simulations. For the Python prototype, as we put a sleep time simulated tasks to simulate CPU time, we simply injected CPU times directly in the simulation. In WRENCH, it simulates CPU time with the number of flops of tasks and the CPU speed of hosts. Thus, for WRENCH and WRENCH-cache, we determined the corresponding number of flops on a 1 Gflops CPU and used these values in the simulation. The simulated the platform and application are available at commit ec6b43561b.

Input size (GB)	CPU time (s)
3	4.4
20	28
50	75
75	110
100	155

Table 1: Synthetic application parameters

We used the synthetic application in three experiments. In the first one (*Exp 1*), we ran a single instance of the application on a single cluster node, with different input file sizes (20 GB, 50 GB, 75 GB, 100 GB), and with all I/Os directed to the same local disk. The information about free memory, amount of dirty data, amount of cache used and free memory is collected using `atop` tool during the execution time of the tasks. We also used `fincore` to after each I/O operation to inspect the cache content with the amount of cached data of each file.

In the second experiment (*Exp 2*), we ran concurrent instances of the application on a single node, all application instances operating on different files stored in the same local disk. Due to the limited capacity of the disk used, we used the file size of 3 GB. We varied the number of concurrent application instances from 1 to 32 since cluster nodes had 32 CPU cores. The read time, CPU time and write time of each instance were logged into log files.

In the third experiment (*Exp 3*), we used the same configuration as the previous one, albeit reading and writing on a 50-GiB NFS-mounted partition of a 450-GiB

remote disk of another compute node. As is commonly configured in HPC environments to avoid data loss, there was no client write cache and the server cache was configured as writethrough instead of writeback. NFS client and server read caches were enabled. Therefore, all the writes happened at disk bandwidth, but reads could benefit from cache hits.

The real application was a workflow of the Nighres toolbox [18], implementing cortical reconstruction from brain images in four steps: skull stripping, tissue classification, region extraction, and cortical reconstruction. Each step read files produced by the previous step, and wrote files that were or were not read by the subsequent step. More information on this application is available in the Nighres documentation at https://nighres.readthedocs.io/en/latest/auto_examples/example_02_cortical_depth_estimation.html. The application is implemented as a Python script that calls Java image-processing routines. We used Python 3.6, Java 8, and Nighres 1.3.0. In Nighres, data is read lazily and written in compressed format. Thus, we patched the application to remove lazy data loading and data compression, which made CPU time difficult to separate from I/O time, and to capture task CPU times to inject them in the simulation. The patched code is available at https://github.com/dohoangdzung/nighres.

We used the real application in the fourth experiment (*Exp 4*), run on a single cluster node using a single local disk. We processed data from participant 0027430 in the dataset of the Max Planck Institute for Human Cognitive and Brain Sciences available at http://dx.doi.org/10.15387/fcp_indi.corr.mpg1, leading to the parameters in Table 2.

Workflow step	Input size (MB)	Output size (MB)	CPU time (s)
Skull stripping	295	393	137
Tissue classification	197	1376	614
Region extraction	1376	885	76
Cortical reconstruction	393	786	272

Table 2: Nighres application parameters

To parameterize the simulators, we benchmarked the memory, local disk, remote

disk (NFS), and network bandwidths (Table 3). Since SimGrid, and thus WRENCH, currently only supports symmetrical bandwidths, we use the mean of the read and write bandwidth values in our experiments.

Bandwidths		Cluster (real)	Python prototype	WRENCH simulator
Memory	read	6860	4812	4812
	write	2764	4812	4812
Local disk	read	510	465	465
	write	420	465	465
Remote disk	read	515	-	445
	write	375	-	445
Network		3000	-	3000

Table 3: Bandwidth benchmarks (MBps) and simulator configurations. The bandwidths used in the simulations were the average of the measured read and write bandwidths. Network accesses were not simulated in the Python prototype.

4.2 Results

4.2.1 Single-threaded execution (Exp 1)

The page cache simulation model drastically reduced I/O simulation errors in each application task (Figure 4). The first read was not impacted as it only involved uncached data. Errors were reduced from an average of 345% in the original WRENCH to 46% in the Python prototype and 39% in WRENCH-cache. Unsurprisingly, the original WRENCH simulator significantly overestimated read and write times, due to the lack of page cache simulation.

