Opportunistic High Energy Physics Computing in User Space with Parrot

Dillon Skeehan, Paul Brenner
Center for Research Computing
University of Notre Dame
Notre Dame, IN, USA
paul.r.brenner@nd.edu

Ben Tovar, Douglas Thain
Dept. of Comp Science & Engineering
University of Notre Dame
Notre Dame, IN, USA
dthain@nd.edu

N. Valls, A. Woodard, M. Wolf, T. Pearson, S. Lynch, K. Lannon
Dept. of Physics
University of Notre Dame
Notre Dame, IN, USA
klannon@nd.edu

***Abstract*—The computing needs of high energy physics experiments like the Compact Muon Solenoid experiment at the Large Hadron Collider currently exceed the available dedicated computational resources, hence motivating a push to leverage opportunistic resources. However, access to opportunistic resources faces many obstacles, not the least of which is making available the complex software stack typically associated with such computations. This paper describes a framework constructed using existing software packages to distribute the needed software to opportunistic resources without the need for the job to have root-level privileges. Preliminary tests with this framework have demonstrated the feasibility of the approach and identified bottlenecks as well as reliability issues which must be resolved in order to make this approach viable for broad use.**

Keywords- grid; opportunistic computing; job eviction; HTCondor; CVMFS; Parrot; XROOTD; user space; remote I/O; high energy physics.

I. INTRODUCTION

In this paper, we propose and implement a framework to optimally utilize opportunistic grid resources without root privileges (user space), in the context of supporting the computational component of the high energy physics (HEP) Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider (LHC) at CERN [1].

CMS is an ongoing experiment searching for rare physical interactions in proton collisions [2]. The rate for proton collisions must be extremely high (20 million to 40 million collisions per second) due the rarity of interesting interactions, which occur at a rate of about 1 in a billion [3]. A fast trigger system selects a subset of rare events and discards the remainder in real time; however over the course of a year, the CMS experiment still generates on the order of 2 petabytes (PB) of collision data [4,5,6].

In order to process all the data coming from the LHC's operation, the Worldwide LHC Computing Grid (WLCG) was formed in a tiered computational facility structure and in concert with existing computing grid infrastructures such as the Open Science Grid and European Grid Infrastructure [7,8,9].

Despite the large scale of current computational resources dedicated to this experiment, the amount of data it generates each year is limited to only half of the maximum possible bandwidth by the available processing power. In the future, this problem is expected to worsen as higher collision rates and more energetic collisions will produce even more data to be analyzed. Additional processing power is thus needed in order to make use of the full data set.

CMS has initiated a collaborative effort to opportunistically utilize computing resources. Opportunistic resources are those that, while not dedicated to CMS, are available during idle times at substantially lower cost than dedicated resources. Opportunistic approaches are particularly well suited to economically handle the spikes in demand that arise because of the frequent, substantial variations in data generation rates that normally occur during LHC operations. In the future, the CMS computing strategy will likely evolve toward using dedicated resources to handle the base level of need, while absorbing spikes using opportunistic resources. Example sources of opportunistic resources include idle servers and desktops on university campuses, lower cost excess capacity on commercial cloud platforms (available through spot pricing) [10,11], and private/public cloud infrastructures with off-peak capacity.

Running CMS jobs in the above environments requires overcoming a number of fundamental challenges. These include the distribution of both the CMS software and data to clusters that have no preconfigured connection to the WLCG infrastructure as well as the potential for a CMS job running on a remote system to be terminated with little or no warning when the priority owner needs the resources. At the moment, the CMS software has no mechanism for recovering the partial progress of interrupted jobs [12].

A large variety of tools and approaches exist for resolving distributed systems challenges of remote data staging, resource-intensive I/O, runtime environment configuration, and job eviction [13,14,15,16]. Our research scope is framed from a subset of existing integrated tools which function within the requirements and limitations of the complex and legacy components of the HEP community. We have focused on the challenges of operating efficiently in an opportunistic computing environment, in which large amounts of data must be accessed remotely via a wide area network (WAN), user jobs cannot make root-level changes/installations, and the opportunistic resources evict stochastically without prior warning.

