AsiaBSDCon 2014
Proceedings

March 13-16, 2014

Tokyo, Japan

Published in Japan, March 2014

INDEX

INDEX

Visualizing Unix: Graphing bhyve, ZFS and PF with Graphite

Michael Dexter <editor@callfortesting.org>
AsiaBSDCon 2014, Tokyo, Japan

"Silence is golden", or so goes the classic Unix tenet and the result is that a traditional Unix system provides us only a command prompt while performing its most demanding tasks. While this is perfectly correct behavior, it provides the operator few insights into the strain that a given system may be experiencing or if the system is behaving unexpectedly. In this study we will explore a strategy for institutionally visualizing Unix system activity using collectd, its site-specific alternatives, Graphite, DTrace and FreeBSD. We have chosen FreeBSD because it includes a trifecta of Unix innovations: the bhyve hypervisor, the PF packet filter and the ZFS file system. While each of these tools provides its own facilities for displaying performance metrics, they collectively present a challenge to quantify their interaction.

Existing Facilities

Complementing the familiar yet verbose `top(1)`, `tcpdump(1)` and `gstat(8)` commands, bhyve, PF and ZFS each have their own dedicated tools for interactively and noninteractively displaying their activity metrics. The first of these, the bhyve hypervisor, includes the most limited quantification facility of the set. The `/usr/sbin/bhyvectl --get-stats --vm=<vm name>` command provides a summary of virtual machine (VM) operation with an emphasis on its kernel resource utilization but few with insights into its relative performance to the host.

The PF packet filter includes the `pflog(4)` logging interface for use with `tcpdump(1)` but the output is consistent with standard `tcpdump(1)` behavior, providing literally a view of active network connections. `tcpdump(1)` is complemented by other in-base tools such as `netstat(1)` but none make any effort to visualize the activity of a given network interface.

Finally, the ZFS file system benefits from in-base tools such as `gstat(8)` which introduce visual aids such as color coding of file system input/output operations and can be further complimented with tools like `sysutils/zfs-stats` but each nonetheless provides a simplistic summary of file system activity and performance.

Environment

While the subject of this study is the bhyve hypervisor itself, bhyve will also serve as the primary environment for the study. This choice was made because of bhyve's indistinguishable host and VM performance with CPU-intensive tasks, precluding the use of a more-familiar virtualization technology such as FreeBSD `jail(8)`. Virtual machines will be used for not only as participant hosts but can also be used for primary hosts as the statistics collection and graphing host. The choice of the bhyve hypervisor allows for this study to be conducted on FreeBSD 10.0 or later, PC-BSD 10.0 or later and its sister distribution TrueOS.

Testing Methodology

While the bhyve hypervisor is fully functional with the included `vmrun.sh` script found in `/usr/share/examples/bhyve/`, this study will elect to use the vmrc system found a `bhyve.org/tools/vmrc/`. vmrc is a virtual machine run command script that facilitates the full provisioning (formatting and OS installation) and management (loading, booting, stopping) of bhyve virtual machines. vmrc usage is as follows:

VM configuration files are located in `/usr/local/vm/` and can be installed with the included `install.sh` script.

vmrc usage is displayed with '`/usr/local/etc/rc.d/vm`' and a VM is provisioned with '`/usr/local/etc/rc.d/vm provision vm0`' which corresponds to the `vm0` configuration file found at `/usr/local/etc/rc.d/vm/vm0/vm0.conf` .

```
# /usr/local/etc/rc.d/vm
Usage: /usr/local/etc/rc.d/vm
[fast|force|one|quiet](start|stop|restart|rcvar|enabled|attach|boot
|debug|delete|destroy|fetch|format|grub|install|iso|jail|load|mount
|provision|status|umount)
```

vmrc's behavior is documented in the included `instructions.txt` file.

For our purposes, each VM will be simply provisioned with the '`provision`' directive and loaded and booted with the '`start`' directive. This study will require at a minimum one or more participant hosts that will be analyzed and optionally a VM for telemetry collection and graphing.

Graphite is a numeric time-series data and graphing system built in the Python scripting language and is best documented at `https://graphite.readthedocs.org` . Legacy and independent documentation sources exist but they vary in reliability and are rarely BSD Unix-oriented. The countless opportunities to customize a Graphite graphing deployments has resulted in a myriad of often site-specific documentation. For our purposes we will demonstrate a minimalistic "stock" Graphite installation that uses default settings whenever possible.

Graphite Components

Graphite is comprised of the `graphite-web` Django web framework "project" that performs all telemetry presentation tasks, the `carbon` telemetry collection daemon (scheduled to be replaced by the `ceres` daemon) and the `whisper` round-robin storage system. Host activity telemetry from one or more hosts is sent to the `carbon` daemon which in turn stores it in `whisper`, which retains or discards telemetry data according to pre-determined criteria, resulting in a fixed-size database after all criteria have been applied. The `whisper` storage system is modular and can be replaced with alternative stores and the `carbon` telemetry collection daemon is data source-agnostic. `carbon` is often supplied host telemetry using the `collectd` or `statsd` system statistics collection daemons but such data can also be provided with a myriad of other tools given that the telemetry data format is very simple:

`<host.data.name> <statistic> <UET timestamp>` which is fed into port `2003` of the `carbon` host using TCP or UDP.

`graphite-web` in turn presents the host telemetry stored in `whisper` in a user-friendly manner by default on port `8080` of the `graphite-web` host which is in turn often served via an established http daemon such as lighttpd, Apache or Nginx via the Python-based Web Server Gateway Interface (WSGI).

Complexity is the enemy of reliability…

…or installability for that matter. The operator will quickly find that the myriad of possible Graphite configurations will greatly complicate the installation and operation of Graphite with most documentation being of a site-specific and advanced-user nature. This study will mercifully provide a minimalistic “default” configuration but in doing so will provide one that is highly user-customizable in exchange for a negligible performance impact. The result is a “Unix” style configuration if you will that leverages familiar Unix conventions and established programmability.

