Tracking FreeBSD in a Commercial Setting

M. Warner Losh
iXsystems, Inc
Broomfield, CO
imp@freebsd.org

October 10th, 2010

Abstract

The FreeBSD project[1] publishes two lines of source code: current and stable. All changes must first be committed to current and then are merged into stable. Commercial organizations wishing to use FreeBSD in their products must be aware of this policy. Four different strategies have been developed for tracking FreeBSD over time:

- **Stock FreeBSD:** A company runs only unmodified releases of FreeBSD.
- **Grab and Go:** A company imports a specific version of FreeBSD's sources once and then never imports new code from FreeBSD again.
- **Stable Branch Tracking:** A company imports new stable branches over time, adding its own changes to that branch, as well as tracking FreeBSD's changes to the branch.
- **Own Branch:** A company tracks FreeBSD's `CURRENT` branch, adding to it their changes and creating their own stable branch as it sees fit.

This paper catalogs the methods different companies have used and explores the pros and cons of each.

1 Problem Statement

While many companies have based their products on FreeBSD, little organized exists to guide new companies' integration of FreeBSD. They have many choices in how they could adapt FreeBSD's code. For some, FreeBSD's binary releases can be used unmodified. Others require extensive modification and additions to FreeBSD. Most companies fall somewhere between these extremes, but little formal documentation exists to guide them. In the other direction, companies need information about the management of contributions back to the community with guidance on when, what and how to contribute.

FreeBSD provides an excellent base technology on which to base products. It is a proven leader in performance, reliability, and scalability[2][3]. Its IEEE 802.11 wireless technology leads the industry[4]. The technology also offers a very business friendly license that allows companies to pick and choose which changes they wish to contribute to the community rather than forcing all changes to be contributed back, or attaching other undesirable license conditions to the code.

However, the FreeBSD project does not focus on integration of its technology into customized commercial products. Instead, the project focuses on producing a good, reliable, fast and scalable operating system and associated packages. The project maintains two lines of development: a current branch, where the main development of the project takes place, and a stable branch which is managed for stability and reliability. While the project maintains documentation on the system, including its development model, relatively little guidance has been given to companies in how to integrate FreeBSD into their products with a minimum of trouble.

Developing a sensible strategy to deal with both these portions of FreeBSD requires careful planning and analysis. FreeBSD's lack of guidelines to companies leaves it up to them to develop a strategy. FreeBSD's development model differs from some of the other Free and Open Source (FOSS) projects. People familiar with those systems often discover that methods that were well suited to them may not work as well with FreeBSD's development model. These two issues cause many companies to make poor decisions without understanding the problems that lie in their future.

Companies looking for formal guidance on integrating FreeBSD into their products will find little organized information. The FreeBSD handbook[5] documents the development process, but gives no guidance on how to integrate FreeBSD into a product. Internet searches reveal several email threads on the topic, but the information is haphazard and contradictory.

2 FreeBSD Branching

The FreeBSD development model strikes a balance between the needs of developers and the needs of users. Developers prefer to have one set of sources that they can change arbitrarily and not have to worry about the consequences. Users prefer to have a stable system that is compatible with the prior systems. These two desires are incompatible and can cause friction between developers and users.

FreeBSD solves these problems by providing two versions of its code. The developer version of the code is called "FreeBSD-current," "CURRENT," or "current". This contains the latest code, including work in progress and other code that might not be completely ready for end users. The quality of the current branch varies over time as new code is committed and bug fixes are made. This works well for the developers, but using current in product requires extreme care and the project generally against using it there. New stable branches are created by the release engineering team from the current branch on a regular basis.

The user version of the code is called "FreeBSD-stable," or just "stable." The release engineering team creates this branch every few years from the current tree. Before the branch is created, the release engineer manages the current tree towards stability: the rate of new features going in is slowed, and bug fixing is generally encouraged. These branches are numbered sequentially and named "RELENG X", the most recent branch is called "RELENG 7." Stable branches are well tested before releases are created from them. Once released, only well tested patches from the current branch are allowed to be merged into the branch. Over the life of the branch, the Application Programming Interface (API) and Application Binary Interface (ABI) are stable and only changed in an upwardly compatible manner. Users can therefore upgrade from point to point within the branch with relative easy. Stable branches tend to have a lifetime of about three to six years.

