Reproducible and User-Controlled Software Environments in HPC with Guix

Ludovic Courtès, Ricardo Wurmus

Reproducible and User-Controlled Software Environments in HPC with Guix

Ludovic Courtès[1] and Ricardo Wurmus[2]

[1] Inria, Bordeaux, France
[2] Max Delbrück Center for Molecular Medicine, Berlin, Germany

Abstract. Support teams of high-performance computing (HPC) systems often find themselves between a rock and a hard place: on one hand, they understandably administrate these large systems in a conservative way, but on the other hand, they try to satisfy their users by deploying up-to-date tool chains as well as libraries and scientific software. HPC system users often have no guarantee that they will be able to reproduce results at a later point in time, even on the same system—software may have been upgraded, removed, or recompiled under their feet, and they have little hope of being able to reproduce the same software environment elsewhere. We present GNU Guix and the functional package management paradigm and show how it can improve reproducibility and sharing among researchers with representative use cases.

1 Introduction

HPC system administration has to satisfy two seemingly contradictory demands: on one hand administrators seek stability, which leads to a conservative approach to software management, and on the other hand users demand recent tool chains and huge scientific software stacks. In addition, users often need different versions and different variants of a given software package. To satisfy both, support teams end up playing the role of "distribution maintainers": they build and install tool chains, libraries, and scientific software packages manually—multiple variants thereof—and make them available *via* "environment modules"[4], which allows users to pick the specific packages they want.

Unfortunately, software is often built and installed in an *ad hoc* fashion, leaving users little hope of redeploying the same software environment on another system. Worse, support teams occasionally have to remove installed software or to upgrade it in place, which means that users may eventually find themselves unable to reproduce their software environment, *even on the same system.*

Recently-developed tools such as EasyBuild [7] and Spack [5] address part of the problem by automating package builds, supporting non-root users, and adding facilities to create package variants. However, these tools fall short when it comes to build reproducibility. First, build processes can trivially refer to tools or libraries already installed on the system. Second, the *ad hoc* naming conventions they rely on to identify builds fail to capture the directed acyclic graph (DAG) of dependencies that led to this particular build.

GNU Guix is a general-purpose package manager that implements the functional package management paradigm pioneered by Nix [2,3]. Many of its properties and features make it attractive in a multi-user HPC context: per-user profiles, transactional upgrades and roll-backs, and, more importantly, a controlled build environment to maximize reproducibility.

Section 2 details our motivations. Section 3 describes the functional package management paradigm, its implementation in Guix, its impact on reproducibility, and how it can be applied to HPC systems. Section 4 gives concrete use cases where Guix empowers users while guaranteeing reproducibility and sharing, while Section 5 discusses limitations and remaining challenges. Finally, Section 6 compares to related work, and Section 7 concludes.

2 Rationale

Recent work on reproducible research insufficiently takes software environment reproducibility into account. For example, the approach for verifiable computational results described in [6] focuses on workflows and conventions but does not mention the difficulty of reproducing full software environments. Likewise, the new Replicated Computational Results (RCR) initiative of the ACM Transactions on Mathematical Software acknowledges the importance of reproducible results, but does not adequately address the issue of software environments, which is a prerequisite. The authors of [13] propose a methodology for reproducible research experiments in HPC. To address the software-environment reproducibility problem they propose two unsatisfying approaches: one is to write down the version numbers of the dependencies being used, which is insufficient, and the other is to save and reuse full virtual machines (VMs), which poses a real challenge for performance and makes verifiability impractical—peers would have to download large images and would be unable to combine them with their own software environment.