As is shown in Figure 4, WRENCH simulation errors substantially decreased as the input file size increased. This is due to the fact that as the input file size grew larger than a specific threshold in the experiment, all files can not fit in the page cache at the same time, and part of files need to be written to disk. The larger the input file size is, the more data is written to the disk, and the smaller proportion

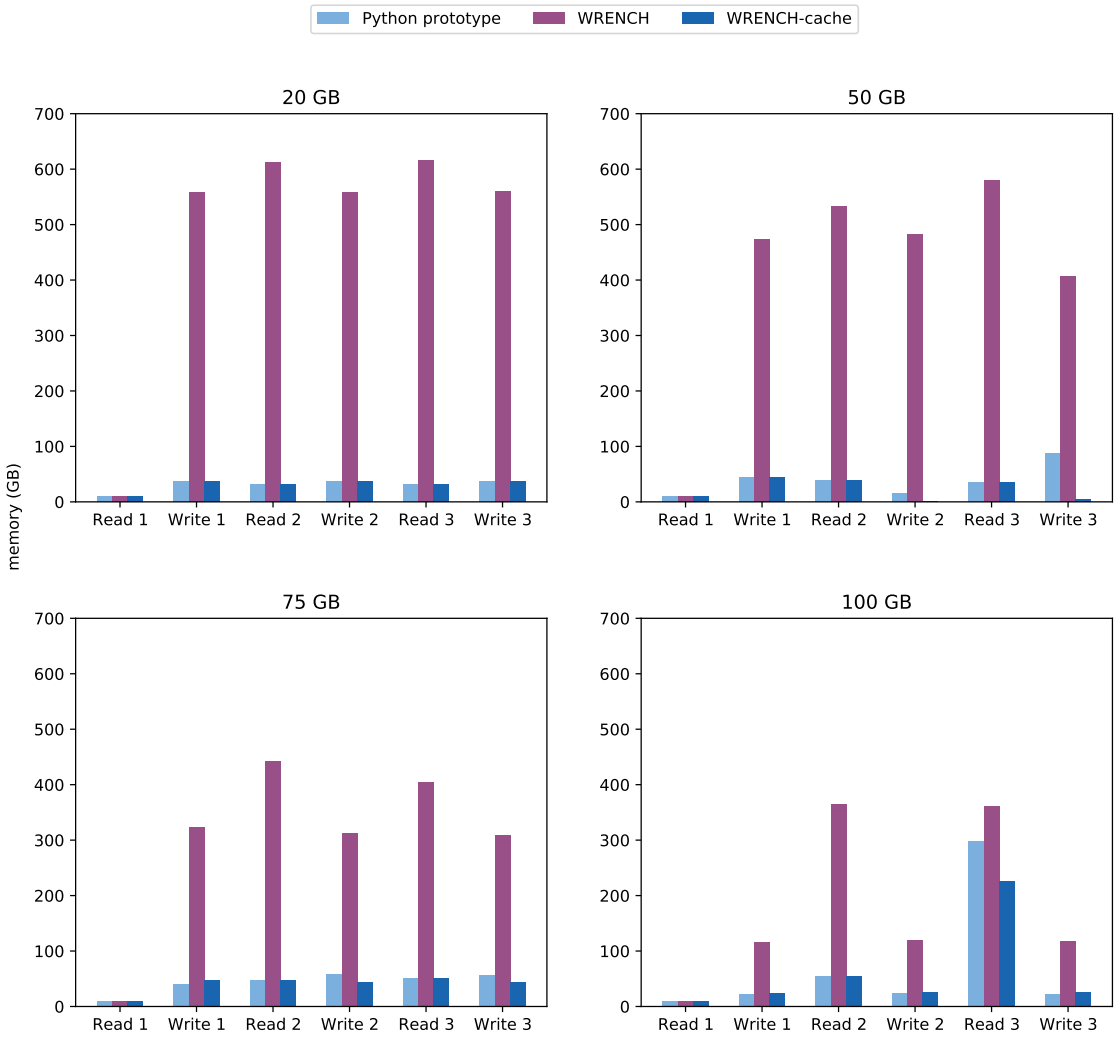

Figure 4: Absolute relative simulation errors of single-threaded application with different file sizes

of total I/O time that the page cache reduced. Conversely, simulation errors of the Python prototype and WRENCH-cache were almost equal with 20 GB, 50 GB and 75 GB. However, the errors of those simulators with 100 GB are considerably higher due to idiosyncrasies in the kernel flushing and eviction strategies that could not be easily modeled.

Simulated memory profiles with different file sizes were highly consistent with the real ones (Figure 5). With 20 GB, 50 GB and 75 GB files, memory profiles almost

