Rebuilding Debian using Distributed Computing

Lucas Nussbaum
Université Claude Bernard Lyon 1
RESO team, LIP, ENS Lyon
Email: lucas.nussbaum@ens-lyon.fr

***Abstract*—Doing Quality Assurance work on Debian, a Linux distribution with more than 12000 packages, requires an impressive amount of computing power, which is usually not available for its developers. In this article, we report on the development of an infrastructure to run quality-assurance tasks on Debian using the Grid'5000 experimental platform. In particular, we focus on the problem of rebuilding all packages in Debian from source. We describe the details of this task, and the infrastructure we developed, with scalability and robustness in mind. The results we obtained are then presented, and we discuss possible improvements and lessons we learnt in the process, which might be useful in the context of other large-scale experiments.**

I. Introduction

The Debian project builds an operating system – Debian GNU/Linux, usually simply called "Debian" – by gathering a very large amount of free software, and turning them into *packages* that can easily be installed by the user. It has been very successful since its creation in 1993, and serves as the basis for other Linux distributions, like Ubuntu. Debian is developed by more than 1000 volunteers, spread across the world and communicating over the Internet. As such, it is often regarded as one of the most important volunteer-based and distributed organizations.

Debian is well renowned for its robustness and its stability, and is known as a good choice for a server's operating system. This level of quality is mainly achieved by a great attention to details by the developers who maintain its 12000+ source packages. But some Quality Assurance (QA) tasks require computing power in addition to manpower, and, since 2006, we have used distributed computing on a Grid infrastructure to find defects in Debian.

We worked on two classes of problems. First, we focused on testing the installation and the removal of packages. Debian has more than 22000 binary packages (source packages are built to generate binary packages, which are installed by users). Each package's meta-data can express relationships with other packages (depends on another package, suggests the installation of another package, conflicts with another package). While the *installability* (whether a package can be installed) of a package can be determined statically [1], other problems might arise during installation, which are harder to detect without actually installing the package: a package could contain the same file as another package without explicitly conflicting with that other package, a script executed after the installation of the package might fail because of a missing dependency, a programming error, or a change in the behaviour of another package since the developer did the initial packaging work. A tool, *piuparts* [2], is available in Debian to perform tests on the installation, upgrade, and removal of packages. Running *piuparts* on all packages is an embarrassingly parallel problem: one could test each one of Debian's 22000 packages in parallel, and the packages that take the longest do not take more than half an hour. Since 2006, we ran several test campaigns using piuparts, and reported about 250 bugs, most of them considered *critical*. However, the result of those installation tests is relatively stable: while bugs might not be easy to find, new bugs are relatively rare, and running those tests does not need to be done on a frequent basis.

The second class of problems we looked at is more challenging. We examined the *buildability* of packages (whether packages can be built successfully). Since Debian contains only free software, the source code for each package is available, and for various reasons, it is important to ensure that it is possible, from a source package, to build the corresponding binary packages. Firstly, during the lifetime of a package, it might be necessary to change something in the source code – to correct a mistake, like a bug or a security problem. In that case, it will be necessary to rebuild the corresponding binary packages after the change has been made. Secondly, for legal reasons: the source code for programs covered by the GNU General Public License must be made available by Debian, and one could argue that shipping a source code that does not allow building the corresponding packages would be a license violation.

Debian packages can be built automatically: all source packages provide a simple interface, based on a Makefile named `debian/rules`, that hides the specifics of each program's build system (use of *Automake* or *CMake*, language-specific tools like Python's *distutils* or Perl's *Makefile.pl*): each package can be built by calling one of the targets of `debian/rules`, or by using a wrapper like `dpkg-buildpackage`, which would use `debian/rules` itself. Unfortunately, packages often become impossible to build, for different reasons. A package needed to build another package (called a *build-dependency*) could be removed from Debian, or modified in a way that makes its *reverse dependencies* impossible to build: a compiler could become more strict by rejecting previously-accepted constructs, the API of a library could change in an incompatible way, the parameters of another program could be modified.