Example host or virtual machine telemetry collection and transmission script

```
#!/bin/sh

host=$( uname -n ) # Used to distinguish the source within carbon
interval=3 # Sampling interval in seconds
destination=localhost # The carbon host
port=2003 # The default carbon port

while :; do # Operate continuously until terminated externally
        sleep $interval
        timestamp=$( date +%s )

# Average load within last minute from uptime(1)
load=$( uptime | grep -ohe 'load average[s:][: ].*' | \
        awk '{ print $3 }' | sed 's/,$//')
        echo "${host}.cpu.load $load $timestamp" | \
        socat - udp-sendto:$destination:$port
        echo "Sent: ${host}.cpu.load $load $timestamp" # Debug

# IOPS from gstat(8)
iops=$( gstat -b | grep ada0 | head -1 | \
        awk '{print $2}' )
        echo "${host}.ada0.iops $iops $timestamp" | \
        socat - udp-sendto:$destination:$port
        echo "Sent: ${host}.ada0.iops $iops $timestamp”
done
```

Example bhyve virtual machine telemetry syntax for the bhyve host:

```
bhyvectl --get-stats --vm=vm0 | \
grep "total number of vm exits" | awk '{print $6}'

bhyvectl --get-stats --vm=vm0 | \
grep "vm exits due to nested page fault" | awk '{print $8}'
```

Example of hard drive temperature monitoring using `sysutils/smartmontools`**:**

```
smartctl -a /dev/ada0 | grep 'Temperature_Celsius' | cut -c1-3
```

Sources of DTrace examples when support is enabled in the kernel:

```
https://wiki.freebsd.org/DTrace
http://www.brendangregg.com/DTrace/dtrace_oneliners.txt
```

Graphite installation script

```
#!/bin/sh
# This must be run as root

# Allow automatic pkg bootstrap
env ASSUME_ALWAYS_YES=YES pkg update

cd /root/

pkg install -y devel/git lang/python databases/py-sqlite3 \
devel/py-pip graphics/py-cairo x11-fonts/xorg-fonts-truetype
net/socat sysutils/tmux

pip install Django django-tagging Twisted pytz pyparsing

git clone https://github.com/graphite-project/graphite-web.git
git clone https://github.com/graphite-project/whisper.git
git clone https://github.com/graphite-project/carbon.git

cd graphite-web ; python setup.py install
cd ../whisper ; python setup.py install
cd ../carbon ; python setup.py install

cp /opt/graphite/webapp/graphite/local_settings.py.example
/opt/graphite/webapp/graphite/local_settings.py

# Pardon the line-wrap
python /usr/local/lib/python2.7/site-packages/django/bin/django-
admin.py syncdb --pythonpath=/opt/graphite/webapp/ --
settings=graphite.settings --noinput

cp /opt/graphite/conf/carbon.conf.example \
/opt/graphite/conf/carbon.conf

# Optionally send via UDP, pardon the line-wrap
sed -i '' -e 's/ENABLE_UDP_LISTENER = False/ENABLE_UDP_LISTENER =
True/' /opt/graphite/conf/carbon.conf

cp /opt/graphite/conf/storage-schemas.conf.example \
/opt/graphite/conf/storage-schemas.conf

mkdir -p /opt/graphite/storage/log/carbon-cache/carbon-cache-a

echo "To start carbon and graphite-web, run:"
echo "python /opt/graphite/bin/carbon-cache.py start"
echo "python /opt/graphite/bin/run-graphite-devel-server.py
/opt/graphite/"
echo
echo "To set the administrator credentials, run:"
echo "python /usr/local/lib/python2.7/site-
packages/django/bin/django-admin.py flush --
pythonpath=/opt/graphite/webapp/ --settings=graphite.settings"
```

Optional collectd installation (note its ZFS ARC Plugin)

```
#!/bin/sh
pkg install -y net-mgmt/collectd5
cat > /usr/local/etc/collectd.conf.diff <<-EOF
14c14
< FQDNLookup   false
---
> #FQDNLookup   true
180c180
< LoadPlugin write_graphite
---
> #LoadPlugin write_graphite
631,632c631,632
< #  Server "ff18::efc0:4a42" "25826"
<    <Server "localhost" "2003">
---
>    Server "ff18::efc0:4a42" "25826"
>    <Server "239.192.74.66" "25826">
1125,1137c1125,1137
< <Plugin write_graphite>
<   <Node "example">
<     Host "localhost"
<     Port "2003"
<     Protocol "udp"
<     LogSendErrors true
<     Prefix "collectd"
<     Postfix "collectd"
<     StoreRates true
<     AlwaysAppendDS false
<     EscapeCharacter "_"
<   </Node>
< </Plugin>
---
> #<Plugin write_graphite>
> #  <Node "example">
> #    Host "localhost"
> #    Port "2003"
> #    Protocol "udp"
> #    LogSendErrors true
> #    Prefix "collectd"
> #    Postfix "collectd"
> #    StoreRates true
> #    AlwaysAppendDS false
> #    EscapeCharacter "_"
> #  </Node>
> #</Plugin>
EOF

patch -f -R /usr/local/etc/collectd.conf \
/usr/local/etc/collectd.conf.diff
```

Administrative Options

Observe that all telemetry data collection in this strategy is obtained using in-base tools such as `uptime(1)` and `gstat(8)`. This decision both leverages and bridges the traditional Unix system activity reporting tools and eliminates dependency and compatibility concerns. Furthermore, it reveals a direct path to creating lower-overhead tools in the C programming language through the extraction of the algorithms within those tools and creating purpose-built telemetry transmitters. Should additional telemetry be desired, familiar `sysctls, dtrace(1)` and tools such as `sysutils/smartmontools` can provide subsystem-specific telemetry such as hard disk temperature with a myriad of parsing options.

