

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

Hoang-Dung Do\*, Valérie Hayot-Sasson\*, Rafael Ferreira da Silva<sup>†</sup>, Christopher Steele<sup>§</sup>, Henri Casanova<sup>†</sup>, Tristan Glatard\*

\*Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada

<sup>†</sup>Department of Information and Computer Sciences, University of Hawai‘i at Mānoa, USA

<sup>‡</sup>Information Sciences Institute, University of Southern California, Marina Del Rey, CA, USA

<sup>§</sup>Department of Psychology, Concordia University, Montreal, Canada

**Abstract**—The emergence of Big Data in recent years has resulted in a growing need for efficient data processing solutions. While infrastructures with sufficient compute power are available, the I/O bottleneck remains. 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.

In this paper, we propose 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.

## I. INTRODUCTION

The Linux page cache plays an important role in reducing filesystem data transfer times. With the page cache, previously read data can be re-read directly from memory, and written data can be written to memory before being asynchronously flushed to disk, resulting in improved I/O performance on slower storage devices. The performance improvements depend on many factors including the total amount of memory, the amount of data being written (i.e., dirty data), and the amount of memory available for written data. All these factors are important when determining the impact of I/O on application performance, particularly in data-intensive applications.

The number of data-intensive applications has been steadily rising as a result of open-data and data sharing initiatives. Due to the sheer size of the data being processed, these applications must be executed on large-scale infrastructures such as High Performance Computing (HPC) clusters or the cloud. It is thus crucial to quantify the performance of these applications on these platforms. The goals 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. Unfortunately, performance studies relying on real-world experiments on compute platforms face several difficulties (high operational costs, labor-intensive experimental setups, shared platforms with dynamic loads that hinder reproducibility of results) and shortcomings (experiments are limited to the available platform/software configurations, which precludes the exploration of hypothetical scenarios). Simulations address these concerns by providing models and abstractions for the performance of computer hardware, such as CPU, network and storage. 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 [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13].

Page caching is an ubiquitous technique for mitigating the I/O bottleneck. As such, it is necessary to model it when simulating data-intensive applications. While existing simulation frameworks of parallel and distributed computing systems capture many relevant features of hardware/software stacks, they lack the ability to simulate page cache with enough details to capture key features such as dirty data and cache eviction policies [5], [6]. Some simulators, such as the one in [14], do capture such features, but are domain-specific.

In this work, we present WRENCH-cache, a page cache simulation model implemented in WRENCH [13], a workflow simulation framework based on the popular SimGrid distributed simulation toolkit [11]. Our contributions are:

- A page cache simulation model that supports both single-threaded and multithreaded applications, and both writeback and writethrough caches for local or network-based filesystems;
- An implementation of this model in WRENCH;
- An evaluation of the accuracy and scalability of our model, and of its implementation, for multiple applications, execution scenarios, and page cache configurations.

## II. RELATED WORK

### A. Page cache

Page cache offsets the cost of disk I/O by enabling I/O to occur directly from memory. When a file is first loaded into memory, the file is read from disk and loaded into the page cache as a series of pages. Subsequent reads to any of the file pages located in memory will result in a *cache hit*, meaning the I/O can occur directly from without disk involvement. Any accessed page not loaded in memory results in a *cache miss*, resulting in the page being read directly from disk.

Written pages can also contribute to future application cache hits. When use of page cache is enabled for a given filesystem, through the enabling of writeback or writethrough cache, 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.

The kernel may also provide improved write performance if writeback cache is enabled. Unlike writethrough, where the data is synchronously written from memory to disk, writeback enables asynchronous writes. With writeback, the application may proceed with execution once the data has been written to memory, even if it has not yet been materialized to disk. 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 [15].

Cache eviction and flushing strategies are integral to proper page cache functioning. Whenever space in memory becomes limited, either as a result of application memory or page cache use, page cache data may be evicted. Only data that has been persisted to storage (clean pages) can be flagged for eviction and removed from memory. Written data that has not yet been persisted to disk (dirty data) must first be copied (flushed) to storage prior to eviction. When sufficient memory is being occupied, the flushing process is synchronous. However, even when there is sufficient available memory, written data will be flushed to disk at a predefined interval through a process known as *periodical flushing*. Periodical flushing only flushes expired dirty pages, which remain dirty in page cache longer than an expiration time configured in the kernel. Different cache eviction algorithms have also been proposed [16].

