Experiences in Managing the Performance and Reliability of a Large-Scale Genomics Cloud Platform

Michael Hao Tong, Robert L. Grossman, Haryadi S. Gunawi
University of Chicago

Abstract

We share our technical experiences in improving the performance of long-running jobs on the Genomic Data Commons (GDC), a large-scale cancer genomics cloud platform. We show how common bioinformatics workloads can cause VMs to age after several days, causing a large number of Extended Page Table (EPT) violations that significantly impact performance. We present host- and VM-level EPT monitoring and evaluate several possible mitigation scenarios. We highlight the long investigative process required for this research, with experiments requiring many days to complete.

1 Introduction

Since the invention of DNA sequencing in 1977, tremendous volumes of genomic sequencing data have been produced, and the number of biological databases has been doubling about every 15 months [1]. A driving force behind this is the exponential drop in the cost for sequencing a human genome, which is shown by the solid purple line in Figure 1 [2]. As can be seen from Figure 1, the cost of sequencing a genome has been decreasing faster than Moore's Law (shown by the dashed green line).

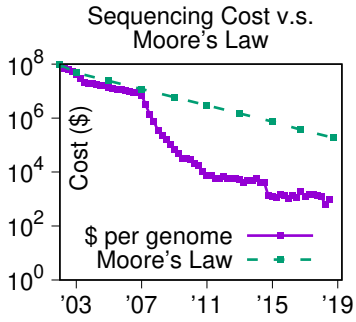

Figure 1: **Sequencing cost.**

With this tremendous growth in biological data, it has become more challenging to manage and analyze the data. As a result, in the recent years, a wide range of bioinformatics methods, techniques, algorithms and tools have been developed to analyze the experimental data and understand the underlying biological mechanisms and their significance [3].

Genomic sequencing data is particularly important in cancer, since cancer is in large part driven by genomic mutations. The GDC was launched in 2016 with the goal of providing a repository for cancer genomics data and for harmonizing data that is submitted to it with a common set of bioinformatics pipelines. The GDC is large-scale cloud based system that stores, analyzes, and shares genomic, clinical and imaging data from patients with cancer. The GDC is a hybrid cloud system and uses both a private on-premise cloud and the public AWS cloud. The GDC is one of the largest bioinformatics platforms that support the cancer research community (see Section 2 for more details).

An important backbone of the GDC is GPAS, the GDC Pipeline Automation System that is used for processing data submitted to the GDC and running a wide range of bioinformatics pipelines over the data. It is important to note that at the scale the GPAS operates, using public compute and storage clouds would be 1.5x or more expensive than using private on-premise clouds. On the other hand, using public clouds is important for GPAS for burst computing, for making use of a wider variety of machine configurations, for running portions of larger, more complex pipelines, and for flexibility in general.

On the compute side, GPAS runs a large on-premise VM cloud powered by OpenStack/KVM, and the on-premise GPAS clusters uses a combination of Ceph and Cleversafe for storage. While there are many interesting experiences to share, this paper focuses on technical matters that might benefit the systems community. In particular, we present our experiences in managing the bioinformatics pipelines and KVM performance in GPAS.

KVM performance (EPT violations under extreme memory fragmentation). We reveal a significant VM performance problem that surfaced in GPAS. We noticed that a significant number of jobs in GPAS exhibited much slower performance compared to other similar jobs, with a *degradation of up to 10×*.

GPAS workloads are I/O intensive, require large memory instances, and, most importantly, are *long running*. For example, many jobs take weeks to complete. These characteristics made the problem difficult to troubleshoot. Online monitoring tools that we typically use only report high-level aggregate metrics and did not pinpoint the root cause. Trying to replicate the problem in an "offline" setting also did not reveal the root cause because of the different environments (fresh vs. aged VMs). Recording more online metrics did not give us quick outcomes because the problem did not appear in the early days of the jobs.

Because of all of these observations, we speculated that the problem could be related to memory fragmentation of aging VMs where many sequential guest pages are not mapped sequentially on the physical pages, causing extreme translation lookaside buffer (TLB) misses. More specifically, this brought us to the root cause, the overhead of hardware-

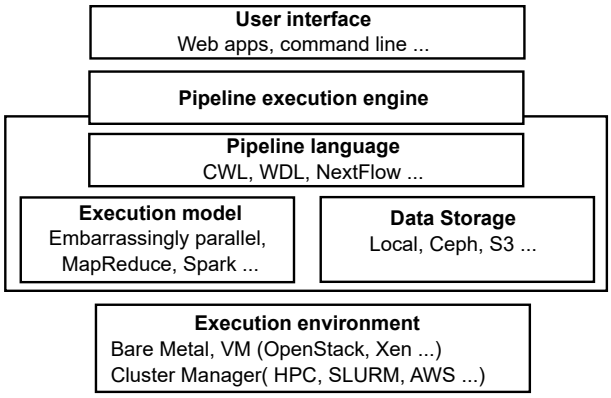

Figure 2: **Full stack bioinformatics pipeline platform.**

assisted Intel Extended Page Table (EPT) [4] technology used by the virtual machine memory management in the VM hypervisor [5]. Interestingly, we confirmed that the problem only surfaces in the OpenStack/KVM stack. We were not able to reproduce the problem in cloud VMs used by Amazon Web Services (AWS) and Google Cloud Platform (GCP). We speculate that public clouds might use their own proprietary virtualization technology (that is not available to us to analyze).

Besides presenting our findings, this paper also shows challenges in monitoring EPT performance issues. We also present several mitigation scenarios that we tried out, such as rebooting the VMs, defragmenting the memory, running on bare metal, and using public clouds, along with their advantages and disadvantages.

2 Background and Motivation

Figure 2 illustrates the full stack of layers of a typical bioinformatics cloud platform from the user/administrative interface to the execution platform. In order to improve portability and replicability, bioinformatics pipelines are often written in a workflow language [6], such as the Common Workflow Language (CWL) [7] or the Workflow Description Language (WDL) [8]. Bioinformatics pipelines are also typically containerized. There are several systems available for executing workflows expressed in workflow languages, including CWLtool [9] for CWL and Cromwell for WDL [8]. As this paper discusses the core systems aspect of GPAS, we have put additional information about the pipelines in the supplemental material [10]. To improve isolation to support security requirements, GPAS runs each containerized pipeline in a virtual machine. GPAS manages virtual machines using OpenStack/KVM. GPAS currently uses CWLtool. CWLtool does not support parallelizing tasks of a pipeline across different machines, however tasks can run in parallel across cores of a machine. Currently, GPAS uses a policy of allocating one VM per physical node, since many GDC pipelines benefit from this allocation. GPAS uses SLURM as the cluster manager for the VM pool and schedules jobs to run on SLURM. In bioinformatics, the terms "pipeline" and "workflow" are often used interchangeably.