Yet, common practices on HPC systems hinder reproducibility. For understandable stability reasons, HPC systems often run old GNU/Linux distributions that are rarely updated. Thus, packages provided by the distribution are largely dismissed. Instead support teams install packages from third-party repositories—but then they clobber the global `/usr` prefix, which sysadmins may want to keep under control, or install them from source by themselves and make them available through environment modules [4]. Modules allow users to choose different versions or variants of the packages they use without interfering with each other. However, when installed software is updated in place or removed, users suddenly find themselves unable to reproduce the software environment they were using. Given these practices, reproducing the exact same software environment on a *different* HPC system seems out of reach. It is nonetheless a very important property: It would allow users to assess the impact of the hardware on the software's performance—something that is very valuable in particular for developers of run-time systems such as StarPU [1]—and it would allow other researchers to reproduce experiments on their system.

Essentially, by deploying software and environment modules, HPC support teams find themselves duplicating the work of GNU/Linux distributions, but why is that? Historical package managers such as APT and RPM suffer from several limitations. First, package binaries that every user installs, such as `.deb` files, are actually built on the package maintainer's machine, and details about the host may leak into the binary that is uploaded—a shortcoming that is now being addressed (see Section 6.)

Second, while it is in theory possible for a user to define their own variant of a package, as is often needed in HPC, this is often difficult in practice. Users of RPM-based systems, for example, may be able to customize a `.spec` file to build a custom, relocatable RPM package, but only the administrator can install the package alongside its dependencies and register it in the central `yumdb` package database. The lower-level `rpm` tool can use a separate package registry, which could be useful for unprivileged users; however RPM package databases cannot be composed, so users would need to manually track down and register the complete graph of dependencies, which is impractical at best.

Third, these tools implement an *imperative* and *stateful* package management model [3]. It is imperative in the sense that it modifies the set of available packages in place. For example, switching to an alternative MPI implementation, or upgrading the OpenMP run-time library means that suddenly all the installed applications and libraries start using them. It is stateful in the sense that the system state after a package management operation depends on its previous state. Namely, the system state at a given point in time is the result of the series of installation and upgrade operations that have been made over time, and there may be no way to reproduce the exact same state elsewhere. These properties are a serious hindrance to reproducibility.

3 Functional Package Management

Functional paradigm. Functional package management is a discipline that transcribes the functional programming paradigm to software deployment: build and installation processes are viewed as pure functions in the mathematical sense—whose result depends exclusively on the inputs—, and their result is a value—that is, an immutable directory. Since build and installation processes are pure functions, their results can effectively be "cached" on disk. Likewise, two independent runs of a given build process for a given set of inputs should return the same value—*i.e.*, bit-identical files. This approach was first described and implemented in the Nix package manager [3]. Guix reuses low-level mechanisms from Nix to implement the same paradigm, but offers a unified interface for package definitions and their implementations, all embedded in a single programming language [2].

An obvious challenge is the implementation of this paradigm: How can build and install processes be viewed as pure? To obtain that property, Nix and Guix ensure tight control over the build environment. In both cases, build processes are started by a privileged daemon, which always runs them in "containers"

as implemented by the kernel Linux; that is, they run in a chroot environment, under a dedicated user ID, with a well-defined set of environment variables, with separate name spaces for PIDs, inter-process communication (IPC), networking, and so on. The chroot environment contains only the directories corresponding to the explicitly declared inputs. This ensures that the build process cannot inadvertently end up using tools or libraries that it is not supposed to use. The separate name spaces ensure that the build process cannot communicate with the outside world. Although it is not perfect as we will see in Section 5, this technique gives us confidence that build processes can indeed be viewed as pure functions, with reproducible results.