The Linux kernel uses a two-list strategy to flag pages for eviction. The two-list strategy is based on a least recently used (LRU) policy and uses an active and inactive list in its implementation. If accessed pages are not in the page cache, they are 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. The lists are also kept balanced by moving pages from the active list to the inactive list when the active list grows too large. Thus, the active list only contains pages which are accessed more than once and not evictable, while the inactive list includes pages accessed once only, or pages that have been accessed more than once but moved from the active list. Both lists operate using LRU eviction policies, meaning that data that has not be accessed recently will be moved first.

### B. Simulation

Many simulation frameworks have been developed to enable the simulation of parallel and distributed applications [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. 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 [17].

In this work, we use the SimGrid and WRENCH simulation frameworks. The years of research and development invested in the popular SimGrid simulation framework [11], 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 [18], [17], [19], [20], [21], [22], [23], [24], [25], [26]. 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 [27]. To remedy this problem, the WRENCH simulation framework [13] 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.

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 [5]. Page cache is also modeled in iCanCloud through a component that manages memory accesses and cached data [6]. 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 [14] 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.

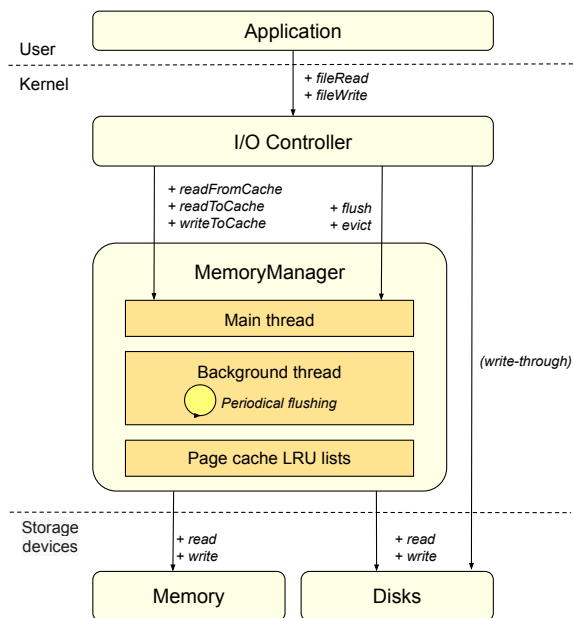


Fig. 1: Overview of the page cache simulator. Applications send file read or write requests to the I/O Controller that orchestrates flushing, eviction, cache and disk accesses with the Memory Manager. Concurrent accesses to storage devices (memory and disk) are simulated using existing models.

### III. 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.

#### A. 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 [21] 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.

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

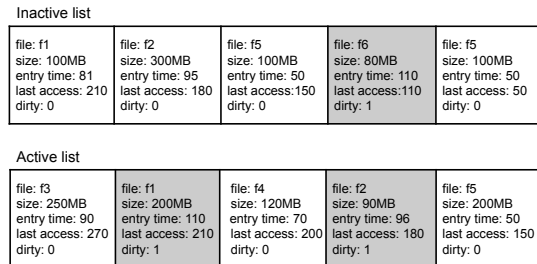


Fig. 2: Model of page cache LRU lists with data blocks.

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 [28], [15].

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.

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) *Flushing and eviction*: The main simulated thread in the Memory Manager can flush or evict data from the memory

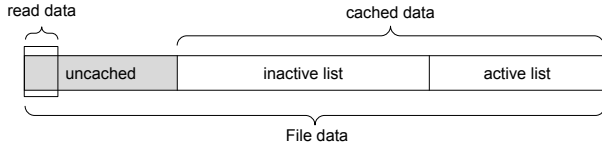


Fig. 3: File data read order. Data is read from left to right: uncached data is read first, followed by data from the inactive list, and finally data from the active list.

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.

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

---

#### 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

```

---

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.

#### B. I/O Controller

---

#### 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)

```

---

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 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.

### C. 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](https://github.com/wrench-project/wrench/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.