By rebuilding all packages in Debian, we not only ensure that Debian is *self-contained* (that all Debian packages can be

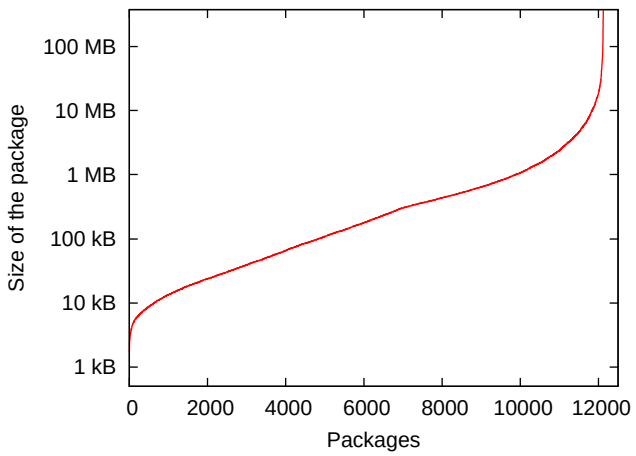

Fig. 1. Distribution of the size of source packages: most packages are quite small.

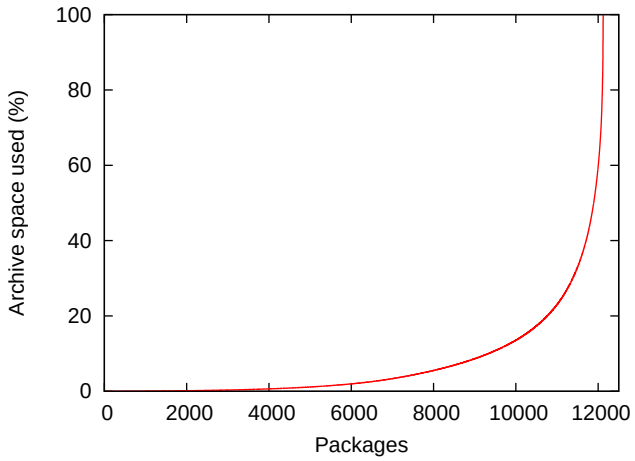

Fig. 2. Distribution of the space used in the archive by packages: the few biggest packages account for a very large part of the archive's size.

rebuilt from Debian), we also stress-test the whole toolchain – the packages that are used to build other packages. This second role is at least as important as the first one: most packages in Debian lack a test suite, and using them to rebuild other packages often serves as some kind of automated test suite.

In the remainder of this paper, we report on the execution of rebuilds of the Debian archive using distributed computing, by providing feedback on improvements implemented since [3]. In section II, we give some information on our workload. In section III, we present Grid'5000, which is the platform that we used to perform those rebuilds, and the specific infrastructure we developed to be able to run those tasks efficiently on Grid'5000. We then present the results we obtained in section IV and discuss possible optimizations in section V, before concluding in section VI.

II. Workload analysis

The implementation decisions that we will have to make depend greatly on the workload we would like to process with our application. In this section, we describe the characteristics of the Debian source packages set. We use Debian 5.0 'Lenny', released in February 2009, on the i386 architecture, as the basis for our study. Previous releases of Debian do not differ significantly from those results, and it can be expected that future releases will not fundamentally differ either, except by increasing the number of packages.

Debian lenny is composed of 12123 source packages, of which about 12000 can be built on the i386 architecture (Debian supports 12 different architectures, and some packages provide functionality that is specific to some architectures, due to specific hardware, for example). The total size of the source packages is 16.3 GB (compressed using gzip).

Figure 1 shows the distribution of the size of packages. A lot of the packages are relatively small (44% smaller than 128 kB, 82% smaller than 1 MB, 99% smaller than 20 MB). However, a few packages are much larger (`openoffice.org` - 346 MB, `nexuiz-data` - 377 MB). As one can see on figure 2, the few largest packages are responsible for most of the archive's size.

Building source packages into binary packages requires several steps. First, a clean build environment is needed. It consists of a minimal *chroot* in which are installed the Debian packages that are always expected to be present when building. This includes the *GCC* compiler, `binutils`, and Debian-specific tools. In our setup, this *chroot* is stored as a tar archive, taking 73 MB compressed (200 MB uncompressed).

Each source package can also specify other packages that must be installed before building. For example, a Fortran program will require the Fortran compiler to be installed, as this package is not expected to be installed by default in the build environment. The installation of those *build-dependencies* can take a significant amount of time, as some packages require the installation of a lot of them: `openoffice.org` requires 485 additional build-dependencies, and `linphone` requires 392 of them. Of the 22311 binary packages in Debian lenny, 5723 are build-dependencies of other packages. Also, those packages have to be fetched from a local mirror before they are installed.

