Improving Reproducibility of Computational Experiments using Linux Containers

Saqib Hussain and Mark Stillwell

School of Engineering
Cranfield University, Cranfield, UK
{s.m.hussain,m.stillwell}@cranfield.ac.uk

Abstract. In recent years, shifts in funding priorities and a culture of "publish or perish" in academia have led to an unprecedented increase in the number of scientific articles submitted for publication. While the goals of hastening scientific progress and maximizing the rate-of-return for research investment are laudable, the pressure on individual scientists can incentivize publication of research results that are not of the highest quality, and a number of high-profile retractions have lead to a crisis in public confidence in scientific research. In order to improve the quality of results and restore public faith in science, researchers within a number of fields have taken on initiatives to improve the repeatability and reproducibility of experiments and analysis, particularly when these are computationally driven. Within this contest, we propose a solution that uses Docker, a version-controlled container management system created by and for the web development community, to improve the reproducibility of parallel scientific experiments. The Linux-kernel based containers managed by Docker provide many of the capabilities offered by virtual machine technology in terms of operating system isolation and resource management, but with minimal overheads and near-native performance.

Keywords: reproducibility,containers,docker

1 Introduction

Reproducibility is almost compulsory in most scientific fields to aid the furthering and verification of research. Scientists and programmers from an extensive range of professions including but not limited to neuroscience and computer science are appealing for a better, more refined paradigm by which new research or findings are submitted into the scientific community[1]. However, computing environments are heterogeneous, and researchers make use of them in a variety of conflicting and poorly documented ways. Consequently, enabling reproducibility and reusability in the general case is a formidable task that encapsulates a host of difficulties[2]. This paper summarises what, in the experience of the authors, are the best techniques to employ when attempting to achieve efficient and comprehensive reproducibility in parallel computation experiments, and proposes a Linux container based solution which incorporates these techniques, addressing many of the common problems with reproducibility.

2 Reproducibility and Repeatability

Reproducibility and repeatability are the two factors that must be accounted for when trying to replicate experimental results. The former can be defined as the degree of consistency of results under changing conditions, the latter as the consistency over multiple executions under static conditions[2]. The demand for both reproducibility and repeatability is seen consistently in scientific environments where research needs to be verified by secondary parties, particularly when papers are submitted for peer review and claim to have made advances into a particular subject area. Most scientific papers contain only "plots and graphs but not the data", and according to Zaino this "makes it impossible to extend the analysis or connect that information with other analysis"[3]. Another factor driving the demand for reproducibility is the possibility of data falsification. The year 2012 saw an approximate tenfold increase in the number of scientific articles that were retracted due to fraud over previous years[4]. One possible explanation could be due to the increased pressure to produce results: while the demand for scientists has remained steady, the number of individuals that are in competition for employment has increased[5].

Many of the implementation details that are required for reproducibility, such as "parameter values and functional invocation sequences", are often omitted from papers[1], [6]. Software being used in an experiment, be it for computation, input/output, or creating graphics, will have its own dependencies and libraries which are constantly being updated and optimised[7]. Therefore, results will inevitably vary if an attempt is made to replicate results using an updated or outdated version of the same software. Software packages may have several interdependencies which aren't accounted for by the programmer because they are automated and run in the background[8]. Attempting to replicate an experiment's results without having these dependencies installed could result in significant deviation from the expected output. It is incredibly difficult to account for every single detail within the working environment, and yet for an experiment to be truly reproducible it has to be captured in its entirety.

Another factor that may inhibit reusability is simply that scientists do not sufficiently document their code[8]. They may leave software unfinished, opting to instead hack their code until it functions well enough for them to produce results, which is not good practice for software reuse[9]. If the original experiment has a poor degree of consistency over several executions, it will be difficult to achieve good correlation with the original results. This is due to most scientists not being programmers but rather having adopted programming in order to complete their research. They are unfamiliar with common software programming practices, but rather concerned with obtaining usable results. Some of these issues cannot be handled entirely by software, instead scientists should adhere to the already well defined software programming standards, particularly those which focus on testing[1]. Consequently, doing so may "reduce the sensitivity of the results to the precise details of the code and environment", this will in turn make the code more reliable and more scientifically valuable[6], and the use of well known and tested libraries will aid the robustness of code[6], [9].