### D. 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. Each compute node had  $2 \times 16$ -core Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz, 250 GiB of RAM,  $6 \times$  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 was the file read by Task 1, etc). The anonymous memory used by the application was released after each task, which we also simulated in the Python prototype and in WRENCH-cache. As our focus was on I/O rather than compute, we measured application task CPU times on a cluster node (Table I), and used these durations in our simulations. For the Python prototype, we injected CPU times directly in the simulation. 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 platform and application are available at commit [ec6b43561b](https://github.com/wrench-project/wrench/commit/ec6b43561b).

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. In the second experiment (*Exp 2*), we ran concurrent instances of the application on a single node, all application

---

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

---

TABLE I: Synthetic application parameters

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 II: Nighres application parameters

instances operating on different files of size 3 GB stored in the same local disk. We varied the number of concurrent application instances from 1 to 32 since cluster nodes had 32 CPU cores. 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 [29], 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](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. 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](http://dx.doi.org/10.15387/fcp_indi.corr).

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 III: 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.

[mpg1](#), leading to the parameters in Table II.

To parameterize the simulators, we benchmarked the memory, local disk, remote disk (NFS), and network bandwidths (Table III). Since SimGrid, and thus WRENCH, currently only supports symmetrical bandwidths, we use the mean of the read and write bandwidth values in our experiments.

## IV. RESULTS

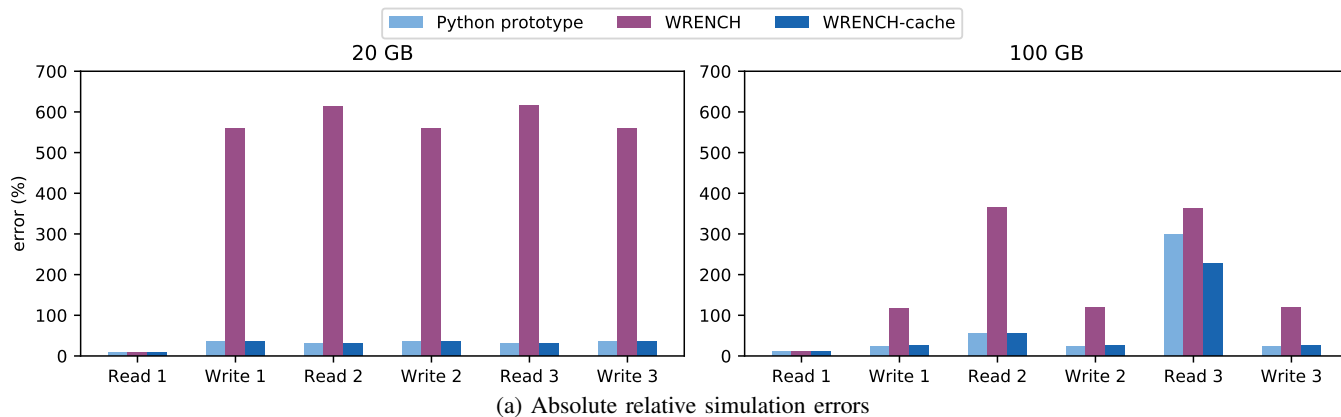
### A. Single-threaded execution (*Exp 1*)

The page cache simulation model drastically reduced I/O simulation errors in each application task (Figure 4a). 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. Results with files of 50 GB and 75 GB showed similar behaviors and are not reported for brevity.

