Scaling Data Intensive Physics Applications to 10k Cores on Non-Dedicated Clusters with Lobster

Anna Woodard, Matthias Wolf, Charles Mueller, Nil Valls, Ben Tovar, Patrick Donnelly, Peter Ivie, Kenyi Hurtado Anampa, Paul Brenner, Douglas Thain, Kevin Lannon, Michael Hildreth

Department of Physics and Department of Computer Science and Engineering, University of Notre Dame

ABSTRACT

The high energy physics (HEP) community relies upon a global network of computing and data centers to analyze data produced by multiple experiments at the Large Hadron Collider (LHC). However, this global network does not satisfy all research needs. Ambitious researchers often wish to harness computing resources that are not integrated into the global network, including private clusters, commercial clouds, and other production grids. To enable these use cases, we have constructed Lobster, a system for deploying data intensive high throughput applications on non-dedicated clusters. This requires solving multiple problems related to non-dedicated resources, including work decomposition, software delivery, concurrency management, data access, data merging, and performance troubleshooting. With these techniques, we demonstrate Lobster running effectively on 10k cores, producing throughput at a level comparable with some of the largest dedicated clusters in the LHC infrastructure.

1. INTRODUCTION

The high energy physics (HEP) community relies upon the WLCG, a global network of computing centers to analyze data produced by multiple experiments at the Large Hadron Collider (LHC). Due to the complexity of the analysis software and the quantity of data involved, each of these data centers employs a high degree of central management. Teams of system administrators ensure that the proper software is installed on every node, that data is delivered in a timely manner, and that users receive service in accordance with data center policies. As a result, the resource management software that has evolved generally assumes that the underlying system is prepared, orderly, and robust.

However, this global network does not satisfy all research needs. Ambitious researchers often wish to harness computing resources that are not integrated into the global network. This might include volunteer computing systems, independent clusters at universities, data centers part of other infrastructure services, or commercial cloud systems. While powerful, these systems are not immediately prepared to service HEP workloads, because the required software stacks, data sharing services, and workload management software are not present. Further, these resources are not dedicated and commonly evict users without warning as resource availability and scheduling policies dictate.

To address this problem, we have developed Lobster, a system for effectively harnessing large non-dedicated clusters for high throughput workloads. Unlike previous solutions, the system is designed to handle the natural consequences of non-dedicated resources in multiple dimensions: task construction, software delivery, data access, output management, and system monitoring. Lobster brings together a variety of existing systems – HTCondor, Work Queue, Parrot, CVMFS, Chirp, Hadoop – along with new capabilities to yield a comprehensive workload management system.

In this paper, we describe our experience in designing, implementing, and operating Lobster on large clusters over the course of a year. We describe how each layer of the software is affected by our assumption of non-dedicated resources, including work decomposition, software delivery, concurrency management, data management, and post-processing. A particular challenge throughout is system-level analysis and troubleshooting. Frequently, a change in the behavior or performance of a system not under our direct control – such as a remote data service – can have cascading effects throughout the system. We have developed a novel technique for tracking and observing the performance of the system comprehensively so as to facilitate this troubleshooting.

Overall, we demonstrate that Lobster is able to scale up to clusters as large as 10k cores, which is comparable in scale to the dedicated resources provided by Tier 2 sites on the WLCG.

2. BACKGROUND

The combined output of the four experiments at the Large Hadron Collider (LHC) is approximately 30 petabytes (PB) of data per year [2]. The needs of the roughly 10,000 scientists associated with the LHC experiments for computing resources to analyze this data has driven the creation of the Worldwide LHC Computing Grid (WLCG) to distribute and process this data. The WLCG is divided into four tiers. Tier-0 (T0), a single site at CERN, provides first basic processing that has to be applied to all data, known as prompt reconstruction. From there, all data is distributed to Tier-1 (T1) sites located at thirteen national-level computing facilities. T1 sites perform archiving, distribution, and reprocessing of data as new calibration data becomes available. 160 Tier-2 (T2) sites located at universities and labs provide the facilities for scientists to interact with the data and run specific analyses to generate physics results. A few hundred Tier-3 (T3) sites consist of local clusters that individual researchers use for analysis jobs that require quick turnaround and a gateway to the wider WLCG.

Computing across the LHC can be roughly divided into two categories: production and analysis.

Production computing involves the prompt reconstruction, distribution of the files to T1 sites, reprocessing, and generation of simulated datasets that provide a common starting point for all other scientific analysis. Production computing capacity is the rate limiting factor for the CMS detector [1], which can record events at 1 kHz, but is limited to approximately half of that because there are insufficient WLCG resources to store and process the full amount. The need for more production capacity has driven a significant amount of work in opportunistic computing [7, 13, 19].

Analysis computing, in contrast, is undertaken by small groups of researchers that process a subset of the production data in order to study a local interest. A typical analysis consumes approximately

0.1 to 1 PB of data, selected via a metadata service, and subsequently processed and reduced through several stages until the final result is generated. Analysis computing needs go through significant spikes and lulls, driven by the activity of individual researchers.

Historically, analysis computing has been performed at T2 and T3 centers at a lower priority than the (more predictable) production computing. But, a problem looms on the horizon: both production and analysis computing needs will grow significantly since the intensity and energy of the LHC will increase by a factor of two in 2015; needs for analysis computing are expected to increase by a factor of three to four due to the increase in event complexity and much larger data sets. Budgetary limits prevent growing the T2 and T3 resources by the same factor, so researchers must find operational fficiencies and harness new resources.

Opportunistic clusters may be able to absorb some of the needs of analysis computing. We define opportunistic clusters to be computing sites that are not specifically dedicated to the WLCG and may include campus clusters, HPC centers, and commercial clouds. These systems present new challenges because they may not have the necessary software and data management infrastructure pre-installed, and may not give preference to CMS computing in scheduling. CMS has already demonstrated the feasibility of leveraging opportunistic resources in the context of production computing [13] and also as a way to handle overflow in demand placed on T2 sites.