With this in mind, we can turn to third party tools to enhance the operator's experience, namely by sending the output of our telemetry transmitting and graphite-web commands to a terminal multiplexer such as `tmux(1)`:

```
tmux new -d -s telemetry "sh send-telemetry.sh"
tmux new -d -s graphite "python /opt/graphite/bin/run-graphite …"
```

These can be accessed with:

```
tmux attach-session -t telemetry
tmux attach-session -t graphite
```

Future Directions

To leverage in-base tools is also to expose their shortcomings. While newer tools such as `bhyvectl(8)` and `zfs(8)` include "`get`" arguments that can produce specific metrics, this new convention has not been backported to well-established tools and to do so may be a worthy pursuit. Similarly, many performance-related utilities have captive user interfaces with only some such as `top(1)` featuring "batch" modes in which they output a telemetry snapshot for easy parsing.

Conclusions

The flexibility of this Unix system activity visualization strategy allows hosts to be monitored with as little as a few-line site-specific shell script or a third-party system statistics collection daemon in accordance to the operator's requirements. For multi-layer environments such as `bhyve(8)` and `jail(8)` virtualization stacks, this strategy can provide host and virtual machine telemetry collection in a performant, low-overhead manner and in fact contain the `graphite-web` graphing function in a virtual machine or `jail(8)`. By employing this strategy, system operators and developers can literally see the interaction of any system component that includes a corresponding reporting utility. In time, this strategy will hopefully result in the institutional availability of queryable reporting utilities, resulting in not only lower overhead at telemetry collection time but also the elimination of complex output parsing. With customization and refinement of this tool set, the systems operator and developer can experience a more visceral understanding of system performance that is the antithesis of the stark Unix command line.

LLVM in the FreeBSD Toolchain

David Chisnall

1 Introduction

FreeBSD 10 shipped with Clang, based on LLVM [5], as the system compiler for x86 and ARMv6+ platforms. This was the first FreeBSD release not to include the GNU compiler since the project's beginning. Although the replacement of the C compiler is the most obvious user-visible change, the inclusion of LLVM provides opportunities for other improvements.

2 Rationale for migration

The most obvious incentive for the FreeBSD project to switch from GCC to Clang was the decision by the Free Software Foundation to switch the license of GCC to version 3 of the GPL. This license is unacceptable to a number of large FreeBSD consumers. Given this constraint, the project had a choice of either maintaining a fork of GCC 4.2.1 (the last GPLv2 release), staying with GCC 4.2.1 forever, or switching to another compiler. The first option might have been feasible if other GCC users had desired the same and the cost could have been shared. The second was an adequate stopgap, but the release of the C11 and C++11 specifications—both unsupported by GCC 4.2.1—made this an impossible approach for the longer term. The remaining alternative, to find a different compiler to replace GCC, was the only viable option.

The OpenBSD project had previously investigated PCC, which performed an adequate job with C code (although generating less optimised code than even our old GCC), but had no support for C++. The TENDRA compiler had also been considered, but development had largely stopped by 2007.

The remaining alternative was Clang, which was still a very young compiler in 2008, but had some significant commercial backing from companies including Apple and Google. In 2009, Roman Divacky and Pawel Worach begin trying to build FreeBSD with Clang and quickly got a working kernel, as long as optimisations were disabled. By May 2011, Clang was able to build the entire base system on both 32-bit and 64-bit x86 and so became a viable migration target. A large number of LLVM Clang bugs were found and fixed as a result of FreeBSD testing the compilation of a large body of code.

3 Rebuilding the C++ stack

The compiler itself was not the only thing that the FreeBSD project adopted from GCC. The entire C++ stack was developed as part of the GCC project and underwent the same license switch. This stack comprised the C++ compiler (g++), the C++ language runtime (`libsupc++`) and the C++ Standard Template Library (STL) implementation (`libstdc++`).

All of these components required upgrading to support the new C++11 standard. The runtime library, for example, required support for dependent exceptions, where an exception can be boxed and rethrown in another thread (or the same thread later).

The FreeBSD and NetBSD Foundations jointly paid PathScale to open source their C++ runtime library (libcxxrt), which was then integrated into the FreeBSD base system, replacing `libsupc++`. The LLVM project provided an STL implementation (`libc++`), with full C++11 and now C++14 support, which was duly integrated.

Using libcxxrt under `libstdc++` allowed C++ libraries that exposed C interfaces, or C++ interfaces that didn't use STL types, to be mixed in the same binary as those that used `libc++`. This includes throwing exceptions between such libraries.

Implementing this in a backwards-compatible way required some linker tricks. Traditionally, `libsupc++` had been statically linked into `libstdc++`, so from the perspective of all linked programs the `libsupc++` symbols appeared to come from `libstdc++`. In later versions in the 9.x series, and in the 9-COMPAT libraries shipped for 10, `libstdc++` became a filter library, dynamically linked to `libsupc++`. This allows symbol resolution to work correctly and allows `libsupc++` or `libcxxrt` to be used as the filtee, which actually provides the implementation of these symbols.

4 Problems with ports

The FreeBSD ports tree is a collection of infrastructure for building around 24,000 third-party programs and libraries. Most ports are very thin wrappers around the upstream distribution's build system, running autoconf or CMake configurations and then building the resulting make files or equivalent. For well-written programs, the switch to Clang was painless. Unfortunately, well-written programs make up the minority of the ports tree. To get the ports tree working with Clang required a number of bug fixes.

4.1 Give up, use GCC

The first stopgap measure was to add a flag to the ports tree allowing ports to select that they require GCC. At the coarsest granularity is the `USE_GCC` flag knob, which allows a port to specify that it requires either a specific version of GCC, or a specific minimum version.

This is a better-than-nothing approach to getting ports building again, but is not ideal. There is little advantage in switching to a new base system compiler if we are then going to use a different one for a large number of ports. We also encounter problems due to GCC's current inability to use `libc++`, meaning that it is hard to compile C++ ports with GCC if they depend on libraries that are built with Clang, and vice versa. Currently around 1% of the ports tree requires this. Quite a few more use the flags exposed in the `compiler` namespace for the port's `USES` flags. In particular, specifying `USES=compiler:openmp` will currently force a port to use GCC, as our Clang does not yet include OpenMP support.

