Bulk building in the many core era

Jörg Sonnenberger ⟨joerg@NetBSD.org⟩

February 4, 2017

Abstract

pbulk, the current generation of the pkgsrc bulk build infrastructure, was created during the GSoC 2007 and remained mostly unchanged. Increased processing power, affordable parallelization, significantly more RAM and SSDs for storage have all changed the environment. This paper investigates different configuration choices and their performance and management impact. The current development towards looser coupling between clients and the build master are presented as well as the necessary changes for a secure integration of package signatures. The impact of the changes is quantified for different native and virtualised setups. The measurements are used to identify improvements for the NetBSD kernel and pkgsrc.

1 Introduction

The pbulk framework for bulk building was created during the GSoC 2007 to address a number of concerns with the old bulk build infrastructure, including:

- No support for building different packages concurrently on the same machine.
- No support for distributing the build across multiple machines.
- Difficult to debug build problems due to partial cleanup between builds.
- Inconsistent rules for when a package has to be rebuilt due to dependency changes.
- Inconsistent handling of redistribution of problematic packages.

As a result both performance and correctness issues have been the major design guide lines for pbulk, with a general preference for the latter.

In the past decade, the computing environment for bulk building has changed significantly. The disk I/O associated with preparing and tearing down the per-package build environment has always been a major time factor, but the recent availability of multiple GB RAM per core and SSDs as backing store has shifted the hot spots to other areas.

The budget requirements for a bulk build running entirely in RAM would have required at least a five digit budget per machine; today typical off-the-shelf dedicated servers in the mid-price range already fit the bill. If a build node had two sockets with a dual core CPU a decade ago, they would now have one or two sockets with eight to sixteen cores each.

Exploiting the available resources efficiently as well as building cost effective systems requires greater care than before. This paper investigates the build time of several natural system configurations. Based on the initial findings, a number of less obvious refinements are compared in order to mitigate severe contention issues. The results pin point issues as well as provide estimations for performance gains from future fixes.

The paper starts with an overview of the pbulk framework and how it extends to a build cluster. Benchmarks for builds of NetBSD/AMD64 follow with configurations ranging from native NetBSD over para-virtualised NetBSD under Xen to fully virtualised configurations under Xen and SmartOS. Finally, the on-going work is presented.

2 Design overview for pbulk

The pbulk framework separates the build into four different phases:

1. Build preparation
2. Tree scanning
3. Package building
4. Post-processing, reporting and uploading

The first phase prepares everything for the new bulk build. This includes the removal of leftovers from earlier builds. Depending on the specific setup, it can also be used to create chroots, spin up virtual machines or synchronize pkgsrc trees.

The second phase is responsible for determining the list of packages and their dependencies. The tree scanning exists in two variations: partial and complete tree. The former option is typically used when packages for a specific set of machines are needed. The user provides a list of directories that should be built and the scan process will iteratively add dependencies as necessary. For the complete tree, a list of all directories is extracted from the top and second level Makefiles and then processed.

While processes similar to the tree scanning can be found in build systems other than pkgsrc, it is normally much easier. Dependencies in pkgsrc are often not specified in the main *Makefile*, but distributed across a tree of included *buildlink3.mk* files. Since pkgsrc supports picking base system dependencies on various platforms, additional computations are often needed. If a package like *devel/cmake* wants to use the Curses library, pkgsrc has to check if the base system provides a compatible implementation. If it finds one, no additional dependency has to be installed, otherwise a fallback to *devel/ncurses* is added. Other complications are optional features in one package changing the behavior of dependees, like the list of visible headers and libraries.

To mitigate the overhead of the build option extraction, pkgsrc recently gained an option[1] for caching the result of those build option extractions. For a typical bulk build, this reduces the time spent in the second phase by 50%. For a complete from-scratch bulk build without reusing earlier packages, the scan phase is responsible for approx. 5% of the total build time. For incremental builds e.g. on a quarterly branch, few rebuilds are typically necessary and the scan phase can dominate the total build time. The scan time can be further reduced in those situations by enabling reuse of old scan results[2]. This will skip the extraction if Makefile (or any of the files included by it) has a newer modification time than the scan result from the last run. This option is not enabled by default as it can give false results if the base system or the build configuration changes. As such, it requires manual intervention from the user to clean the stale files in those situations.