In addition to the stable branch, FreeBSD creates releases from time to time from the stable branch. Releases have an even higher level of testing than the stable branch, and includes integrated packages built from the FreeBSD ports tree[6]. After the release, a branch is maintained to incorporate security fixes and critical bug fixes. Users can upgrade their release to the latest version of that release's branch, or to newer releases by rebuilding sources or with a binary update program. While technically different from the stable branch, the rest of the paper will lump all these branches into the parent stable branch.

The FreeBSD project defined its ideal release and branching strategy. Every two years, the project releases a new major version and creates a new RELENG branch. The project releases minor versions on the RELENG branch every three to six months for about two years. After this active phase, the RELENG branch is maintained for critical bug fixes and security issues for about another year. After about three years, the branch is then abandoned in favor of the newer RELENG branch. Figure 3 shows this graphically. The horizontal axis is time, measured in years. The vertical access is cumulative change to each branch over time.

Figure 4 shows the actual branching and release history. This data was collected from the subversion repository. There are three features of this graph that need to be called out. The first one is that branches tend to live beyond the three year "ideal" case. Second, the branching is less regular than the ideal plan suggests. After the 5.x misstep, we've branched on time for 6.x, 7.x and are on track for an 8.x branch. Third, although it looks like there's a flattening out of commit rates in the project, the graphs is misleading because it omits data. In 2002 the project started using perforce for work in progress. In 2008, the project moved from CVS to subversion for Source Code Management (SCM), and started to move work in progress from perforce to subversion. Figure 5 shows these additional changes, which shows where the apparently "missing" commits in Figure 4 have gone.

The FreeBSD ports system (which is used to generate the packages that appear in FreeBSD's releases) is not branched at all. Instead, it supports both the current branch, as well as the active stable branches of the project. For each release, the tree is tagged so that it can be reproduced in the future if necessary. These policies are different than the main source tree. Tracking ports is beyond the scope of this paper.

3 Branching Choices

A wide range of companies use FreeBSD in their products today. On the simplest end, companies load FreeBSD onto boxes that they ship. On the most complex end, companies modify FreeBSD extensively to make it fit their needs. Over the years four different approaches to tracking FreeBSD have evolved.

The simplest method uses the stock FreeBSD releases unmodified. Companies download FreeBSD at its release points and make no changes to the base software. They just change the configuration settings and install packages. They typically don't track the sources and only install binary packages from FreeBSD's web pages. Some companies keep the sources to FreeBSD in escrow for regulatory compliance. Most companies move from release to release as necessary, or install security updates.

The next simplest method involves grabbing a release of FreeBSD and using that as a basis for their product. The company makes whatever modifications necessary for their product. The company doesn't update the sources to newer versions of FreeBSD. Almost always, the company doesn't contribute any of its changes back to the FreeBSD project. Effectively, they have created a fork of FreeBSD.

Companies often set up repositories of FreeBSD stable branches. In this model, the tip of a stable branch (or the latest release point) is imported into some SCM. The company makes fixes and improvements to its private branch. They also import newer versions of FreeBSD from the parent stable branch. Many companies loop back changes to FreeBSD to reduce the number of changes they must maintain and to simplify their upgrade path.

The most complicated method mirrors FreeBSD's development process. The company imports some version of the FreeBSD development branch. Updates happen frequently, often automatically daily or weekly. They make changes to FreeBSD in this mainline of development. Rather than using FreeBSD's stable branches, the company will decide when and where to branch its version. Once branched, it will control what fixes are merged into its branch.

3.1 Stock FreeBSD

By far the simplest way to use FreeBSD is to download releases and use them unmodified or with the latest security fixes. They layer packages on top of FreeBSD, typically a mix of stock packages from the release and their own additional scripts or programs. They customize and configure the system using standard tuning knobs in FreeBSD's configuration files. The focus of these companies is to have a system that they can deploy and use for a particular purpose. Most commercial users of FreeBSD use this method. Some of these companies will also compile customized kernel configurations. Some companies will also build custom versions of FreeBSD using the NanoBSD or TinyBSD build scripts.

Customization of the system is typically tracked in some kind of SCM system. These customizations include the `/etc/rc.conf` file (which controls most of the global settings for the system), as well as configuration files and other data used by other programs in the system. The number of files that a company needs to manage using this method is typically less than 20.

Apart from security updates, these companies typically upgrade only when new features or hardware support forces them to upgrade. Once they find a stable version they stick with it until they need something from a newer version. This could be support for newer hardware (drivers or architectures), new application level features such as threading support, or performance improvements for their workloads. Updates via freebsd-upgrade(8) are common for machines performing services, but fairly rare for deployed systems.