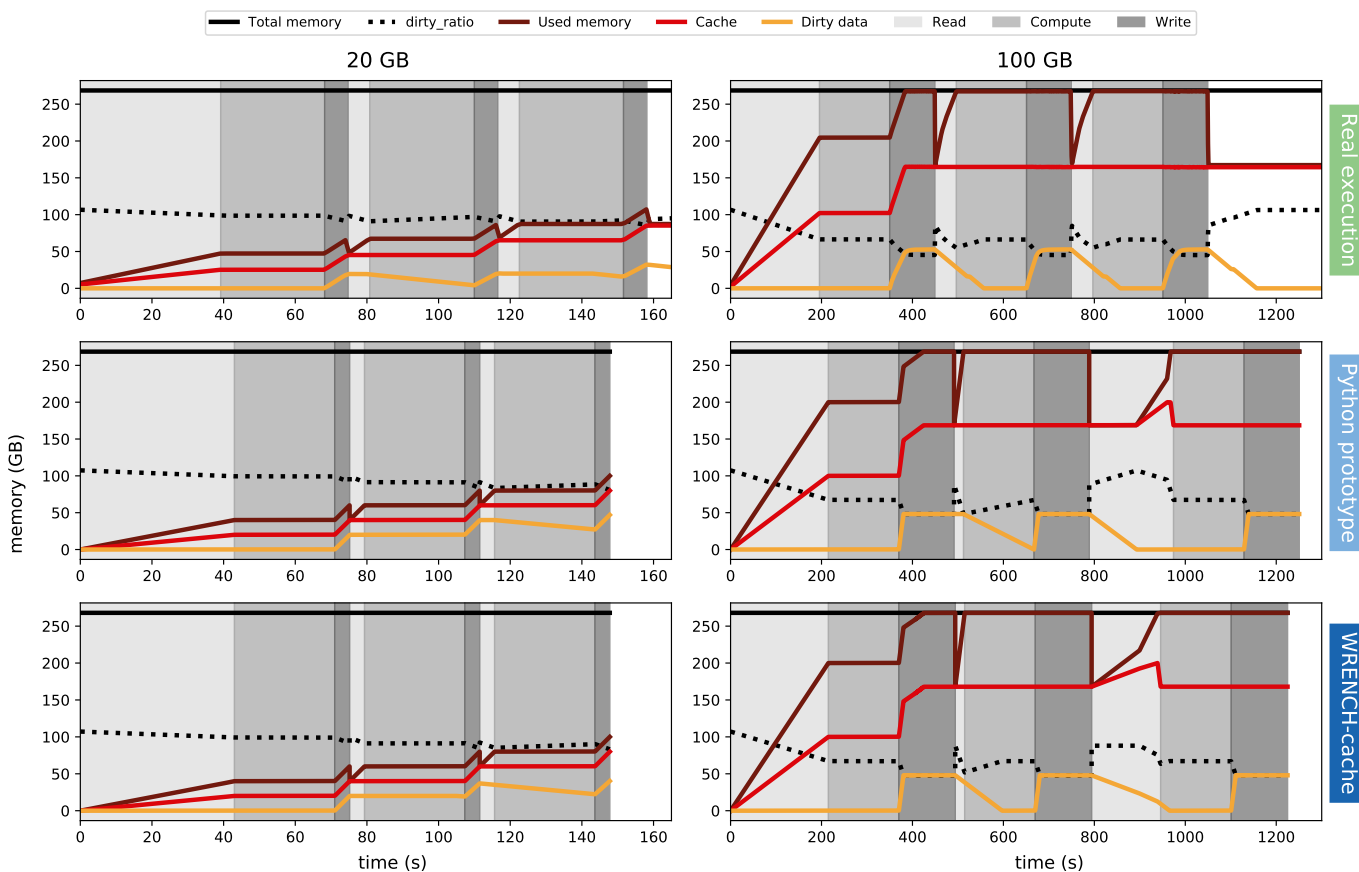
WRENCH simulation errors were substantially lower with 100 GB files than with 20 GB files, due to the fact that part of the 100 GB file needed to be read and written to disk, the only storage device in WRENCH, as it did not fit in cache. Conversely, simulation errors of the Python prototype and WRENCH-cache were higher with 100 GB files than with 20 GB files, due to idiosyncrasies in the kernel flushing and eviction strategies that could not be easily modeled.

Simulated memory profiles were highly consistent with the real ones (Figure 4b). With 20 GB files, memory profiles almost exactly matched the real ones, although dirty data seemed to be flushing faster in real life than in simulation, a behavior also observed with 100 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. 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.