In this work, we consider how to harness opportunistic clusters in order to augment T3 resources for analysis computing. To accomplish this, several challenges must be met:

First, the job scheduling and execution system must permit each independent user control over what resources are harnessed and how jobs are scheduled. The current CMS workflow management tools (WMAgent [8] for production and CRAB [17, 6] for T2 analysis) use the GlideInWMS [15] framework for job management. PanDa [14], the workflow management tool at ATLAS, uses a similar approach. While this solution is efficient, it provides a single central-

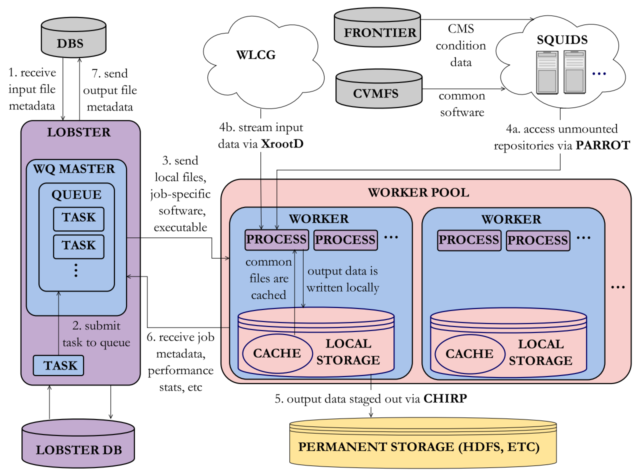
DBS
1. receive input file metadata
7. send output file metadata
LOBSTER
WQ MASTER
QUEUE
TASK
TASK
⋮
2. submit task to queue
TASK
LOBSTER DB
WLCG
FRONTIER
CVMFS
CMS condition data
common software
SQUIDS
...
4b. stream input data via XrootD
4a. access unmounted repositories via PARROT
3. send local files, job-specific software, executable
6. receive job metadata, performance stats, etc
WORKER POOL
WORKER
PROCESS
PROCESS
...
common files are cached
output data is written locally
CACHE
LOCAL STORAGE
WORKER
PROCESS
PROCESS
...
CACHE
LOCAL STORAGE
...
5. output data staged out via CHIRP
PERMANENT STORAGE (HDFS, ETC)

Figure 1: Lobster Architecture

ized scheduling point for the entire collaboration, making it impossible to harness and schedule a resource for the sole use of a single user. To address this, we employ the Work Queue [3] execution framework, which can be easily deployed on a per-user basis.

Second, each job must consume some fraction of the enormous CMS data. By definition, opportunistic resources do not provide direct, local access to CMS data, so jobs must access the data by streaming it over the WAN from WLCG repositories. This challenge has already been successfully met by the CMS "Any Data, Anytime, Anywhere" (AAA) [12] data federation using the XrootD software [10] framework. In a similar way, each job depends on a complex custom software stack which is also not installed on an opportunistic resource. For this, we rely on the Parrot virtual file system to provide transparent access to the CERN Virtual Machine File System (CVMFS) [4] .

Third, each component of the system must be designed under the assumption that running jobs will, sooner or later, be preempted by other jobs or the resource owner as availability or scheduling policies change. The costs of these preemptions are magnified by the amount of state (software and data) on the preempted node, so the system must be designed to pull the minimum amount of state and share it among jobs to the maximum extent possible.

Last but not least, the user of an opportunistic resource can only expect to have ordinary user permissions. The owner of the resource is unlikely to install software, modify kernels, or elevate privileges for a transient user on a large number of machines. Every component of the system must be able to operate effectively with a minimum of privilege.

3. ARCHITECTURE OF LOBSTER

Lobster is a job management system designed to run millions of data intensive analysis codes on tens of thousands of cores over long times scales. These resources are assumed to be non-dedicated in that they are not necessarily prepared with the required application software nor the input data. Further, the availability of the resources may vary – machines may be added and removed from the system at runtime due to scheduling policies, system outages, and

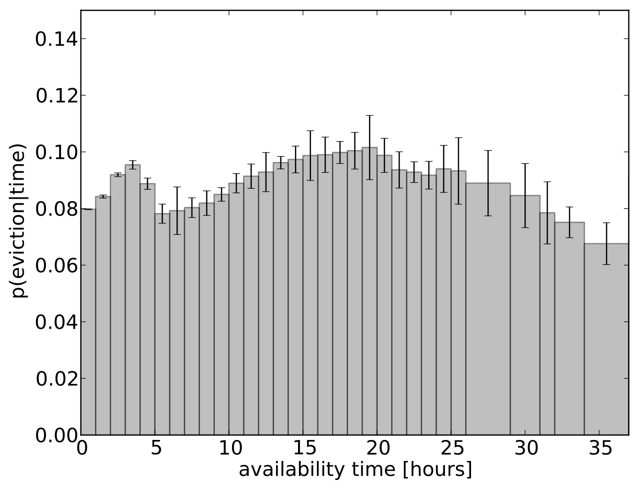

Figure 2: Worker Eviction Probability
Probability of worker eviction as a function of of its availability time, taken from physics analysis runs performed over several months. Uncertainties are estimated using the binomial model.

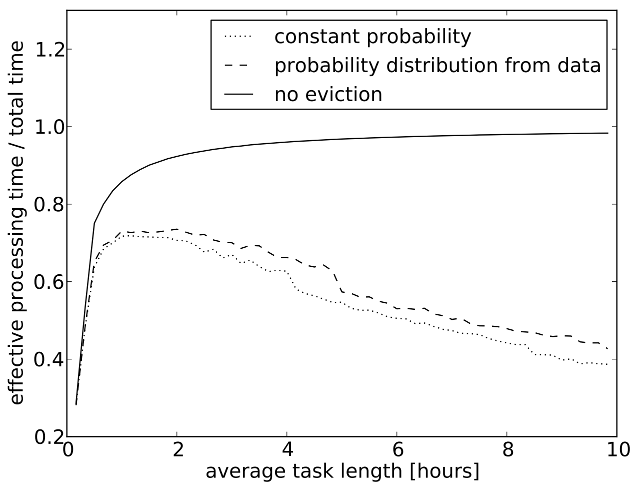

Figure 3: Simulated Efficiency by Task Length
Efficiency, calculated as the ratio of effective processing time to total time, as a function of the average task length for the simulated processing of 100,000 tasklets and assuming a constant probability of eviction (dotted), a probability derived from observation (dashed), or no eviction (solid.)

economic concerns change.

Figure 1 shows the architecture of Lobster. An execution begins with the main Lobster process that is invoked by the user to initiate a workload. The user provides a configuration file which describes the input data sources and the analysis code which is to be run on each input data source. The main Lobster process creates a local SQlite database (Lobster DB) which persistently records the mapping from tasklets to tasks. [1]

The tasks themselves are executed by the Work Queue (WQ) distributed execution system, which consists of a **master** and a large number of **workers** The main Lobster process creates an instance of a master, generates individual tasks, records them in the Lobster-DB, and then submits the tasks to the master. The master passes these tasks to workers, where the tasks are executed. As tasks complete, notification is returned to the master. As tasks are returned from the master to the main Lobster process, the Lobster DB is updated appropriately.