FreeBSD meets the needs of these companies fairly well. They don't require additional features or bug fixes not in the current releases. They don't need to optimize FreeBSD for any given platform beyond what the standard system tunables provide for them. The main advantage for these companies is that FreeBSD is a drop in solution. There's very little overhead necessary to get their machines and applications running and FreeBSD's standard install tools can be used to create images for their products (if they even need separate images at all). Some of these companies participate in the community and contribute bug reports, bug fixes, documentation or user support to the community.

3.2 Grab and Go

Another easy way to use FreeBSD sources is the grab and go method. In this method, the company grabs FreeBSD and modifies it for their needs. The company doesn't attempt to track changes in FreeBSD and pulls in little or no bug fixes. They layer in their own build and packaging system often times. Sometimes they port to a new architecture. FreeBSD typically is the base for a more extensive application or appliance which the company has total control over.

There are a few advantages to this method. The company can concentrate on making their product work without new versions distracting its engineers. The company manages its risk by doing everything themselves since external changes won't affect them at all. The company can keep any information about what they are doing from being inferred by competitors looking at their bug submissions to FreeBSD. The company's employees are not distracted by interactions with the FreeBSD community. Without these distractions, the company can bring its product to market more quickly, at least in theory.

However, there are many disadvantages to this method. The biggest problem is older versions of FreeBSD are poorly supported by the community. If there are problems in the base that require community involvement to solve, the company may have difficulty finding users in the community that care enough about the old release to help them. Most of the active members in the community use only recent versions, so are unable to help out with problems in older versions. Experience over the years has shown that community support can often save many hours troubleshooting time. Often times, interaction with the community on problems for recent releases of the software can save tremendous amounts of time for the company's employees because they can leverage the knowledge of others who have had similar problems.

Second, bug fixes in newer version of FreeBSD can be difficult to back port since the fixes often depend on other fixes. This is not a big problem when the software is relatively recent, but can be a big problem if FreeBSD is on a new major version.

Companies often times think they are in total control of the hardware platform, but in reality this is a mistaken assumption. Hardware platforms are made of up chips that one buys from manufacturers. These chips go obsolete at an alarming rate sometimes, forcing changes to the underlying hardware to even be able to continue to build it. These new chips often times require changes to the drivers to work optimally. Just as often, others in the community have used the newer parts and have migrated the necessary changes into FreeBSD. Companies using the grab and go method often must rework these changes to fit the older version of FreeBSD they are using.

Some companies have managed to start out with this method and later transition to one of the other methods described in this paper. One is even rumored to have recently completed the jump from FreeBSD 2.1.6 (released in 1996) to FreeBSD 6.2 (released in 2008) and are now using the stable branch tracking method described below. This method is viable when the time lines are relatively short, but becomes difficult to sustain over more than a few years.

3.3 Stable Branch Tracking

One nice feature of FreeBSD's stable branches is their stability. The FreeBSD project manages the branch to ensure this stability by limiting the number of changes to the branch. Almost all changes must be proven in "CURRENT" for at least two weeks before being eligible for merging to a stable branch. All APIs and ABIs are managed to minimize incompatible changes. The stable branch almost always builds and rarely has serious problems. The stable branch tracking strategy takes advantage of these features.

Companies using this approach import a suitable version of FreeBSD's stable branch. Typically, they import it into a SCM using the SCM's "vendor branch" features. A private branch is then created from this "vendor branch" where bug fixes found by the company's engineers are committed. The company makes their product releases from this private branch. Over time, new version can be imported into the vendor branch and merged into the private branch.

This separation offers many benefits. First, it isolates the company from change in the FreeBSD project. Since the importing and merging operations are under the control of the company, they can choose when to do them. Testing can be done in side branches for the stability of a candidate import as well. These tasks can be done as resources allow. The importing can also be automated via a cron(8) job.

Second, it allows for easier patch management since the company's changes are isolated from FreeBSD's changes. The SCM tracks all changes, so a record of when and why changes were made is kept. The

SCM can be used to generate patches to be submitted to FreeBSD. With frequent updates, and the cooperation of a FreeBSD committer[1], the company can minimize differences to stock FreeBSD, help the community with bug fixes and benefit from code reviews of submitted changes to improve patches.