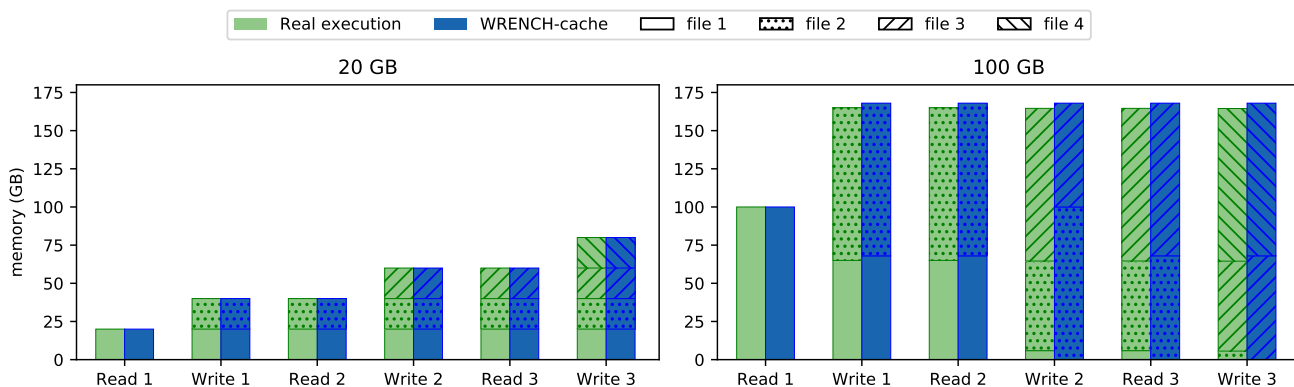
The content of the simulated memory cache was also highly consistent with reality (Figure 4c). With 20 GB files, the simulated cache content exactly matched reality, since all files fitted in page cache. With 100 GB files, a slight discrepancy was observed after Write 2, which explains the simulation 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.



(a) Absolute relative simulation errors



(b) Memory profiles



(c) Cache contents after application I/O operations

Fig. 4: Single-threaded results (*Exp 1*)

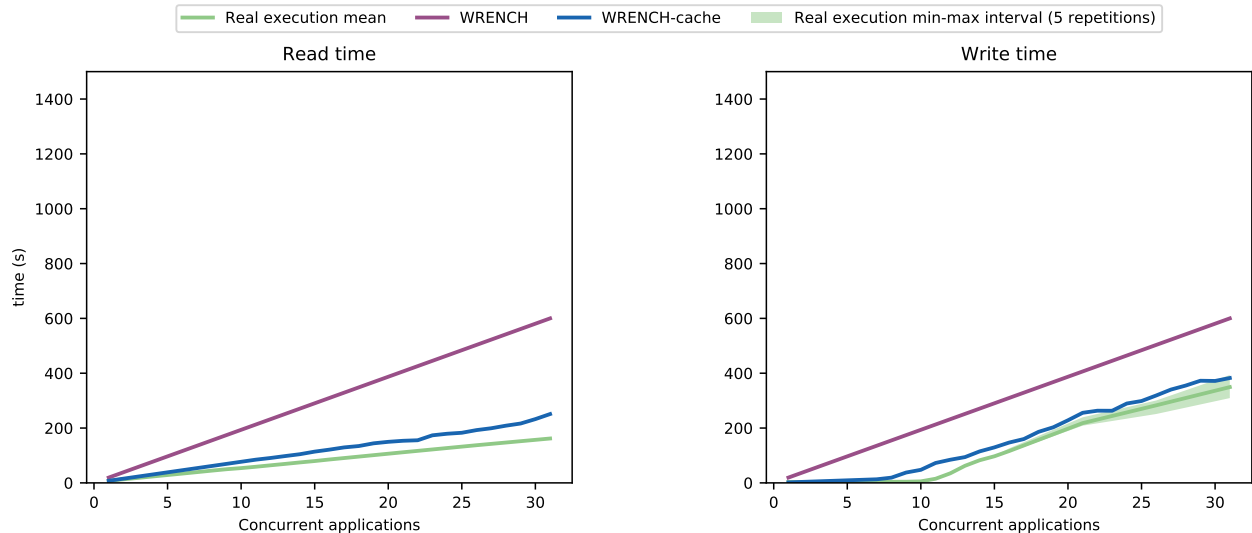


Fig. 5: Concurrent results with 3 GB files (*Exp 2*)

### B. Concurrent applications (*Exp 2*)

The page cache model notably reduced WRENCH’s simulation error for concurrent applications executed with local I/Os (Figure 5). For reads, WRENCH-cache slightly overestimated runtime, due to the discrepancy between simulated and real read bandwidths mentioned before. For writes, WRENCH-cache retrieved a plateau similar to the one observed in the real execution, marking the limit beyond which the page cache was saturated with dirty data and needed flushing.

### C. Remote storage (*Exp 3*)

Page cache simulation importantly reduced simulation error on NFS storage as well (Figure 7). This manifested only for reads, as the NFS server used writethrough rather than write-back cache. 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 applications since before this threshold, most reads resulted in cache hits.