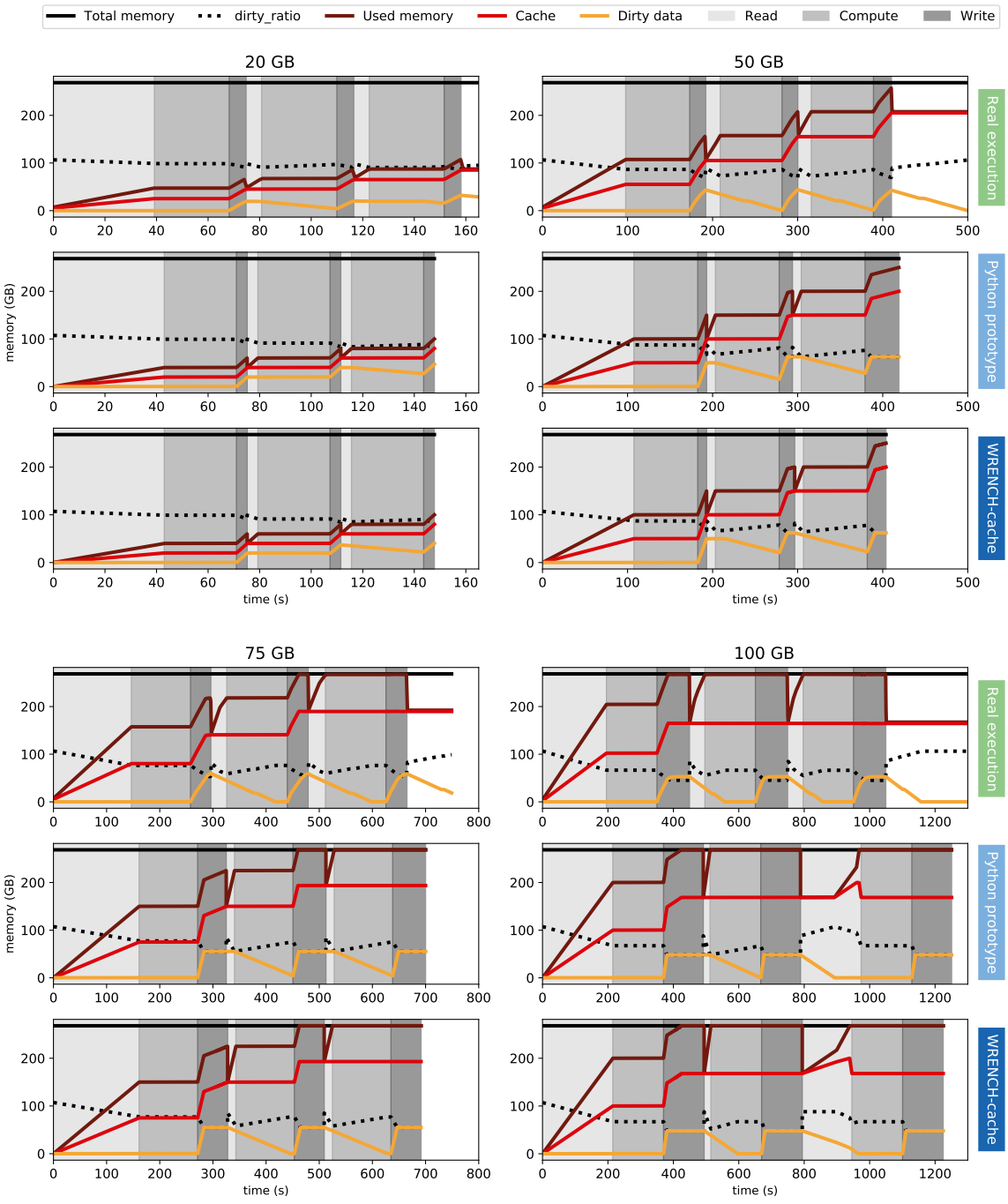

Figure 5: Single-threaded application memory profiles with different file sizes

exactly matched the real ones, although dirty data seemed to be flushing faster in real life than in simulation. With 50 GB files, this slower dirty data flushing led to a larger amount of dirty data after Read 3 in simulation than in reality, which caused longer write time of Write 3 when less data was written to cache. The simulated memory profiles of 75 GB files were also matched with the real ones, except that there were plateaus in file writes, which also induced longer write time as in Read 3 with 50 GB files. With 100 GB files, used memory reached total memory during the first write, triggering dirty data flushing, and dropped back to cached memory when application tasks released anonymous memory. Simulated cached memory was highly consistent with real values, except toward the end of Read 3 where it slightly increased in simulation but not in reality. This occurred due to the fact that after Write 2, File 3 was only partially cached in simulation whereas it was entirely cached in the real system. Thus, Read 3 happened in memory in the real system, but part of File 3 was read from disk in simulation, leading longer simulated read time. In all cases, dirty data remained under the dirty ratio as expected. The Python prototype and WRENCH-cache exhibited nearly identical memory profiles, which reinforces the confidence in our implementations.

The content of the simulated memory cache was also highly consistent with reality (Figure 6). With 20 GB and 50 GB files, the simulated cache content exactly matched reality, since all files fitted in page cache. With 75 GB files, the amount of File 1 cached after Write 2 and Read 3 in reality were slightly less than the simulated amount, but the cached data amount of files in simulation matched reality in overall. With 100 GB files, a slight discrepancy was observed after Write 2, which explains the simulation error previously mentioned in Read 3. In the real execution indeed, File 3 was entirely cached after Write 2, whereas in the simulated execution, only a part of it was cached. This was due to the fact that the Linux kernel tends to not evict pages that belong to files being currently written (File 3 in this case), which we could not easily reproduce in our model.