Third, it allows "cherry picking" fixes from many sources. One can upgrade a small portion of the tree to a newer stable version, to a "CURRENT" version, or even to patches posted in a mailing list. This allows companies that need to be on the bleeding edge for some hardware to use that hardware and leverage work done elsewhere in the project to support the hardware without needing to take all changes to the tree to get that support in many cases.

Forth, this method leverages the extensive release engineering that is performed on the stable branch on an ongoing basis. Because the branch is managed, and the focus of the project near releases, its quality remains high, limiting the risk for pulling from this branch. Since the APIs and ABIs are carefully managed for compatibility in the stable branch, fixes from newer versions of the branch very often are "drop in" compatible with whatever version of stable is in use at the company at the moment.

Finally, since modern SCMs allow for easy branching, the merging of new versions can be done on a branch of the mainline. The stability of the APIs in the branch makes this process easy, and the branches allow for testing before merging into the mainline. Branches can also be used to pin different projects to a specific point in time with only critical fixes applied.

As new stable branches of FreeBSD become available, this process can be repeated for them in a separate module or directory in the SCM. Moving from one stable branch to another can also be used as an opportunity to examine what patches may make sense to contribute to FreeBSD to help ease the burden of moving from branch to branch in the future, and to help keep the project strong.

There are a few disadvantages for this approach. First, to fully leverage the FreeBSD community, it is desirable to push back bug fixes to the community in a timely fashion. When this isn't done, as is often the case when deadlines are tight, the chore up upgrading increases because one must bring forward all of the changes to the system. Second, if the company makes extensive changes that aren't merged back into FreeBSD and want to migrate to the next major version, they will need to redo their changes after the next major branch is created. If they are in an area of FreeBSD that has changed between the two branches, this can take quite a bit of time and effort.

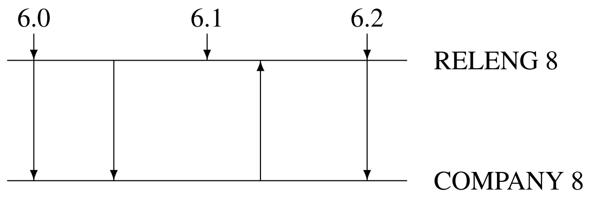

Figure 1: Code Flow between FreeBSD RELENG 8 and Company's version of 8.x

Figure 1 shows this graphically. This figure shows an idealized flow of patches into the company tree and back to FreeBSD.[2] The branches are shown as flat lines, rather than a graph of the number changes to the branches like Figures 3 and 4 presented above. The downward arrows pointing to the RELENG branch represent FreeBSD releases from that branch. The arrows from the RELENG branch to the COMPANY branch represent merges of code from FreeBSD into the company's repository. The arrows from COMPANY to RELENG represent patches that have successfully been contributed back into FreeBSD and have been merged into FreeBSD's RELENG tree. Notice that FreeBSD releases and code integrations from the RELENG tree can be decoupled.

3.4 Own Branching

Another way to keep current in FreeBSD is to track FreeBSD "current." Most developers use perforce or subversion to implement this and find that it works well. This method follows that practice, and adds stable branches, akin to FreeBSD's stable branches in concept, but not tracking any specific FreeBSD release.

Similar to what's described in the *Stable Branch Tracking* section, the company imports FreeBSD "current" into a vendor branch. The company creates a development branch where they commit changes to FreeBSD to. The vendor branch is updated from time to time, typically automatically. Merging the new FreeBSD into the company's private branch happens as time and resources permit. This method automatically gives developers in

[1]An individual with commit access to the FreeBSD repository.

[2]For simplicity, this fgture neglects to picture the required trip through FreeBSD current required for all patches to be committed to stable branches.

the company an easy way to generate patches to integrate into FreeBSD.

The company also emulates FreeBSD's branching practices. When the tree is in a good state to branch, possibly driven by delivery schedules for its end products, the company branches their own stable branch from their current branch. They merge bug fixes and new features from their current branch into this stable branch and build products from this stable branch.

The main advantage of this approach is that it is easier to keep current with FreeBSD than the stable branch tracking approach. To generate patches, a simple diff(3) between the FreeBSD sources and the company sources will generate the patches. As patches are merged with FreeBSD, the next pull will automatically include those changes and the delta between the company's sources and FreeBSD's will drop. By controlling the branching times, there's no need to wait for FreeBSD to create new a stable branch, so the company can drive released schedules more easily than companies tracking stable branches.