III. Software Infrastructure

A. Grid'5000

Grid'5000 [4], [5] is an experimental platform for research on large-scale parallel and distributed systems. Grid'5000 is being developed under the INRIA ALADDIN development action, with support from CNRS, RENATER, and several universities as well as other funding bodies.

Grid'5000 consists of about 2000 compute nodes, split in a dozen of clusters, located in 9 locations in France. Those 9 sites are connected with a dedicated 10 Gbps backbone (see figure 3).

Grid'5000 aims at providing a reconfigurable, controllable, and monitorable experimental platform. As such, once compute nodes have been reserved, it is possible to deploy one's

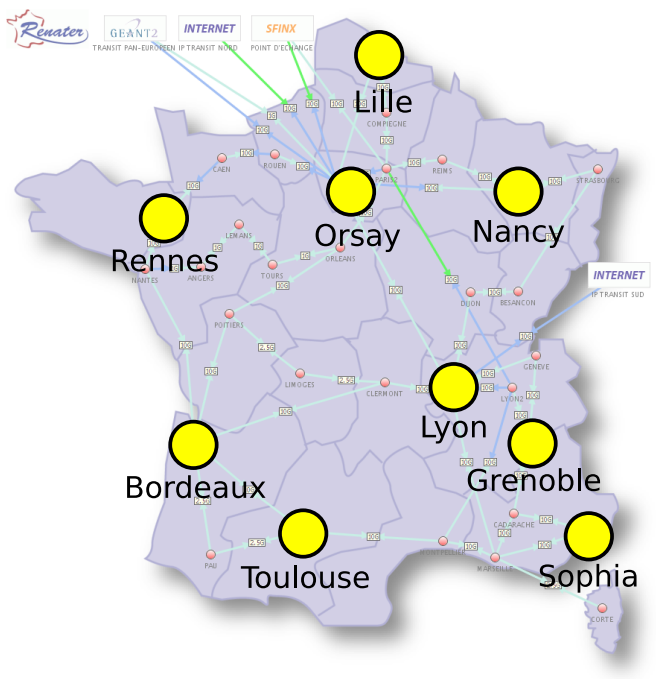

Fig. 3. Grid'5000 sites on top of the Renater 5 network infrastructure. 10 Gbps links connect the various Grid'5000 sites together.

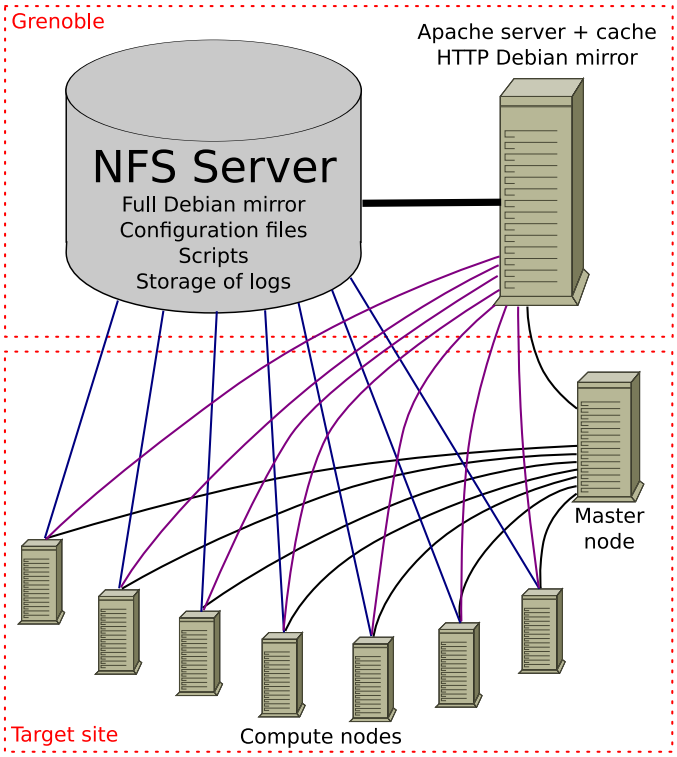

Fig. 4. Software infrastructure

own work environment using Kadeploy [6]. This allows installing specific software (including kernel) and to get administrator (*root*) access on the nodes.