After the extraction of the packages and their dependencies, precise dependencies according to the rules pkg_add uses during installation are computed. This means that discrepancies between scan and build phase show up as errors due to missing dependencies in the latter phase. Some diagnostic data is also provided for maintainers, e.g. if the preferred dependency is not found in the expected location. This can happen when packages share the same name root like PHP versions, but also when alternative patterns are used (e.g. to depend on either an OSS or ALSA plug-in to be installed).

One important consideration for the scan phase is that it is effectively read-only. No packages have to be installed and no temporary files in the build area are created. This allows sharing the same virtual machine or chroot with multiple scan jobs. As the memory requirements per scan jobs are also very moderate, it is easy to run one scan job per core.

The third phase is responsible for building the individual binary packages. A build job in a virtual machine or chroot will first decide whether a pre-existing binary package is good enough to be reused. This check normally includes:

- The exact dependencies for the current build match the state of the old build.
- The recorded RCS IDs of important files like the main *Makefile* or *distinfo* match the state of the pkgsrc tree.
- The binary package is newer the packages it requires.

[1] Added 2016-12-18, set *PBULK_CACHE_DIRECTORY* in *mk.conf* and create/empty it in the preparation step.

[2] Added 2012-11-23, set *reuse_scan_results* to *yes* in *pbulk.conf*.

If all checks pass, no further work is necessary. If there is no pre-existing binary package or it is considered out-dated, a full package build is started. The build job will first tear down the package prefix (e.g. */usr/pkg* per default on NetBSD).

The actual build processes in stages:

1. pre-cleanup
2. installation of listed dependencies
3. fetching distribution sources and patches and matching the checksums against *distinfo*
4. configuration
5. build
6. installation to staging directory
7. creation of binary package
8. test installation

The output of the stages is logged separate for easier debugging and preserved for failing builds. Depending on the result, either a fresh binary package exists or build log for the problems. After notifying the build scheduler, the next build is executed.

The fourth and final phase is responsible for building the SHA512 index over all packages as well as uploading the results (if applicable) and providing a human and machine readable report.

Both scan phase and build phase provide a basic scheduler mode to process jobs sequentially from an integrated client. This mode is functionally very similar to the previous generation of the bulk build system. It is reasonable when doing a small set of packages or when operating on constrained hardware (e.g. with only one or two cores). Pre-populating a local copy of distribution sources should still be considered to avoid the delays from fetching during the build from the internet.

Alternatively, scan and build jobs can run in parallel in their respective phase, either across a different chroots or across different (virtual) machines. The framework provides a pull-based scheduler. A hook is provided to execute a script after the master process is ready and this in turn will start the scan or build clients via chroot[3] or ssh. Additional clients can be added on-demand. The protocol currently provides no mechanism to restart a failed build of an individual package, even if it is a side effect of a machine crash.

[3] Added 2016-12-18, by specifying an absolute path as client in *pbulk.conf*.

3 Build cluster infrastructure

The pbulk framework depends on a number of shared resources for efficient builds:

- distribution files and patches,
- logs for failing builds,
- binary packages and
- the pkgsrc tree itself.

The past recommendation for bulk builders was to use shared file systems like null mounts or NFS. While this works reasonably, it is also regular a source of headaches in some cases. For the distribution files, if two build jobs try to fetch the same file at the same time, they can stumble across each other, making one of them fail. As the build order of variant packages like Python modules tend to schedule different variants of the same module near each other, this has been a regular problem. In the case of the log directory on the other hand, NFS is known to create build failures when intermediate file operations fail like attempt at locking.

This section includes a new set of recommendations based on those experiences.

3.1 Distribution files and patches

For a single machine build, having a local directory null mounted into each build chroot is still a reasonable low-overhead option. The danger of the occasional failures has been mentioned. Sharing across machines with NFS is no longer recommended.