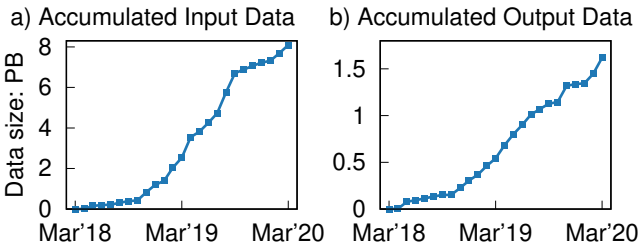

Figure 3: **Accumulated input consumption and output production over the months, §2.**

Clouds	Approx. Hourly	Approx. Annual
On premise	$0.31	$691,000
AWS (Spot)	$0.75	$1,693,000
AWS (3-yr Reserved)	$1.06	$2,386,000
AWS (On-Demand)	$2.50	$5,645,000

Table 1: **The approximate costs of on-premise vs. public cloud, §2.** *For AWS, we assume that EC2 i3.8xlarge instances are used, which are roughly similar to the on-premise instances.*

On average, over 3,000 researchers use GDC every day, with over 100,000 unique researchers using the GDC each year. Over 1 PB of data is accessed or downloaded from the GDC in a typical month. GDC currently manages over 15 PB of data across almost 100 storage servers, including the released data, the data being processed for release, temporary files, and backup copies. There are over 200 compute nodes containing 7500 cores, 50 TB of RAM, and 900 TB of local disk/SSD storage. Figures 3a and 3b show the accumulated amount of data that GPAS has processed and produced by month for a two year period. Interested readers can refer to the GDC Documentation [11] for more details.

The GPAS on-premise cloud annually processes about 322,000 workflows requiring 35 million core hours. At this scale, the costs using a public cloud would be significantly higher, as shown in Table 1. For this comparison, on-premise costs assume a 4-year amortization of equipment costs and a 15-year amortization of data center costs. The comparison assumes that on-premise nodes are used at 100% utilization.

3 Workloads in GPAS

3.1 The bioinformatics pipelines and tools

GPAS uses a wide range of pipelines that include various bioinformatics analytical tools and serve different analysis purposes. At this time, there are 10 open-sourced pipelines in the GDC GitHub repository [12].

Table 2 lists the open-source pipelines and the main tools used in these pipeline. GPAS uses a large variety of tools that include in-house software or scripts, such as HTSeq Tool and GDC VEP Tool, and widely used third-party bioinformatics tools, such as BWA and GATK. In addition to those listed in

Pipeline	Main tools used
DNA-Seq Alignment	BWA [13], Biobambam2 [14], Picard Tools [15], GATK [16]
RNA-Seq Alignment	STAR [17]
miRNA Alignment and Profiling	BWA [13]
RNA-Seq HTSeq Quantification	HTSeq Tool [18]
WXS Variant Calling	MuSE [19], SomaticSniper [20], VarScan2 [21], MuTect2 [22]
WXS Variant Filtering	Picard Tools [15]
WGS Variant Calling	CGP WGS [23]
VEP Variant Annotaton	GDC VEP Tool [24]
Mutation Annotation File (MAF) Generation	MAF Tools [25]
SNP6 Segmentation	snp6cbs [26]

Table 2: **Open-source GDC Pipelines and the main tools used in the pipelines.**

Tool	Language	I/O Intensity	Computing Intensity
samtools	C	High	Low
BWA	C	High	High
FastQC	Java	Medium	Low
MuSE	C++	Low	High
MuTect2	Java	Low	High
SomaticSniper	C	High	High
VarScan2	Java	High	High

Table 3: **Characteristics of the tools used in GPAS.** *The observations are based on the way we used the tools in* GPAS*, the results may differ if they are used in different ways*

the table, FastQC [27] and samtools [28] are also commonly used across all the GDC pipelines.

Due to the large number of tools that exist in bioinformatics, it is difficult to characterize the computing resource demand and performance of each tool before putting them into use. We list the the characteristics of some tools in Table 3 that are used in the DNA-Seq alignment and whole exome sequencing (WXS) variant calling workflows, two of the longer running workflows in GPAS. Note that the way GPAS uses the tools in Table 3 is that, for samtools, FastQC, MuSE, MuTect2, SomaticSniper, VarScan2, GPAS spawns multiple processes each of which runs the same tool on separate input files, even though the tool may support multi-threading by itself [10]; for BWA, GPAS uses BWA's own multi-threading functionality.

Table 3 shows that even though MuSE, MuTect2, SomaticSniper and VarScan2 are all somatic variant calling tools, due to different algorithms and program designs, they expose different patterns in I/O intensity. The GPAS pipelines are quite varied, and some GPAS pipelines, such as the MAF generation pipeline, require significantly less time to complete.

3.2 Pipeline Job Performance

For simplicity of discussion, we mainly measure job-level performance. A pipeline job is an execution of the pipeline that handles a specific set of input data. Readers who are interested in knowing the logical abstraction and composition of the pipeline can read our extended report [10].

The majority of the jobs in GPAS process a large amount of input data and take a long time to complete. Thus, a simple metric for understanding job performance and its degradation is the processing rate:

$$processing\ rate = \frac{job\ execution\ time}{input\ data\ size} \quad (1)$$

The unit of *processing rate* is $seconds/GB$ or $hours/GB$, thus a larger *processing rate* value means means the performance is worse (as more time is needed for processing one GB of input data).

Due to the complexity of bioinformatics pipelines in GPAS, the job performance shows quite interesting patterns. The job performance shown in this section are collected from jobs that ran on bare metal nodes, so that we avoid the interference from VMs (which will be discussed later).

Through linear regression and significance-test analysis, it is found that, among all the parameters known at the time a job starts (such as number of input files, number of CPUs allocated, etc.), input data size exhibits the strongest correlation with *processing rate*.

To better illustrate the correlation between input size and *processing rate*, jobs are sorted by their *processing rate*, and divided into 20 buckets according to the percentile of *processing rate*. Average input size of each bucket is shown in Figure 4 and 5. There can be either positive or negative correlation between input size and *processing rate*, *depending upon the specific pipeline*. There are three relationships that emerge from this analysis:

Larger input size means slower/larger *processing rate* (positive correlation): As shown in Figure 4, *processing rate* is positively correlated with input size for some pipelines. This is the case for sequence alignment pipelines. Note that, as can be seen in the graph to the right, *processing rate* of this pipeline is generally very large, more than 1500 seconds/GB. Sequence alignment involves multiple string searches and exhibits positive correlation between input data size and *processing rate*.

Larger input size means faster/smaller *processing rate* (negative correlation): Contrary to the previous result, Figures 5 shows that *processing rate* is negatively correlated with input size for some pipelines. For example, this is the case for some utility pipelines that involve accessing and extracting files from cloud buckets for processing. Note that *processing rate* in this pipeline is much faster than for the jobs in the previous one. With a high-bandwidth network and I/O, *processing rate* is higher. However, since the execution time for such pipeline jobs is short, the overhead of

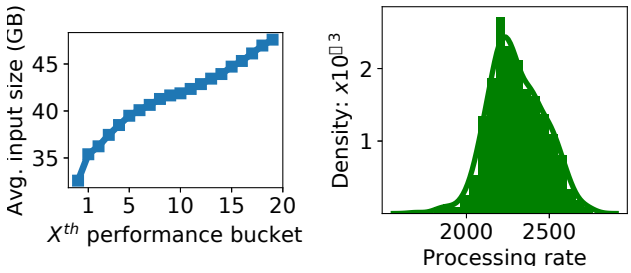

Figure 4: **A DNA-Seq alignment pipeline shows larger input size has slower/larger *processing rate*.** *Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2*

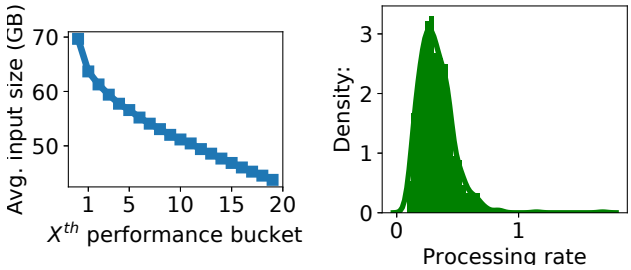

Figure 5: **An internal light-weighted pipeline shows larger input size has faster/smaller *processing rate*.** *Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2*

spawning containers and other components is significant in comparison. As a result, smaller input size leads to worse *processing rate*.