The main disadvantage of this approach is that the company loses the work done by the FreeBSD community to keep its stable branches stable and useful. Since there is no connection between the company's stable tree and FreeBSD's stable tree, improvements to FreeBSD's stable branch aren't automatically reflected in the company's stable branch. An engineer will need to watch changes going into either the current branch from FreeBSD, or into FreeBSD's stable tree and manually pull them into their own stable branch. Typically, there are on the order of 100-200 commits to a FreeBSD stable branch a month, so this load can be quite large. In addition, except around the time a new branch is cut in FreeBSD, FreeBSD's current branch may have periods of instability and it can be quite difficult to know when a good time to branch might be as many of the stability or quality problems that are in FreeBSD's current branch often lay undiscovered for months or years because it doesn't get the intensity of testing that a FreeBSD stable branch receives.

In talking to different companies about how they implement this, they use many variations on this method. The ones that merge early and often report many advantages: easier to track down problems; easier to integrate in faster moving parts of FreeBSD; small amount of work often seems easier to schedule than one big chunk of work. The disadvantage of this comes when there's an incompatible change in FreeBSD that takes more time than the simple merge. If you have a fairly junior person doing the merging, they can have problems at these events. Companies report that having a senior person closely supervising the junior person works well to help smooth over these events.

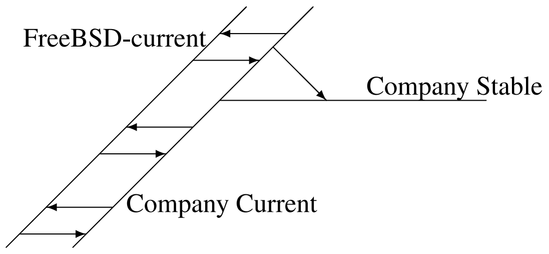

Figure 2: Relationship between FreeBSD-current and company branches

Figure 2 shows this graphically. This figure shows an idealized flow of patches into the company tree and back to FreeBSD. The two parallel current branches are shown diagonally, with the company's custom stable branch shown horizontally, much like Figures 1 and 2 presented above. No FreeBSD release points are included, since they are largely irrelevant to the method. The exact delta between the two current branches is also abstracted out, as this will ebb and flow over time and needlessly complicates the graph. The arrows represent changes being merged from one branch to another, either between the two current branches, or from the company's current branch to its stable branch.

4 SCM Tools

Managing the change for any of these methods can be a challenge. Modern Source Code Management (SCM) tools can help. There are two main types of SCM in use today. Centralized SCMs such as perforce, CVS and svn, and distributed SCMs such as svk, Mecurial (hg) and git. A complete tutorial on all these technologies is beyond the scope of this paper. However, methods for using some of these tools have been well documented. They will be presented here, with references to the original documentation as a resource to the reader.

4.1 svk

The svk[7] program is a decentralized version control system built on top of subversion[8]. Its primary purpose in life is to provide a detached mode of operation for subversion users, as well as providing more intelligent algorithms for merging changes between branches.

John Baldwin has an excellent writeup on how to use svk to mange what he calls FooBSD[9]. In the technical works, “foo” is a generic meta variable that substitutes to anything. This allows a high level of automation for users trying to track the FreeBSD tree. While John’s write up is centered on the “stable branch” method described above, it can easily be adapted to the “own branch” method as well.

4.2 git svn

git[10] is another distributed version control system. It was originally written by Linus Torvalds of Linux fame. It has heavy use in the Linux community, and known by many users in that community. It offers a different approach to version control synchronization than svk.

A complete tutorial of git is beyond the scope of this paper. The author notes that the documentation on how to use git svn[11] is extensive and easy to understand.

5 Acknowledgments

I would like to thank the crew at Timing Solutions: Ben Mesander, John Hein, Patrick Schweiger, Steve Passe, Marc Butler, Matthew Phillips, and Barb Dean for their insight and implementation of the ’Stable Branch Tracking’ method described in this paper. We deployed it across 4 major versions of FreeBSD.

I would like to thank Julian Elischer for the many conversations that we have had about development method. He provided much of the input into the ’Own Branching’ section.

I would like to acknowledge many others who contributed to the descriptions of the “grab and go” method on the condition of anonymity.

I’d like to thank John Baldwin for providing the community his experience with FooBSD.

References

[1] *FreeBSD Home Page*. n.d. FreeBSD Project. May 2009. http://www.freebsd.org/.