A scalable and reliable mechanism is to use a local mirror with automatic upload of new files. For this purpose, *MASTER_SITE_OVERRIDE* should point to the local HTTP or HTTPS server. It can run on the same machine as the build master, but any other local server works as well. The second part of the puzzle is *POST_FETCH_HOOK*, a script that is called with path of the newly obtained file and the URL the file was obtained from. The author uses WebDAV for uploading new files,

as requires less configuration than sftp or scp only SSH upload. An example script can be found in listing 18. Finally, *DISTDIR* can be set to $WRKDIR/.distdir, so that files are removed automatically once the build job is finished. In this case, *use_unprivileged_checksum*[4] should be set to 'yes' in *pbulk.conf* as well.

Some package have distribution files which can't be obtained automatically. A typical example is the EULA for Oracle's JRE and JDK builds. If the user uploads those files to the local mirror by hand, the *IGNORE_INTERACTIVE_FETCH* option can be used to skip the check, so that *MASTER_SITE_OVERRIDE* is actually tried.

3.2 Build logs

Similar to distribution files, having a directory null mounted into each build chroot is reasonable for single-machine builds. The file system is primary modified by appending to files, so lock contention not a big factor. For cross-machine machine builds, it is recommended to use either scp/sftp based uploads or WebDAV. NFS comes with a noticeable performance penalty and the earlier mentioned mysterious error cases. The *sync_buildlog* option[5] in *pbulk.conf* can be pointed to either a shell script or function to provide the necessary functionality. An example can be found in listing 7.

The long-term plan is to integrate the uploading of the build logs into pbulk itself to reduce the number of moving parts.

3.3 Binary packages

The constraints for binary packages are a bit different. Unlike build logs or distribution files, the binary packages are accessed directly as part of the check for skipping rebuilds. It is difficult to reproduce this logic for packaged served via HTTPS. As such, a shared file system like NFS is unavoidable for the cluster case. Since the check uses time stamps, use of NTP is strongly recommended for all build machines.

It is possible to use a read-only NFS file system in combination with the *sync_package* option[6] in *pbulk.conf*. The primary use case is allowing package signing from a separate environment, so that keying material is never accessible to the builds. The basic approach is illustrated by listing 10, but the given script shouldn't be used in production due to the obvious security concerns.

The long-term plan is to ingrate the uploading into the pbulk protocol to reduce the number of moving parts as well as to allow tighter constraints on the package content. The build master knows the package that is supposed to be signed and can detect fancy attempts for creating signed core packages when a minor source has been trojaned. This has been deferred until the infrastructure side for multi-package builds (e.g. one build creating multiple sub-packages) has been resolved. The work was started by Leonardo Taccari's GSoC 2016 project.

3.4 Sharing of the pkgsrc tree

The pkgsrc tree is read-only during the build and especially during the scan phase highly contended. The combination of those properties make it attractive to distribute a copy to all build nodes e.g. via rsync in the first phase of the build. Even for diskless clients like building on small ARM boards, it is often better to have a copy of the pkgsrc tree on a local file system over iSCSI compared to plain NFS, since NFS provides only limited caching. Multiple scan/build jobs on a single node can share the same copy via null mounts. No noticeable lock contention has been observed on NetBSD at this point for shared pkgsrc trees. The *client_prepare_chroot* and *client_prepare_action* options[7] can be used for this purpose.

4 Comparative benchmarks of different configurations

In this section, different configurations will be evaluated for a whole tree from-scratch bulk build. Common base configuration is a NetBSD current on AMD64 from mid-2016 with clang as system compiler. Some of the Xen and SmartOS based tests use a newer kernel from around November 2016 for technical reasons, but it is believed to not have any significant impact on the build times.

[4] Added 2015-09-08
[5] Added 2015-09-13
[6] Added 2015-09-13
[7] Added 2016-12-18

Run	nullfs root		tmpfs root		Difference	
	scan	total	scan	total	scan	total
1	97	2667	102	2111		
2	113	2862	119	2211		
3	112	2862	98	2211		
Avg	107	2797	106	2178	-1%	-22%
σ	4.2	53	5.3	27.2		

Figure 1: Scan and total build time (in minutes) for nullfs root vs tmpfs root

A number of serializing packages are explicitly marked as broken like *math/openaxiom*, as they would skew numbers artificially

Test platform is a dual Intel Xeon E5-2630v3 with 64GB RAM and SSD for persistent storage. Hyper-threading is active. Persistent file systems use FFS with WAPBL enabled. MAKE_JOBS is set to 16, developer checks are enabled. The use of cwrappers is enabled as well.