II. DATA ACCESS ON OPPORTUNISTIC RESOURCES

Opportunistic computing requires access to data as well as software. CMS data is stored in several globally distributed data centers and provided to analyses as needed.

The process is I/O intensive because of the extraordinary amount of data, typically about 400 terabytes, required to complete each single analysis. The I/O strategy must take into account the capabilities and limitations of the host, system software, user software, network bottlenecks, and priorities and policies of the available hosts.

In the absence of systems software homogenity or a fully virtualized and homogeneous cloud platform, the common grid (distributed) computational environment requires accomodation for high degrees of heterogenity, fault tolerance, and variable performance. Opportunistic computing on such infrastructure layers adds the challenge of jobs being evicted from the system without warning and opportunistic users being rarely capable of making root-level changes. We have architected a framework for opportunistic I/O-intensive computation on remote data, taking into account the capabilities and limitations of the resources and social policies of dynamically volunteered systems.

A. CMSSW Software via CVMFS and Parrot

To better handle the challenges of heterogeneous software environments, we capitalized on existing tools for handling remote I/O and software repositories such as Parrot and the CERN Virtual Machine Filesystem (CVMFS) respectively [17,18,19,20,21].

CVMFS [22,23,24] is a read-only filesystem that is aggressively cached and downloaded on demand, providing experimental analysis software to the remote host in a fast and scalable way. On the security side, data integrity is ensured via cryptographic checksums. To increase reliability, the repositories can be mirrored and load balanced amongst different proxy servers.

In regular operation, before a processing node can access a CVMFS repository, the repository has to be mounted through the FUSE kernel module, which requires root access. This presents a challenge when using opportunistic resources, as applications may have to run with restricted permissions and root access is often not permitted. We use Parrot to address these issues. Parrot is an interposition agent that transparently translates file operations of existing programs written for local file systems to remote file systems. This is done by capturing system calls using the ptrace debugging interface. Parrot is able to access CVMFS repositories without mounting them first, which eliminates the need for root access, at the expense of increased system call latency due to the translations.

We have evaluated three different CVMFS cache modes for Parrot instances running on the same node: dedicated, single-lock, and single-shared.

In **dedicated mode**, each Parrot instance has its own cache, which is always empty (i.e., "cold") at the start of the instance execution; dedicated mode produces the highest network band-width and file-system utilization, as each needed file has to be copied once per instance.

In **single-lock mode**, there is a single cache that can be used only by one instance at a time. Only the first instance to execute will find the cache cold, as further instances can reuse the cache. Single-lock mode reduces network band-width and file-system utilization, as each needed file has to be copied once. We recommend single-lock mode for debugging and testing, but not for production, as it greatly increases turnaround time with no two Parrot instances making progress concurrently.

In **single-share mode**, multiple instances of Parrot access the single cache on each machine. and we recommended it for production. Single-shared mode is implemented on top of the CVMFS option alien-cache, thus it assumes that all of the Parrot instances are being run under the same UNIX user group (GID), and that such credential is enough to have read/write rights on the file-system. If this assumption is false, we recommend the dedicated cache mode as a fallback.

B. Configuring and Submitting CMSSW Jobs

The workflow in Figure 1 was developed for the University of Notre Dame (ND) Condor computing resource pool and can be executed from any of the ND Center for Research Computing head nodes.

At the start of a representative workflow execution, an update script is run to verify currency of a locally-installed copy of the grid software. Parrot access is set on user login via modification of the user shell startup configuration file to include Parrot's path, the http proxy for Parrot to access, and the http address of the local CMS site.

The CVMFS and CMSSW initialization script is more standard. It sources both the CMS Remote Analysis Builder (CRAB) startup file and the cmsset_default.sh file to give access to CRAB and the Source Configuration Release and Management (SCRAM) commands. This should be done inside Parrot, as SCRAM may only be accessed in the CMSSW repository.