Each build process produces one or more files in directories stored in a common place called *the store*, typically the `/gnu/store` directory. Each entry in `/gnu/store` has a name that includes a hash of *all the inputs* of the build process that led to it. By "all the inputs", we really mean all of them: This includes of course compilers and libraries, including the C library, but also build scripts and environment variable values. This is recursive: The compiler's own directory name is a hash of the tools and libraries used to build, and so on, up to a set of pre-built binaries used for bootstrapping purposes—which can in turn be rebuilt using Guix [2]. Thus, for each package that is built, the system has access to the *complete DAG* of dependencies used to build it.

```
(define openmpi
  (package
    (name "openmpi")
    (version "1.8.1")
    (source (origin
              (method url-fetch)
              (uri (string-append
                    "http://www.open-mpi.org/software/ompi/v"
                    (version-major+minor version)
                    "/downloads/openmpi-" version ".tar.bz2"))
              (sha256
               (base32
                "13z1q69f3qwmmhpglarfjminfy2yw4rfqr9jydjk5507q3mjf50p"))))
    (build-system gnu-build-system)
    (inputs `(("hwloc" ,hwloc)
              ("gfortran" ,gfortran-4.8)
              ("pkg-config" ,pkg-config)))
    (arguments '(#:configure-flags `("--enable-oshmem")))
    (home-page "http://www.open-mpi.org")
    (synopsis "MPI-2 implementation")
    (description "This is an MPI-2 implementation etc.")
    (license bsd-2)))
```

Fig. 1. Guix package recipe of Open MPI.

Package recipes in Guix are written in a domain-specific language (DSL) embedded in the Scheme programming language. Figure 1 shows, as an example, the recipe to build the Open MPI library. The `package` form evaluates to a

package object, which is just a "regular" Scheme value; the `define` form defines the `openmpi` variable to hold that value.

```
;; Query the direct and indirect inputs of Open MPI.
;; Each input is represented by a label/package tuple.
(map (match-lambda
       ((label package)
        (package-full-name package)))
     (package-transitive-inputs openmpi))
```

... yields:

```
("hwloc-1.9" "pciutils-3.2.0" "numactl-2.0.9" "gfortran-4.8.4" "pkg-config-0.28")
```

Fig. 2. Querying the dependencies of a package object.

Line 14 specifies that the package is to be built according to the GNU standards—*i.e.*, the well-known `./configure && make && make install` sequence (similarly, Guix defines `cmake-build-system`, and so on.) The `inputs` field on line 15 specifies the direct dependencies of the package. The field refers to the `hwloc`, `gfortran-4.8`, and `pkg-config` variables, which are also bound to package objects (their definition is not shown here.) It would be inconvenient to specify all the standard inputs, such as Make, GCC, Binutils so these are implicit here; they are provided by `gnu-build-system`, which is responsible for compiling package objects to an intermediate representation. Since we are manipulating "normal" Scheme objects, we can use the API of Guix to query those package objects, as illustrated with the code in Figure 2, which queries the name and version of the direct and indirect dependencies of our package[3].

With that definition in place, running `guix build openmpi` returns the directory name `/gnu/store/z0cx27hlk1x57bfks3v6qjw728yfv0i2-openmpi-1.8.1`. If that directory did not already exist, the daemon spawns the build process in its isolated environment with write access to this directory. Of course users never have to type these long `/gnu/store` file names. They can install packages in their *profile* using the `guix package` command, which essentially creates symbolic links to the selected `/gnu/store` items. By default, the tree of symbolic links is rooted at `~/.guix-profile`, but users can also create independent profiles in arbitrary places of the file system. For instance, a user may choose to have GCC and Open MPI in the default profile, and to populate another profile with Clang and MPICH2.

It is then a matter of defining the search paths for the compiler, linker, and other tools *via* environment variables. Fortunately, Guix keeps track of that and the `guix package --search-paths` command returns all the necessary environment variable definitions in Bourne shell syntax. For example, when both the GCC tool chain and Open MPI are installed, the command returns definitions for the `PATH`, `CPATH`, and `LIBRARY_PATH` environment variables, and these definitions can be passed to the `eval` shell built-in command.

[3] This document is an "active paper" written in Skribilo, a Scheme-based authoring tool, which allows us to use Guix and run this code from the document.

4 Use Cases

We explore practical use cases where Guix improves experimentation reproducibility for a user of a given system, supports the deployment of complex software stacks, allows a software environment to be replicated on another system, and finally allows fine customization of the software environment.