3 Related Work

Researchers as far back as 30 years have recognised the need for reproducibility and have taken their own initiatives to improve the state of their practice[1]. A few examples include journals such as *Science and the Proceedings of the National Academy of Sciences* (PNAS) making data and code publication mandatory, the creation of data sharing platforms such as Machine Learning Open Source Software and researchers from "bioinformatics, applied mathematics, computer science, law, and many other fields" have already attempted to produce specialised requirements concerning "code sharing in support of reproducible computational research"[1]. A group of neuroscience researchers from the UK have employed the aforementioned techniques of data standardisation to collate findings from numerous studies from different universities in order to assist and promote the furthering of their research[10]. Such efforts, while laudable, have had limited impact across scientific disciplines because of differing processes and requirements.

4 Provenance Tracking Systems

Davison lists many examples of provenance tracking systems, each of which can be separated into three distinct categories: literate programming, workflow management systems and environment capture [11].

4.1 Literate Programming

Literate programming (LP) almost reverses the traditional programming paradigm of code littered with comments, and instead involves writing in natural language fused with traditional code; minimal impositions from the compiler allow code to be written logically from a human perspective and compilation results in the production of both human readable documentation and executable code. LP examples include Sweave and TeXmacs. Similar to LP are "interactive notebooks" such as IPython and Mathematica which place any textual and graphical outputs in line with documentation. LP allows the developers to more effectively communicate what they did with in the code instead of describing what the aimed to do with the code, hopefully helping the reproduction and furthering of their research. Complex algorithms and functions can also more easily be conveyed to and understood by an external reader. Although these systems aid reproducibility by "inextricably binding the code and results together", they are not as applicable in parallel or computationally intensive environments [11]. Trying to do so is typically more difficult as LP tools have poor interfaces and require the user to learn enigmatic languages.

4.2 Workflow Management Systems

Workflow management systems (WMS) are an infrastructure to link many small but unambiguous and distinct tools in a pipeline, providing an overall orchestration of the desired task. They are particularly useful in scientific environments

where it is commonplace that much of the code for sub-systems has already been written by someone else. WMS are usually GUIs and therefore easy to use when components already exist, however, should the need arise to write a new component it can prove problematic as intricate details of WMS architecture are required [11]. Although WMS can aid data manipulation by taking manual control away from users, much of this is still scripted by the users themselves. Many WMS are lacking in availability across computing platforms; workflow pattern, tool and change management support. WMSy can be difficult to install and often require the support of system administrators because they frameworks have complex daemons that run for lengthy durations.

4.3 Environment Capture

The most prominent method for accomplishing reproducibility is through the use of virtualisation to abstract computer resources. This is typically done with virtual machines (VMs) and a hypervisor which dictates access to resources. VMs provide an easy solution to encapsulate the entire system upon which computation is run; any "data, software, dependencies, notes, logs, scripts, and more" can be duplicated by creating an exact replica of a machine, known as an image. Images can then be cited in papers and easily be pulled from repositories to extend research or reproduce results. [8]. Virtual machines significantly aid portability and with minimal disruption to the researcher's workflow and impositions on language unlike LP or WMS.

However, they suffer from hypervisor overheads and lengthy boot times. These are particularly undesirable in a parallel computationally intensive environment where bottlenecks and resource provisioning are important factors, and where users are consistently running one experiment after another; an inordinate amount of time would be spent booting VMs. Furthermore, final VM images are often very large in size and may not be easy to version control, requiring large images pushes every time a change is made to the working environment. There is also the possibility that any results produced under virtualisation will be sensitive to the particular configuration environment configuration, reducing the code's reproducibility.