This framework allows ports to specify the exact features of GCC that they require, allowing them to be switched to using Clang once the

4.2 The default dialect

One of the simplest, but most common, things to fix was the assumption by a lot of ports that they could invoke the `cc`, program and get a C89 compiler. POSIX97 deprecated the `cc` utility, because it accepts an unspecified dialect of C, which at the time might have been K&R or C89. Over a decade later, some code is still trying to use it. Today, it may require K&R C (very rare), C89 (very common), C99 (less common), or C11 (not yet common), and so should be explicitly specifying a dialect. This was a problem, because `gcc`, when invoked as `cc` defaults to C89, whereas `clang` defaulted to C99 and now to C11.

This is not usually an issue, as the new versions of the C standard are intended to be backwards compatible. Unfortunately, although valid C89 code is usually valid C99 or C11 code, very little code is actually written in C89. Most C ports are written in C plus GNU extensions. In particular, C99 introduced the **inline** keyword, with a different meaning to the **inline** keyword available as a GNU extension to C89. This change causes linker failures when C89 code with GNU-flavoured inline functions is compiled as C99. For most ports, this was fixed by adding `-fgnu89-inline` to the port's CFLAGS.

4.3 C++ templates

Another common issue in C++ code relates to two-phase lookup in C++ templates. This is a particularly tricky part of the C++ stack and both GCC and Microsoft's C++ compiler implemented it in different, mutually incompatible, wrong ways. Clang implements it correctly, as do new versions of other compilers. Unlike other compilers, Clang does not provide a fallback mode, accepting code with GNU or Microsoft-compatible errors.

The most common manifestation of this difference is template instantiations failing with an unknown identifier error. Often these can be fixed by simply specifying **this**$->$ in front of the variable named in the error message. In some more complex programs, working out exactly what was intended is a problem and so fixing it is impossible for the port maintainer.

This is currently the largest cause of programs requiring GCC. In particular, some big C++ projects such as the Sphinx speech recognition engine have not had new releases for over five years and so are unlikely to be fixed upstream. Several of these ports will only build with specific version of GCC as well and so are still built with GCC in the ports tree. Fortunately, many these (for example, some of the KDE libraries) are now tested upstream with Clang for Mac OS X compatibility and so simply updating the port to a newer version fixed incompatibilities.

4.4 Goodbye tr1

C++ Technical Report 1 (TR1) is a set of experimental additions to C++ that were standardised in between C++03 and C++11. It provided a number of extensions that were in headers in the `tr1/` directory and in the std :: tr1 namespace. In C++11, these were moved (with some small modifications) into the standard header directory and namespace.

The new C++ stack is a full C++11 implementation and does not provide the TR1 extensions to C++98. This means that code that references these will fail, complaining about a missing header. The simple fix for this is just to globally delete tr1 from the source files. Getting the code to also build with GCC is somewhat more problematic, but can be accomplished with a relatively small set of **#ifdef**s.

4.5 _Generic problems

In FreeBSD 10, we improved some of the generic macros in `math.h` to use the C11 _Generic expressions or GCC's type select extension if available. The old code dispatched arguments to the correct function by comparing **sizeof**(arg) against **sizeof**(**double**) and so on. Now, we are able to explicitly match on the type. Macros such as isnan() and isinf () will now raise compile-time errors if they are invoked with a type that is not one of the compatible ones.

This is something that we consider a feature. If you pass an **int** to isnan(), then you probably have a bug because there are no possible values of an **int** that are not numbers. Unfortunately, a surprising amount of code depends on the previous buggy behaviour. This is particularly prevalent in configure scripts. For example, Mono checks whether isnan(1) works, which checks whether there is a version of isnan() that accepts an integer argument. If it doesn't find one, then it provides an implementation of isnan() that accepts a **double** as the argument, which causes linker failures.

Fixing these was relatively easy, but time consuming. Most of the errors were in configure scripts, but we did find a small number of real bugs in code.

4.6 OpenMP

One of the current limitations of Clang as a C/C++ compiler is its lack of OpenMP support. OpenMP is a pragma-based standard for compiler-assisted parallelism and so is increasingly important in an era when even mobile devices have multiple cores. Intel has recently contributed an OpenMP implementation to Clang, but the code has not yet been integrated. This implementation also includes a permissively licensed OpenMP runtime, which would replace the GNU OpenMP library (`libgomp`).

Work is currently underway to finish importing the OpenMP support code into Clang. This is expected to be completed by LLVM 3.5, although some extra effort may be required to build the OpenMP support library on FreeBSD (Linux and Mac OS X are its two current supported configurations).

5 Looking forwards

Having a mature and easily extensible library-based compiler infrastructure in the base system provides a number of opportunities.

5.1 Install-time optimisation

A common misconception of LLVM, arising from the VM in its name, is that it would allow us to easily compile code once and run it on all architectures. LLVM uses an intermediate representation (IR) in the middle of the compiler pipeline. This is not intended as a distribution format or as a platform-neutral IR, in contrast to .NET or Java bytecode. This is an intrinsic problem for any target for C compilation: once the C preprocessor has run, the code is no longer target-neutral and much C code has different paths for things like byte order or pointer size.

Although LLVM IR is not architecture neutral, it is *microarchitecture* neutral. The same LLVM IR is generated for a Xeon and an Atom, however the optimal code for both is quite different. It would be possible for a significant number of ports to build the binary serialisation of LLVM IR ('bitcode') and ship this in packages. At install time, the `pkg` tool could then optimise the binaries for the current architecture.

To avoid long install times, packages could contain both a generic binary and the IR, allowing the IR to be stripped for people who are happy to run the generic code, or used for optimisation as a background task if desired. It's not clear how much overhead this would add to installation. Building large ports can be time consuming, however the slowest to build are typically C++ ports where the build time is dominated by template expansion. Generating a few megabytes of optimised object code from LLVM IR typically only takes a few seconds on a modern machine.