To get work done, the user must start workers by one means or another. Workers need not all be on the same system; they can be running simultaneously on any systems to which the user has access. Worker processes can be started individually from the command line, but more commonly the request for workers is submitted in bulk to a batch system which can start hundreds to thousands of workers simultaneously. When a worker is ready to accept a task, it makes a TCP connection back to the master, which sends tasks.

A single master will eventually reach a limit in the number of workers that it can drive directly. Scalability can be improved in two ways. First, a single worker can be configured to manage multiple cores on a machine, and run multiple tasks simultaneously, sharing a single cache directory, and a single connection to the master. In addition, the number of workers can be increased by introducing **foremen** between the master and the workers to create a hierarchy of arbitrary width and depth. In this work, we use one intermediate rank of four foreman driving a variable number of workers managing eight cores each.

[1]Conceptually, the scheduler node could be implemented as a replicated service, storing its state in a service such as Zookeeper [11]. To date, this has not been necessary because the system state is quickly and automatically recovered if the scheduler node should crash and reboot.

HEP applications generally require a fairly complex execution environment. To this end, each task consists of a **wrapper** which performs pre- and post-processing around the actual application. The pre-processing steps checks for basic machine compatibility, obtains the software distribution and the input data from the data tier, and starts the application. The application runs with either FUSE or Parrot to access software at runtime via the CVMFS global file system. The post-processing step sends output data back to the data tier and summarizes task statistics, which are sent back to the master.

4. COMPONENT EVALUATION

Each component of Lobster, from task execution to data access, contains elements that must first be evaluated independently for the opportunistic environment before considering the whole system. In this section, we consider each component in turn.

4.1 Task Size Selection

For convenience, we define three terms: A **tasklet** is the smallest element into which the overall workflow can be divided and still be submitted as a self-contained piece of work to a remote worker. The complete list of tasklets is created at the beginning of the workflow. A **task** is a group of tasklets that are assigned to run on a single worker core. Tasks are created and assigned dynamically; a buffer of 400 tasks is maintained to be assigned as workers become available. A **workflow** can be divided into individual tasks of any integer number of tasklets, where the task size is set

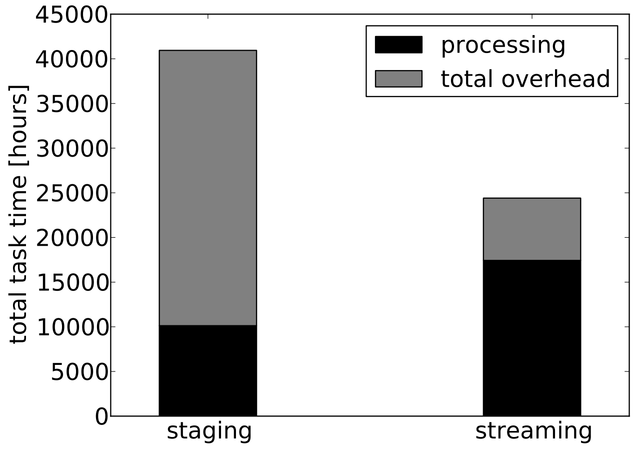

Figure 4: Data Access Methods Compared
The overall runtime for two different data access methods split into data processing and general overhead. Staging of files before and after execution results in less CPU utilization but overall runtime longer than streaming the data into the task as it runs.

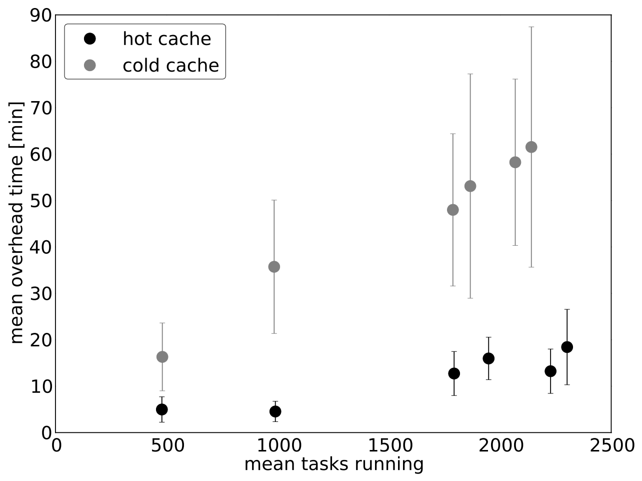

Figure 5: Proxy Cache Scalability
Mean task overhead times as a function of number of tasks sharing one proxy cache, for both cold and hot worker caches. One proxy cache can support approximately 1000 hot worker caches.

by the user and can be adjusted over the course of the run. Tasks which are too large may be evicted from a worker by the system owner before they are complete, in which case the work is lost. However, splitting the workflow into too many small tasks is also inefficient, since each additional task creates processing overhead. Consequently there is generally an optimal task size, which can vary depending on network traffic and other externalities.

We created a simple simulation model to determine the optimal task size, taking into account the distribution of task availability times, and the distribution of worker overheads, task overheads, and task execution times.

Worker availability was observed by collecting logs from multiple runs of Lobster spanning multiple months, marking the times at which a worker joined and left the system, usually due to eviction by HTCondor. The probability of worker eviction as a function of of these availability intervals is shown in Figure 2.

Overhead costs are consolidated into two categories: costs which are incurred only when a worker is started, such as populating the cache, and those which are incurred for each task, such as transferring output files. The per-worker and per-task overheads are taken to be 5 and 20 minutes, respectively. Tasklet completion times are taken as Gaussian distributed with μ=10 minutes and σ=5 minutes. The total number of tasklets to process is set at 100,000.

A pseudo-random sample of worker survival times for 8,000 workers is drawn. Tasklet processing times are drawn for each worker, incurring per-task overhead costs at task-size intervals. The sum of processing times for each completed task-size interval is added to the total effective processing time. If the total time exceeds the survival time drawn for a given worker, it is considered ”evicted”. It incurs an additional per-worker overhead cost, and a new survival time is drawn. All processing time since the start of the evicted task is considered lost and is not included in the effective processing time. Efficiency is calculated as the ratio of effective processing time to total time.

Figure 3 shows the results of running the simulation for task lengths ranging from 1 to 10 hours. Three eviction scenarios are considered: a constant probability of 0.1, the probability derived from observed availability times shown in Figure 2, and no eviction. In all three cases, the efficiency is low while task times are shorter than the average worker startup costs. Without eviction, these costs are incurred only once, so effective processing time soon dominates and the efficiency asymptotically approaches 1. This simulation is not sensitive to differences between the observed probability and a constant one; in both cases, the maximum CPU efficiency is around 70% at a task length of one hour. We consider this to be the upper limit of achievable efficiency under non-dedicated circumstances.