4.1 Usage Patterns on an HPC Cluster

One of the key features of Guix and Nix is that they securely permit unprivileged users to install packages in the store [3]. To build a package, the `guix` commands connect to the build daemon, which then performs the build (if needed) on their behalf, in the isolated environment. When two users build the exact same package, both end up using the exact same `/gnu/store` file name, and storage is shared. If a user tries to build, say, a malicious version of the C library, then the other users on the system will not use it, simply because they cannot guess its `/gnu/store` file name—unless they themselves explicitly build the very same modified C library.

Guix is deployed at the Max Delbrück Center for Molecular Medicine (MDC), Berlin, where the store is shared among 250 cluster nodes and an increasing number of user workstations. It is now gradually replacing other methods of software distribution, such as statically linked binaries on group network shares, relocatable RPMs installed into group prefixes, one-off builds on the cluster, and user-built software installed in home directories. The researchers use tens of bioinformatics tools as well as frameworks such as Biopython, NumPy, SciPy, and SymPy. The functional packaging approach proved particularly useful in the ongoing efforts to move dozens of users and their custom software environments from an older cluster running Ubuntu to a new cluster running a version of CentOS, because software packaged with Guix does not depend on any of the host system's libraries and thus can be used on very different systems without any changes to the packages. Research groups now have a shared profile for common applications, whereas individual users can manage their own profiles for custom software, legacy versions of bioinformatics tools to reproduce published results, bleeding-edge tool chains, or even for complete workflows.

```
;; This file can be passed to 'guix package --manifest'.
(use-modules (gnu packages base) (gnu packages gcc)
             (my-openmpi))

(packages->manifest
 (list glibc-utf8-locales gnu-make gcc-toolchain openmpi))
```

Fig. 3. Declaring the set of packages to be installed in a profile.

Guix supports two ways to manage a profile. The first one is to make transactions that add, upgrade, or remove packages in the profile: `guix package --install openmpi --remove mpich2` installs Open MPI and removes MPICH2 in a single transaction that can be rolled back. The second approach is to *declare*

the desired contents of the profile and make that effective: the user writes in a file a code snippet that lists the requested packages (see Figure 3) and then runs `guix package --manifest=my-packages.scm`.

This declarative profile management makes it easy to replicate a profile, but it is symbolic: It uses whatever package objects the variables are bound to (`gnu-make`, `gcc-toolchain`, etc.), but these variables are typically defined in the `(gnu packages ...)` modules that Guix comes with. Thus the precise packages being installed depend on the version of Guix that is available. Specifying the Git commit of Guix in addition to the declaration in Figure 3 is all it takes to reproduce the exact same `/gnu/store` items.

Another approach to achieve bit-identical reproduction of a user's profile is by saving the contents of its transitive closure using `guix archive --export`. The resulting archive can be transferred to another system and restored at any point in time using `guix archive --import`. This should significantly facilitate experimentation and sharing among peers.

4.2 Customizing Packages

Our colleagues at Inria in the HiePACS and Runtime teams develop a complete linear algebra software stack going from sparse solvers such as PaStiX and dense solvers such as Chameleon, to run-time support libraries and compiler extensions such as StarPU [1] and hwloc. While developers of simulations want to be able to deploy the whole stack, developers of solvers only need their project's dependencies, possibly several variants thereof. For instance, developers of Chameleon may want to test their software against several versions of StarPU, or against variants of StarPU built with different compile-time options. Finally, developers of the lower-level layers, such as StarPU, may want to test the effect of changes they make on higher-level layers.

```
(define starpu-1.2rc              ;release candidate
  (package (inherit starpu)
    (version "1.2.0rc2")
    (source (origin
              (method url-fetch)
              (uri (string-append "http://starpu.gforge.inria.fr/files/"
                                  "starpu-" version ".tar.gz"))
              (sha256
               (base32
                "0qgb6yrh3k745grjj14gc2vl6a99m0ljcsisfzcwyhg89vdpx42v"))))))

(define starpu-with-simgrid
  (package (inherit starpu)
    (name "starpu-with-simgrid")  ;name shown the user interface
    (inputs `(("simgrid" ,simgrid)
              ,@(package-inputs starpu)))))
```

Fig. 4. Defining variants of the default recipe for StarPU.