4.2.2 Concurrent applications (Exp 2)

Figure 7 presents the average read and write time of each pipeline in the concurrent applications experiment. As is shown in the figure, the page cache model notably reduced WRENCH's simulation error for concurrent applications executed

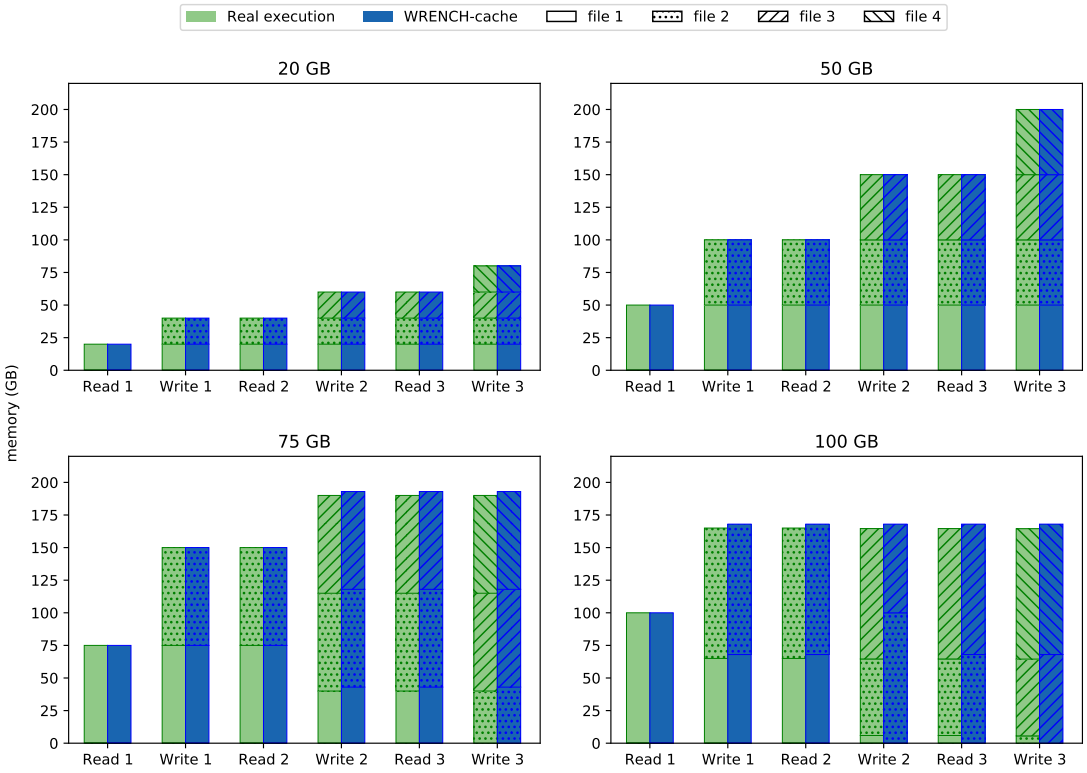

Figure 6: Cache contents after single-threaded application I/O operations

with local I/Os. For reads, WRENCH-cache slightly overestimated runtime, due to the discrepancy between simulated and real read bandwidths mentioned before, in which simulated read bandwidth is slower than the real one. For writes, WRENCH-cache retrieved a plateau similar to the one observed in the real execution. This was marked with the limit at which all data of pipelines could still fit into the page cache. Beyond this limit, the page cache was saturated with dirty data and needed flushing.

4.2.3 Remote storage (Exp 3)

Similar to the the previous experiment, the average read and write time with concurrent applications on remote storage are illustrated in Figure 8. The figure shows that page cache simulation importantly reduced simulation error on NFS storage as well. This manifested only for reads, as the NFS server used writethrough rather than