There also exist image build systems with the ability to generate experiment environments in a reproducible manner. These can be separated into two categories, bootstrap (BoxGrinder, Ubuntu VMBuilder) or full installation (veewee, Oz). Kameleon is a bootstrap image build tool which uses recipes written in YAML to provide high level descriptions of the environments to be created, supporting numerous hypervisors, custom, raw, and even Docker images, it can be considered a comprehensive example. Inbuilt is the ability to cache all packages used in the original build into a .tar file and any subsequent builds, if containing a '–from_cache' argument, will ensure an exact replica of the original environment is reproduced. It can be argued that systems such as these are becoming obsolete given that popular virtualisation management engines such as Vagrant or Docker provide a built in method for archiving images indefinitely, most scientists are more than likely to have high speed internet and thus the ability to

download a one off image in little time, making the need to build-on-demand almost redundant.

Build systems do not address all the issues with reproducibility in a parallel computational environment, in particular, resource usage and boot times. These issues have been recognised by many researchers, including a group in Switzerland who have developed a type of lightweight VM known as a virtual appliance (VA). Their VA is a minimalistic image containing only a kernel and CernVM-FS with any additional files being downloaded on demand [12]. Their attempt to mitigate the drawbacks of VMs results in faster boot times and lower resource consumption, however the solution still does not address hypervisor overhead and it would require each compute node in a cluster which is running a VA to have permanent access to the internet.

Another solution is CDE (Code, Data, Environment), a Linux environment capture tool which can track and record all dependencies during the execution of a program run on the command line, including any spawned sub-processes. These dependencies are automatically packaged up and can be distributed with research to replicate results as any subsequent runs would use the original packaged dependencies. CDE's use is restricted to modern Linux distributions and any replication of experiments should be carried out on the same major kernel version which was used in the original experiment, although, there are methods to work around this. As with VMs, CDE also suffers from sensitivity to environment configuration. CDE mitigates many of the drawbacks of virtualisation, including the large VM image size but the major advantage is its relative unobtrusiveness to existing workflows and simplicity because it does not require the use of hypervisor, but rather works alongside the executed program [13].

5 Containers

Containers are a streamlined alternative to virtual machines. From the viewpoint of the user they provide a similar type of resource abstraction but are much more lightweight. VMs often require hardware virtualisation technologies to run more efficiently and a hypervisor to dictate access to system resources. Containers on the other hand, are an OS level virtualisation technique and share the host kernel, they can therefore make system calls directly and avoid hypervisor overheads. The OS kernel provides isolation and security for different user spaces, similar to a "chroot jail". There are many different implementations of containerisation, the most prominent being LXC and OpenVZ, both of which are open source and available only on Linux. The only real containerisation solution for a Windows OS is that of Parallels Virtuozzo Containers, however this requires a proprietary licence which is not desirable for scientific, reproducible use and only the Linux based solutions will be explored.

LXC utilises control groups for resource quotas and access restriction to devices; sandboxing kernel mechanisms such as seccomp to isolate processes from system resources, thus avoiding malicious system calls; and namespaces for isolation to secure containers. As of version 1.0 LXC containers are by default

unprivileged, therefore, should a user manage to access a host resource they will not have root access to the host system [14]. With the recent major updates to LXC, the security and resource isolation between it an OpenVZ are similar, however, OpenVZ provides additional checkpointing, live migration and limiting kernel memory usage functionality. The disadvantage to OpenVZ is that it requires patched kernels to run all its features, while LXC is available by default with the Linux kernel.

Since the shared kernel is used to dictate access to resources, there is no traditional virtualisation of an entire OS, only the separation of a userspace, thus the overheads of a hypervisor are mitigated, resulting in faster deployment, shutdown and near native CPU, memory and network performance. From this we can also infer that VMs are inherently more secure as they emulate every aspect of the OS, despite this, security requirements may not be as imperative on a local compute clusters which are only accessed by trusted users. Unlike VMs, containers can be provisioned in a matter of milliseconds, without being memory intensive; dozens of isolated containers all running at near native performance could be launched before they began to consume as much memory as a virtual machine of a similar calibre. This is accomplished by use of a "copy-on-write" mechanism which shares files until a modification occurs, meaning little disk activity is actually required to start a container. When a file is changed, only then is a copy is made for a given container. It is also worth noting that while Linux kernels provide services and user space, they do not break user space but instead implement the libc interface for applications to make calls [15]. Therefore, any updates to the kernel will not break containers, similarly, any updates to the libc library won't affect containers either as this is not shared with the host. Regrettably, the nature of containerisation does not allow for different kernel OSs to be virtualised within each other, this however, is not a problem in the situation this paper describes as each container would run the same OS, it is not improbably that the conatiner and host OSs are the same.