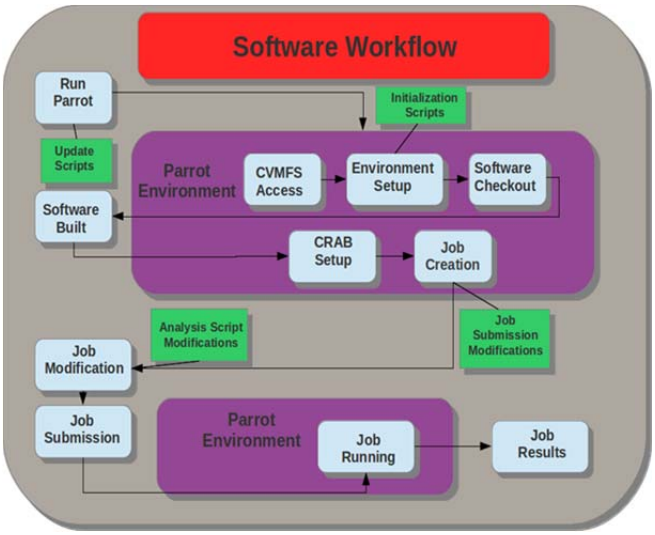

Figure 1. Software workflow

For our implementation, the grid initialization script includes an environmental variable specifying the directory of the local copy of the grid repository and a command to source the particular version of the grid software being used. To make this work for a local copy of the grid software not residing in CVMFS, the grid-env.sh script was modified so that all instances of the grid path refer to the environmental variable set in the grid initialization script. As a note, the initialization of the grid environment should occur outside of Parrot due to security concerns.

In Parrot, analysis software can then be checked out from the repository to a local setup site individual to each user. However, compilation cannot be performed while in a Parrot environment but must be performed directly on a local host. This is due to a current Parrot limitation when wrapping multithreaded applications. The software build step currently requires a local installation of the CMSSW tools. Most sites have dedicated local resources for CMS analyses, which include such tools.

After setup of the software on the remote host, CRAB assembles the data and software needed for the analysis. However, CRAB only creates the Condor submission script when given the submit command. A workaround to this problem consists of submitting and subsequently killing the jobs. With the Condor submission script and the CMSSW shell script in hand, modifications were made so that job requirements are rewritten to enable Parrot running on each worker node. A CMSSW-fixer script (cmssw_fixer) and Condor job-fixer script (job_fixer) were implemented to facilitate this change. The cmssw_fixer script targets the CMSSW.sh script, which is the job control script that is submitted to the Condor host. The CMSSW.sh file undergoes a targeted change that causes Condor to be considered a valid middleware manager. This allows each node that receives a job to set up the proper environment for the CMSSW analysis. The rest of the script is unmodified in order to ensure no broken dependencies.

The job_fixer script targets the Condor submission script, a .jdl file hidden in the .condor_temp folder in the CRAB directory. This file is modified to support Parrot as the executable being run on each worker node. The argument variable is changed so that Parrot receives the CMSSW.sh file as the script to run on the previous, old arguments. Finally, the getenv variable is set to true in order to make sure the Parrot shell running on the worker node has all proper environmental variables set. Commands to stream the output and errors can optionally be set as well in order to monitor the success or failure of the CMSSW analysis. Jobs can then be submitted to the Condor pool, where the runtime analysis proceeds normally. It should be noted that the Condor submission requires the user be running in a Parrot-free shell. Otherwise, jobs are halted immediately, as each worker node will receive the environment of host user which was running in a Parrot shell. An error will be thrown, as Parrot cannot be run inside itself.

C. HEP Experimental Data via CRAB and XROOTD

Unlike access to CMSSW, which can be performed by any user regardless of their affiliation to the CMS experiment, access to both collision and simulation data requires verified credentials from a sponsoring user group organization such as CMS or ATLAS. Grid credentials, known as certificates, grant the access needed in order to find and transfer data stored in different institutions around the world. These certificates indicate that the user is trustworthy and is capable of accessing and writing back data that is correct. This also provides a way to monitor and report on which user groups are utilizing the various grid resources (CMS, ATLAS, etc.).