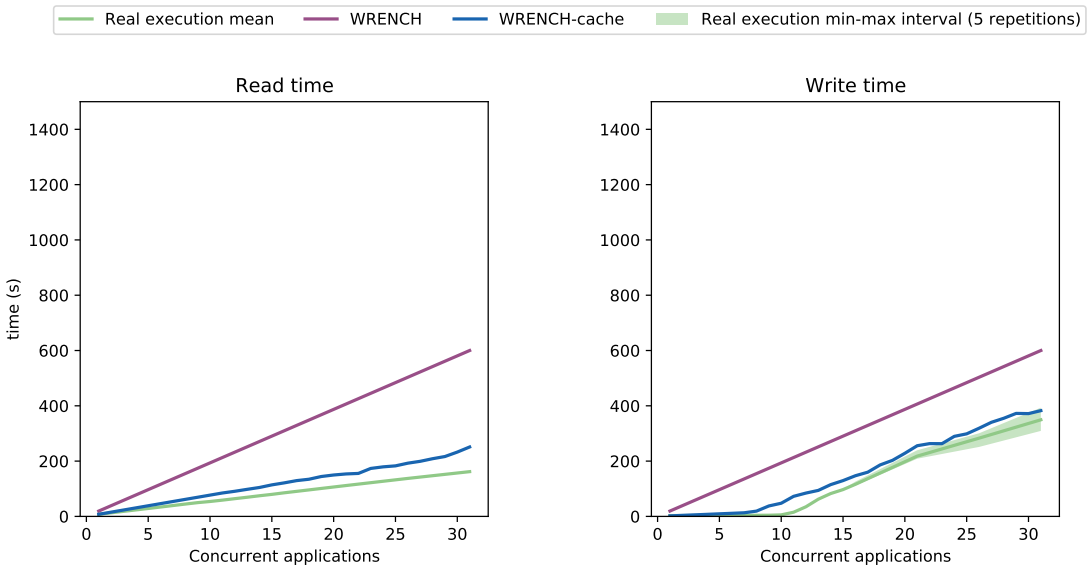

Figure 7: Results of concurrent applications on local storage with 3 GB files (*Exp 2*)

writeback cache, which means all write operations happened at disk write bandwidth. Both WRENCH and WRENCH-cache underestimated write times due to the discrepancy between simulated and real bandwidths mentioned previously. For reads, this discrepancy only impacted the results beyond 22 concurrent applications. Before this threshold, most reads resulted in cache hits, while after this threshold, WRENCH-cache did not accurately simulate data flushing and cache eviction, similar to what we observed in the single-threaded experiment with 100 GB files and leading to less cache hits and more data read from disk in simulation than in reality.

4.2.4 Real application (Exp 4)

Similar to the synthetic application, simulation errors of the real application were substantially reduced by the WRENCH-cache simulator compared to WRENCH (Figure 9). On average, errors were reduced from 337 % in WRENCH to 47 % in WRENCH-cache. The first read happened entirely from disk and was therefore very accurately simulated by both WRENCH and WRENCH-cache.

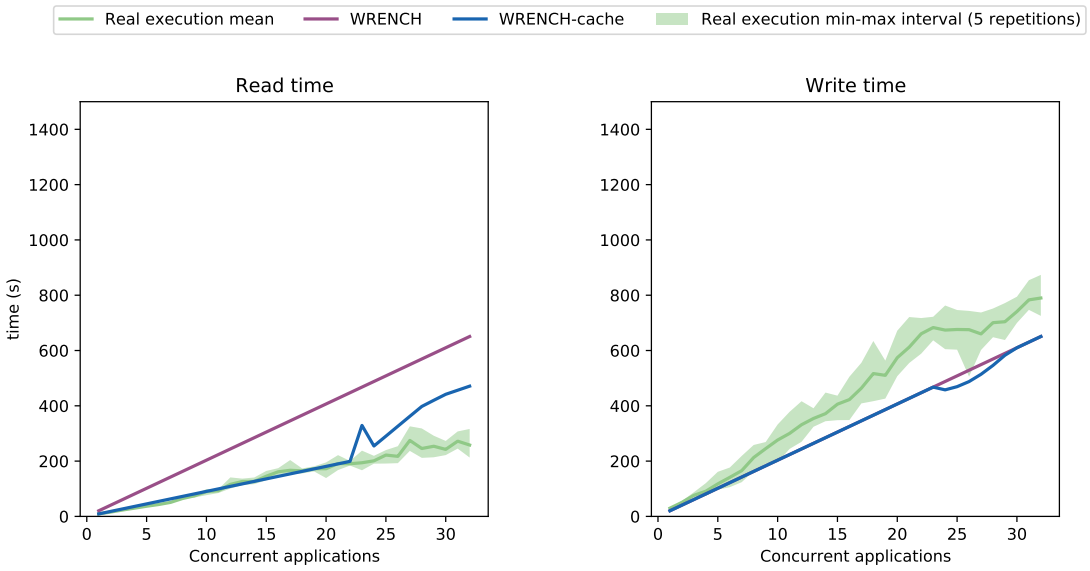

Figure 8: Results of concurrent applications on NFS with 3 GB files (*Exp 3*)