Microarchitectural optimisations are not the only applicable kind that could benefit from this approach. Link-time optimisation can give a significant speedup by doing interprocedural analysis over an entire program and using these results in optimisation. Typically, the boundary for this is a shared library, because you can not rely on code in a shared library not changing. If we are shipping both LLVM IR and binaries, however, it becomes possible to specialise shared libraries for specific executables, potentially generating much better code. The down side of this is that you end up without code shared between users of a library, increasing cache churn.

Fortunately, there is information available on a system about whether this is likely to be a good trade. The package tool is aware of how many programs link to a specific library and so can provide hints about whether reduction in code sharing is likely to be a problem. If you have a shared library that is only used by a single program, obviously you don't get any benefits from it. The kernel may also be able to profile how often two programs using the same library are running simultaneously (or after a short period) and so gaining any benefit from the sharing.

Of course, these are just heuristics and it may be that some library routines are very hot paths in all of their consumers and so would benefit from inlining anyway.

5.2 Code diversity

LLVM has been used by a number of other projects. One interesting example is the Multicompiler [3], which implements code diversity in LLVM with the goal of making return-oriented programming (ROP) more difficult. ROP turns the ability for an attacker to run a small amount of arbitrary code (e.g. control the target a single jump, such as a return instruction) into the ability to run large amounts of code. This works by stringing together short sequences of instructions ('gadgets') in a binary, connected by jumps. Gadgets are particularly common in code on x86, because the variable-length instruction encoding and byte alignment of instructions mean that a single instruction or instruction pair can have a number of different meanings depending on where you start interpreting it.

The Multicompiler combats this in two ways. First, it can insert nops into the binary, breaking apart particularly dangerous accidental sequences. Second, using a random seed, it performs various permutations of the code, meaning that different compiles can end up with code (including the surviving gadgets) in different places.

We are currently working to incorporate the multicompiler into the ports tree, so that users building site-local package sets can set a random seed and get deterministic builds that are nevertheless different in binary layout to those produced by everyone else. This makes generating an exploit that will work on all FreeBSD systems very difficult. We will also be able to incorporate this into the FreeBSD-provided binary packages, quickly running diversified builds when a vulnerability is found, requiring attackers to create new versions of their exploits. By rolling these out in a staggered fashion, we can make it hard to write an exploit that will work on all FreeBSD users, even within a single package version.

5.3 Sanitiser support

Clang, on Linux and Mac OS X, supports a number of 'sanitisers', dynamic checkers for various kinds of programming error. The compiler identifies particular idioms and inserts checks that are evaluated at run time and may potentially call routines in a supporting library. These include:

AddressSanitizer was the first of the family and is intended to provide similar functionality to Valgrind [6], with a much lower overhead. It detects out-of-bounds accesses, use-after-free and other related memory errors.

MemorySanitizer checks for reads of uninitialised memory. This catches subtle bugs where code can work fine on one system because memory layout happens to contain valid values, but fail on another.

ThreadSanitizer is intended to detect data races.

UndefinedBehaviorSanitizer performs run-time checks on code to detect various forms of undefined behaviour. This includes checking that **bool** variables only contain **true** or **false** values, that signed arithmetic does not overflow, and so on. This is very useful for checking portable code, as undefined behaviour can often be implemented in different ways on different platforms. For example, integer division by zero may trap on some architectures but may silently give a meaningless result on others.

DataFlowSanitizer allows variables to be labelled and their flow through the program to be tracked. This is an important building block for a category security auditing tools.

All of these require a small runtime library for supporting functionality, including intercepting some standard C library functions (e.g. malloc() and free()). These have not yet been ported to FreeBSD, but would provide significant benefits if they were. In particular, running the FreeBSD test suite with dynamic checks enabled on a regular basis would allow early detection of errors.

5.4 Custom static checks

The Clang static analyser provides generic functionality for understanding control and data flow inside compilation units. It also includes a number of checkers for correct usage of the relevant languages, for example checking that variables are not used uninitialised and NULL pointers are not dereferenced in all possible control flows. The more useful checks are those that incorporate some understanding of API behaviour.

By default, the analyser can check for correct usage of a number of POSIX APIs. Apple has also contributed a number of checkers for OS X kernel and userspace APIs. The framework is sufficiently generic that we can also provide plugins for FreeBSD APIs that are commonly misused.

Some of the checkers would be of more use if we provided more annotation in the FreeBSD code. For example, WITNESS allows dynamic lock order checking, but Clang can also perform some of these checks statically. It can also do some more subtle checks, for example ensuring that every access to a particular structure field has a specific lock acquired. Ideally, the static analyser would be combined with WITNESS, to elide run-time checks where static analysis can prove that they are not required.

5.5 Other analysis tools

The LLVM framework has been used to implement a number of other analysis tools. Of particular relevance to FreeBSD are SOAAP [4] and TESLA [1], which were both developed at the University of Cambridge with FreeBSD as the primary target.

TESLA is a framework for temporal assertions, allowing the programmer to specify things that must have happened (somewhere) before a line of code is reached, or which must happen subsequently. A typical example is that within a system call, by the time you get to the part doing I/O, some other code must have already performed a MAC check and then something else must later write an audit log event. These complex interactions are made harder to understand by the fact that the kernel can load shared libraries. TESLA uses Clang to parse temporal assertions and LLVM to instrument the generated code, allowing them to be checked at run time. A number of TESLA assertions were added to the FreeBSD kernel in a branch and used to validate certain parts of the system.

SOAAP is a tool to aid compartmentalising software. This is an important mitigation technique, limiting the scope of compromises. The Capsicum [9] infrastructure provides the operating system functionality required for running low-privilege sandboxes within an application but deciding where to place the partitions is still a significant engineering challenge. SOAAP is designed to make it easy to explore this design space, by writing compartmentalisation hypotheses, ensuring that all shared data really are shared, and simulating the performance degradation from the extra process creation and communication.

We anticipate a lot more tools along these lines being developed over the coming years and intend to take advantage of them.