B. Infrastructure for Debian rebuilds

To rebuild all Debian packages efficiently on Grid'5000, we developed our own software infrastructure (figure 4). We had the following goals in mind:

- most of the infrastructure should be deployed dynamically during the rebuilds, using Kadeploy;
- it should be robust. The rebuilds are supposed to be run unattended, and should not fail;
- it should be scalable. As we will see in section IV, we will be able to run the rebuild on 50 to 100 compute nodes at the same time.

Our infrastructure is composed of two parts: a static part, located in the Grenoble Grid'5000 site, and a dynamic part that can be deployed on any Grid'5000 site, depending on where resources are available.

The static part of the infrastructure consists of an NFS server hosting all the necessary data:

- A full Debian mirror internal to Grid'5000;
- The scripts and configuration files, as well as some data files needed by the compute nodes;
- The logs generated by the builds.

An Apache web server is also configured next to the NFS server, and serves the Debian mirror over HTTP. This proved to be more efficient than distributing the packages directly using NFS, and also provides an opportunity for caching, thus reducing the load on the NFS server.

The deployment of the dynamic part of the infrastructure is done in several steps.

1) Nodes are reserved using the OAR Batch Scheduler;
2) The reserved nodes are deployed using Kadeploy. A standard environment, available on all Grid'5000 clusters, is used. The deployment is managed by Katapult, to allow the failed nodes to be re-deployed if necessary. This takes 3 to 5 minutes;
3) From the frontend, a script is executed (over SSH) on one of the deployed nodes (the *master node*). Basic configuration is done on the node (like the mounting of a shared NFS directory);
4) From the frontend, a script located on the shared NFS directory is executed on the *master node* to continue the configuration of the nodes;
5) From the frontend, a last script located on the shared NFS directory is executed. This script will control the rest of the operations;
6) The script running on the *master node* executes the same process on the other nodes: it first copies a script to the nodes, executes it to mount the shared NFS directory, then run another script to finish the preparation of the nodes;
7) When all the nodes are properly prepared, the *master node* starts scheduling and executing tasks on them. At the beginning of each build, the *chroot* is uncompressed from a tar archive, to ensure that the build environment is always clean. The build log is stored locally until the end of the build, and is then copied to the NFS directory.

8) After all the tasks have been executed, all the nodes are given back to the batch scheduler. It is possible that, at the end of the rebuild, there are no remaining tasks to run on some nodes, which could therefore be freed. But the OAR batch scheduler does not allow releasing some nodes earlier than others, which can lead to a waste of resources in this case.

While NFS is not efficient over high-latency networks, it proved to be an easy way to push configuration files and scripts to the nodes. Also, we made sure not to use the NFS server for performance-critical steps during the process.

We also chose to use a standard deployment environment, instead of a customized one. This allows us to use an environment maintained by Grid'5000's system administrators, and available everywhere. After deployment, we install the necessary software packages, like `sbuild` (the Debian tool used to build packages in a *chroot*) and `approx`, a Debian mirror proxy. Installed on each node, it allows caching locally build-dependencies that are frequently downloaded, and alleviate the load on the central Debian mirror.

Finally, our infrastructure has obvious reliability issues: both the *master node* and the static part of the infrastructure (NFS server, Debian mirror) are single points of failure. However, due to the length of full Debian rebuilds on Grid'5000 (less than 10 hours in practice), we do not consider this to be an important issue: if a grave problem occurs during a rebuild, it is still possible and relatively cheap to restart the whole rebuild. Regarding compute nodes, the script responsible of running the tasks on them tries to detect problems that might arise during a build. When they occur, the failed build is restarted on another node, and the compute node is removed from the list of nodes used in the rebuild.

IV. Results

Using our architecture, we rebuilt all the packages in Debian lenny using 49 nodes of the *azur* Grid'5000 cluster in Sophia. Each compute node is a server with 2 Opteron 246 (2.0 GHz) CPUs, and 2 GB of RAM. It took a total of 9 hours and 20 mins, of which 13 minutes where spent deploying the infrastructure. The sum of the build time of all packages (sequential time) is 17 days and 4 hours. The logs generated by all builds use 2.0 GB on the NFS directory.

Figure 5 shows the distribution of the packages' build time. One can see that most packages take a very short time to build – 62% take less than a minute, while 90% take less than 3 minutes. However, a few packages take a lot more time (table I). Those packages also are responsible for the majority of the build time (figure 6): the 5% longest packages account for 50% of the build time.