[2] *MySQL Database performance*. June 2008. The FreeBSD Project. May 2009. http://people.freebsd.org/ kris/scaling/mysql.html.

[3] *Introducing FreeBSD 7.0*. October 2007. The FreeBSD Project. May 2009. http://people.freebsd.org/ kris/scaling/7.0

[4] *Wireless Networking in the Open Source Community*. May 2006. The FreeBSD Project. May 2009. http://people.freebsd.org/ sam/SAN2006-WIRELESS.pdf.

[5] *FreeBSD Handbook*. n.d. FreeBSD Doc Project. May 2009. http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/.

[6] *Installing Applications: Packages and Ports*. n.d. FreeBSD Doc Project. May 2009. http://www.freebsd.org/doc/en/books/handbook/ports.html.

[7] *The SVK version control system*. n.d. Best Practices. July 2010. http://svk.bestpractical.com/view/HomePage.

[8] *Apache Subversion*. n.d. Apache Foundation. July 2010. http://subversion.apache.org/.

[9] *Using svk and svn to Maintain a FooBSD*. n.d. July 2010. http://blogs.freebsdish.org/jhb/2009/05/26/using-svk-foobsd/.

[10] *Git - Fast Version Control*. n.d. github hosting. July 2010. http://git-scm.com/.

[11] *git-svn(1) Manual Page*. n.d. kernel.org. July 2010. http://www.kernel.org/pub/software/scm/git/docs/git-svn.html.