5.6 The rest of the toolchain

We currently include code from either GNU binutils or the ELF Toolchain project. Most of this duplicates functionality already in LLVM. In particular, every LLVM back end can parse assembly and generate object code, yet we still have the GNU assembler. Various other tools, such as `objdump` have direct replacements available in LLVM and some others (e.g. `addr2line` would be simple wrappers around the LLVM libraries). The only complex tool is the linker.

There are two possible linkers available, both based on LLVM: MCLinker [2] and lld [8]. MCLinker is currently able to link the entire FreeBSD base system on i386, but lacks support for version scripts and so the resulting binaries lack symbol versions. It is a promising design, performing well in terms of memory usage and speed.

Lld is developed primarily by Sony and is part of the LLVM project. It is currently less mature, but is advancing quickly. Both use a scalable internal representation, with some subtle differences, inspired in part by Apple's 64-bit linker. MCLinker aims to be a fast ELF-only linker, whereas lld aims to link all of the object code formats supported by LLVM (ELF, Mach-O and PE/COFF). We are likely to import one of these in the near future.

We have already imported LLDB, the LLVM debugger, into the base system, although it was not quite ready in time for the 10.0 release. LLDB uses numerous parts of LLVM. When you type an expression into the GNU debugger command line, it uses its internal parser, which supports a subset of the target language. In LLDB, the expression is parsed with Clang. The parsing libraries in Clang provide hooks for supplying declarations and these are supplied by LLDB from DWARF debug information. Once it's parsed, the Clang libraries emit LLVM IR and the LLVM JIT produces binary code, which is copied into the target process's address space and executed.

5.7 Other compilers in FreeBSD

When people think of compilers in FreeBSD, the C and C++ compilers are the most obvious ones. There are a number of others, for domain-specific languages, in various places. For example, the Berkeley Packet Filter (BPF) contains a simple hand-written JIT compiler in the kernel. This produces code that is faster than the interpreter, but not actually very good in absolute terms.

Having a generic compiler infrastructure for writing compilers allows us to replace some of these with optimising compilers. In a simple proof of concept for an LLVM-based BPF JIT (a total of under 500 lines of code, implementing all of the BPF bytecode operations), we were able to generate significantly better code than the current in-kernel JIT. The LLVM-based JIT in its entirety (excluding LLVM library code) was smaller than the platform-dependent code in the in-kernel JIT and will work for any architecture that LLVM supports, whereas the current JIT only supports x86[-64].

It is not a simple drop-in replacement, however. LLVM is large and does not gracefully handle low-memory conditions, so putting it inside the kernel would be a terrible idea. There are two possible solutions to this. The first is to run the JIT in userspace, with the kernel streaming BPF bytecodes to a device that a userspace process reads, compiles, and then writes the generated machine code back into. The kernel can use the interpreter for as long as the userspace process takes to perform the compilation. The alternative is to use the NetMap [7] infrastructure to perform packet filtering entirely in userspace.

This is less attractive for BPF, where rule sets tend to be fairly simple and even the interpreter is often fast enough. It is more interesting for complex firewall rules, which change relatively infrequently (although the state tables are updated very often) and which can be a significant bottleneck.

6 Platform support

FreeBSD currently supports several architectures. We have enabled Clang/LLVM by default on x86, x86-64, and ARMv6 (including ARMv7). This leaves older ARM chips, SPARC, PowerPC, MIPS, and IA64 still using GCC. Support is progressing in LLVM for SPARC, PowerPC and MIPS.

We are able to compile working PowerPC64 kernels without optimisation, but there are still some optimisation bugs preventing Clang from becoming the default compiler on this architecture. On 32-bit PowerPC, LLVM still lacks full support for thread-local storage and position-independent code. SPARC support is progressing in LLVM, but it has not been recently tested.

We are currently compiling significant amounts of MIPS code (including FreeBSD libc) with LLVM and a large patch set. This includes significant improvements to the integrated assembler, but also support for MIPS IV. Currently, LLVM supports MIPS32, MIPS32r2, MIPS64 and MIPS64r2. The earlier 64-bit MIPS III and MIPS IV ISAs are still widespread. The changes required to support these in the back end are not very complex: simply disable the instructions that are not present in earlier ISA revisions. They should be upstreamed before LLVM 3.5 is released.

The (unfinished) IA64 back end in LLVM was removed due to lack of developer interest. It is unlikely that this architecture will ever be supported in LLVM, and it is doubtful that it has a long-term future in FreeBSD, as machines that use it are rare, expensive, and unlikely to be produced in the future.

7 Summary

Importing LLVM and Clang into the FreeBSD base system and switching Tier 1 platforms to use it was a significant amount of effort. So far, we have only just started to reap the benefits of this work. Over the next few years, LLVM is likely to be an important component of the FreeBSD base system.

This paper has outlined a few of the possible directions. It is likely that there are more that are not yet obvious and will emerge over time.

8 Acknowledgements

The task of importing LLVM into FreeBSD was begun by Roman Divacky and Pawel Worach in 2009 and without their initiative none of this work would have been possible. Since then, a large number of FreeBSD developers have worked to improve the state of the LLVM toolchain in FreeBSD, including Dimitry Andric, Ed Maste, Brooks Davis, Ed Schouten, and many others.

Portions of this work were sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advan ced Research Projects Agency or the Department of Defense.

References

[1] Temporally enhanced security logic assertions (TESLA). `http://www.cl.cam.ac.uk/research/security/ctsrd/tesla/` (accessed 31/1/2014).

[2] Chinyen Chou. MCLinker BSD. In *BSDCan*, 2013.

[3] Michael Franz, Stefan Brunthaler, Per Larsen, Andrei Homescu, and Steven Neisius. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[4] Khilan Gudka, Robert N. M. Watson, Steven Hand, Ben Laurie, and Anil Madhavapeddy. Exploring compartmentalisation hypotheses with soaap. In *Proceedings of the 2012*

IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW '12, pages 23–30, Washington, DC, USA, 2012. IEEE Computer Society.