4.2 Data Access

Physics analyses within the HEP community rely on statistical analysis of many particle collisions, in form of events, produced by the detectors at the LHC. While an individual event may be small in size, on the order of 100kB, depending on the information stored, the overall amount of data analyzed can easily exceed 100TB. Storing such amounts of data locally for several analysis groups is not feasible for the majority of the institutions involved with the LHC, and most of the data is stored at large central computing facilities around the world. To run analyses, jobs either run on remote sites where the data is located, or rely on remote access. For the latter, the HEP community provides the XrootD protocol, which facilitates global access to data with full Globus authentication via directory servers. This is also the primary access mode to CMS data within Lobster.

Apart from the actual information recorded by the LHC, HEP analysis jobs also depend on configuration and calibration information, which is distributed from CERN through a network of proxies, using the Frontier [5] protocol.

Normally, a HEP analysis job contains only a fraction of the information present in the input data. Hence, output files normally range in the size of megabytes, where the pro-

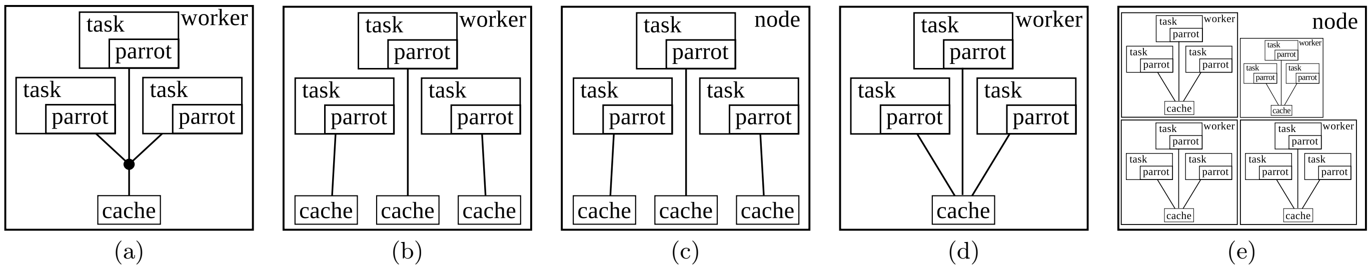

Figure 6: Parrot Cache Configurations
(a) Single cache in a worker with single read/write access. (b) Each worker has an independent cache. (c) Similar to (b), tasks may run as individual condor jobs in a computing node, with separate caches. (d) Per worker there is a single cache with single write access, but multiple read access. (e) As in (d), but a single computing node may run several workers at once.

cessed input is at least an order of magnitude larger. Yet the amount of output data from several thousand jobs is enough to overload the data handling offered by Work Queue. To thus facilitate concurrent transfer of the job outputs to a storage element, we use a Chirp [18] user level file server to provide access to a backend Hadoop cluster.

The user can provide a list of input data files directly or specify a dataset in the CMS Dataset Bookkeeping System (DBS). In the latter case Lobster communicates with the DBS and obtains the list of data files, experiment runs, and lumisections in the dataset. The actual data is then provided to the worker in one of three ways:

1. via XrootD: The worker is given the logical file name, which is uniquely specifies any file in the CMS-wide data federation. XrootD accesses a database server which looks up the physical location of the file and streams the data back to the worker.

2. via WQ: the Work Queue master process has direct access to the system on which the input files are stored and copies the data to the worker.

3. via Chirp: The user starts a Chirp server on the system where the data is stored. The Lobster wrapper starts a Chirp client on the worker, which requests files from the server.

These file access methods can be grouped into two modes: staging and streaming. For Lobster, the former encompasses WQ and Chirp, while the later is comprised of XrootD. A performance assessment of these data access modes is given in Figure 4. Compared to streaming the input files, staging incurs a penalty resulting in larger overhead that is not compensated in gains due to data locality while processing.

4.3 Scalable Software Delivery

Common components of the CMS software are obtained from the CernVM File System (CVMFS), a read-only file system from which files can be downloaded on demand. On dedicated computing resources, the CVMFS repository is mounted using the FUSE kernel module. This allows applications to run locally in a transparent manner even when their executables are kept on remote machines.

FUSE requires root access, which is not available on opportunistic clusters. On these systems we use Parrot which is able to access remote CVMFS repositories without mounting them first. When a CMS application is run with Parrot, it intercepts file access system calls and translates them as necessary using LibCVMFS. System call translation allows the remote storage system to appear as a local file system without requiring root access, recompilation, or changes to the original application.

LibCVMFS allows FUSE and Parrot to use the HTTP protocol for communications between local nodes and the CVMFS repository. This makes it possible to use Squid proxy servers, which cache HTTP requests to reduce the load when accessing CVMFS repositories.

Parrot optimizes operations using remote file systems by constructing a cache in the local file system of the worker node. For example, a `seek` operation is done with the local copy whenever possible, minimizing requests to the remote file system. When a worker node is first created, its cache is empty (“cold”), and filling it for the first time adds to overhead. Cache overhead is much lower for subsequent tasks on the same worker as the cache is already filled (“hot”) and most of the required files will be common to both tasks. This can be seen in Figure 5, which shows the mean overhead time for either cold or hot caches. The overhead increases with the number of concurrent workers running; this is mainly due to limitations in both communication bandwidth and the capacity of the Squid proxy servers. We see that one proxy is able to sustain about 1000 workers before performance begins to suffer. After that point, more proxies are needed.

When multiple instances of Parrot run on the same computing worker, however, they race to access the same default local directory to construct the cache (see Figure 6(a).). To avoid data corruption on concurrent write access on a file, Parrot instances will try to lock such file when trying to create it or modify it. Parrot instances will block and wait until they can obtain the lock for the file. When the cache is cold, that is, empty, only the Parrot instance with the writing lock can make any progress, while other instances wait for the lock to be released. When further instances obtain the lock, they will find the cache already populated, allowing for several tasks to be completed concurrently. This is of course inefficient, as only one instance may have writing access to the cache at any time.