### D. Real application (*Exp 4*)

Similar to the synthetic application, simulation errors were substantially reduced by the WRENCH-cache simulator compared to WRENCH (Figure 6). 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.

### E. Simulation time

As is the case for WRENCH, simulation time with WRENCH-cache scales linearly with the number of concurrent applications (Figure 8,  $p < 10^{-24}$ ). However, the page cache model substantially increases simulation time by application, as can be seen by comparing regression slopes in Figure 8. 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.

## V. 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  $9\times$  (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.



Fig. 6: Real application results (*Exp 4*)



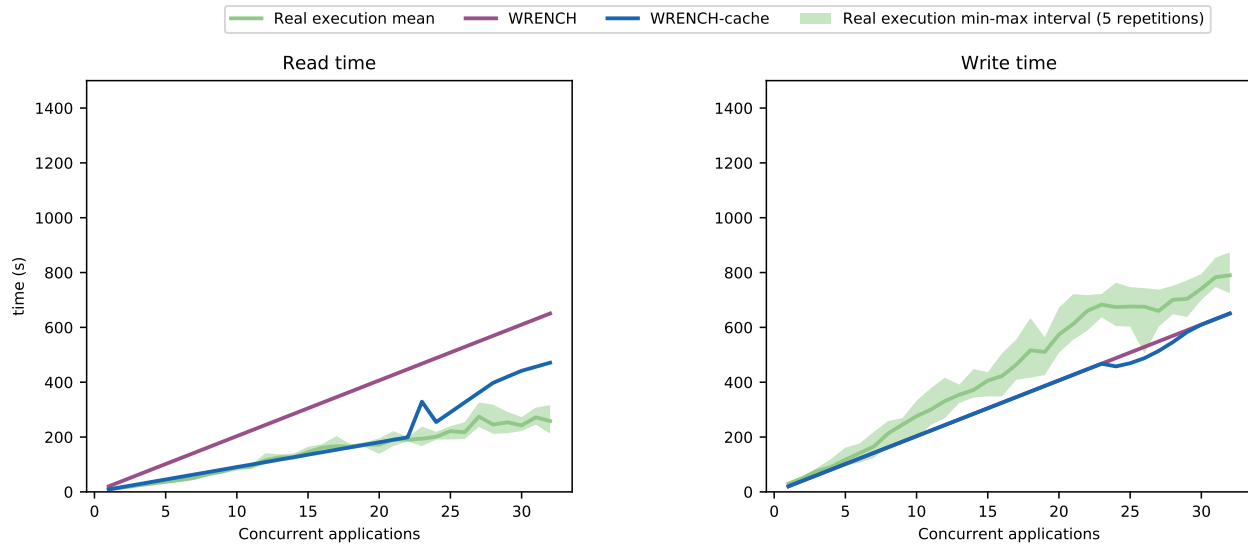


Fig. 7: NFS results with 3 GB files (*Exp 3*)

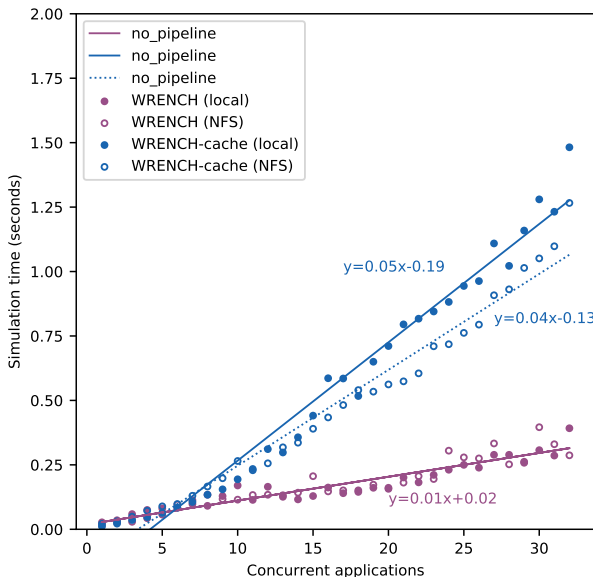


Fig. 8: Simulation time comparison. WRENCH-cache scales linearly with the number of concurrent applications, albeit with a higher overhead than WRENCH.

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 [30]. Our simulator could also be leveraged to evaluate solutions that reduce the impact of network file transfers on distributed applications, such as burst buffers [31], hierarchical file systems [32], active storage [33], or specific hardware architectures [34].