External factors interference: In addition, there are some cases that may not comply with the previous behaviors. Figure 6 shows a distribution with two peaks for the job processing rate of a pipeline. By inspecting job logs, it is found that jobs at the right peak (slower jobs) spent significantly longer times downloading data. That suggests that there might have been network congestion in the cluster during that time. Hence, in some cases, we need to take the external environment into account to understand the *processing rate* of jobs.

To conclude, through statistical analysis of the jobs, although there are some degrees of variation, we find that *processing rate* for pipelines is highly correlated with a job's input size. The exact correlation depends on the workloads associated with the pipeline. Sometimes, the *processing rate* might also be subject to external factors during some parts in the job. It is important to understand the nature of the job so that good interpretation can be formed.

In the next section, we will discuss performance issues that arise and create long tail distributions when jobs are executed on VMs.

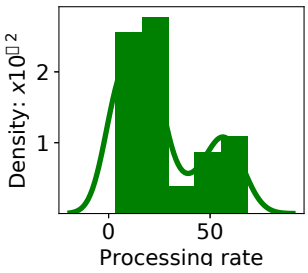

Figure 6: **Abnormal *Processing rate* Distribution.** *Each performance bucket contains 5% jobs, and buckets that have larger number demonstrate worse performance, §3.2*

4 Performance Issues: Aging VMs and EPT Violations

The GDC uses over 20 complex pipelines for processing data. Some of the pipeline components are I/O bound and some are memory bound. Complicating matters, some of the pipelines are long-running and can take several weeks to complete. After several months, it became apparent that some of the pipelines failed to complete, especially the longer running ones, which required retrying the pipelines, and which decreased the overall throughput of the system. After some experimentation, it appeared that long running pipelines performed better on bare metal nodes versus using virtual machines, and some of the nodes running virtual machines were replaced with bare metal nodes for this reason.

Figure 7a shows the success rate for jobs in each month. GPAS serves more than 20,000 jobs per month and has achieved over 90% monthly success rate for the jobs. The figure shows the success rate of jobs running on VMs (red) and bare metal (green). As noted, for long running jobs (those that require weeks), performance, as measured by GB processed per hour, tends to decline, and failures tend to become more common. Some jobs have to be stopped in the middle due to their very poor performance.

Starting in June 2019, GPAS began to run some jobs on bare metal nodes, while continuing to run most jobs on virtual machines. Although using bare metal nodes improves job completion rates, it significantly complicates the management of GPAS, since the GDC in general is designed around the setup, management, and monitoring of virtual nodes. For this reason, identifying the root cause of the higher failure rate for long-running jobs and mitigation strategies for improving the completion rate was important.

4.1 Job Performance Variance

Our next step was to quantify the slowdown. Originally, the GPAS compute pool used only VMs managed by OpenStack.

We divide pipeline jobs into comparable groups [10], and Figure 7b shows the cumulative distribution function (CDF) of *processing rate* of jobs on VMs and bare metal nodes that

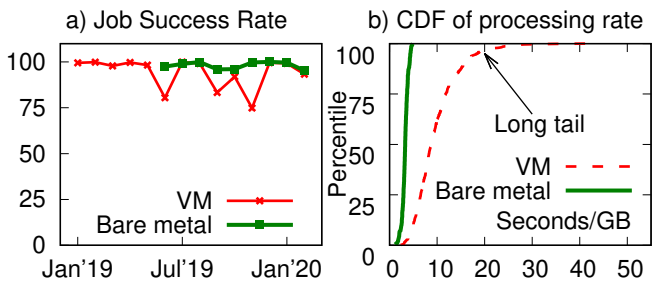

Figure 7: **Performance of jobs running on VMs and bare metal nodes, §4.** *a) shows the success rate over the months for jobs running on VMs and bare metal nodes (which were added to the computing pool at a later time); b) shows processing rate CDF comparison between VMs and bare metal nodes.*

we profiled in our deployment for one of the groups. For this figure, there are 796 and 247 jobs on VMs and bare metal, respectively, and all the jobs run the same type of pipeline (called "variant-filtration.pindel"), hence all the jobs should observe similar performance. In Figure 7a, the nearly vertical solid line represents *stable processing rate* of jobs on bare metal nodes. However, the dashed line of Figure 7b shows that *processing rate* on VMs is generally worse than *processing rate* on bare metal nodes. While this is expected, the issue lies in the *tail* performance especially at high percentiles. According to the zoomed graph in Figure 7b, starting from the 80th percentile, *processing rate* on VMs is 3 times worse than bare metal nodes, and in higher percentiles, the performance gets worse by a even larger magnitude.

We also would like to note again that the jobs in Figure 7b are jobs that come from one type of job pipeline ("variant-filtration.pindel") that generally only spends not more than 40 seconds/GB. However, there are other more CPU/memory-intensive pipelines that spend 1000 – 3000 seconds for each GB. The problem raised in this paper becomes worse for these even more intensive pipelines.

Besides, the performance variation is a wide-spread issue across all of the pipelines mentioned in Section 3. And in fact, we divided all the pipeline jobs (around 200K) into 474 comparable groups at the time of writing this paper. Among the largest 20 groups (constituting 36.5% of all the jobs), all of them exhibit a long tail in performance, and 95^{th} percentile performance is larger than 1000 seconds/GB for 12 groups.

To show that the behavior in Figure 7b is not because of degraded machines or hardware, we take six VMs that run on six different machines and show the statistics of the job performance on these VMs in Figure 8 in a boxplot. In every VM (*e.g.*, VM1 on machine1), users can run the same job pipeline repeatedly. Maximum and minimum *processing rate* are represented by the top and bottom bars, and 75^{th} and 25^{th} percentiles are represented by the top and bottom edges of the rectangle box, and median value is represented by the line inside the rectangle box. As shown in the figure, the *processing rate* in a single VM varies (even though the same pipeline) and the performance across the VMs also varies (even though the VMs have the same configuration).

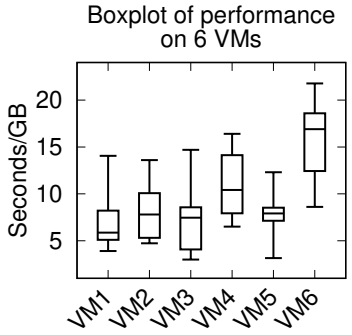

Figure 8: **Performance variance across VMs, §4.1.**

4.2 Application-Level Measurement

We manually noticed that jobs started running slower on VMs that have been running for *several days*. Thus, in order to understand the root cause to the issue, we conducted a set of experiments that included micro-benchmarks and real workloads. All experiments were conducted on a pair of VMs with the same virtual hardware setup and the same software setup, except that one of the VMs was cold-restarted before the experiment. We refer this cold-restarted VM as a *Fresh VM*. The other VMs (*Aged VM*) had been running for a few days and were selected for close monitoring after slow jobs were observed. Each VM was the only tenant on its host and configured to have 40 vCPUs, 226GB RAM and 2.5 TB storage. The hosts were equipped with two 2.20GHz 12-core 24-thread Intel Processors Xeon E5-2650 v4, 504GB RAM and 7.3 TB SSD RAID-5 storage. Linux-4.4 kernel, libvirt 1.3.1 and OpenStack Nova 13.1.4 were installed for virtual machine support.