A straightforward solution is to direct the Parrot instances not to use the default directory (see Figure 6(b)). This automatically allows for several tasks to be completed concurrently per worker, and it offers similar efficiency as running several tasks in the same computing node as individual condor jobs (see Figure 6(c)). We note, however, HEP analysis

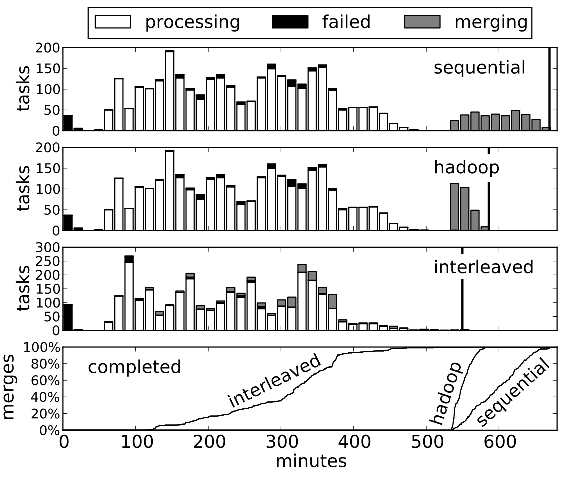

Figure 7: Merging Modes Compared
Number of finished analysis and merge tasks as a function of time for the sequential, hadoop, and interleaved merging modes. The time of completion of the last merging task is denoted with a vertical bar.

jobs use common support software, which means that the same files are requested from the CVMFS repositories over and over again. In effect, this increases the bandwidth required in direct proportion to the number of tasks running per worker or computing node. For a typical HEP analysis job, this bandwidth requirement is about 1.5 GB per cache.

These issues can be vastly improved by observing that CVMFS is a read-only file system. Once a file is in the cache, it will not be modified. Several Parrot instances can then populate the cache concurrently, with each file being copied to each worker only once, and with all Parrot instances working concurrently (see Figure 6.(d)). Not only this, but several workers may run as individual condor jobs per computing node (see Figure 6.(e)). Support for concurrent CVMFS cache access (known as alien cache) has been activated in Parrot with good results [16].

4.4 Data Merging

As noted above, Lobster task sizes are tuned to achieve acceptable performance, taking into account evictions and overhead times. However, this leads to significantly more and smaller output files compared to regular CMS workflows. While these files could be published as-is, it would require a significant amount of metadata, which increases the expense of publication and further handling. To offset these penalties, we implemented several ways to merge completed output files up to a desired file size. Typically, files of 10-100 MB are merged into files of 3-4 GB.

Sequential Merging After all data has been processed, Lobster will group the finished tasks by output size to form *merge tasks*, yielding an output file size close to a user-specified value. These merge tasks will concatenate the output files, and also merge the associated metadata. Merge tasks run in the same way as analysis tasks, transferring data via XrootD (input files only), Chirp, or, as a fall-back, Work Queue. The workload is complete when all merge tasks are done.

Merging via Hadoop. Within CMS, Hadoop is typically used to take advantage only of the bulk storage capabilities, not the Map-Reduce programming model. In this case, Map-Reduce seems to be a natural way to conduct the merge tasks, which access large amounts of data, but do not need large amounts of computation.

We created an implementation of merging that is executed entirely within Hadoop after all analysis tasks are completed. This eliminates the need for data to flow in and out of the Chirp server to the rest of the cluster. In this implementation, the Map phase is used to collect the list of small files from Lobster and group them (by name) to produce the desired size of merged output files. The grouped names are passed to the Reduce phase. In each reducer, each small file transferred to the local machine where the HEP environment is created and the local files are merged together. Then the new, larger file is copied back into HDFS.

Interleaved Merging. The sequential merge procedure as described first above places a high load on the Chirp server, as merging tasks have a short runtime and thus more output transfers occur concurrently than with data processing. This severely limits the amount of concurrent merge tasks that Lobster is able to handle. To reduce the time spend merging output files after processing, Lobster is also able to interleave merge tasks with regular tasks.

When doing interleaved merging, Lobster attempts to create merge tasks for every task/dataset that is more than to 10 % processed. Output files will only be merged once, and merge tasks will only be created when enough processing tasks have finished to create a sufficiently large merged output file. In this mode, some small number of merge tasks will run at the same time as analysis tasks.

Figure 7 compares the three different merging modes in different runs of Lobster. The bottom-most graph compares the cumulative completion of all three methods. Each graph shows the number of analysis tasks (white bars) and the number of merge tasks (gray bars) completed in each time period. For sequential and interleaved merging, XrootD was used to transfer merging inputs, while merged files were staged out via Chirp. As can be seen, sequential merging takes the longest, and suffers from a long-tail effect, like any distributed computation. Merging via Hadoop is more efficient and has a shorter tail. Interleaved merging is less efficient in use of resources, but completes faster overall because it can be done concurrently with analysis. Lobster currently uses the latter.

5. PERFORMANCE MONITORING

Due to the large number of interacting components in Lobster, troubleshooting problems can be very challenging. A hiccup in the performance of one element can have a cascading impact on the rest of the system. For example, if the performance of a data cache is impeded for some reason, this will be reflected as low bandwidth in data transfers, resulting in low CPU efficiency for individual tasks, and then eventually in extended run times. To address this problem, we have implemented a comprehensive monitoring system that covers almost every aspect of the system and the infrastructure. This enables us to disentangle the root cause of many runtime performance problems.

As a first measure, we record general information, such as workers connected to the Lobster master, tasks running, the number of failed and successful tasks over time, and the output written to disk to provide an overview over the

Task Phase	Time (h)	Fraction (%)
Task CPU Time	171036	53.4
Task I/O Time	65356	20.4
Task Failed	44830	14.0
WQ Stage In	22056	6.9
WQ Stage Out	8954	2.8
Total	320462	

Figure 8: Data Processing Runtime

throughput of the system.

To enable further drill-down, the wrapper script that runs every user task is heavily instrumented. It is broken down into logical segments: environment initialization, task execution, output transfer, etc. Each segment records a timestamp and performs an internal test for success or failure, with a unique failure code that can be emitted for each segment. A record of the performance and success of each segment is returned back to the master. The master itself adds to this record by providing timing information that is not visible to the task itself, including input transfer times, task dispatch times, overall runtime including eviction, and time spent waiting for responses.

All of these records are stored in the Lobster DB, so that it becomes easy to generate histograms and time lines showing the distribution of behavior at each stage of of the execution. Of course, data by itself does not yield a diagnosis, but this collection of data has allowed us to understand and fix problems like the following:

- High values of lost runtime suggest that the target task size is too high, because eviction limits the available computation time.
- Long sandbox stage-in times or long wait times for finished task collection suggest the usage of more foremen, to spread the load of sending out the sandbox.
- Consistently long setup times hint at an overloaded squid proxy, which can be addressed by increasing the number of cores per worker, or deploying more proxies.
- Increased stage-in and stage-out times suggest an overloaded Chirp server, which can be corrected by adjusting the number of concurrent connections permitted.

That said, troubleshooting in a non-dedicated environment remains challenging, because the underlying system is rarely in a constant state for more than a few hours at a time. Monitoring and correcting a complex set of interacting processes still requires human intervention and is a fruitful area for further research.