All distribution files are served by a HTTP server in the local network, connected by Gigabit Ethernet. Build logs are pushed via rsync similar to listing 7. Packages are written directly to nullfs or NFS (for multi-VM configurations). The state of the pkgsrc tree is identical across all builds. Para-virtualised Xen builds have a couple of additional failures related to Emacs, but this is believed to be a minor impact.

4.1 Base line with native kernel

The original configuration used a FFS file system for the root template of the chroots. Each chroot has a read-only null mount of this root template with tmpfs instances mounted over it for /etc, /var, /tmp, /usr/pkg and the work area. Eight chroots are used for build jobs and each chroot played host to four scan jobs. Two additional chroots are provisioned for the build master and potential testing, but are otherwise not used during the builds.

Investigations of lockstat output suggested a high contention of the vnode for the top level of the root template. This lead to the first variation with a read-only tmpfs copy of the root template for every chroot. Root template contains the 'base', 'comp', 'etc', 'games', 'misc' and 'text' sets with small pruning by removing the lib*_pic.a files which are not relevant for package builds. The resulting tmpfs requires 518MB per chroot.

Timing results can be found in figure 1. The difference in the scan time is within the noise. This was unexpected as the scan phase normally hits the file system and memory subsystem much more often due to the high fork/exec rate. An easy explanation like "the scan phase is read-only" doesn't work as the file system itself is read-only in both cases. The lock protocol for nullfs clearly needs to be investigated.

The tmpfs root numbers will be used as base line for all further comparisons

4.2 Para-virtualisation with Xen

The first alternative to running NetBSD on bare-metal hardware is para-virtualisation with Xen. This is used on the build cluster of The NetBSD Foundation for the official bulk builds, so the overhead here is direct interest. The reference configuration is a NetBSD DOM0 configured with 1GB RAM running under Xen 4.6.3 as hyper-visor In the single DOMU case, one NetBSD guest is using the remaining 62GB RAM and 31 cores. For the four DOMUs case, all DOMUs have 15GB RAM assigned and eight cores, except one DOMU with only seven cores.

Timing results can be found in figure 2. The single DOMU case was expected from unpublished benchmarks by the author in the 2007-2008 time frame. The real surprise was the four DOMUs case though. An investigation with lockstat showed a very high contention on the page queues. When splitting the system, this contention has cut down significantly and compensates for the system call and context switch overhead of Xen. Since this overhead is still present in the four DOMUs numbers, they provide a good lower bound for the expected gains from more granular page queue locking.

4.3 Full virtualised builds with Xen-HVM and SmartOS-KVM

The reference configuration for Xen-HVM uses a NetBSD DOM0 configured with 1GB RAM and a four HVM guests with 15GB RAM each. The hyper-visor is Xen 4.6.3. The guests all have eight

Run	Base line		1 DOMU		Difference		4 DOMUs		Difference	
	scan	total	scan	total	scan	total	scan	total	scan	total
1	102	2111	185	2897			34	1917		
2	119	2211	188	2894			33	1913		
3	98	2211	186	2909			33	1909		
Avg	106	2178	186.3	2900	+75%	+33%	33	1913	-69%	-12%
σ	4.2	53	0.72	3.7			0.27	1.9		

Figure 2: Scan and total build time (in minutes) for base line vs 1 DOMU vs 4 DOMUs

Run	Base line		Xen-HVM		Difference		SmartOS KVM		Difference	
	scan	total	scan	total	scan	total	scan	total	scan	total
1	102	2111	39	2880			35	2721		
2	119	2211	36	2748			48	2727		
3	98	2211	38	2927			43	2774		
Avg	106	2178	38	2852	-65%	+31%	42	2741	-61%	+26%
σ	4.2	53	0.72	44			3.1	14		

Figure 3: Scan and total build time (in minutes) for base line vs Xen-HVM vs SmartOS KVM

cores assigned each, except one with only seven cores.