This use case leads to two requirements: that users be able to customize and non-ambiguously specify a package DAG, and that they be able to reproduce any variant of their package DAG. Guix allows them to define variants; the code for these variants can be stored in a repository of their own and made visible to the `guix` commands by defining the `GUIX_PACKAGE_PATH` environment variable. Figure 4 shows an example of such package variants: based on the pre-existing `starpu` variable, the first variant defines a package for a new StarPU release candidate, simply by changing its `source` field, while the second variant adds the optional dependency on the SimGrid simulator—a variant useful to scheduling practitioners, but not necessarily to solver developers.

These StarPU package definitions are obviously useful to users of StarPU: They can install them with `guix package -i starpu` and similar commands. But they are also useful to StarPU developers: They can enter a "pristine" development environment corresponding to the dependencies given in the recipe by running `guix environment starpu --pure`. This command spawns a shell where the usual `PATH`, `CPATH` etc. environment variables are redefined to refer precisely to the inputs specified in the recipe. This amounts to creating a profile on the fly, containing only the tools and libraries necessary when developing StarPU. This is notably useful when dealing with build systems that support optional dependencies.

```
(define (make-chameleon name starpu)
  (package
    (name name)
    ;; [other fields omitted]
    (inputs `(("starpu" ,starpu)
              ("blas" ,atlas) ("lapack" ,lapack)
              ("gfortran" ,gfortran-4.8)
              ("python" ,python-2)))))

(define chameleon
  (make-chameleon "chameleon" starpu))
(define chameleon/starpu-simgrid
  (make-chameleon "chameleon-simgrid" starpu-with-simgrid))
```

Fig. 5. Defining a function that returns a package object for the Chameleon solver.

Now that we have several StarPU variants, we want to allow direct and indirect users to select the variant that they want. A simple way to do that is to write, say, a function that takes a `starpu` parameter and returns a package that uses it as its input as show in Figure 5. To allow users to refer to one or the other variant at the command line, we use different values for the `name` field.

This approach is reasonable when there is a small number of variants, but it does not scale to more complex DAGs. As an example, StarPU can be built with MPI support, in which case Chameleon also needs to be explicitly linked against the same MPI implementation. Users may want to replace all occurrences of an MPI implementation in the DAG rooted at Chameleon with a different implementation. One way to do that is by writing a function that recursively adjusts the package labeled `"mpi"` in the `inputs` field of packages. This is what

```
(define (override-input original label replacement)
  ;; Return a variant of ORIGINAL where inputs corresponding
  ;; to LABEL are replaced by REPLACEMENT, recursively.
  (package (inherit original)
    (inputs (map (lambda (input)
                   (match input
                     ((input-label package)
                      (if (string=? input-label label)
                          `(,label ,replacement)
                          `(,label
                            ,(override-input package
                                             label replacement))))))
                 (package-inputs original)))))

(define chameleon/mpich
  (override-input chameleon "mpi" mpich2))
```

Fig. 6. Rewriting a package DAG by changing a specific input.

the `override-input` function in Figure 6 does. No matter how complex the transformations are, a package object unambiguously represents a reproducible build process.

5 Limitations and Challenges

Privileged daemon. Nix and Guix address many of the reproducibility issues encountered in package deployment, and Guix provides APIs that can facilitate the development of package variants as is useful in HPC. Yet, to our knowledge, neither Guix nor Nix are widely deployed on HPC systems. An obvious reason that limits adoption is the requirement to have the build daemon run with root privileges—without which it would be unable to use the Linux kernel container facilities that allow it to isolate build processes and maximize build reproducibility. System administrators are wary of installing privileged daemons, and so HPC system users trade reproducibility for practical approaches.