[5] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[7] Luigi Rizzo and Matteo Landi. Netmap: Memory mapped access to network devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 422–423, New York, NY, USA, 2011. ACM.

[8] Michael Spencer. lld - the LLVM Linker. In *EuroLLVM*, 2012.

[9] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

NPF - progress and perspective

Mindaugas Rasiukevicius
The NetBSD Project
rmind@netbsd.org

January 2014

Abstract

NPF – is a NetBSD packet filter which can do TCP/IP traffic filtering, stateful inspection and network address translation with a development focus on performance, scalability and modularity. The packet filter was first introduced with the NetBSD 6.0 release. This paper and the presentation will cover the main NPF features: its filtering engine, stateful inspection and NAT, application level gateways (ALGs), tables and extensions. It will give an overview of some internals, e.g. key data structures used in NPF and the rationale behind some design decisions. Finally, the latest developments in NetBSD -current will be discussed, including: addition of BPF coprocessor and just-in-time (JIT) compilation which lead to NPF being fully switched to BPF byte-code, support for tables which use CDB with perfect hashing, stateless NAT with NPTv6 and work in progress towards lockless state lookup.

1 Introduction

NPF was introduced with the NetBSD 6.0 release. The expectations for the first release were to let users try the technology, get wider testing and thus get the core features stable, as well as gather general feedback. There have been many improvements, fixes and developments since then. In this section we will briefly review the main features of NPF. Then will take a look at the recent developments in section 2. This work will be appear in the NetBSD 7.0 release. Finally, the work in progress will be discussed in section 3.

Some understanding and knowledge of syntax will be assumed in the examples. Comprehensive information about NPF capabilities, configuration and the syntax can found in the NPF documentation web page [npf14], as well as npf.conf(5) and other manual pages.

1.1 Multi-core scalability

NPF was designed and developed with a focus on high performance and scalability. Multi-core systems became prevalent in the last decade. In 2009, the Linux Netfilter project announced nftables, with one of the main features being: *"the core is completely lockless ..."* [McH09]. At that time there was no SMP-optimised packet filter in *BSD. The NPF idea was partly a BSD response to nftables. The main motivation was to design a packet filter for multi-core systems from the very beginning, as well as use byte-code based processing to have protocol independence and flexibility. For example, L7 filtering can be done without any kernel modifications.

To achieve high concurrency, NPF employs various lockless techniques, e.g. ruleset inspection uses passive serialisation[1] and implements completely lockless processing. The details of state lookup will be discussed in further sections. Large IP sets can be stored in the tables for very efficient and concurrent lookups, which will be discussed in section 2.2. Other components of NPF where very high concurrency is not a concern use fine-grained locking.

1.2 Stateful filtering

NPF supports stateful filtering – a required feature for any modern packet filter. It performs full tracking of TCP connections. This means tracking and

[1] Similar concept to RCU, but patent-free.

inspecting not only the source and destination IP addresses and port numbers, but also TCP state, sequence numbers and window sizes [Roo01]. The current data structures to store tracked connections will be discussed in section 3.1.

1.3 NAT

Another major feature is network address translation. Currently, NPF supports dynamic (stateful) NAT which can perform network address port translation (NAPT, also known as masquerading), as well as other variations of inbound and outbound NAT, including bi-directional NAT. An interface for application level gateways (ALGs) is provided to supplement NAT, e.g. detect traceroute packets and perform the address translation in the embedded payload.

1.4 Dynamic rules

NPF has gained support for dynamic rules with the NetBSD 6.1 release. This allows us to add or remove rules at run-time without reloading the whole configuration, in a similar way to Linux *iptables*. For example:

```
$ npfctl rule "test-set" add \
    block proto icmp from 192.168.0.6
OK 1
$ npfctl rule "test-set" list
block proto icmp from 192.168.0.6
$ npfctl rule "test-set" add \
    block from 192.168.0.7
OK 2
$ npfctl rule "test-set" list
block proto icmp from 192.168.0.6
block from 192.168.0.7
$ npfctl rule "test-set" rem \
    block from 192.168.0.7
$ npfctl rule "test-set" rem-id 1
$ npfctl rule "test-set" list
```

Each rule gets a unique identifier which is returned on addition. As shown in the example, the rules can be removed using the identifier or by passing the exact filter criteria. In the latter case, the SHA1 hash is computed on a rule metadata to identify it.

1.5 Modularity and extensions

Another focus of NPF development was modularity and extensibility. Each component in NPF is abstracted and has its own strict interface. *Rule procedures* in NPF are the main interface to implement custom extensions. An extension consists of two parts: a dynamic module (.so file) supplementing the npfctl(8) utility and a kernel module. The syntax of npf.conf supports arbitrary procedures with their parameters, as supplied by the modules. It takes about 160 lines of code, including the comments and a license text, for a demo extension which blocks an arbitrary percentage of traffic. The extension does not require any modifications to the NPF core or npfctl(8). Facilities such as traffic normalisation and packet logging are provided as extensions.

1.6 Running, testing and debugging in userspace

For testing, NPF uses NetBSD's RUMP (Runnable Userspace Meta Programs) framework – a kernel virtualisation and isolation technique, which enables running of the NetBSD kernel or parts of it in userspace, like a regular program. For example, NetBSD's TCP/IP stack could be run in userspace [Kan09] and other applications be passed through it. This makes debugging or profiling significantly easier due to the availability of tools such as gdb(1). NPF regression tests are integrated into NetBSD's test suite and thus are part of the periodic automated runs.

One of the reasons to emphasize modularity and strict interfacing within the NPF code base was to ease testing of the individual components or mechanisms: there are unit tests for every NPF subsystem. They are available within npftest(8) – a program containing both the tests and the NPF kernel part running as a userspace program. npftest(8) can also read and process tcpdump pcap files with a custom npf.conf configuration. This enables the analysis of a particular stream or connection in userspace. The npfctl(8) utility has a "debug" command which can print disassembled BPF byte-code and dump the configuration in the format sent to the kernel.