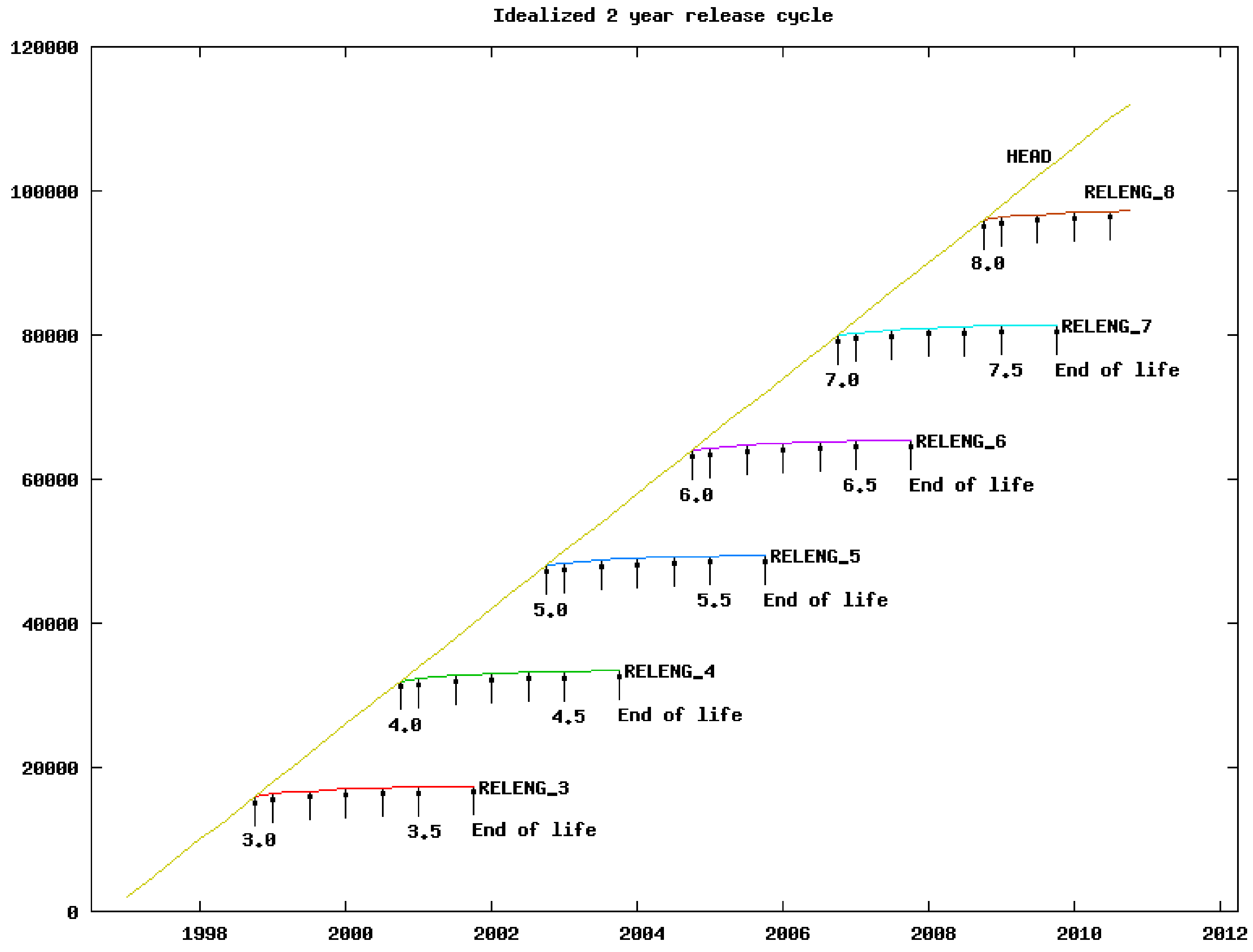

Figure 3: Idealized branching model

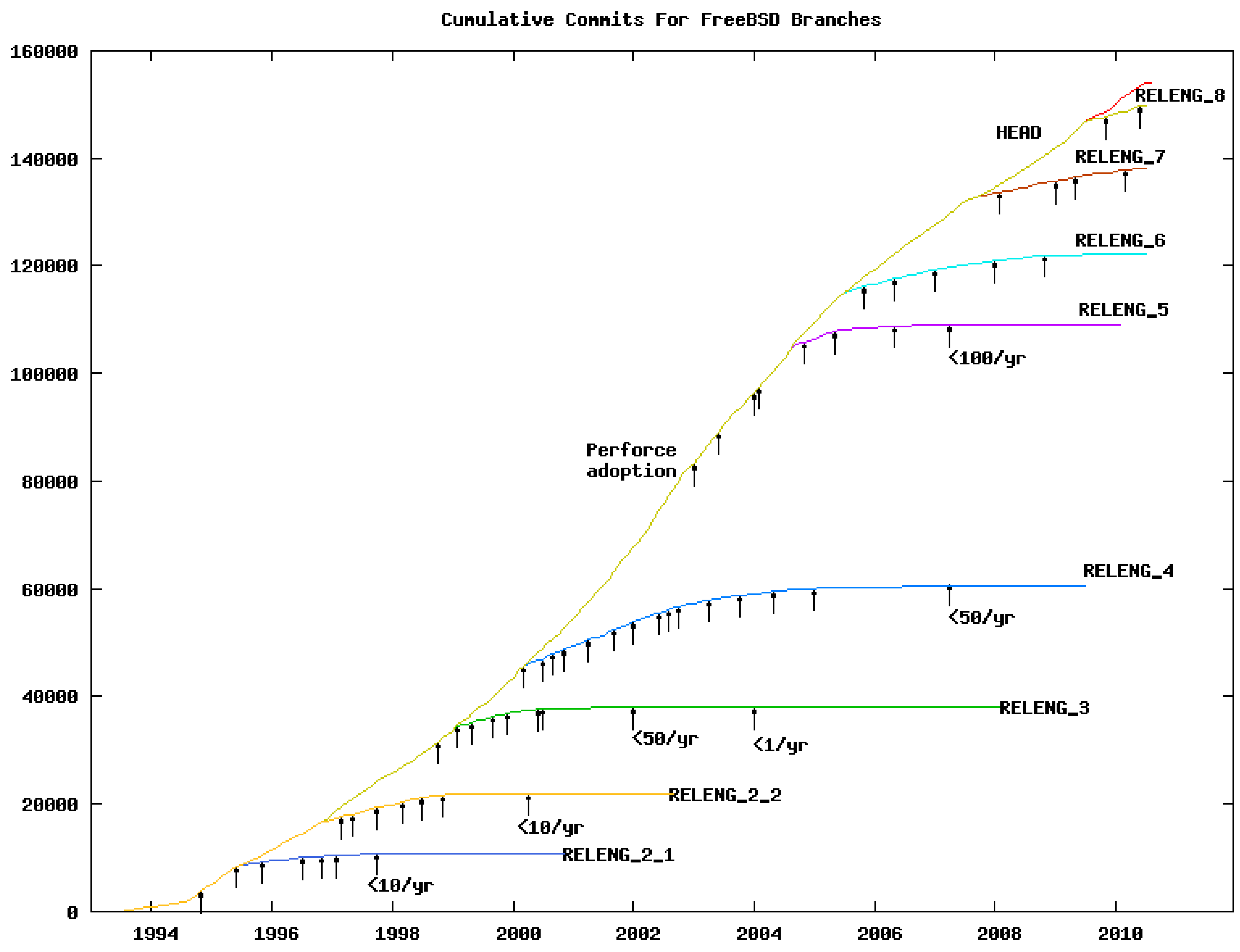

Figure 4: Actual FreeBSD branching history