Small complications were discovered between Parrot and the mechanism for obtaining and verifying grid credentials based on certificates. Repository switching in Parrot is a new, experimental feature, so for this implementation a local copy of only the necessary grid software is maintained and updated daily to ensure that all grid security certificates are up-to-date.

The reason behind this requirement is that the grid software directory maintains a security paradigm, known as Certificate Revocation Lists (CRLs). The CRLs are unique to CERN and are maintained at regular intervals inside the grid-security directory in the grid repository in CVMFS. If a user tries to access grid resources, such as a proxy, with old CRLs a security error will be thrown and the request will be denied. Attempts to use the fetch-crl script to automatically update the CRLs were not successful. Thus, we simply copy the contents of the grid security directory from CVMFS to the local grid repository on a daily basis. This requires a negligible amount of time and ensures that all CRLs are up-to-date when attempting to access grid resources.

Despite the large size of the grid repository, only a fraction of the volume is needed by an institution, since the repository maintains multiple versions of the grid software. For our purpose, only version 3.2.11-1 was used along with the /etc directory housing the grid security and related subdirectories. The default directory was symbolically linked to this version, in order to satisfy commands that refer to this default setting.

Because jobs are submitted outside of Parrot, the environment should be set up also outside of Parrot in order to make sure all job dependencies are successfully communicated to the Condor scheduler. The grid credentials are then passed through to the jobs, where they are used to authenticate against remote sites that store data to be processed. After successful authentication, these data are streamed to the job using the XROOTD protocol [25,26], on a per-file basis. The credentials are also used after a successful job completion to authenticate against the site to store output data. Output files are copied to the storage site (local or remote) via the SRM protocol.

III. HANDLING EVICTION AND RUNTIME BOTTLENECKS

In section II we detailed the numerous steps required to get CMSSW jobs successfully running in an opportunistic environment. The next major set of challenges includes the study and handling of CMSSW job failures, evictions, and runtime resource bottlenecks on the voluntary heterogeneous resources.

Figure 2 illustrates the physical infrastructure configuration used to test the workflow execution and validity of all software modifications. In addition to the previous software workflow, the grid software is packaged along with the job so that each worker node can access data as needed for its respective analysis.

A. Workflow Execution on Opportunistic Versus Dedicated Computational Resources.

To perform a comparison of the opportunistic computing framework and the dedicated resources, runtime analysis was performed on workflows run in each framework. Jobs submitted were typical examples of CMS analysis, i.e. processing of Monte Carlo-simulated events, approximately 2500 each, and very similar to one another in nature. Approximately 3100 jobs were submitted in each scenario, all to a single Condor queue.

Figures 3, 4, and 5 show the overhead, processing, and total runtime respectively for jobs submitted to the dedicated resources and for jobs submitted to opportunistic ones (split by three different Parrot cache mechanisms). Table 1 contains the average times in each scenario, showing serious performance degradation in the case of the single locked cache during overhead operations (which involve a number of I/O transactions). While results using separate caches are comparable to those with a single shared cache, the latter eliminates redundant network traffic and is therefore preferred. Average processing times (Fig 4) are comparable as Parrot is not involved in that stage.