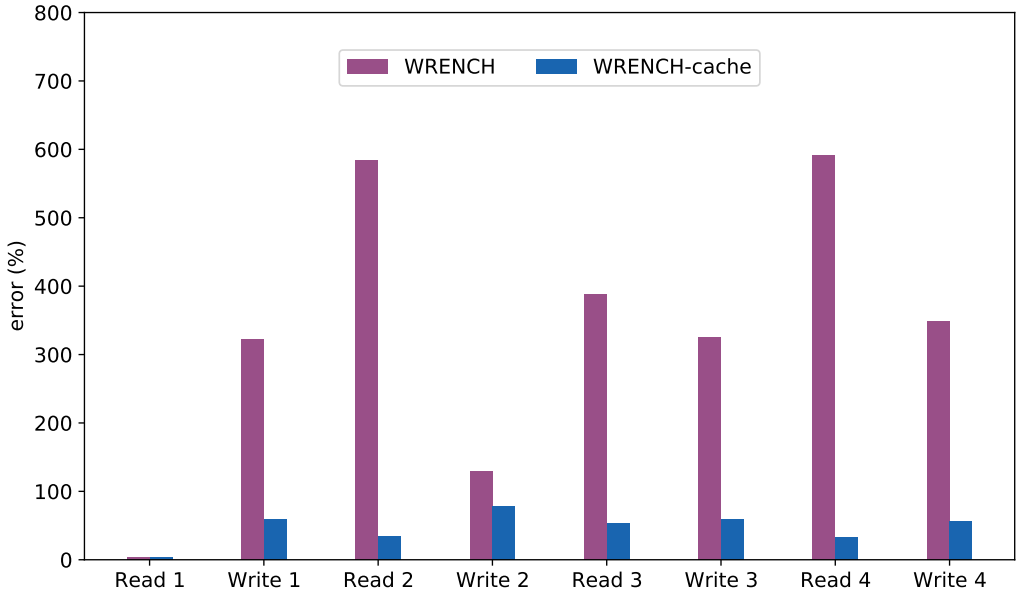

Figure 9: Real application results (*Exp 4*)

4.2.5 Simulation time

As is the case for WRENCH, simulation time with WRENCH-cache scales linearly with the number of concurrent applications (Figure 10, p $<10^{-24}$). However, the page cache model substantially increases simulation time by application, as can be seen by comparing regression slopes in Figure 10. Interestingly, WRENCH-cache is faster with NFS I/Os than with local I/Os, most likely due to the use of writethrough cache in NFS, which bypasses flushing operations.

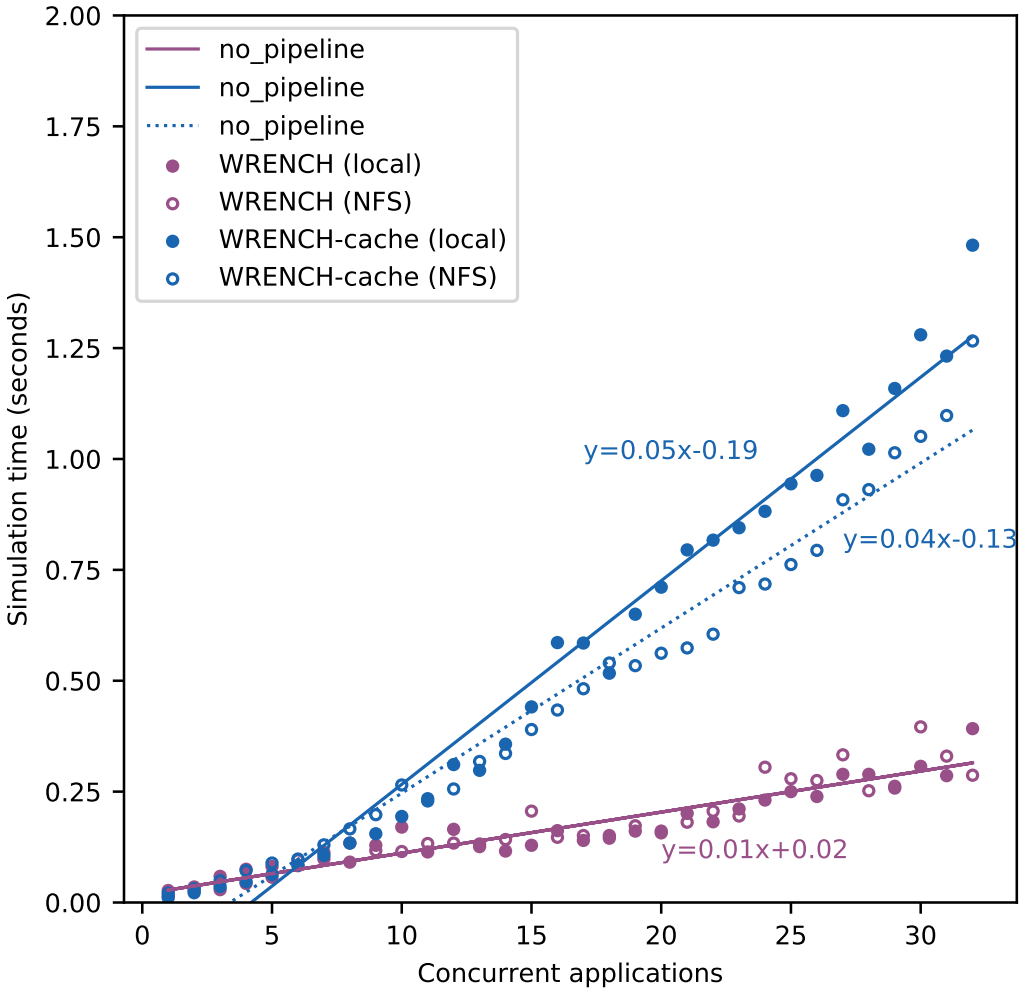

Figure 10: Simulation time comparison. WRENCH-cache scales linearly with the number of concurrent applications, albeit with a higher overhead than WRENCH.