6 Docker

Docker is a Linux based engine which manages the deployment of these containers. Using Docker means no longer having issues with missing/differing dependencies or libraries; compatibility; deployment; or hypervisor overhead. Docker containers can be launched in virtually any Linux environment, even in the cloud, and are able to execute anything that can be executed on the native machine. Docker provides the ability to use numerous execution drivers, including the aforementioned LXC and OpenVZ. However, the default driver for Docker is a custom library, libcontainer, written entirely in Go, which can access the Linux kernel directly without additional dependencies. This reduces situations with missing or incompatible libraries. It also enables additional features such as checkpointing and there are even ongoing efforts to port the library to ASP.NET. Docker enables portability by bundling the environment into a single

entity which may be deployed and reproduced anywhere, this is not possible using just an execution driver.

Docker containers are built from base images available via public or private repositories. These images can then be set up manually with scripts or configuration management tools, there is also an option to use a Dockerfile to automate the building of an image which automatically commits changes until the image is created. Docker is entirely open source, as such, it has a lot of public interest with over 500 contributors. Several extensions have been developed to aid its use in a variety of situations, among them are drivers and plug-ins to support its use and orchestration in popular cloud environments such as OpenStack and a toolkit, libswarm, for orchestration across multiple hosts and infrastructure providers. It even boasts the ability to bind mount the entire host file system within containers.

Docker harnesses existing techniques employed by source control technologies to only store the differences between containers when a change is made and committed, whereas VMs only have the ability to create large snapshots of the entire operating system. This allows layers upon layers to be built on the base image until the desired container is composed. If a secondary user already has the base image present on their local Docker machine, they need only to pull the committed changes, thus the whole sharing process becomes much quicker and more manageable. Should a scientist update their production environment to run more tests, downloading the changes would be a matter of seconds for pulling "diffs" instead of the entire image. Furthermore, this allows the production environment to be version controlled, providing historical tracking and the ability to revert to an earlier layer if bugs are introduced into the system, users can even query the history of an image using the "docker history" command. The Dockerfile itself can also be version controlled. This functionality aids reusability by allowing users to take an image from any point from within layers and continue building a different configuration. Docker uses aufs (another union file system) to implement layers which does bring drawbacks. Currently, aufs supports a maximum of 127 layers, this number has already been increased after users began to hit the previous limit. One way to circumvent this is to export a layered image and import it back with just one layer but the history will be forfeit. Furthermore, since layers are read only, removal of a file followed by a commit does not actually remove said file from an earlier layer, but rather adds a new layer to hide the file. So it is easy to see how layered file systems could easily become very complex, thus the use of a Dockerfile is recommended because it takes advantage of cache.

Docker then, implements features from build systems and source control technology, it also has the small file size advantages of CDE, requiring only "diffs" to be pushed to the repository when changes are made. It is not, however, without its flaws; Docker suffers from similar reproducibility verses repeatability issues as many of the methods already discussed. But it can aid in lessening some repeatability issues with its packaging mechanisms for portability allowing users to deploy code on local machines, clusters, inside VMs and in the cloud.

7 Preliminary Results

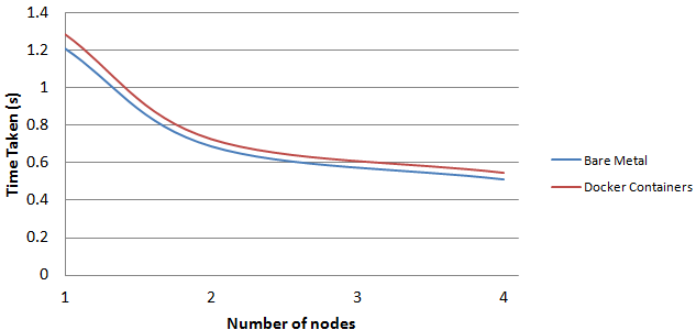

Fig. 1: Graph to show CPU utilisation of booting 15 VMs and containers.