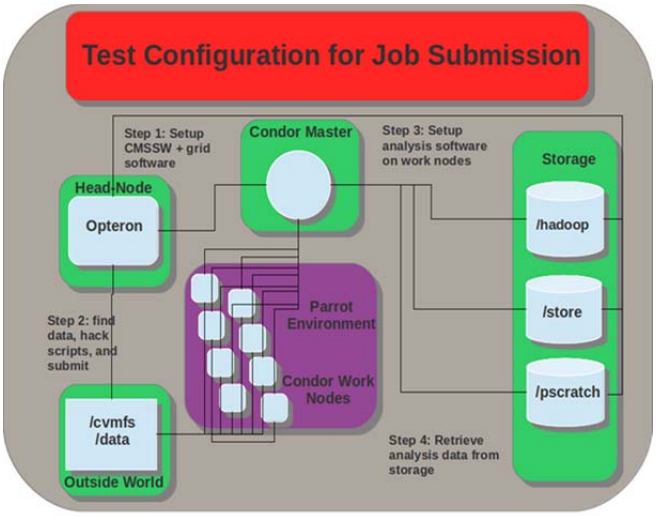

Figure 2. Testbed infrastructure

Understanding eviction rates and runtime of analysis jobs is crucial to finding the ideal number of events to be processed per job. While overhead time on opportunistic resources cannot be improved by the user and may only be reducible to a certain extent, the number of events to be processed by each job can be easily adjusted and optimized. A balance between large and small event multiplicity is required in order to reduce the number of evictions while minimizing network traffic inherent to a large number of job submissions.

Typical job failure rates are on the order of 1%, mostly due to transient xrootd server problems when serving the input data files. Automatic resubmission upon identification of such transient issues is being studied currently.

	Overhead	Processing	Total
Dedicated	4.5	110	110
Opportunistic, single locked cache	320	89	410
Opportunistic, separate caches	8.5	120	130
Opportunistic, single shared cache	4.2	120	130

Table 1. Average job times (in minutes)