Chapter 5

Conclusion

We designed a model of the Linux page cache and implemented it in the SimGrid-based WRENCH simulation framework to simulate the execution of distributed applications. Evaluation results show that our model improves simulation accuracy substantially, reducing absolute relative simulation errors by up to 9× (see results of the single-threaded experiment). The availability of asymmetrical disk bandwidths in the forthcoming SimGrid release will further improve these results. Our page cache model is publicly available in the WRENCH GitHub repository.

Page cache simulation can be instrumental in a number of studies. For instance, it is now common for HPC clusters to run applications in Linux control groups (cgroups), where resource consumption is limited, including memory and therefore page cache usage. Using our simulator, it would be possible to study the interaction between memory allocation and I/O performance, for instance to improve scheduling algorithms or avoid page cache starvation [37]. Our simulator could also be leveraged to evaluate solutions that reduce the impact of network file transfers on distributed applications, such as burst buffers [13], hierarchical file systems [19], active storage [32], or specific hardware architectures [16].

Not all I/O behaviors are captured by currently available simulation models, including the one developed in this work, which could substantially limit the accuracy of simulations. Relevant extensions to this work include more accurate descriptions of anonymous memory usage in applications, which strongly affects I/O times through writeback cache. File access patterns might also be worth including in the simulation models, as they directly affect page cache content.

Bibliography

[1] P. Bedaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, F. Suter, and B. Videau. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. In Proc. of the 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, 2013.

[2] William H. Bell, David G. Cameron, A. Paul Millar, Luigi Capozza, Kurt Stockinger, and Floriano Zini. OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies. IJHPCA, 17(4):403–416, 2003.

[3] Daniel Bovet and Marco Cesati. Understanding The Linux Kernel. O'Reilly & Associates Inc, 3rd edition, 2005.

[4] Rajkumar Buyya and Manzur Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. Concurrency and Computation: Practice and Experience, 14(13-15):1175–1220, December 2002.

[5] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms. Software: Practice and Experience, 41(1):23–50, January 2011.

[6] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: A High-Performance, Low Memory, Modular Time Warp System. In Proc. of the 14th ACM/IEEE/SCS Workshop of Parallel on Distributed Simulation, pages 53–60, 2000.

[7] Henri Casanova, Rafael Ferreira da Silva, Ryan Tanaka, Suraj Pandey, Gautam Jethwani, William Koch, Spencer Albrecht, James Oeth, and Frédéric Suter.

Developing accurate and scalable simulators of production workflow management systems with WRENCH. Future Generation Computer Systems, 112:162–175, 2020.

[8] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. Journal of Parallel and Distributed Computing, 74(10):2899–2917, June 2014.

[9] Amit S Chavan, Kartik R Nayak, Keval D Vora, Manish D Purohit, and Pramila M Chawan. A comparison of page replacement algorithms. International Journal of Engineering and Technology, 3(2):171, 2011.

[10] Andrea De Mauro, Marco Greco, and Michele Grimaldi. A formal definition of big data based on its essential features. Library Review, 2016.

[11] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, and F. Suter. Simulating MPI applications: the SMPI approach. IEEE Transactions on Parallel and Distributed Systems, 28:2387–2400, 2017.

[12] M. Eisler, R. Labiaga, and H. Stern. Managing NFS and NIS: Help for Unix System Administrators. O'Reilly Media, 2nd edition, 2001.

[13] Rafael Ferreira da Silva, Scott Callaghan, Tu Mai Anh Do, George Papadimitriou, and Ewa Deelman. Measuring the impact of burst buffers on data-intensive scientific workflows. Future Generation Computer Systems, 101:208–220, 2019.

[14] K. Fujiwara and H. Casanova. Speed and Accuracy of Network Simulation in the SimGrid Framework. In Proc. of the 1st Intl. Workshop on Network Simulation Tools, 2007.

[15] Mel Gorman. Understanding the Linux virtual memory manager. Prentice Hall Upper Saddle River, 2004.

[16] Valérie Hayot-Sasson, Shawn T Brown, and Tristan Glatard. Performance benefits of Intel® Optane™ DC persistent memory for the parallel processing of large neuroimaging data. In 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pages 509–518. IEEE, 2020.