6. LOBSTER IN PRODUCTION

We have used Lobster over the last year to process data for the Notre Dame CMS group on nearly 20k opportunistic cores at the University of Notre Dame. While there are still opportunities for improvement, this opportunistic resource (at peak) is competitive with many of the dedicated resources within the CMS collaboration. We briefly describe our experience in running a data processing workload on O(10k) cores and a simulation workload on O(20k) cores.

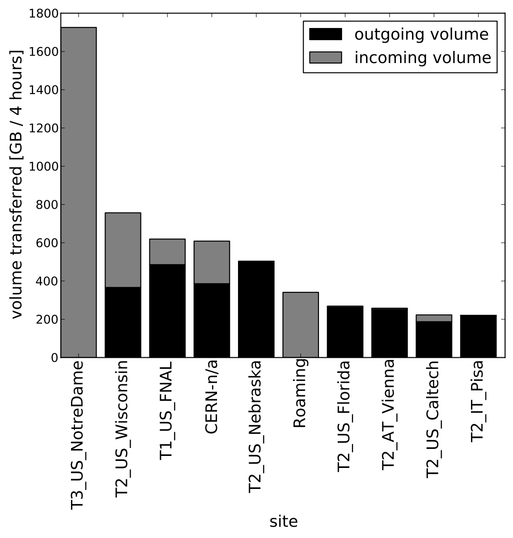

Figure 9: Data Processing Volume
Volume of data transferred via XrootD for the top ten consumers in the CMS collaboration during a 4 hour period on January 17, 2015. During this time Lobster was running around 9000 tasks at Notre Dame.

Data Processing Run. Figures 8, 9 and 10 tell the story of a large data processing run that peaked at nearly 10,000 cores over the course of two days. Figure 8 gives the overall consumption of CPU time, broken down by segments of the workload, Figure 9 compares the data consumed by Lobster compared other CMS sites, and Figure 10 shows the time line of the run in tasks running, completed, and failed.

To assess performance, the overall runtime of the tasks can be broken down as in Figure 8, which shows that about three quarters of the total runtime were spent in the task itself, either executing on the CPU or accessing data. The most significant loss of efficiency is failed tasks, caused by temporary XrootD access problems, also visible in the middle part of Figure 10. This is followed by a wall-clock time noticeably exceeding the used CPU time. The higher contribution of I/O processing time was to be expected, as the campus bandwidth, 10 Gbit/s, was entirely used up by the running tasks. Nevertheless, the overall efficiency, as seen in the bottom of Figure 10, peaks close to the expected maximum, even with the aforementioned dip in efficiency due to data access problems. Figure 9 shows that Lobster was the biggest consumer of XrootD data within CMS at the time of running, as measured by the global CMS dashboard.

Simulation Run. Running simulation tasks, on the other hand, gives a very different picture. The limitation of external bandwidth to access remote data is lessened by several orders of magnitude, as external data is only needed to overlay pile-up interactions, i.e. noise, on top of the simulated process. This allowed us to scale Lobster up to 20,000 concurrently running simulation tasks as shown in Figure 11. New challenges arise in that the squid deployed had trouble serving up the data required to create the software environment fast enough (Figure 11, second from above), which could have been mitigated by an increased number of squid servers. Alternatively, increasing the number of cores sharing a Parrot cache should lower the cost of the software

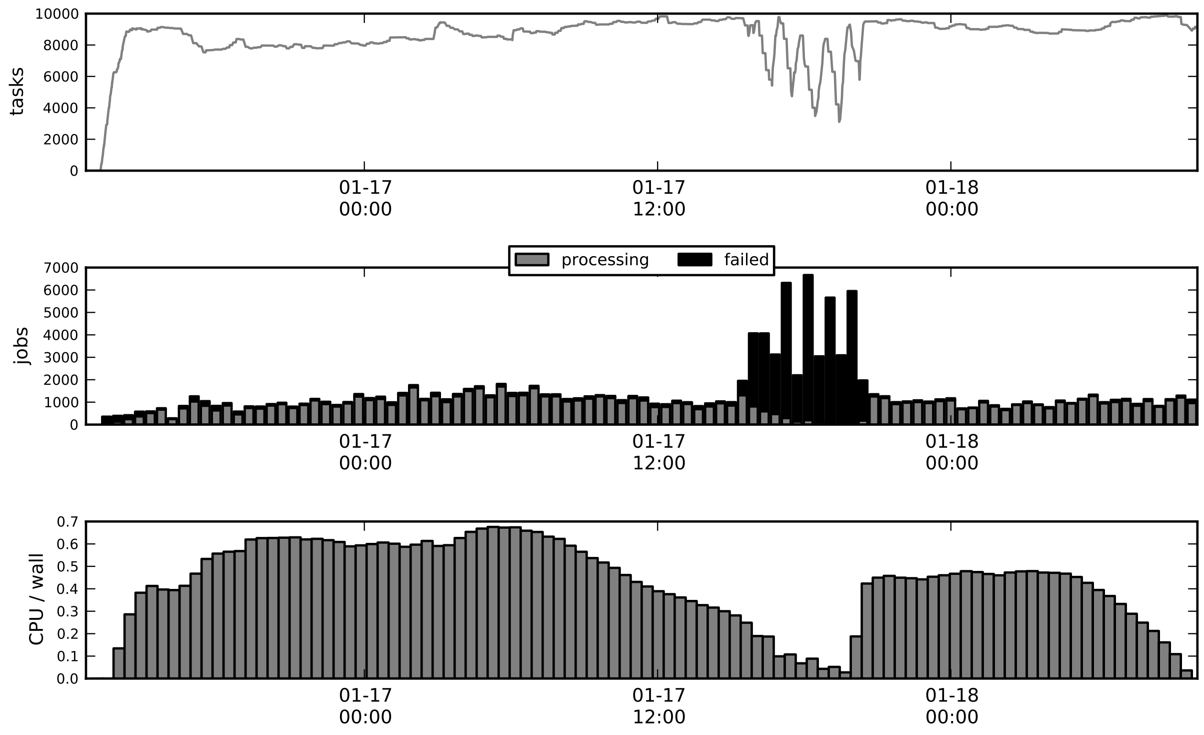

Figure 10: Timeline of Data Processing Run

The time evolution of a data processing run on nearly 10K cores over two days. The top graph shows the number of concurrent tasks running, the middle show the number of tasks completed or failed in each time unit, and the bottom graph shows the (CPU-time/wall-clock) ratio in each time unit. Note that the maximum possible ratio is approximately 70%, as discussed in Section 4.1. The burst of failures midway is due to a transient outage of the wide-area data handling system.

setup. As data is cached on the worker, the setup cost is significantly lowered for subsequent tasks running.