First, we ran a simple application to reproduce our observation about a *Fresh VM* vs an *Aged VM*. To simplify the experiment, we broke down a widely used pipeline in GPAS (Somatic Variant Calling) and only selected one of the tools, VarScan2 [21]. Job pipelines in GPAS are spawned as multiple processes, hence VarScan2 can run as multiple processes. The details and the reason for parallelism is explained in [10].

Since the input data for all the tests in this experiment are the same, here we do not use *processing rate*, but instead *execution time* as the performance metric. A higher execution time implies worse performance (the same as in *processing rate*).

We define an "n-process VarScan2 *task*" as a task that uses n VarScan2 processes to process the input data. We define a "*test*" as an experiment that concurrently runs 5 tasks of the 8-process VarScan2 on a single VM. The input data is replicated 5 times and each task performs the same computation on the same content but distinct replicas of the data. After a test completes, we measure and record the *average execution*

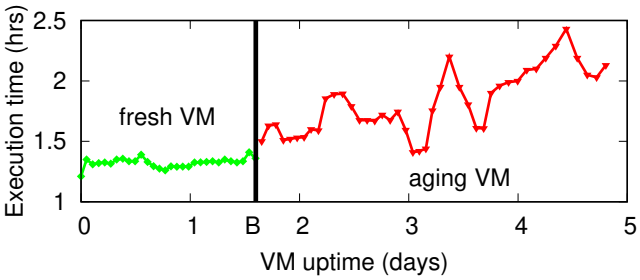

Figure 9: **VarScan2 experiment, §4.2.** *Each point in the figure represents a test of five 8-process VarScan2 tasks.*

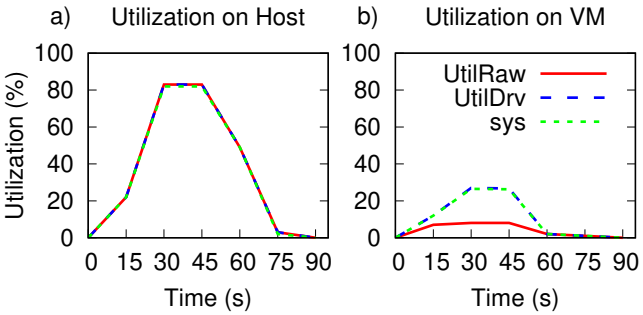

Figure 10: **CPU utilization of `sysbench`, §4.3.** *CPU utilization during `sysbench` file I/O test is presented in this figure. The graph to the left shows CPU utilization collected from the host OS, while the one to the right is collected from an Aged VM. The VM is the only tenant on its host.*

time of the 5 tasks in the test and then repeat the test until 5 days have elapsed.

Figure 9 shows the average execution time of the tasks across several days. Every y point represents a test; the y value of a point shows the average execution time of the 5 concurrent tasks in the test. We can observe that between day 0 and 1.6 (marked by line B), the execution time of the test is relatively fast with low variance. However, after approximately 1.6 days, the performance starts to degrade. We observe that VarScan2 tasks perform normally on a *Fresh VM*, but perform much worse on an *Aged VM*.

4.3 Kernel-Level Measurement

To understand the slow performance in an *Aged VM*, we conducted further experiments including micro-benchmarks and in-kernel measurements in the *Aged VM*.

We use `sysbench` [29], a configurable multi-threaded benchmark tool that provides a variety of tests for benchmarking CPUs, multi threading, memory operation, and file I/O performance. We performed many varieties of experiments (not shown here for space) and found that most benchmarking results do not reveal much difference between an *Aged VM* and a *Fresh VM* (not shown), except for file I/O performance. Not only does the I/O throughput show different results; but, more interestingly, the monitored CPU utilization on *Aged VM* and *Fresh VM* are quite different when compared to that of the host OS.

A common way to monitor CPU utilization is calculating the difference of accumulated values in the pseudo file system (`/proc/stat`). Simply speaking, the values represent how many time slices have been used for each type (user, system, etc) and for each CPU. There are many tools that profile CPU utilization including `top` and `scollector`. We use the latter as it collects and saves raw data, and we can use other tools to calculate and visualize it in desired ways.

Figure 10 shows the comparison of CPU utilization collected from the host OS versus inside an *Aged VM*. In the VM, we ran a `sysbench` file I/O test (on an SSD). It is worth noting that there are two lines in the figure (`UtilRaw` and `UtilDrv`) representing two ways we calculate CPU utilization. `UtilRaw` is the raw CPU utilization number (in %) that `scollector` outputs in every second. In addition to the raw CPU utilization, `scollector` also outputs more detailed information, including `cpuUserSlices`, `cpuSysSlices`, and `cpuIdleSlices`. We define `UtilDrv` (for derived) by summing the user and system time slices and dividing by all slices ((`cpuUserSlices`+`cpuSysSlices`)/`allSlices`). Again the difference is that `UtilDrv` simply sums the `cpuUserSlices` and `cpuSysSlices` without considering the `idleSlices`. The figure also shows another line `sys`, which represents `cpuSysSlices` divided by all the slices (in %).

We make the following important observations: **(a)** System (`sys`) CPU utilization is high on the host and equal to the overall CPU utilization. A similar observation can be found in the *Aged VM*. We consider this abnormal because the workload is I/O bound. **(b)** At the peak utilization, the CPU utilization observed on the host is 82%, while it is only 27% on the *Aged VM*. Note again that there is only one VM on the host and no other heavy workloads running on the host. This implies that *the hypervisor works intensively, a hidden CPU overhead*. **(c)** On the VM, there is a *gap* between the two ways we measure CPU utilization (the gap between `UtilDrv` and `UtilRaw`). Normally, these two lines should overlap as in the host-level measurement (left graph). What happens here is that `scollector` assumes the VM always gets all the CPU slices from the host OS. However, with our method, `UtilDrv`, it shows there is a "loss of time" due to the hidden overhead in the hypervisor. CPU time that is supposed to be used for tasks in the VM was used for other system tasks in the host. **(d)** On a separate measurement on a *Fresh VM* (not shown for space), we found no such high system CPU usage nor a gap between the two lines.