[17] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A. Orgerie, and M. Quinson. Predicting the energy-consumption of MPI applications at scale using only a single node. In 2017 IEEE International Conference on Cluster Computing (CLUSTER), pages 92–102, 2017.

[18] Julia M Huntenburg, Christopher J Steele, and Pierre-Louis Bazin. Nighres: processing tools for high-resolution neuroimaging. GigaScience, 7(7):giy082, 2018.

[19] Nusrat Sharmin Islam, Xiaoyi Lu, Md Wasi-ur Rahman, Dipti Shankar, and Dhabaleswar K Panda. Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture. In 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pages 101–110. IEEE, 2015.

[20] G. Kecskemeti. DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds. Simulation Modelling Practice and Theory, 58(2), 2015.

[21] G. Kecskemeti, S. Ostermann, and R. Prodan. Fostering Energy-Awareness in Simulations Behind Scientific Workflow Management Systems. In Proc. of the 7th IEEE/ACM Intl. Conf. on Utility and Cloud Computing, pages 29–38, 2014.

[22] Adrien Lebre, Arnaud Legrand, Frédéric Suter, and Pierre Veyre. Adding storage simulation capacities to the SimGrid toolkit: Concepts, models, and API. In Proceedings of the 15th IEEE/ACM Symposium on Cluster, Cloud and Grid Computing (CCGrid 2015), pages 251–260, Shenzhen, China, May 2015. IEEE/ACM.

[23] S.H. Lim, B. Sharma, G. Nam, E.K. Kim, and C.R. Das. MDCSim: A multi-tier data center simulation platform. In Intl. Conference on Cluster Computing and Workshops (CLUSTER), 2009.

[24] Robert Love. Linux Kernel Development. Addison-Wesley Professional, 3rd edition, 2010.

[25] A. W. Malik, K. Bilal, K. Aziz, D. Kliazovich, N. Ghani, S. U. Khan, and R. Buyya. CloudNetSim++: A toolkit for data center simulations in OMNET++. In 2014 11th Annual High Capacity Optical Networks and Emerging/Enabling Technologies (Photonics for Energy), pages 104–108, 2014.

[26] Alberto Núñez, Javier Fernández, Rosa Filgueira, Félix García, and Jesús Carretero. SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications. Simulation Modelling Practice and Theory, 20(1):12–32, 2012.

[27] Alberto Núñez, Jose L Vázquez-Poletti, Agustin C Caminero, Gabriel G Castañé, Jesus Carretero, and Ignacio M Llorente. iCanCloud: A flexible and scalable cloud infrastructure simulator. Journal of Grid Computing, 10(1):185–209, 2012.

[28] Simon Ostermann, Radu Prodan, and Thomas Fahringer. Dynamic Cloud Provisioning for Scientific Grid Workflows. In Proc. of the 11th ACM/IEEE Intl. Conf. on Grid Computing (Grid), pages 97–104, 2010.

[29] Laurent Pouilloux, Takahiro Hirofuchi, and Adrien Lebre. SimGrid VM: Virtual Machine Support for a Simulation Framework of Distributed Systems. IEEE transactions on cloud computing, September 2015.

[30] T. Qayyum, A. W. Malik, M. A. Khan Khattak, O. Khalid, and S. U. Khan. FogNetSim++: A Toolkit for Modeling and Simulation of Distributed Fog Environment. IEEE Access, 6:63570–63583, 2018.

[31] A. S. M. Rizvi, T. R. Toha, M. M. R. Lunar, M. A. Adnan, and A. B. M. A. A. Islam. Cooling energy integration in SimGrid. In 2017 International Conference on Networking, Systems and Security (NSysS), pages 132–137, 2017.

[32] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W. Liao, and A. Choudhary. Enabling active storage on parallel I/O software stacks. In 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pages 1–12, 2010.

[33] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and accurate simulation of multithreaded sparse linear algebra solvers. In 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pages 481–490, 2015.

[34] P. Velho and A. Legrand. Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. In Proc. of the 2nd Intl. Conf. on Simulation Tools and Techniques, 2009.

[35] P. Velho, L. Mello Schnorr, H. Casanova, and A. Legrand. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. ACM Transactions on Modeling and Computer Simulation, 23(4), 2013.

[36] Jianwen Xu, Kaoru Ota, and Mianxiong Dong. Saving energy on the edge: In-memory caching for multi-tier heterogeneous networks. IEEE Communications Magazine, 56(5):102–107, 2018.

[37] Zhenyun Zhuang, Cuong Tran, Jerry Weng, Haricharan Ramachandra, and Badri Sridharan. Taming memory related performance pitfalls in Linux cgroups. In 2017 International Conference on Computing, Networking and Communications (ICNC), pages 531–535. IEEE, 2017.