Similarly, the stage-out times via Chirp, second to last in Figure 11, display a periodic behavior, in which transfer times are increased due to an overloaded Chirp server. As simulation tasks require a Chirp transfer of some of their input files from the local storage element to the worker, additional load is placed on the server. Together with limited concurrent connections, which keep the underlying hardware from becoming completely unresponsive, the Chirp server handles incoming connections sequentially, and waves of tasks finishing at the same time lead to the displayed behavior.

At this scale, there is a small but continuous trickle of failed tasks, shown at the bottom of Figure 11. The most common failures are related to the setting up the required software, and are most likely caused by timeouts in connecting to the squid proxy cache. The remaining failures are miscellaneous CMS software failure modes and are transient. This underscores the need for continuous failure monitoring and compensation.

7. LOBSTER IN CONTEXT

In designing Lobster, our goal was to enable the high-throughput, data-intensive applications of HEP to run on non-dedicated resources, such as those available through campus clusters or commercial clouds. We have successfully demonstrated this capability at scales of up to 8-10k simultaneously running tasks. At this scale, limitations in network bandwidth for delivering the input data, and bottlenecks in our caching infrastructure begin to have a significant impact on application efficiency, as measured by the ratio of CPU time to wall time consumed by tasks. We have also demonstrated the ability to dispatch and run up to 20k simultaneous tasks. At this scale, we observe a precipitous growth in task overhead because the caching infrastructure becomes overwhelmed during the process of launching the first workers that need to populate their local caches. In addition, the Chirp server receiving task output becomes overloaded periodically, causing periods of long stage-out times. At the moment, we are working to overcome all of these limitations simply by deploying more cache and Chirp resources.

To place these achievements in context, it is useful to compare to the dedicated WLCG resources used by the CMS experiment. A recent survey of U.S. CMS T3 sites finds that the total deployment of T3 resources is 8899 CPU cores. The seven U.S. CMS Tier 2 facilities currently deploy a total of 43,628 CPU cores, with the individual sites ranging from 4,126 to 11,144 cores. The U.S. CMS T1 site at Fermi National Accelerator Laboratory (Fermilab) deploys approximately 11,000 CPU cores. Therefore, the scale of resources currently capable of being harnessed through Lobster is comparable to the resources deployed at the T1 at Fermilab or the largest of the U.S. CMS T2 sites. It's larger than the resources available at the average U.S. CMS T2 site and equivalent to approximately one-quarter of the entire U.S. CMS T2 resources. When running at the approximately

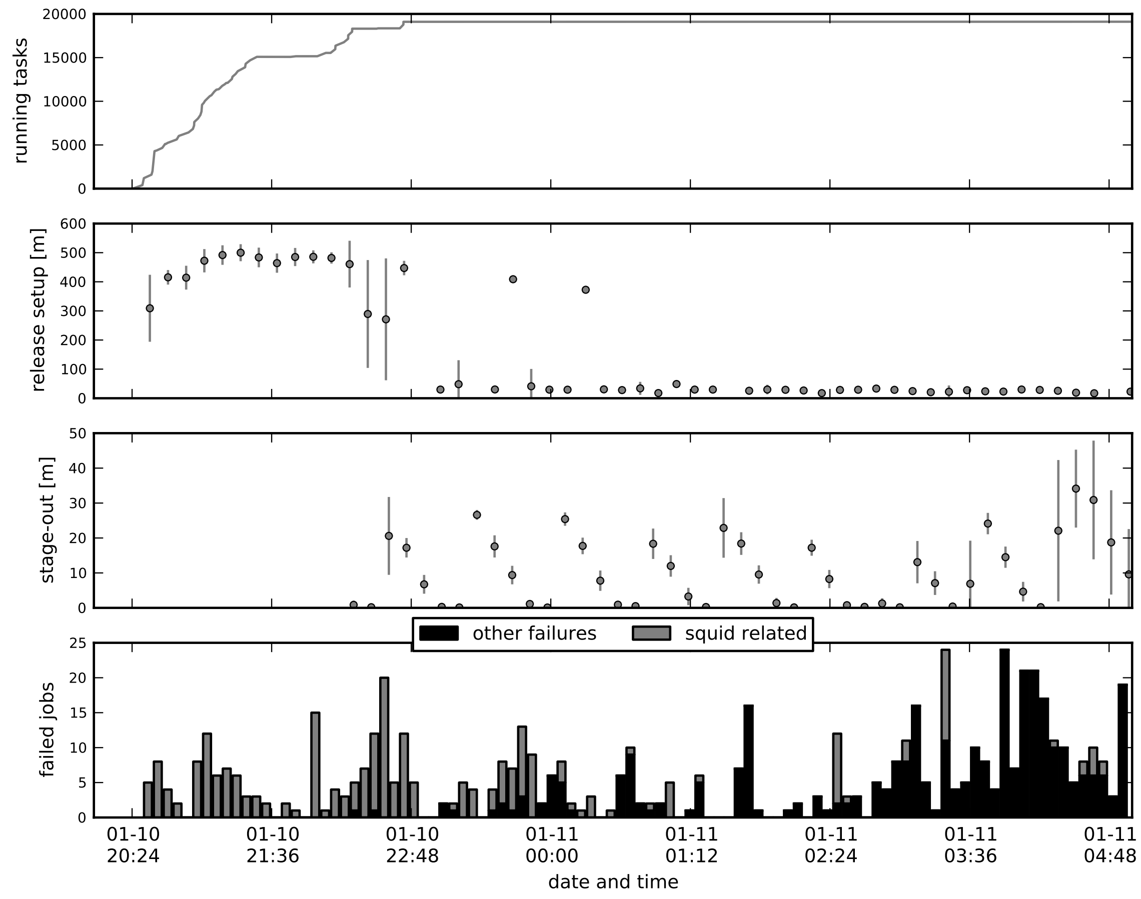

Figure 11: Timeline of Simulation Run

The time evolution of a simulation run on nearly 20K cores over eight hours. From the top: number of concurrent tasks running; time to setup the software release and initialize the environment; time to stage-out data from local to permanent storage; and exit code of failed tasks as a function of time. At the beginning of the run, the release setup time peaks around 400 minutes as cold worker caches are filled simultaneously. During this period, high load on the squid proxy cache is responsible for a small number of task failures (shown in gray in the bottom panel.) After most caches are filled, the release setup time drops, as does the prevalence of tasks exiting with squid related failures.