4.4 Memory Fragmentation

A major problem of aging resources is fragmentation. We started suspecting there was a memory fragmentation problem where sequential guest pages were not mapped sequentially on the physical pages. To get more evidence, we ran concurrent processes and analyzed read operations. Roughly

Tasks	1×1	1×8	4×8	5×8
Steps #1-4	< 1	< 2	< 12	< 20
Step #5	*5*	*92*	*290*	*512*
Total	6	94	306	552

Table 4: **Read latency break-down, §4.4.** *Average total time (in seconds) that each process spends in the five steps during read operations are listed in the table. $x \times n$ at the top row means a test with x number of n-process VarScan2 tasks.*

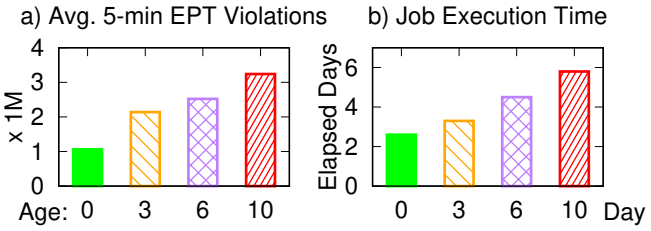

Figure 11: **Correlations between EPT violations and job execution time, §4.5.** *Each bar represents the statistics for jobs when the VM is 0, 3, 6, and 10 days old. EPT violations are recorded by enabling tracing on the physical host.*

speaking, a file read operation goes through five steps: 1) check the page cache to see whether the data is already in memory; 2) conduct a synchronized read request at the file system level; 3) send out an asynchronized read request at the block level; 4) wait for completion; and, 5) copy the data from the kernel to the user space. Step (5) represents the most memory-intensive step among all the steps.

We conducted an experiment that records the time spent in each of the steps. We set up four tests. The first test runs a single 1-process VarScan2 task, and the other four tests run 1, 4, and 5 (concurrent) 8-process tasks, respectively.

Table 4 shows the average time *every* process spends on each I/O step. Note that most of the time is spent in Step #5 (memory copying). With just 1 process (1x1), a process only takes 5 seconds in total for memory copying. With 40 processes (5x8), *every* process now takes 512 seconds in step #5, which is roughly a *100x slowdown*. We note that there is other contention in the SSD (as can be seen in steps #1-4); but, even with 40 processes, the I/O waiting time is not as severe as the slowdown from memory-copying. We also note that we have a 48-core machine, hence CPU contention should be almost negligible with 40 processes.

4.5 The Root Cause: EPT Violation

In this section, we quantify the root cause. After considerable investigation, we found that the root cause resides in the *extended page table (EPT)*, which is a technology invented to increase virtual memory performance for VMs. The use of EPT by the hypervisor is designed to be transparent to VM users. To our knowledge, EPT is used by only certain hypervisor implementations, such as Linux KVM. In a nutshell, EPT serves as a page table that stores the mapping between the VM memory address and the host physical memory address. Modern CPUs use the translation look-aside buffer (TLB) to store a small subset of the EPT entries. Whenever there is a TLB miss, an *EPT violation* occurs, which causes the hypervisor to interrupt the VM to handle the violation.

Figure 11 shows an experiment with 5 jobs using 30 cores running repeatedly on a fresh state VM for ten days. Figure 11a shows the average number of EPT violations (in millions) observed in every 5 minutes. The figure clearly shows that the longer the VM has been running, the higher the number of EPT violations. Figure 11b confirms the correlation between the number of EPT violations and job execution time.

In conclusion, VM aging leads to more frequent EPT violations causing the hypervisor to interrupt the VM more frequently. In the next two sections, we describe how we monitor and mitigate the problem.

5 Performance Management

To manage this performance problem, we discuss the two parts of our solution: monitoring and mitigation.

5.1 Monitoring

We suggest two methods to detect VM aging: monitoring EPT violation (in the host) or CPU utilization gap (in the VM), along with with the challenges.

Host-Level EPT Violation Monitoring One direct way to measure the problem is to count the number of EPT violations observed in the hypervisor, however the result is relative—how do we know whether the number represents a higher than normal number of EPT violations. We found another more concrete metric to measure this problem: *address distance of subsequent EPT violations* (which basically attempts to measure the level of memory fragmentation). For example, if two subsequent violations at time T and $T+t$ are about translation misses of guest pages #100 and #2000, respectively, then the distance recorded is 1900×4KB. Essentially, we argue that when the addresses of subsequent EPT violations are farther apart, the memory tends to be more fragmented and more EPT violations will occur, causing more time to be spent managing EPT violations.

We return to the experiment in Section 4.5 and this time plot the distribution of address distances of subsequent EPT violations. Figure 12 shows different distributions categorized based on the age of the VM age. Notice that older VMs have distinctly larger address distances. For example, in a 10-day old VM, we can see a distance of at least 50GB in roughly 40% of the time (x=50GB, y=0.6).

This method requires access to the host. It can provide performance alerts in advance. For example, the distance distribution in a 3-day old VM can be clearly distinguished from a fresh VM. Here, the resulting job execution time has

Application / VM	*vCPU Efficiency* (%)	Execution Time
Heavy / *Fresh VM*	99	16.0 hrs
Heavy / *Aged VM*	83	39.0 hrs
Light / *Fresh VM*	99	5.4 hrs
Light / *Aged VM*	99	5.6 hrs

Table 5: ***vCPU Efficiency*, §5.1.** *vCPU Efficiency and execution time for applications on the Fresh VM and Aged VM are listed in the table. There only shows a difference in vCPU Efficiency for the "heavy" application.*

been increased by 27% (which was hard to observed in the middle of the job). Thus, this kind of monitoring allows us to predict job performance degradation even before the job ends. In terms of performance overhead, sampling violations in an online manner may bring an impact to system performance. However, our experiment where the trace is enabled for five minutes every ten minutes does not show a negative impact on performance.

VM-Level /proc/stat Monitoring While the above method requires host-level access, we now present another method that can be done at the VM level. As explained before, the Linux kernel implements the `proc` pseudo-filesystem which provides an interface to kernel data structures. The `/proc/stat` file provides time-slice statistics across user, system, and idle processes from the time the system boots up. In our deployment, the Linux time slice is configured to 10ms.

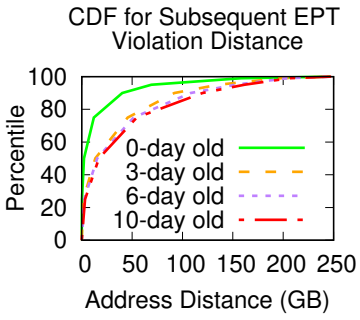

Figure 12: **EPT violation distance CDF, §5.1.**

Inspired by the "gap" shown in Figure 10 earlier, it is possible for users to monitor the gap at the VM level. *The gap is caused by a phenomenon where a time slice in the VM is actually (and significantly) less than 10ms because the hypervisor is handling EPT violations.* Thus, we can introduce a simple metric, *vCPU Efficiency*, which is the sum of all the time-slice values in `/proc/stat` (user, sys, and idle values) divided by the real time slices that have elapsed (since the last time we read `/proc/stat`). The metric should be near 100%, but when EPT violation is high, it is expected that the efficiency will be much lower than 100%.

Table 5 shows *vCPU Efficiency* and the job execution time for the same experiments we ran before. Here, we select one "heavy" and one "light" application, where the heavy application uses all 6 available cores per job and the light application uses 1 core per job. We can see that for heavy workloads, there is a large difference in *vCPU Efficiency* and execution time. For example, when *vCPU Efficiency* drops from 99% to 83%, the resulting job execution time increases from 16 to 39 hours.

Mitigation	Pros	Cons
Using Huge Pages	Performance is close to bare metal nodes.	Huge page VMs are more complicated to configure, and less flexible. Benefits of huge pages could be offset with even larger memory size and memory usage.
Restarting VM	Performance is best after restarting.	The system must support job checkpointing. Longer down time.
Defragmenting Memory	Performance is improved, short down time.	Cannot provide the best performance. Improvement is only temporary.
Running on bare metal	Performance is best and sustainable.	Does not provide the flexibility, management advantages, nor security isolation of VMs
Using public clouds	Based on one-week experiments, the performance is best and sustainable. Easy to manage.	higher costs for some workloads.