Looking at various system counters during the builds, we could determine that the tasks are both CPU- and I/O-bound. Memory usage generally stays quite low (but may vary greatly between packages). Network does not play an important role: common build-dependencies are cached on the node, and it is only used for control besides that.

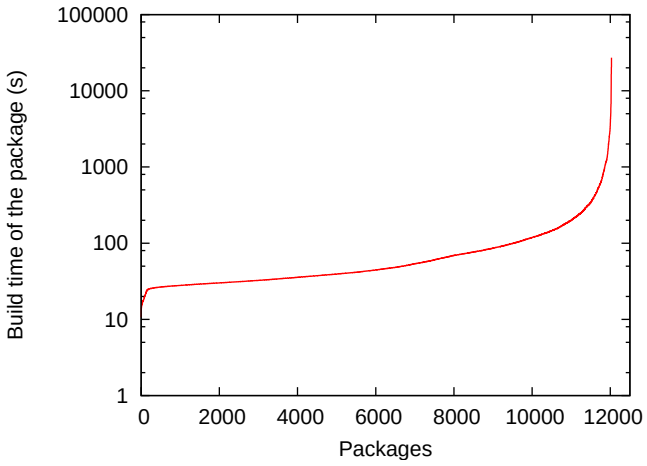

Fig. 5. Distribution of the build time of packages. Most packages are fast to build.

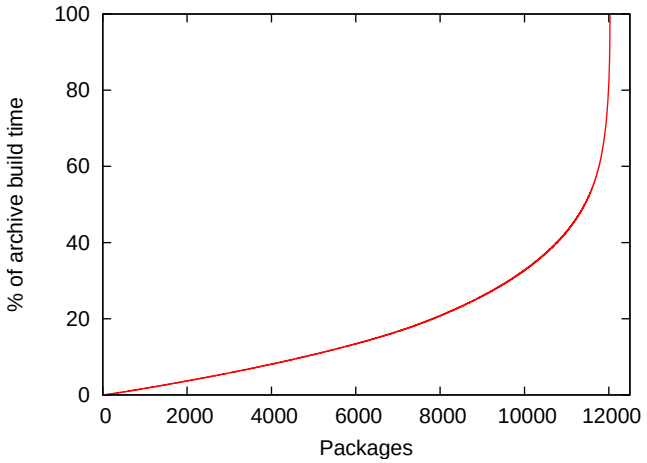

Fig. 6. Share of the build time taken by each package. The few longest packages account for a large part of the archive's build time.

TABLE I
PACKAGES THAT TAKE MORE THAN 2 HOURS TO BUILD

Package	Time
openoffice.org	7 h 33 m
openjdk-6	5 h 42 m
insighttoolkit	5 h 38 m
gecode	4 h 51 m
latex-cjk-chinese-arphic	4 h 38 m
linux-2.6	4 h 33 m
gcc-4.3	4 h 21 m
gcc-4.2	3 h 38 m
installation-guide	3 h 28 m
qt4-x11	2 h 12 m

V. Optimizations

Two objectives can be targeted when trying to improve the process:

- Reduce the makespan, possibly increasing the number of

necessary machines. The main reason for this is that, if possible, we could use a lot more machines on Grid'5000: it is generally considered less disturbing to use more nodes during a shorter period of time, than to use less nodes during a longer period. To reduce the makespan, the main problem to address is the build time of the longest packages;

- Reduce the number of machines without increasing the makespan. This requires making the build process more efficient for all packages.

A. Scheduling of the tasks: longest-first

There are huge differences between the build time of all the packages. While most packages are extremely fast to build, a few packages take a very long time. To minimize the makespan, it is important to schedule those long packages early in the rebuild process: if we schedule them too late, we might reach a point where we all tasks are finished except one, and we are only waiting for that (long) task to finish.

A simple optimization is therefore, after we have determined the time taken to build each package, to schedule them starting with the longest packages.

Using this scheduling, and the results described in section IV, we can estimate that the optimal scheduling of the rebuild of all packages would take 7 h 33 m (time taken to build `openoffice.org`), using 55 compute nodes (this does not include the time needed to configure the environment at the start of the job): with more nodes, some nodes would be idle at the end of the process while we wait for `openoffice.org`; with less nodes, we would still have some tasks to process after `openoffice.org` is finished.

B. Adding parallel building support to long packages

The makespan is limited by the time taken by the longest package – `openoffice.org`. An interesting way to reduce its build time is to make use of parallel building (often known as `make -j`). This consists in running several steps of the build process in parallel to make use of several CPUs or to allow to continue to perform CPU-intensive operations in some threads while other threads are blocked on I/O [7].