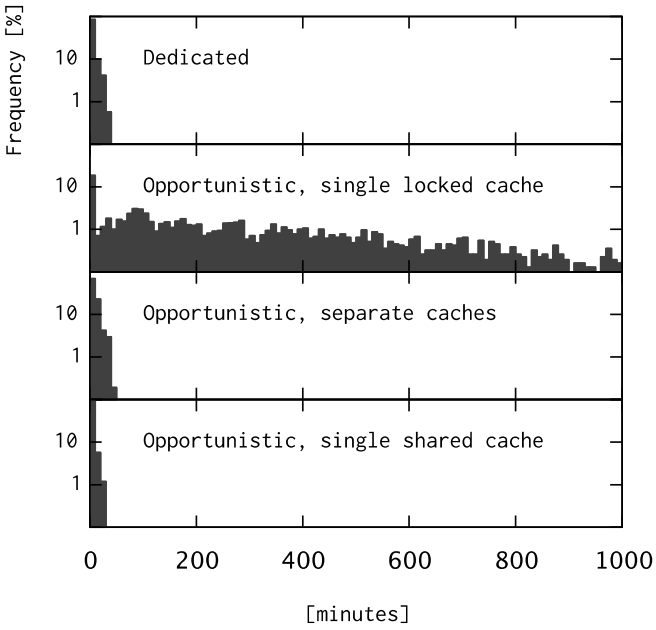

Figure 3. Job overhead time

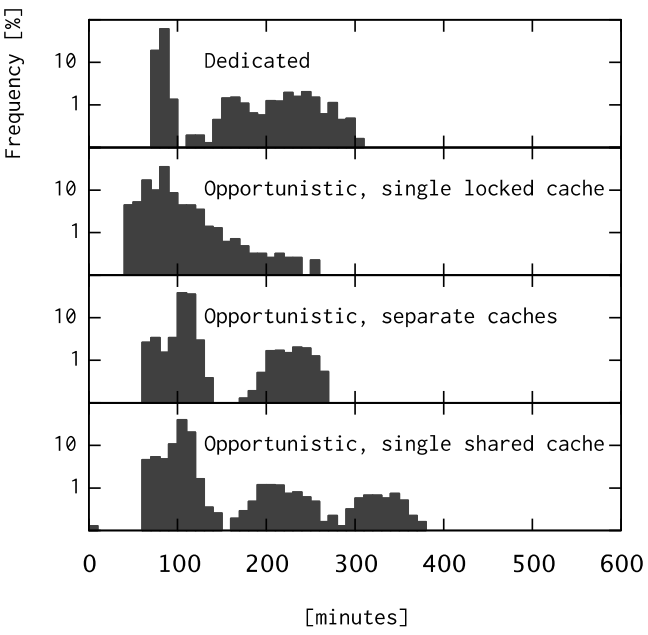

Figure 4. Job processing time

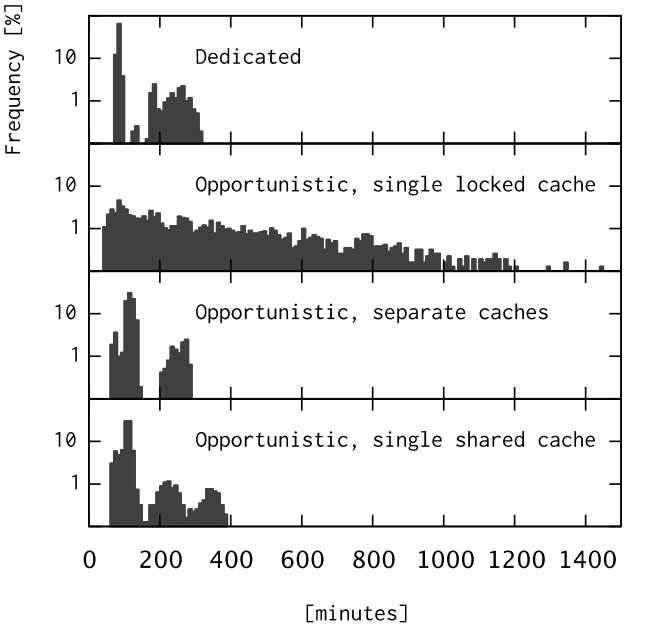

Figure 5. Total job time (aggregate of overhead and processing times)

IV. Conclusion

The paper describes a framework to enable CMS software to run on opportunistic computing resources. It addresses the fundamental obstacle of providing access to complex CMS software stack despite the lack of root privileges for the opportunistic job. The solution leverages the existing Parrot and CVMFS software. Initial tests demonstrate the feasibility of this approach, but work remains to reduce the overhead for starting jobs and improve the reliability.

V. Future Work

The next major obstacle to address to achieve optimal use of opportunistic resources involves addressing situations in which an opportunistic job is preempted. Currently, the CMS software lacks the ability to migrate a running job to another available resource upon preemption or to restart the job from a partially completed output. To address this, several improvements can be pursued.

A. Writing Results Back to Local and Remote Repositories

CMS jobs currently store their output locally during execution and only write the output to the final storage repository at the end of successful execution. For preempted jobs, this output is lost and the job must be restarted from the beginning. Alternatively, if the jobs either periodically copied or continuously streamed output to temporary storage buffer, then partial progress could be retained, allowing interrupted jobs to continue from the point of interruption. Addressing this issue would also involve developing a new mechanism for collecting temporary outputs from jobs and copying them to their final repository.

B. Writing and Restarting from Incremental Outputs

The CMS software currently writes out only complete output files. Interruption of the program execution currently results in a corrupted output file. However, there is no fundamental limitation preventing incremental output from being saved. Due to the nature of the CMS workflow, sufficient information could be extracted from a partially completed output file and the initial job parameters to allow the job to be continued from the point of execution, thus allowing a strategy for mitigating the loss of efficiency from preemption.

C. Fully Automated Complete Checkpointing

Managing and retaining partial output files from preempted jobs would provide an option for addressing preemption in an opportunistic environment. However, the solution would be specific to the CMS application and would have to be re-implemented for any other software. A more general solution involving, for example, virtualization and automated virtual machine migration or checkpointing when in the face of preemption would provide an alternative solution. A fully automated approach would require substantially more effort: in addition to the software framework described in this paper, enhancements would need to be made to the workflow management and batch scheduling tools to incorporate the migration or checkpointing capabilities. This investment of additional effort would be worthwhile if it leads to a fully general solution to the problem.