10k simultaneous task scale, Lobster provides more CPU resources than the entire set of U.S. CMS T3 resources. Admittedly, Lobster can only provide this scale of resources if they are physically available from one or more clusters. In our experiences, we were able to obtain this scale of resources in short bursts opportunistically from clusters hosted in our campus computing center. Furthermore, Lobster's design makes it possible to harvest resources from several clusters, and even commercial clouds, together to achieve the desired scale. Thus, Lobster makes it possible for a single user to harness a scale of resources, at least in bursts, which is significant on the US-CMS national scale.

Another way to gauge the significance of this work is to compare the scale of running tasks achieved with Lobster to the scale achieved by the dedicated global job submission infrastructure of CMS. CMS has recently merged all job submission, including production jobs managed via WMAgent and analysis jobs managed via CRAB, into a single job submission pool implemented via GlideInWMS, known as the Global Pool [9]. The Global Pool, operated on dedicated WLCG hardware by an international team providing continuous monitoring and support, has achieved a record of just over 110k simultaneously running jobs across all CMS WLCG T1 through T3 resources. The Global Pool team is currently focused on resolving bottlenecks to achieve a scale of 200k simultaneously running jobs. Lobster empowers a single user to access a scale of opportunistic resources approximately 10% the size of the global pool without intervention from systems administrators.

While the current incarnation of Lobster contains code specific to the CMS experiment, expanding the scope to other LHC experiments or HEP collaborations is feasible without much effort. As the used software delivery and data access methods are already popular within the HEP community, changes would mainly be required for the input metadata acquisition, the task sandbox, the task execution scaffolding on the worker, and the treatment of file merging. Input metadata discovery is already structured modularly, and merging treatment can easily be abstracted in a similar fashion. Code specific to the task execution on the worker would have to be completely replaced, as the setup of the working environment and the post-execution analysis of CMSSW framework reports are very CMS-specific.

8. CONCLUSION AND FUTURE WORK

The future evolution of Lobster is targeted first at finding ways, either through operational efficiencies or by deploying more network and caching resources, to exceed the approximately 10k task scale and reach the 20k scale. Beyond this, we are focused on finding ways to boost processing efficiency. A key lesson learned from Lobster to date is the importance of monitoring every aspect of opportunistic resources so that the user can quickly diagnose problems arising from transient failures or shifts in resource performance. Thus, we continue to work on improvements to monitoring. Furthermore, we are investigating ways to make use of the rich monitoring data collected via Lobster to enable automatic performance optimization through dynamic adjustment of task size in the face of changing eviction rates and resource performance. Achieving this goal will allow us to address a major source of observed processing inefficiency, resulting from the inevitable delay between a change in opportunistic running conditions and the appropriate response from the user monitoring the running tasks. Finally, we are in the process of working with the broader CMS software and computing team to implement selected features from Lobster as part of the CRAB workload management tool, to make the features of Lobster available to the wider CMS user community.

Acknowledgments

We are grateful to many people who assisted in constructing and troubleshooting this complex system. Jakob Blomer, Dan Bradley, and the CVMFS team all contributed to the interaction between Parrot and CVMFS. Dave Dykstra, Barry Blumenfeld and the Frontier team worked tirelessly to monitor and tune the wide area caching network. Sergeui Fedorov and the CRC staff were instrumental in deploying and operating the centralized services.

This work was supported in part by National Science Foundation grant OCI-1148330 and a Department of Education GAANN fellowship.

9. REFERENCES

[1] CMS Experiment Public web site, `http://cms.web.cern.ch`.

[2] Worldwide LHC Computing Grid, `http://wlcg.web.cern.ch/`.

[3] M. Albrecht, D. Rajan, and D. Thain. Making Work Queue Cluster-Friendly for Data Intensive Scientific Applications. In *IEEE International Conference on Cluster Computing*, 2013.

[4] J. Blomer, C. Aguado-Sánchez, P. Buncic, and A. Harutyunyan. Distributing LHC application software and conditions databases using the CernVM file system. *Journal of Physics: Conference Series*, 331(4):042003, Dec. 2011.

[5] B. Blumenfeld, D. Dykstra, L. Lueking, and E. Wicklund. CMS conditions data access using FroNTier. *Journal of Physics: Conference Series*, 119(7):072007, July 2008.

[6] M. Cinquilli and et al. CRAB3: Establishing a new generation of services for distributed analysis at CMS. *Journal of Physics: Conference Series*, 396(3):032026, 2012.

[7] D. Colling and et al. Using the CMS High Level Trigger as a Cloud Resource. *Journal of Physics: Conference Series*, 513(3):032019, June 2014.

[8] E. Fajardo, O. Gutsche, S. Foulkes, J. Linacre, V. Spinoso, A. Lahiff, G. Gomez-Ceballos, M. Klute, and A. Mohapatra. A new era for central processing and production in CMS. *Journal of Physics: Conference Series*, 396(4):042018, 2012.

[9] O. Gutsche and et al. Using the glideinWMS system as a common resource provisioning layer in CMS. 2015. Computing in High Energy and Nuclear Physics.

[10] A. Hanushevsky and D. Wang. Scalla: Structured cluster architecture for low latency access. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1168–1175, May 2012.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[12] Kenneth Bloom and the CMS Collaboration. CMS Use of a Data Federation. *Journal of Physics: Conference Series*, 513(4):042005, 2014.

[13] P. Kreuzer and et. al. Opportunistic Resource Usage in CMS. *Journal of Physics: Conference Series*, 513(6):062028, June 2014.

[14] T. Maeno. Panda: distributed production and distributed analysis system for atlas. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.

[15] I. Sfiligoi, D. C. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Wurthwein. The Pilot Way to Grid Resources Using glideinWMS. In *World Congress on Computer Science and Information Engineering (CSIE)*, pages 428–432, 2009.

[16] D. Skeehan, P. Brenner, B. Tovar, D. Thain, N. Valls, A. Woodard, M. Wolf, T. Pearson, S. Lynch, and K. Lannon. Opportunistic High Energy Physics Computing in User Space with Parrot. In *IEEE Symposium on Cluster, Cloud, and Grid Computing (CCGrid)*, pages 170–175, 2014.

[17] D. Spiga and et al. The CMS remote analysis builder (CRAB). In *High Performance Computing (HiPC)*, 2007.

[18] D. Thain, C. Moretti, and J. Hemmes. Chirp: A Practical Global Filesystem for Cluster and Grid Computing. *Journal of Grid Computing*, 7(1):51–72, 2009.

[19] D. Weitzel, I. Sfiligoi, B. Bockelman, J. Frey, F. Wuerthwein, D. Fraser, and D. Swanson. Accessing opportunistic resources with Bosco. *Journal of Physics: Conference Series*, 513(3):032105, 2014.