The drawbacks of VM hypervisors are already well known but as a proof of concept, a series of experiments were carried out in order to analyse how similar the performance between Docker containers and bare metal would be under parallel computationally intensive conditions. An MPI matrix multiplication program was written in C along with a C++ application to generate a matrix of random floating point numbers. The nodes were running on a Ubuntu 12.04 OS, Intel i7 870 processor with 16GB of DDR3 RAM. The code used for matrix generation and running MPI, along with the raw data can be cloned from the following repository: https://github.com/SaqibHussain/REPPAR14.git.

Figure 1 shows the results of running matrix multiplication on bare metal and inside containers for a 512x512 matrix. Each experiment was run 20 times and averages were taken. The container execution time was almost on par with that of bare metal. The difference here between the two is an average of 6.5%. This is low but the execution times were predicted to be far closer, moreover, 6.5% is a difference which could be noticed in a production envrionment if experiments run for hours or even days . Therefore, the same experiment was now carried out for a larger problem domain to analyse how this affected the outcome.

Figure 2 shows the differences in execution time when the matrix size was doubled. Immediately noticeable is the fact that as more nodes are used, the execution times begin to coincide. When using just one node for both bare metal and containers the difference between the results is approximately 20% but when 4 nodes are reached this difference has dropped to 0.6%. Containers also showed an average relative standard deviation of execution times of only 2.4% while bare metal was slightly higher at 5.6%, therefore these results can be considered somewhat accurate and a useful insight into the usefulness of containers.

To confirm the results, the matrix size was double once more and the experiments rerun. Figure 3 depicts these results. A similar type of curve is seen between the it and the previous experiment. Differences between the execution times of bare metal and containers were now even lower, dropped to an average

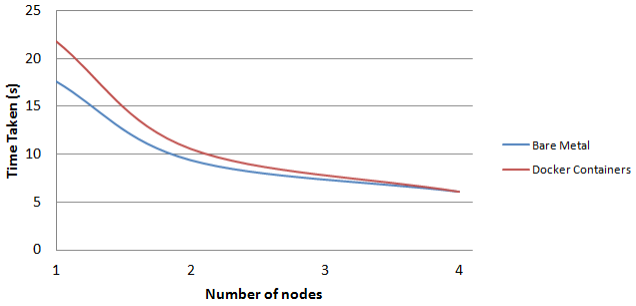

Fig. 2: Graph to show CPU utilisation of booting 15 VMs and containers.

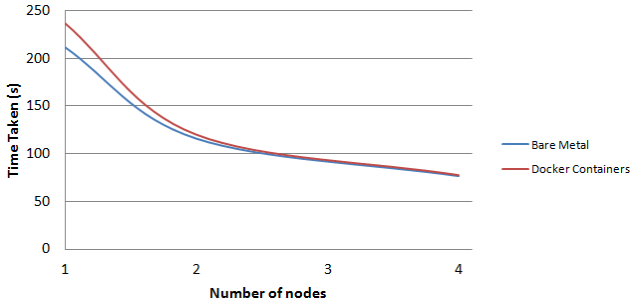

Fig. 3: Graph to show CPU utilisation of booting 15 VMs and containers.

difference of only 5.8%. This time, the relative standard deviation of results was a very low 3.15% for bare metal and 2.2% for containers. Another conclusion which can be drawn from these experiments is that the problem domain may have to be over a certain size for any performance gains to be seen when using containers over VMs.