ACKNOWLEDGMENT

We would like to thank the National Science Foundation (NSF), who sponsored Dillon Skeehan's contributions through a Research Experience for Undergraduates program hosted at the ND Center for Research Computing (NSF #1063084). The Notre Dame HEP CMS research team is supported through NSF grants #0955765 and #1312842. The ND Collaborative Computing Lab's work integrating Parrot and CVMFS was partially supported by NSF grant #1148330. Brian Bockleman (University of Nebraska, Lincoln) and Dan Bradley (University of Wisconsin, Madison) provided valuable technical support relative to OSG/Condor/Xrootd performance tuning and troubleshooting. The authors would also like to thank ND Center for Research Computing members Rich Sudlow and Serguei Fedorov for technical support during the development and testing phases.

REFERENCES

[1] R. Adolphi et al. The CMS experiment at the CERN LHC. JINST, 0803:S08004, 2008.

[2] Lyndon Evans and Philip Bryant. LHC machine. Journal of Instrumentation, 3:S08001–S08001, August 2008.

[3] G.L. Bayatian et al. CMS technical design report, volume II: Physics performance. J.Phys.G, G34:995–1579, 2007.

[4] S. Dasu et al. CMS. The TriDAS project. Technical design report, vol. 1: The trigger systems. 2000.

[5] P. Sphicas et al. CMS: The TriDAS project. Technical design report, Vol. 2: Data acquisition and high-level trigger. 2002.

[6] D. Bonacorsi. From commissioning to collisions: Preparations and execution of CMS computing. Nucl.Phys.Proc.Suppl., 215:79–81, 2011.

[7] Worldwide LHC Computing Grid (WLCG). http://lcg.web.cern.ch/lcg/public/.

[8] Geoff Brumfiel. High-energy physics: Down the petabyte highway. Nature, 469:282–283, 2011

[9] Open science grid (OSG). https://www.opensciencegrid.org.

[10] Andrew Melo and Paul Sheldon. Integrating Amazon EC2 with the CMS production frame-work. J.Phys.Conf.Ser., 368:012007, 2012.

[11] Amazon elastic compute cloud (EC2). http://aws.amazon.com/ec2/.

[12] Any data, anytime, anywhere (NSF awards 1104549, 1104447, and 1104664). http://osg-docdb.opensciencegrid.org/cgi-bin/ShowDocument?docid=1025.

[13] Matt Mutka and Miron Livny. Profiling workstations' available capacity for remote execution. Performance, pages 529–544, December 1987.

[14] Marvin Solomon and Michael Litzkow. Supporting checkpointing and process migration outside the Unix kernel. In USENIX Winter Technical Conference, pages 283–290, 1992.

[15] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In Symposium on Operating Systems Design and Implementation, 2002.

[16] Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In Proceedings of SciDAC, 2006.

[17] Parrot website. http://www.nd.edu/ ~ ccl/software/parrot/.

[18] Douglas Thain and Miron Livny. Parrot: Transparent User-Level Middleware for Data Intensive Computing. In Workshop on Adaptive Grid Middleware at PACT, 2003.

[19] Douglas Thain and Miron Livny. Parrot: An Application Environment for Data-Intensive Computing. Scalable Computing: Practice and Experience, 6(3):9–18, 2005

[20] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. Concurrency and Computation: Practice and Experience, 17(2-4):323–356, 2005.

[21] Gabrielle Compostella, Simone Pagan Griso, Donatella Lucchesi, Igor Sfiligoi, and Douglas Thain. CDF Software Distribution on the Grid using Parrot. In Computing in High Energy Physics, 2009.

[22] P Buncic et al.; 2010 J. Phys.: Conf. Ser. 219 042003, CernVM – a virtual software appliance for LHC applications

[23] J Blomer et al.; "CernVM-FS: delivering scientific software to globally distributed computing resources"

[24] J Blomer et al.; 2012 J. Phys.: Conf. Ser. 396 052013 "Status and future perspective of CernVM-FS"

[25] Hanushevsky, A., Furano, F., & Dorigo, A. (2004). The next generation ROOT file server.

[26] Dorigo, A., Elmer, P., Furano, F., & Hanushevsky, A. (2005). XROOTD-A Highly scalable architecture for data access. WSEAS Transactions on Computers, 1(4.3).