Unfortunately, most Debian packages lack support for building using several threads. An interface for that was recently added to Debian's build system, and we worked together with some package maintainers to help them implement it. The results presented in section IV include results obtained with parallel builds for some packages, like `openoffice.org` (where only a small part of the build process can be done in parallel), `linux-2.6` or `latex-cjk-chinese-arphic`.

C. Reducing the local I/O bottleneck

Building packages is I/O intensive, especially for small packages where the build time is dominated by the creation of the *chroot*, and the installation of build-dependencies. We investigated ways to alleviate this problem. First, the EXT3 file system used on the compute nodes issues a `sync()` every 5 seconds by default, to ensure that all the data and meta-data is written to disk. We modified that parameter by re-mounting the build partition with the `commit=86400` option to delay *syncs*. Unfortunately, it seems that `sync()` system calls are also issued by some applications during the build process. As a consequence, we didn't notice any improvement.

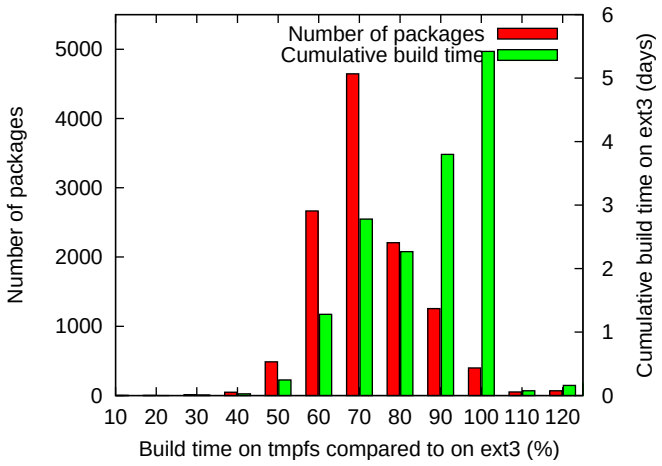

Fig. 7. Distribution of the packages' build time on tmpfs. Most packages reduce their build time, but short builds benefit more than long builds.

Since the data written during the build is only used temporarily, we investigated another solution: building in memory, using Linux's `tmpfs` file system, which stores its content in virtual memory (RAM or swap). As our compute nodes have at least 2 GB of RAM, most packages could be built without ever swapping some memory pages to disk.

This approach has some drawbacks. Firstly, We needed to add more swap space to the compute nodes by adding a swap file, and creating such a file takes a long time during node preparation. We chose to create a 32 GB swap file, and this file is not allowed to contain holes – it cannot be a sparse file. Creating and writing a 32 GB file takes 12 minutes on our compute nodes – limited by the disk writing speed, since the file has to be filled with zeroes. It is possible that the new EXT4 file system will solve this problem by implementing the `fallocate()` system call.

Secondly, some packages failed to build on `tmpfs`, for various reasons that still need to be investigated. However, this approach is promising: the build time (when comparing only packages that built fine with both configurations) was reduced by 13%. But some packages took more time to build on tmpfs, as seen on figure 7. Also, it seems that this optimization mainly benefits packages that are quick to build, while packages that take a long time to build do not benefit as much.

D. Building several packages concurrently

Since most packages lack support for building using several parallel threads ("`make -j`"), another solution is to build several packages concurrently, on the same compute node: for example, the same compute node would build 4 different packages concurrently. This allows to reduce the total number of compute nodes used for the rebuild, without increasing

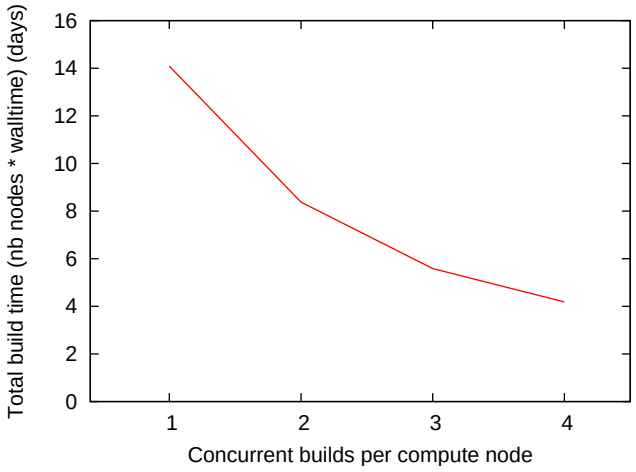

Fig. 8. Overall build time when building several packages concurrently on the same node.