In the SmartOS case, the KVM guests were assigned 14GB each. A stable build was not possible with 4GB RAM reserved for the SmartOS kernel, even with manual tuning to limit the size of the ARC buffer. NetBSD uses virtio for both disk and network I/O under KVM.

Timing results can be found in figure 3. The scan time results can be readily explained by the page queue contention as discussed in the last section. Both systems have comparable results with SmartOS winning over all. The differences can likely be attributed to optimizations in the page table trapping or context switching, but has not been analyzed in detail. The cost of complex device emulation vs the simpler virtio protocols is a factor that may favor SmartOS for the longer builds though. The virtualisation overhead on sixth generation Xeons with Extended Page Tables and other features is significantly lower compared to the third generation Xeons, where a bulk build took over 100% longer in Xen-HVM compared to native.

5 Work in progress

There are currently five on-going projects for the pbulk framework:

1. Integration of a better process monitor for BSD systems
2. Rebuilding the storage mechanism by moving to SQLite
3. Extending the communication protocol to allow retrying failed individual builds.
4. Extending the communication protocol to separate up-to-date check and build jobs.
5. Investigate metrics for resource consumption and build load for improved scheduling.

Long-term a move to a new HTTP(S)-based communication protocol is planned, but no details exist yet.

5.1 Process monitoring with kqueue

Build jobs sometimes hang for a long time. There are two common problematic situations:

1. Infinite loops resulting in wasted CPU time.
2. Dead locks and other conditions where no progress is made.

The first category can be solved easily with process limits on the CPU time, but this doesn't work for processes that wait for a very long time on events that might not never happen. Regular occurrences with pkgsrc are Qt's MOC with bugs in the process cleanup, Mono with internal dead locks or the job accounting problems in GNU make 4.2.1.

The author has created a new process monitor for this purpose. It uses the kqueue event notification system to listen for process creation and termination. If no process is created within a given time period, the job is considered to hang. It can also specify a fixed hard limit for the total time a job is allowed to take.

As a side effect of monitoring the process tree, it can also detect leftover process. At least one package in the pkgsrc tree is known to leak 'bonobo-activation' instances, but it is not yet known which. The processor monitor would allow hunting for this kind of garbage.

The tool is currently tested internally and will be published after some more polishing. It should work on all systems with a kqueue implementation, assuming the kernel implementation works correctly.

5.2 Moving towards SQLite storage

When pbulk was originally written in 2007, only the old Berkeley Database 1.85 was used by pkgsrc infrastructure. Experience with SQLite at the time was too limited for taking it into consideration. As a consequence pbulk is using flat files in append-only mode. After crashes, manual cleanup was often necessary to resume a build and code flexibility was also limited.

Since then, SQLite has seen use in different NetBSD and pkgsrc related projects:

- NetBSD's *apropos* uses the Full Text Search of SQLite.
- The repository mirror of NetBSD's CVS tree for Fossil and Git depend on SQLite for storage and query logic.
- The *pkgin* front-end depends on it.

Switching pbulk to using SQLite allows further developer and simplifies improvements to the data model. When the scan phase was extended to allow reusing old results, it had the side effect of increasing the data volume by a factor of six. This directly affects the memory requirement accordingly. Normalization would avoid most of the overhead.

5.3 Protocol extensions for retrying individual builds

The build scheduling is currently a one-way street. Every package is scheduled for build at some point after its dependencies are built successfully. The build status is appended to one of two files (*error* or *success*). This makes it difficult to provide a mechanism for developers to say "I fixed foo-1.2, try again please". The only way is currently to stop the build, edit the *error* file by hand and then restart the build. As this doesn't magically stop current builds and those can take a long time to finish, it is clearly suboptimal. With the new storage backed, it is much easier to extend the communication protocol with an option for rescheduling a failing package or just doing the manipulation directly in a way that it will be picked up on the next decision point.