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.

## VI. ACKNOWLEDGMENTS

The computing platform used in the experiments was obtained with funding from the Canada Foundation for Innovation. This work was partially supported by NSF contracts #1923539 and #1923621.

## REFERENCES

- [1] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini, "OptorSim - A Grid Simulator for Studying Dynamic Data Replication Strategies," *IJHPCA*, vol. 17, no. 4, pp. 403–416, 2003.
- [2] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, pp. 1175–1220, Dec. 2002.
- [3] S. Ostermann, R. Prodan, and T. Fahringer, "Dynamic Cloud Provisioning for Scientific Grid Workflows," in *Proc. of the 11th ACM/IEEE Intl. Conf. on Grid Computing (Grid)*, 2010, pp. 97–104.

- [4] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, Jan. 2011.
- [5] A. Núñez, J. Fernández, R. Filgueira, F. García, and J. Carretero, "SIMCAN: A flexible, scalable and expandable simulation platform for modelling and simulating distributed architectures and applications," *Simulation Modelling Practice and Theory*, vol. 20, no. 1, pp. 12–32, 2012.
- [6] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente, "iCanCloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, no. 1, pp. 185–209, 2012.
- [7] S. Lim, B. Sharma, G. Nam, E. Kim, and C. Das, "MDCSim: A multi-tier data center simulation platform," in *Intl. Conference on Cluster Computing and Workshops (CLUSTER)*, 2009.
- [8] G. Kecskemeti, "DISSECT-CF: A simulator to foster energy-aware scheduling in infrastructure clouds," *Simulation Modelling Practice and Theory*, vol. 58, no. 2, 2015.
- [9] 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)*, 2014, pp. 104–108.
- [10] 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*, vol. 6, pp. 63 570–63 583, 2018.
- [11] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014.
- [12] 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*, 2000, pp. 53–60.
- [13] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, and F. Suter, "Developing accurate and scalable simulators of production workflow management systems with WRENCH," *Future Generation Computer Systems*, vol. 112, pp. 162–175, 2020.
- [14] J. Xu, K. Ota, and M. Dong, "Saving energy on the edge: In-memory caching for multi-tier heterogeneous networks," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 102–107, 2018.
- [15] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.
- [16] H. M. Owda, M. A. Shah, A. I. Musa, and M. I. Tamimy, "A comparison of page replacement algorithms in Linux memory management," *memory*, vol. 1, p. 2, 2014.
- [17] 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*, vol. 23, no. 4, 2013.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] A. Lebre, A. Legrand, F. Suter, and P. 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)*. Shenzhen, China: IEEE/ACM, May 2015, pp. 251–260.
- [22] L. Pouilloux, T. Hirofuchi, and A. Lebre, "SimGrid VM: Virtual Machine Support for a Simulation Framework of Distributed Systems," *IEEE transactions on cloud computing*, Sep. 2015.
- [23] 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*, vol. 28, pp. 2387–2400, 2017.
- [24] 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)*, 2017, pp. 132–137.
- [25] 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)*, 2017, pp. 92–102.
- [26] 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)*, 2015, pp. 481–490.
- [27] 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*, 2014, pp. 29–38.
- [28] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.
- [29] J. M. Huntenburg, C. J. Steele, and P.-L. Bazin, "Nighres: processing tools for high-resolution neuroimaging," *GigaScience*, vol. 7, no. 7, p. giy082, 2018.
- [30] Z. Zhuang, C. Tran, J. Weng, H. Ramachandra, and B. Sridharan, "Taming memory related performance pitfalls in Linux cgroups," in *2017 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 2017, pp. 531–535.
- [31] R. Ferreira da Silva, S. Callaghan, T. M. A. Do, G. Papadimitriou, and E. Deelman, "Measuring the impact of burst buffers on data-intensive scientific workflows," *Future Generation Computer Systems*, vol. 101, pp. 208–220, 2019.
- [32] N. S. Islam, X. Lu, M. Wasi-ur Rahman, D. Shankar, and D. 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*. IEEE, 2015, pp. 101–110.
- [33] 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)*, 2010, pp. 1–12.
- [34] V. Hayot-Sasson, S. T. Brown, and T. 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)*. IEEE, 2020, pp. 509–518.