Table 6: **Pros and cons of five mitigation methods, §5.2.**

The disadvantage of this method is that we cannot detect VM aging unless we run a heavy application that can be impacted by the aging. Table 5 shows that for a light application, there is no visible difference in the *vCPU Efficiency* and execution time, even though the VM is already degraded at the time of running. We also want to emphasize that VM aging *cannot* be crudely defined by the number of days a VM has been up running. In our deployment, VMs age fast (after 3-6 days) because we ran complex bioinformatics pipelines.

5.2 Mitigations

This section describes several ways that we tried to address the problem. The choice of which mitigation to use depends upon the system requirements, and system adminstrators will need to decide which mitigation technique works best for them. Table 6 summarizes the pros and cons of the mitigation techniques we discuss below.

Using Huge Pages Just like a standard page table, the EPT size increases as the memory grows larger. Also the smaller the page size, the higher the probability of misses. Increasing the page size to 1MB for example (*i.e.* using "huge" pages) shrinks the size of the EPT table and reduces the probability

Restart Strategy	Jobs per day
Proactive restart	1.92
Slowdown-triggered restart	1.69
No restart	1.23

Table 7: **Rebooting VM mitigation, §5.2.**

of misses. However, in GPAS this is not an easy option to adopt. Huge page VMs are less flexible. Configuring and debugging huge page VMs in a production environment takes time. Hence, it is not easy to reconfigure and troubleshoot all the machines with a huge page configuration. In addition, a huge page size that is "huge" enough for now is not a permanent solution given the growing sizes of memory and the increasing application memory usage expected in the future [30]. Another option is to keep the 4KB page unit, but allocate smaller VMs to reduce memory fragmentation. However, in GPAS, resource requirements differ across jobs, and many jobs require large memory VMs.

Restarting VMs to avoid performance degradation Another method is to restart the VMs occasionally to "reset" the memory fragmentation. According to Figure 11, it is possible to detect EPT violations early before performance degrades significantly. With such a detection, we can decide when is a proper time to restart. We conducted a simple experiment with the same jobs as in Figure 11. Table 7 shows that restarting increase the number of jobs finished per day. Here, "proactive restart" implies restarting the VM after every job finishes (*i.e.*, do not reuse the VM across jobs) and "slowdown-triggered restart" implies restarting when the monitored *vCPU Efficiency* drops below 90%. The throughput numbers in Table 7 might also suggest that restarting in the middle of a long job might improve its execution time. However, this requires checkpointing the job progress, which a feature that is not supported in GPAS.

Defragmenting Memory The burst of EPT violations in aging VMs is essentially caused by the loss of data locality of the VM memory on the host physical memory. We can use the built-in memory defragmentation tool in Linux to reorganize the memory layout. A simple experiment with VarScan2 workloads shows that defragmentation indeed helps decrease EPT violations. As shown in Table 8, the test runs 21% faster after defragmentation when the VM has been heavily used for 7 days. The number of EPT violations during the test is decreased by 58%. However, comparing to performance of the fresh state, this method is still 44% slower and EPT violations are nearly 100 times more. In other words, memory defragmentation can be a temporary method to improve performance by a small margin, but restarting VMs leads to a better outcome.

Running on Bare Metal In GPAS, the most viable alternative is to run jobs directly on bare metal nodes without

VM age	Exec. Time (s)	EPT violations
0 day	587	630,636
7 days	1,073	140,677,142
7 days, defragmented	847	58,607,955

Table 8: **Memory defragmentation, §5.2.**

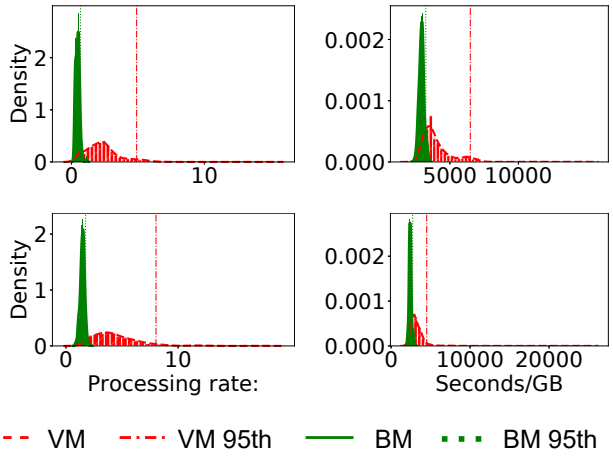

Figure 13: **Gaussian density estimation on *processing rate*, §5.2.** *Gaussian density estimation has been applied to processing rate of jobs for four pipelines Histograms are also shown for each pipelines. Vertical lines show where the 95th percentile processing rate resides for bare metal nodes and VMs.*

VMs. The caveat is that not all research projects (jobs) can run in this mode; some research projects require the security isolation provided by running the jobs in a secure VM. For this reason, GPAS now uses a combination of bare-metal and VM deployments. The statistics presented below are from job pipelines that have more than one hundred jobs on bare metal nodes.

In Figure 13, a Gaussian kernel density estimation is calculated for jobs of four particular pipelines on bare metal (BM) nodes and VMs. Bare-metal jobs exhibit much less variance than VM jobs (the bell shapes of bare-metal jobs are more localized across the x-axis). In addition, bare-metal jobs have higher *processing rate* than VM jobs. The 95th percentile *processing rate* is marked in the graphs. Note that the *processing rate* tail of the bare metal jobs is relatively short.

Table 9 shows the performance improvements of using bare metal vs VMs for the fifteen GPAS pipelines that had at least 100 or more bare metal jobs. The *processing rate* improvement (in %) at is shown in the table for different percentiles. For example, the 95th percentile *processing rate* is improved by between 22% and 95%. The average performance improvement is between 18% and 87% (not shown).

In summary, given these benefits, the GDC platform uses a combination of bare metal and VM nodes, which provides good performance, but more complexity when scheduling jobs and more overhead when managing nodes.

Using Public Clouds We next turn to the question of whether there are problems with long running bioinformatics

Percentile	Improvement	Percentile	Improvement
25^{th}	14 – 78	95^{th}	22 – 95
$median$	18 – 79	97^{th}	20 – 97
75^{th}	19 – 81	max	28 – 100

Table 9: **Running on bare metal, §5.2.** *Processing rate improvements on different percentiles by running jobs on bare metal nodes are listed in the table.*

pipelines on the virtualization stacks used in public clouds. Different virtualization stacks are likely to use different approaches, which may, or may not, have the same issues with EPT violations. Some vendors [31, 32] recommend in their documentation that VMs with large memories use configurations with huge page memory, but with no more specific guidance given. On the other hand, hypervisors such as Xen, use an approach to implement virtual machine memory management that directly maps guest virtual address to host physical address (called direct paging). For this reason, they do not incur any additional overhead when resolving the mapping from guest to host, as KVM does.

We did experiments using Amazon Web Services (AWS) and Google Cloud Platform (GCP) to understand whether there are any problems with long running bioinformatics pipelines in their virtualization stacks. To simplify the experiments, we ran VarScan2 using open access data so that the full security and compliance infrastructure normally required by GPAS would not have to be used.