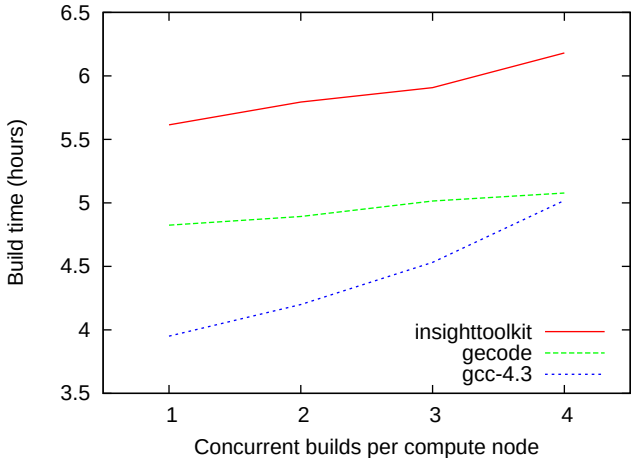

Fig. 9. Build time for some packages that take a long time to build, when running several other builds concurrently on the same nodes.

the makespan. One problem with this approach, however, is that the individual packages take more time to build. While computing power can be shared equally between tasks, the concurrent tasks on the same node will be fighting for the I/O bandwidth. This is not a problem for small packages, but might be a problem for packages that already take a very long time to build: slowing down those packages would result in an increase of the makespan.

We mitigated this issue by setting process priorities based on the duration of the build: long builds get an higher priority than short builds.

Figure 8 describes the overall build time (number of compute nodes used, multiplied by walltime) when running several concurrent builds on the same compute nodes. Running one build per compute nodes, 14 days of compute time on Grid'5000 was used (which could translate into using 42 nodes for 8 hours, for example). Running 4 concurrent builds, the total time decreases to about 4 days (12 nodes for 8 hours). However, in practice, the slowdown caused by I/O concurrency is a major problem: even after having added processes priorities based on the duration of the build, the long packages still take more time when they are built concurrently with other packages (figure 9). A solution could be to schedule long packages alone on a compute nodes, while the shorter packages would be built concurrency with others. This has not been implemented yet.

VI. Conclusion

With this infrastructure, we performed several full rebuilds of Debian during the lenny development cycle, and reported more than 2300 critical bugs on packages that failed to build from source.

In addition to that, this work was the basis of a small shift in the Debian development processes: since it was easy to rebuild the Debian archive with a custom setup, we performed some builds with custom environments to evaluate the consequences of proposed changes to build tools. In the same spirit, several rebuilds were also performed with newer development versions of base software, like the *GCC* compiler: rebuilding all packages in Debian with a beta version of *GCC* allowed to find several important regressions that were fixed before the final *GCC* release. We think that there are other opportunities where such environments could be helpful for the free software community.

This work would not have been possible without the flexibility offered by Grid'5000. This application has very specific and demanding requirements, like the fact that a special environment has to be deployed on the nodes, and that root access is required for several steps. Despite being "experimental" in terms of software used, Grid'5000 proved reliable enough to fully automate the complex processes needed by this work.

References

[1] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in foss distributions: details and challenges," in *HotSWUp '08: Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*. New York, NY, USA: ACM, 2008, pp. 1–5.

[2] "Piuparts: .deb package installation, upgrading and removal testing tool," http://packages.debian.org/unstable/devel/piuparts.

[3] L. Nussbaum, "Use of grid computing for debian quality assurance," in *Research Room @ FOSDEM 2007*, Brussels, Belgium, 02 2007.

[4] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard, "Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform," in *Grid'2005 Workshop*. Seattle, USA: IEEE/ACM, November 13-14 2005.

[5] "Grid'5000 website," https://www.grid5000.fr/.

[6] Y. Georgiou, J. Leduc, B. Videau, J. Peyrard, and O. Richard, "A tool for environment deployment in clusters and light grids," in *Second Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS'06)*, Rhodes Island, Greece, April 2006.

[7] B. Cantrill and J. Bonwick, "Real-world concurrency," *Communications of the ACM*, vol. 51, no. 11, pp. 34–39, 2008.