Figure 4 and figure 5 are taken from a presentation given at DockerCon14 which explored the performance of Docker containers verses VMs [16]. The author used KVM 1.0 as a hypervisor as is it is widely acknowledged for its efficiency and speed, Docker 0.10.0 and OpenStack with the nova-docker virt driver to enable the use of Docker as a hypervisor. The compute nodes housed two Intel Xeon-Westmere 5620-Quadcore CPUs clocked at 2.4GHz, 16GB of DDR3 RAM operating on a Ubuntu 12.04 64bit OS. The experiment consisted of booting 15 VMs and 15 containers simultaneously, waiting 5 minutes and destroying them. Evident from figure 4 is that containers launch quicker and with less impact on the CPU. Once booted and idle, containers have virtually no additional effect on the system until they are destroyed, whereas VMs still show flutters of activity. VMs would then slightly reduce the computing power available to the user.

Figure 5 depicts the RAM usage. Here we ascertain just how much more RAM friendly containers are over a VMs. In fact, each VM used approximately 6x more RAM than the equivalent container. In a real world scenario, this would diminish the amount of memory available for computation. It may also be argued that boot times of bare metal are higher than VMs, however, in a cluster environment,

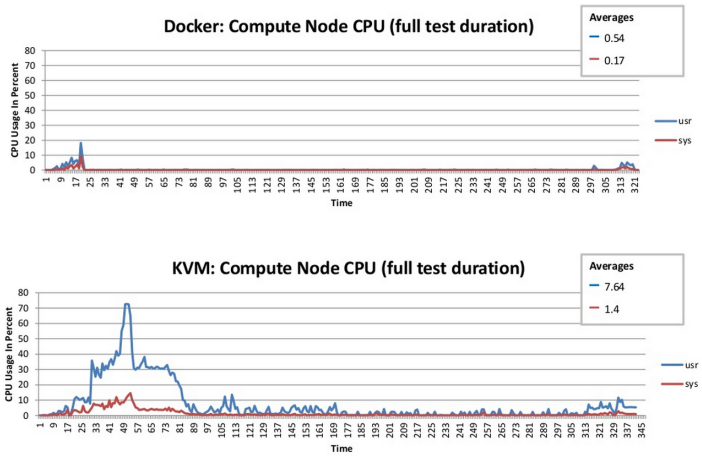

Fig. 4: Graph to show CPU utilisation of booting 15 VMs and containers [16].

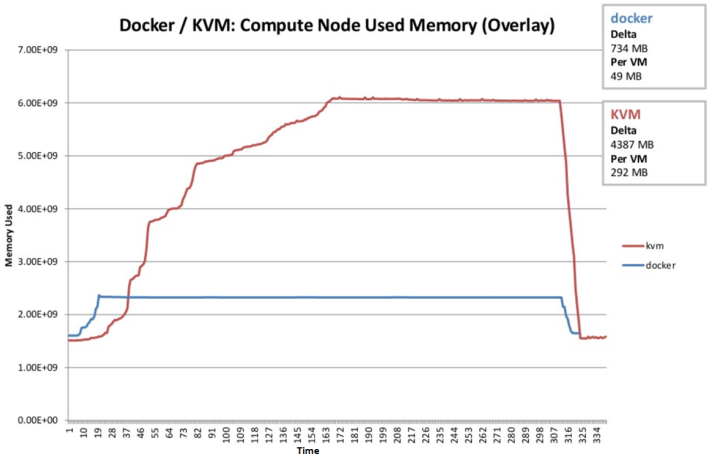

Fig. 5: Graph to show RAM utilisation of booting 15 VMs and containers [16].

which is the focus of this study, it is commonplace that a cluster remain on-line once running, whereas the VMs launched on a cluster will be destroyed once no longer in use.

8 Planned Development

Docker itself is still relatively new and under heavy development and care should be taken not to use it within a production environment but it is developing quickly and if security is not a huge concern, it could well be used sparingly. There still needs to be a unified and more user friendly system which uses Docker as a back end to provide reproducibility to the scientific community. Such a system could take textual configuration files from which it could automatically build Dockerfiles, provision resources and run the scientific code provided. This would mean that researchers could effectively supply a single code repository link with their research which anyone could clone and use to run their own copy of the Docker based system and reproduce the results. The most important future development is that of Dockers own container engine, which in time will aid running containers cross platform and further aid the reproducibility of computational code. Additional experimentation is still required as those in this paper are not exhaustive, notably in a more parallel environment with more communication between nodes. This would allow assessment of the networking capabilities of container and whether execution times for containers begin to drop at a certain threshold.