Amazon Web Services has developed their own Nitro system based on KVM. We rented a dedicated host (z1d) and allocated a large memory VM (12xlarge). The one-week experiment does *not* show performance degradation. For the dataset we used, the minimum execution time is 3.6 hours and the maximum is 3.9 hours, as shown in Table 10. Nitro may offload many tradition virtualization functions to dedicated hardware, which we suspect avoids the address translation overhead.

Google Cloud Platform also developed their hypervisor based on KVM [33]. We rented a 96-core sole-tenant host to avoid sharing with other users and allocated a 90-core 576GB VM on the host. We repeated the same experiment while scaling the maximum number of jobs to 14. The experiment results also does *not* show any degradation either. The minimum and maximum execution times are 5.7 and 6.0 hours, respectively. Although they use KVM, the same problem might not appear due to one of the following potential reasons: they use huge pages; they use software-based memory management instead of EPT; or the CPUs they use have larger TLBs.

In summary, public cloud platforms do not have the problems with performance degradation for long running pipelines that we observed with our on-premise OpenStack/KVM platform. On the other hand, as mentioned, the large scale GPAS on-premise system has lower costs than public clouds for many of the GDC workloads.

	Tests	Min (hrs)	Max (hrs)
One on-prem VM	36	10	15.3
Amazon Web Services	69	3.6	3.9
Google Cloud Platform	98	5.7	6.0

Table 10: **DNA alignment workload execution time on public cloud, §5.2.**

For all of the reasons stated in this section, the GDC platform uses a combination of VM and bare-metal, with occasional VM restarts. The GDC also uses public clouds for some workloads, to provide flexibility, and for burst computing. In summary the GDC today uses a hybrid on-premise/public cloud to take advantage of the flexibility of public clouds and the lower costs provided by large scale on-premise clouds for some workloads.

6 Future Challenges

After addressing the EPT violation problem, our future goal is to improve resource utilization of our cluster. The GPAS job management system runs jobs that are encapsulated as CWL workflows [7] and Dockerized. Data that is submitted to the GDC are organized into projects or portions of a project called batches. First, GPAS must schedule different competing projects/batches. For simplicity, we will just describe the process for scheduling projects, since scheduling batches is similar but more complicated. Projects typically contain multiple data types and the appropriate CWL pipelines must be run over each data type. In the first step, GPAS schedules the running of the required CWL pipelines over the different data types in each project. In the second step, CWLtool processes the DAG in the CWL pipeline and schedules each step in the DAG. In the third step, SLURM manages the running of jobs submitted by CWLtool on the VM pool of the available compute nodes.

Currently, each bioinformatics pipeline is encapsulated in a single CWL workflow that is containerized, and the container is scheduled and assigned to a virtual machine for execution. The pipelines were developed by the research community over a period of time and some employ threads efficiently, while others are less efficient. Since portions of pipelines may be either CPU-bound or I/O bound, inefficiencies can arise. For example, Figure 14 shows five Somatic-Variant-Calling (I/O intensive) jobs consuming different input files. The left figure shows that I/O utilization is full but the total bandwidth is lower than the maximum bandwidth of the hard drive (due to seek contention). The right figure shows that for most of the time, CPU is spent in waiting for I/O. The end part (5PM) of the figure also highlights the problem in the last paragraph where there is low I/O activity but the CPUs are not fully utilized.

```
      | CURRENT:                  |  PROPOSED:
   P1: | A1  A2  A3  B1  B2  B3    |  A1  A2  A3
   P2: |         A3          B3    |  B1  B2  A3  B3
   P3: |                           |              B3
 Time: |  1   2   3   4   5   6   |   1   2   3   4
```

Inefficiencies can also arise due to the fact that currently GPAS assigns a VM per CWL container, and different portions of the CWL workflow benefit from different number of processors. As a specific example, assume we have a machine with three processors (P1–P3), and two jobs A and B, each of which has three serialized tasks (1, 2, 3) and task# 3 runs as threads. Assume that the maximum resource that A and B need are two processors, but that these are just needed for a portion of the pipeline. In this case, we have a classical scheduling allocation problem, if we devote two processors for A and B, then there are periods in which the CPUs are under utilized as shown in the Figure (the white spaces).

It would be more efficient to allocate the processors as shown on the right, where we allow both jobs A and B to be assigned on the same machine even though the total processors needed exceeds the number of processors ($4 > 3$). A simple simulation of some jobs in our cluster shows such an approach like this can increase CPU utilization up to 20% and reduce job execution time up to 15%.

While there is a vast literature on job/task scheduling (see Section 7), the challenge here is that the CWL worfklows must either be rewritten and an appropriate scheduling algorithm used or the CWL execution engine itself must have a greater ability to parallelize portions of the DAG being executed. Both of these approaches are currently being developed.

7 Related work

We now discuss related work briefly for space (please see our supplemental material for more [10]).

The introduction of EPT is aimed to increase the performance of VM memory management. In its early stage of development, many found it to be effective in reducing the translation overhead compared to software-based solutions [34]. Multiple recent studies [4, 35–37] show that *short-term* benchmarks running on more advanced modern CPUs, the overhead from TLB misses and EPT violations is small and the VM performance is comparable to bare-metal performance. We found this is not true for our large bioinformatics jobs. Huge pages can reduce TLB misses to a large extent [38, 39], but it is not always a viable solution. More recent research in the community [30, 40–44] has devoted themselves to devise a better virtual memory management or mitigate the overhead of TLB misses. Another lesson that can be taken here is the need for tools to quickly reproduce memory (allocation) aging just like the popular utility of filesystem aging tools [45–53].

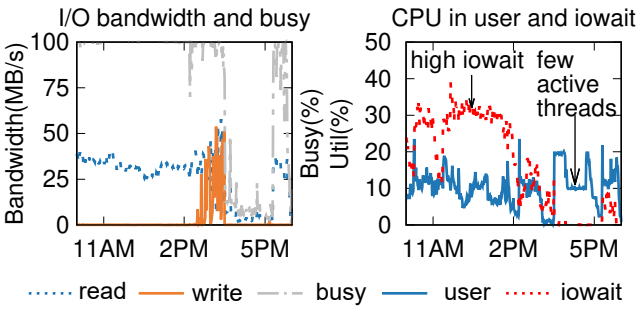

Figure 14: **CPU under utilization.**

On task scheduling, the literature is also rich on techniques that suggest finer-grained scheduling [54–56], better fairness [57–61], VM/Container backup/migration [62–64], hybrid scheduling [65–67]. The majority of the literature is concerned with optimizing new jobs that are written for a system, while the challenge with GPAS is parallelizing batches of older pipelines, most of which were designed for a single multi-core machine running a single job rather than for processing large datasets of multiple heterogeneous jobs running on a distributed platform.

8 Conclusion

To the best of our knowledge, we are the first to conduct a prolonged performance evaluation of virtualization stack for jobs that are both long running and memory intensive, such as bioinformatics jobs, and hence causing extreme virtual memory fragmentation. Diagnosing this problem has been a long and onerous process, primarily because the problem cannot be quickly reproduced; every experiment must be repeated for days to provide the evidence required. We hope the contributions of this paper can help other deployments similar to ours and lead to new research activities (*e.g.*, memory aging tools).

Acknowledgment

We thank James Bottomley, our shepherd, and the anonymous reviewers for their tremendous feedback and comments.