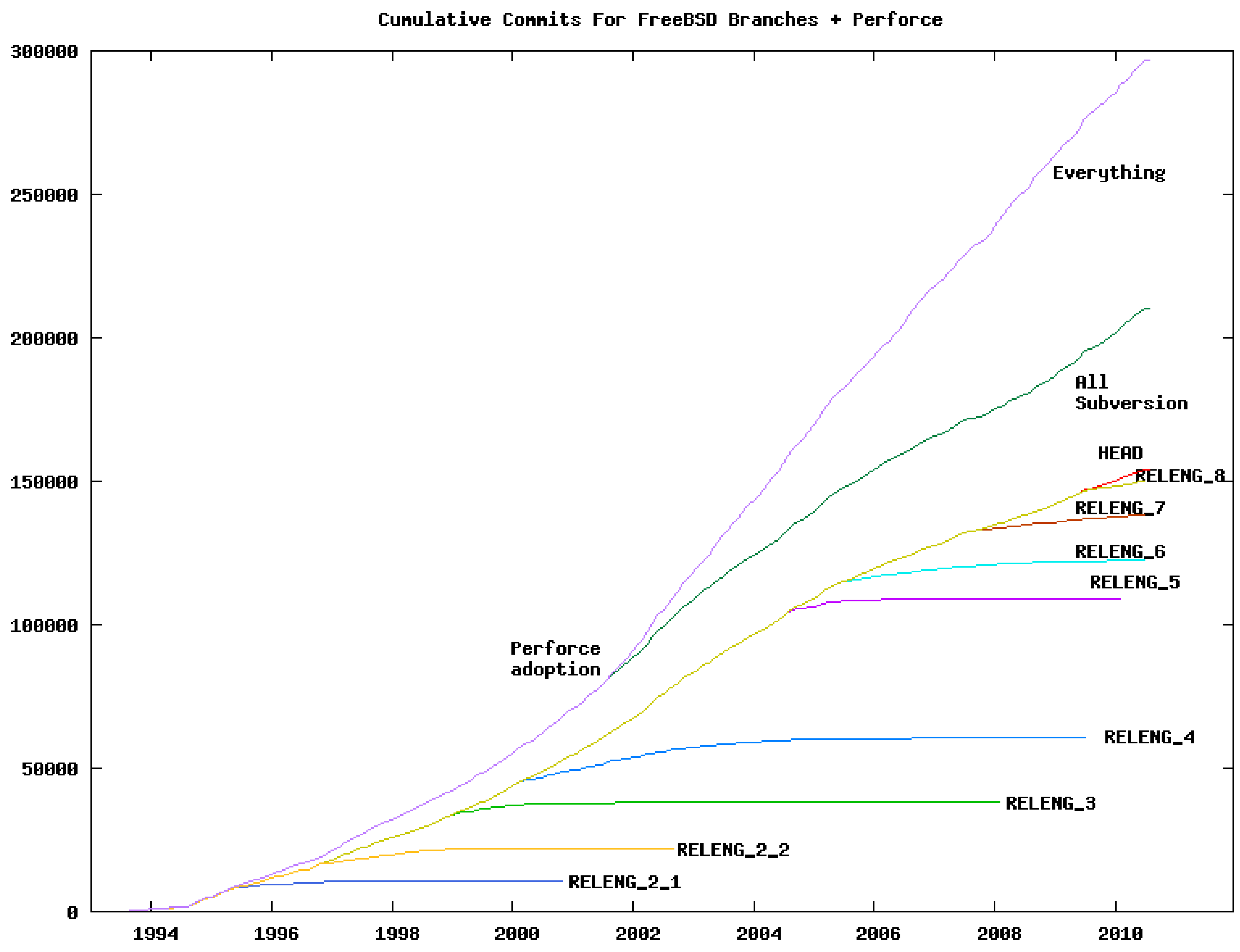

Figure 5: Commits to all release branches, FreeBSD-current and other development branches