9 Conclusion

Although containers don't alleviate all the problems associated with VMs, it is evident that containers are a much more efficient method of virtualisation, even if they don't provide the same degree of flexibility. In a computation environment, performance is paramount and the restriction to Linux when using containers should not pose much of a concern considering many of today's intensive computation environments already operate on some version of Linux. There are some additional security concerns with using containers if not configured properly or run with privileges, but it is unlikely that these be of major concern in a scientific environment. The container based paradigm that has been discussed would mean minimal disruption to the researchers, they can work on their local systems running containerised experiments which provide version controlling, bare metal performance and easy distribution, all without fear of incompatibility or missing dependencies. Furthermore, Docker will help ease some of the pressure on scientists by allowing them to reuse others' systems easily and efficiently, save time and money by producing results slightly quicker, and containers are denser If adopted by the scientific community, it would significantly aid the furthering and verification of published results.

References

[1] R. LeViqije, I. Mitchell, and V. Stodden, "Reproducible research for scientific computing: tools and strategies for changing culture," *Computing in Science and Engineering*, vol. 14, no. 4, 2012.

[2] X. Xia, Q. Zhou, and J. Zhu, "Evaluation for repeatability and reproducibility of information poor process," in *Proceedings of the 2010 International Conference on Image Analysis and Signal Processing*, 2010, pp. 528–531.

[3] J. Zaino, *CHAIN-REDS project enhances semantic search and extends reproducibility of scientific data*, `http://semanticweb.com/chain-reds-project-enhances-semantic-search-extends-reproducibility-scientific-data_b42239`, 2014.

[4] F. Fang, R. Steen, and A. Casadevall, "Misconduct accounts for the majority of retracted scientific publications," in *Proceedings of the National Academy of Sciences of the United States of America*, 2012.

[5] H. Dbouk, *"publish or perish" and the plague of scientific misconduct*, `http://ascb.org/ascbpost/index.php/compass-points/item/170-publish-or-perish-and-the-plague-of-scientific-misconduct`, 2013.

[6] A. Davison, "Automated capture of experiment context for easier reproducibility in computational research," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 48–56,

[7] D. Milajevs, *Software in your reproducible research*, `http://qmcs.io/software-in-your-reproducible-research.htm`, 2014.

[8] B. Howe, "Virtual appliances, cloud computing, and reproducible research," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 36–41, 2012.

[9] Software Sustainability Institute, *Cw14 discussion session 3*, `http://www.software.ac.uk/collaborations-workshop-2014-cw14-software-your-reproducible-research/discussion-sesssion-3`, 2014.

[10] B. Howe, "Reproducibility, virtual appliances, and cloud computing," in *Implementing Reproducible Research*, Chapman and Hall, 2013.

[11] A. Davison, *Workflows for reproducible research in computational neuroscience*, `http://rrcns.readthedocs.org/en/latest/`, 2013.

[12] J. Blomer, D. Berzano, P. Buncic, I. Charalampidis, G. Ganis, G. Lestaris, R. Meusel, and V. Nicolaou, "Micro-cernvm: slashing the cost of building and deploying virtual machines," *CoRR*, vol. abs/1311.2426, 2013.

[13] P. Guo, *Cde v1.0 documentation*, `http://www.pgbovine.net/cde/manual/index.html`, 2012.

[14] S. Graber, *Lxc 1.0: security features*, `https://www.stgraber.org/2014/01/01/lxc-1-0-security-features/`, 2014.

[15] A. Brouwer, *The linux kernel*, `http://www.win.tue.nl/~aeb/linux/lk/lk.html`, 2003.

[16] R. Boden, `http://blog.docker.com/2014/06/dockercon-video-performance-characteristics-of-vms-vs-docker-containers-by-boden-russel-ibm/`, 2014.