Cluster setup. All the `guix` commands are actually clients of the daemon. In a typical cluster setup, system administrators may want to run a single daemon on one specific node and to share `/gnu/store` among all the nodes. At the time of writing, Guix does not yet allow communication with a remote daemon. For this reason, Guix users at the MDC are required to manage their profiles from a specific node; other nodes can use the profiles, but not modify them. Allowing the `guix` commands to communicate with a remote daemon will address this issue.

Additionally, compute nodes typically lack access to the Internet. However, the daemon needs to be able to download source code tarballs or pre-built binaries from external servers. Thus, the daemon must run on a node with Internet access, which could be contrary to the policy on some clusters.

Non-determinism. Despite the use of isolated containers to run build processes, there are still a few sources of non-determinism that build systems of packages might use and that can impede reproducibility. In particular, details about the operating system kernel and the hardware being used can "leak" to build processes. For example, the kernel Linux provides system calls such as `uname` and interfaces such as `/proc/cpuinfo` that leak information about the host; independent builds on different hosts could lead to different results if this information is used. Likewise, the `cpuid` instruction leaks hardware details.

Fortunately, few software packages depend on this information. Yet, the proportion of packages depending on it is higher in the HPC world. A notable example is the ATLAS linear algebra system, which fine-tunes itself based on details about the CPU micro-architecture. Similarly, profile-guided optimization (PGO), where the compiler optimizes code based on a profile gathered in a previous run, undermines reproducibility. Running build processes in full-blown VMs would address some of these issues, but with a potentially significant impact on build performance, and possibly preventing important optimization techniques in the HPC context.

Proprietary software. GNU Guix does not provide proprietary software packages. Unfortunately, proprietary software is still relatively common in HPC, be it linear algebra libraries or GPU support. Yet, we see it as a strength more than a limitation. Often, these "black boxes" inherently limit reproducibility—how is one going to reproduce a software environment without permission to run the software in the first place? What if the software depends on the ability to "call home" to function at all? More importantly, we view reproducible software environments and reproducible science as a tool towards improved and shared knowledge; developers who deny the freedom to study and modify their code work against this goal.

6 Related Work

Reproducible builds. Reproducible software environments have only recently become an active research area. One of the earliest pieces of work in this area is the Vesta software configuration system [10]. Vesta provides a DSL that allows users to describe build operations, similar to Nix [3]. More recently, projects such as Debian's Reproducible, Fedora's Mock, or Gitian have intended to improve reproducibility and verifiability of mainstream package distributions. Google's recent Bazel build tool relies on container facilities provided by the kernel Linux and provides another DSL to describe build operations.

Reproducibility can be achieved with heavyweight approaches such as full operating system deployments [11], VM deployments [8], and full-system container-based deployments [12]. In addition to being resource-hungry, these approaches are coarse-grain and do not compose: if two different VM or Docker images provide useful features or packages, the user has to make a binary choice and cannot combine the features or packages they offer. A side issue is security: it

was recently reported that many official Docker images are plagued with serious unfixed security vulnerabilities [9].

HPC package management. In the HPC community, efforts have focused primarily on the automation of software deployment and the ability for users to customize their build environment independently of each other. The latter has been achieved by "environment modules", a simple but efficient tool set that is still widely used today [4]. Build and deployment automation is more recent with the development of specialized package management tools such as EasyBuild [7] and Spack [5].

Both EasyBuild and Spack have the advantage of being installable by unprivileged users since they do not rely on privileged components, unlike Guix and Nix. The downside is that they cannot use the kernel's container facilities, which seriously hinders build reproducibility. When used in the user's home directories, each user may end up rebuilding the same compiler, libraries, etc., which can be costly in terms of CPU, bandwidth, and disk usage. Conversely, Nix and Guix support safe sharing of builds.

EasyBuild aims to support multiple package variants, such as packages built with different compilers, or linked against different MPI implementations. To achieve that, it relies on directory naming conventions; for instance, `OpenMPI/-1.7.3-GCC-4.8.2` contains packages built with the specified MPI implementation and compiler. Such conventions fail to capture the full complexity of the DAG and configuration space. For instance, the convention arbitrarily omits the C library, linker, or configuration flags being used.