5.4 Up-to-date-check nodes

As detailed in section 3.3 pbulk checks whether binary packages can be reused or not. For this check direct access to the binary packages is necessary and this is the only part of a bulk build still depending on shared file systems.

Three possible solutions exist:

1. Rewrite the check to deal with protocols like HTTP.
2. Move the check into the build master.
3. Create a new set of jobs that can be run from the machine hosting the packages.

The first option is somewhat difficult as it requires a custom client code and the protocol is not that friendly towards such time checks. The second option has been considered, but the build master currently needs no access to the actual pkgsrc tree and it would be detrimental to change that back. This

leaves the last option of splitting the build jobs into two parts. This requires extending the protocol to support three different clients: build-only, check-only, combined build and check. That's a moderately small change. The master needs to keep track of a second to-do list and some additional book keeping. Overall, this is the most promising approach.

5.5 Tracing resource consumption

From a theoretical perspective, an optimal build scheduling should take the availability of restricted resources and optimize the job scheduling to minimize the target function (here: the total build time). For bulk builds, CPU and total build time, concurrency, memory requirements and volume of disk I/O are all possible factors for the build time. The prediction of the interaction of multiple builds on the same machine is highly complicated. During the configure phase, many package will utilize only a single core and little memory. During the build phase, many cores can be used, but compiler and linker can require a lot of memory for a single task. The installation phase, but also linking of debug builds, can create a lot of disk I/O. For a bulk build, many of those interactions will even out towards a mean. From regular observations of bulk builds one pattern for improvement is obvious: avoiding building of large packages near the end. This normally happens naturally as some very large packages like LibreOffice are not necessary for building anything else. Building a database of at least the total CPU time, the wall time and the peak RSS would allow making better decisions. Some of those measurements can be obtained easily. The move towards SQLite will also simplify integration of such data.

6 Summary and outlook

The paper presented modern recommendations for configuring bulk builds for pkgsrc with a focus on NetBSD. While some of the details will differ on other operating systems, the core tenets remains.

Real-world performance tests have shown clear scalability issues in NetBSD. Para-virtualisation can provide a 10% boost to the over-all build time on the tested hardware. It is expected that the benefit grows for systems with even more cores.

Full virtualised solutions like Xen-HVM and KVM on SmartOS have been shown to have a reasonable overhead of 25% to 30% on Haswell Xeons, compared to native NetBSD builds. This shows that cloud-based bulk builds are technically feasible without wasting too much in terms of resource utilization The missing cooperation between guest OS and hyper-visor clearly comes at a significant price.

The integration of better process monitoring tools for the build is an on-going project. The pbulk framework will see changes to its communication protocol and storage mechanisms in the near future. This will ease the deployment on smaller machines and improve the experience for developers.

```
#!/bin/sh
mirror=distfiles.example.com

case "$2" in
https://${mirror}/*)
    ;;
*)
    # cacert.pem is not in the default search path.
    # Pick password from distfile_netrc, i.e. in the form:
    # machine distfiles.example.com
    # login myuser
    # password mysecret
    /pbulk/bin/curl -T "$1" \
      --cacert /pbulk/share/mozilla-rootcerts/cacert.pem \
      --netrc-file /pbulk/etc/distfile_netrc \
      "https://${mirror}/$1" ;;
esac
```

Listing 1: Sample upload script

```
push_buildlog() {
    # Don't depend on rsync in PATH
    /pbulk/bin/rsync --rsync-path /pbulk/bin/rsync \
       --delete --archive -e ssh "$1/$2" ${master_ip}:$1/
}
sync_buildlog=push_buildlog
```

Listing 2: *pbulk.conf* fragment for build logs

```
push_package() {
    ssh ${master_ip} mkdir -p /packages/All
    scp "$1" ${master_ip}:/tmp/${pkgname}${pkg_sufx}.tmp
    ssh ${master_ip} /pbulk/sbin/pkg_admin gpg-sign-package \
      /tmp/${pkgname}${pkg_sufx} \
      /packages/All/${pkgname}${pkg_sufx}
    ssh ${master_ip} rm /tmp/${pkgname}${pkg_sufx}
}
sync_package=push_package
```

Listing 3: Insecure proof of concept for signing package remotely