This project was funded in part with Federal funds from the National Cancer Institute, National Institutes of Health, agreement 14X050 and task order T02 under agreement 17X147 under contract HHSN261200800001E and NSF Grant Numbers CNS-1563956 and CCF-2028427. The content of this publication does not necessarily reflect the views or policies of the Department of Health and Human Services or the National Science Foundation, nor does mention of trade names, commercial products or organizations imply endorsement by the US Government.

References

[1] Nicholas M Luscombe, Dov Greenbaum, and Mark Gerstein. What is bioinformatics? a proposed definition and overview of the field. *Methods of information in medicine*, 40(04):346–358, 2001.

[2] K. A. Wetterstrand. Dna sequencing costs: Data from the nhgri genome sequencing program. www.genome.gov/sequencingcostsdata, 2020.

[3] L Koumakis, C Mizzi, and G Potamias. Bioinformatics tools for data analysis. In *Molecular Diagnostics*, pages 339–351. Elsevier, 2017.

[4] Timothy Merrifield and H Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 25–35, 2016.

[5] Sheng Yang. Extending kvm with new intel virtualization technology. In *KVM forum*, 2008.

[6] John Vivian, Arjun Arkal Rao, Frank Austin Nothaft, Christopher Ketchum, Joel Armstrong, Adam Novak, Jacob Pfeil, Jake Narkizian, Alden D Deran, Audrey Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314–316, 2017.

[7] Peter Amstutz, Michael R Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, et al. Common workflow language, v1. 0. 2016.

[8] Workflow description language specification. https://software.broadinstitute.org/wdl/documentation/spec, 2019.

[9] Cwltool repository. https://github.com/common-workflow-language/cwltool, 2020.

[10] Extended report, including extended background, related work, explanations and code. https://tinyurl.com/biosys-tr, 2020.

[11] National cancer institute gdc documentation. https://docs.gdc.cancer.gov/, 2021.

[12] Overview of gdc harmonization workflows. https://github.com/NCI-GDC/gdc-workflow-overview, 2020.

[13] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[14] G Tischler. biobambam2. *https://github.com/gt1/biobambam2*, 2017.

[15] Picard toolkit. http://broadinstitute.github.io/picard/, 2019.

[16] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[17] Alexander Dobin, Carrie A Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R Gingeras. Star: ultrafast universal rna-seq aligner. *Bioinformatics*, 29(1):15–21, 2013.

[18] Gdc htseq expression quantification workflow. https://github.com/NCI-GDC/htseq-cwl, 2020.

[19] Yu Fan, Liu Xi, Daniel ST Hughes, Jianjun Zhang, Jianhua Zhang, P Andrew Futreal, David A Wheeler, and Wenyi Wang. Muse: accounting for tumor heterogeneity using a sample-specific error model improves sensitivity and specificity in mutation calling from sequencing data. *Genome biology*, 17(1):178, 2016.

[20] David E Larson, Christopher C Harris, Ken Chen, Daniel C Koboldt, Travis E Abbott, David J Dooling, Timothy J Ley, Elaine R Mardis, Richard K Wilson, and Li Ding. Somaticsniper: identification of somatic point mutations in whole genome sequencing data. *Bioinformatics*, 28(3):311–317, 2011.

[21] Daniel C Koboldt, Qunyuan Zhang, David E Larson, Dong Shen, Michael D McLellan, Ling Lin, Christopher A Miller, Elaine R Mardis, Li Ding, and Richard K Wilson. Varscan 2: somatic mutation and copy number alteration discovery in cancer by exome sequencing. *Genome research*, 22(3):568–576, 2012.

[22] Gatk mutect2. https://gatk.broadinstitute.org/hc/en-us/articles/360037593851-Mutect2, 2019.

[23] dockstore-cgpwgs. https://github.com/cancerit/dockstore-cgpwgs, 2020.

[24] Gdc vep annotation workflow. https://github.com/NCI-GDC/vep-cwl, 2020.

[25] Maf-lib: the mutation annotation format library. https://github.com/NCI-GDC/maf-lib, 2020.

[26] snp6cbs: Segment tcga snp6 tangent normalized data. https://github.com/NCI-GDC/dnacopy-tool, 2020.

[27] Simon Andrews et al. Fastqc: a quality control tool for high throughput sequence data, 2010.

[28] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[29] Alexey Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[30] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D Hill, and Michael M Swift. Efficient virtual memory for big memory servers. *ACM SIGARCH Computer Architecture News*, 41(3):237–248, 2013.

[31] Huge pages in vmware vcloud nfv openstack edition. https://docs.vmware.com/en/VMware-vCloud-NFV-OpenStack-Edition/3.0/vmwa-vcloud-nfv30-performance-tunning/GUID-1F05987F-012B-4BC4-9015-CDE3C991C68C.html, 2018.

[32] How is the hugepages feature enabled for virtual machines? https://docs.oracle.com/cd/E64076_01/E64081/html/vmcon-vm-hugepages.html, 2018.

[33] Google compute engine faq. https://cloud.google.com/compute/docs/faq, 2020.

[34] Nikhil Bhatia. Performance evaluation of intel ept hardware assist. *VMware, Inc*, 2009.

[35] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):1–25, 2016.

[36] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[37] Jeffrey Buell, Daniel Hecht, Jin Heo, Kalyan Saladi, and R Taheri. Methodology for performance analysis of vmware vsphere under tier-1 applications. *VMware Technical Journal*, 2(1):19–28, 2013.

[38] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 39–50, 1993.

[39] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.

[40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, 2016.

[41] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 679–692, 2018.

[42] Jayneel Gandhi, Christopher J Rossbach, and Timothy Merrifield. Decoupling memory metadata granularity from page size, September 12 2019. US Patent App. 15/916,173.

[43] Jayneel Gandhi, Arkaprava Basu, Mark D Hill, and Michael M Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–189. IEEE, 2014.

[44] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. Smartmd: A high performance deduplication engine with mixed pages. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 733–744, 2017.

[45] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R Ganger. Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, 2018.

[46] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[47] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A Bender, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, et al. File systems fated for senescence? nonsense, says science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, 2017.

[48] Nitin Agrawal, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)*, 5(4):1–30, 2009.

[49] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O'hallaron. //trace: parallel trace replay with approximate causal events. 2007.

[50] Akshat Aranya, Charles P Wright, and Erez Zadok. Tracefs: A file system to trace them all. In *FAST*, pages 129–145, 2004.

[51] Alan D Brunelle. Block i/o layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 57, 2006.

[52] Nikolai Joukov, Timothy Wong, and Erez Zadok. Accurate and efficient replaying of file system traces. In *FAST*, volume 5, pages 25–25, 2005.

[53] Zev Weiss, Tyler Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Root: Replaying multithreaded traces with resource-oriented ordering. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 373–387, 2013.

[54] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 184–200, 2017.

[55] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. Improving resource utilization by timely fine-grained scheduling. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[56] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, 2016.

[57] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[58] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, 2016.

[59] Călin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its

application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 97–109, 2017.

[60] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[61] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[62] Wei Chen, Jia Rao, and Xiaobo Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263, 2017.

[63] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 620–634, 2017.

[64] Daeyong Jung, SungHo Chin, Kwang Sik Chung, and HeonChang Yu. Vm migration for fault tolerance in spot instance based cloud computing. In *International Conference on Grid and Pervasive Computing*, pages 142–151. Springer, 2013.

[65] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[66] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: scheduling of long running applications in shared production clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[67] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.