EasyBuild is tightly integrated with environment modules [4], which are familiar to most users of HPC systems. While modules provide users with flexible environments, they implement an imperative, stateful paradigm: Users run a sequence of `module load` and `module unload` commands that *alter* the current environment. This can make it much harder to reason about and reproduce an environment, as opposed to the declarative approaches implemented by `guix package --manifest` and `guix environment`.

Like EasyBuild and similarly to Guix, Spack implements build recipes as first-class objects in a general-purpose language, Python, which facilitates customization and the creation of package variants. In addition, Spack provides a rich command-line interface that allows users to express variants similar to those discussed in Section 4.2. This appears to be very convenient for common cases, although there are limits to the expressivity and readability of such a compact syntax.

7 Conclusion

Functional package managers provide the foundations for reproducible software environments, while still allowing fine-grain software composition and not imposing high disk and RAM costs. Today, GNU Guix comes with 1,922 packages, including many of the common HPC tools and libraries as well as around 40 bioinformatics packages. It is deployed on the clusters of the MDC Berlin, and

being discussed as one of the packaging options by the Open Bioinformatics Foundation, a non-profit for the biological research community. We hope to see more HPC deployments of Guix in the foreseeable future.

GNU Guix benefits from contributions by about 20 people each month. It is the foundation of the Guix System Distribution, a standalone, reproducible GNU/Linux distribution.

Acknowledgments

We would like to thank Florent Pruvost, Emmanuel Agullo, and Andreas Enge at Inria and Eric Bavier at Cray Inc. for insightful discussions and comments on an earlier draft. We are grateful to the Guix contributors who keep improving the system.

References

1. C. Augonnet, S. Thibault, R. Namyst, P-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proc. of the 15th Int. Euro-Par Conf.*, Springer International Publishing, August 2009.
2. L. Courtès. Functional Package Management with Guix. In *European Lisp Symp.*, June 2013.
3. E. Dolstra, M. d. Jonge, E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proc. of the 18th Large Installation System Administration Conf. (LISA '04)*, pp. 79–92, USENIX, November 2004.
4. J. L. Furlani. Providing a Flexible User Environment. In *Proc. of the 5th Large Installation System Administration (LISA V)*, pp. 141–152, June 1991.
5. T. Gamblin. Spack Web Site. 2015. *http://scalability-llnl.github.io/spack/* .
6. M. Gavish, D. Donoho. A Universal Identifier for Computational Results. In *Procedia Computer Science*, 4(0) , 2011, pp. 637–647.
7. M. Geimer, K. Hoste, R. McLay. Modern Scientific Software Management Using EasyBuild and Lmod. In *Proc. of the 1st Workshop on HPC User Support Tools (HUST'14)*, pp. 41–51, IEEE Press, 2014.
8. P. V. Gorp, S. Mazanek. SHARE: a web portal for creating and sharing executable research papers. In *Procedia Computer Science*, 4(0) , 2011, pp. 589–597.
9. J. Gummaraju, T. Desikan, Y. Turner. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. May 2015. *http://www.banyanops.com/blog/analyzing-docker-hub/* .
10. A. Heydon, R. Levin, Y. Yu. Caching Function Calls Using Precise Dependencies. In *Proc. of the ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation*, PLDI '00, pp. 311–320, ACM, 2000.
11. E. Jeanvoine, L. Sarzyniec, L. Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning. In *USENIX ;login:*, 38(1) , February 2013, pp. 38–44.

12. C. Kniep.
Reproducability of CAE-computations through Immutable Application Containers. . *http://doc.qnib.org/HPCAC2015.pdf*.
13. L. Stanisic, A. Legrand.
Effective Reproducible Research with Org-Mode and Git. In *Euro-Par 2014: Parallel Processing Workshops*, Springer International Publishing, pp. 475–486, 2014.