This set of tools which work in the userspace was a major factor which made NPF development much easier and faster.

2 Improvements

2.1 BPF and JIT-compilation

In 2012, NetBSD imported sljit[2] – a stack-less platform independent just-in-time (JIT) compiler, which supports various architectures. It is used by the PCRE library and is reasonably tested and benchmarked. The primary purpose of the addition was to support JIT-compilation for BPF. This opened the possibility to consider using BPF byte-code in NPF.

However, the original instruction set lacked certain capabilities which would be useful for NPF and potentially other applications. In particular, it was missing ability to perform more complex operations from the BPF program. For example, while most of the packet inspection logic can stay in the byte-code, operations such as looking up an IP address in some container or walking the IPv6 headers and returning some offsets have to be done externally.

Hence, honouring the tradition of the RISC-like instruction sets, it was decided to add support for a BPF coprocessor – a generic mechanism to offload more complex packet inspection operations. Two new instructions were added to the misc (BPF_MISC) category: BPF_COP and BPF_COPX. These instructions allow BPF programs to call predetermined functions by their index. It is important to note that there is no default coprocessor which could be invoked via /dev/bpf and the userlevel cannot set one. Only a kernel subsystem, which is in full control of the coprocessor, can set it (the in-kernel API was extended with bpf_set_cop(9), bpf_validate_ext(9) and bpf_filter_ext(9) routines to support this). Each BPF caller in the kernel would have its own independent context (state) and therefore different callers would not affect each other. The functions are predetermined and cannot change during the life-cycle of the BPF context. The coprocessor can inspect the packet in a read-only manner and return some numeric values. It cannot alter the flow of the program, therefore this functionality does not make BPF programs Turing-complete.

[2]http://sljit.sourceforge.net [accessed: 31 January 2014]

Originally, NPF had its own instruction set called n-code. The main motivation for n-code was to provide CISC-like complex instructions and thus reduce the processing overhead. With BPF coprocessor and JIT-compilation support, it became possible to use BPF byte-code in NPF (and, in fact, unify all packet classification engines under BPF). Therefore, the original n-code became redundant and was dropped in favour of BPF byte-code in NPF.

This was a major change and simplification of the NPF core. It also had an important user-visible effect – the availability of libpcap and its syntax (as described in the pcap-filter(7) manual page) which is widely used in tcpdump(1). For example:

```
block out final pcap-filter \
    "tcp and dst 10.1.1.252"
block in final pcap-filter \
    "ip[2:2] > 576"
```

As illustrated in the example, BPF supports byte-level inspection and virtually any filter pattern can be constructed and passed as a rule.

2.2 Tables and perfect hashing

Another major improvement was support for a new NPF table type. Initially, there were two types: hash and tree. The first provides a hash table which was recently improved to use lockless lists per bucket. This table structure provides amortised $O(1)$ lookup time and high concurrency, but in a case of increasing number of elements, it may suffer from the collisions. The future work would be to improve the implementation to use efficient and concurrent hash resizing techniques [TMW11].

The second is implemented using a PATRICIA tree which provides $O(k)$ lookup time (where k is a key length) and supports prefix matching. However it uses read-write locks and thus has limited scalability.

The third, new type is cdb – a constant database which uses perfect hashing and thus guarantees $O(1)$ complexity and provides completely lockless lookup. In case of a static set of data, this provides highest performance and ideal scalability. NetBSD has a general purpose interface to produce and access constant databases, which is provided as a part of libc. The API is described in the cdbr(3) and cdbw(3) manual pages.

2.3 Stateful ends

In NPF, the state is uniquely identified by a 6-tuple: source address with port, destination address with port, protocol and the interface identifier. Remember that if a packet belongs to a connection which has a state entry, it will completely bypass the ruleset inspection on that interface. It was a deliberate choice to include the interface as a state identifier so it would match only on the interface where it was created. Bypassing the ruleset on other interfaces can have undesirable effects, e.g. a packet with a spoofed IP address might bypass ingress filtering. Associating a state with two interfaces (forwarding case) may also cause problems if the routes change.

However, there are legitimate cases when bypassing on other interfaces is safe and useful, e.g. when in case of forwarding the ruleset on one interface is larger and the administrator ensures that there are no security or other implications. For this case, a special keyword *stateful-ends* was added in NPF to perform the state lookup on other interfaces as well. This may lead to higher performance in certain configurations and may also handle some asymmetric routing cases. The administrator is free to choose whether *stateful* or *stateful-ends* is more suitable.

2.4 Stateless NAT and NPTv6

An important addition was stateless (static) NAT. The addition is relatively easy given that NPF already supports stateful (dynamic) NAT – the policy is always looked up by inspecting the NAT ruleset. The policy implements a particular algorithm of the translation. Consider the following syntax of "map" (in a simplified Backus-Naur Form):

```
map =
  "map" interface
  ( "static" algo | "dynamic" )
  net-seg ( "->" | "<-" | "<->" ) net-seg
  [ "pass" filt-opts ]
```

The translation happens between two network segments (syntactically, defined on the left and the right hand sides respectively, separated by an arrow which defines the translation type). Currently, the simplest supported form of stateless NAT is when both segments consist of only one host – that is, a 1:1 translation of two IP addresses.

The second supported form is the IPv6-to-IPv6 Network Prefix Translation, shortly NPTv6, which is an algorithmic transformation of IPv6 prefixes as described in RFC 6296 [WB11].

2.5 Dynamic interface handling

One feature, which has been constantly requested by the users and finally added, is dynamic handling of interface arrivals and departures. In the latest NPF, it is possible to create rules for the interfaces which do not exist at the moment of configuration load. That is, instead of using the interface index (see if_nametoindex(3) routine), the interface is always identified by the name (e.g. *ppp0*) regardless of its existence.

NPF internally maps interface names to the generated interface IDs which are assigned to the interfaces when they arrive. Therefore, matching an interface is an $O(1)$ operation merely matching the IDs. It is important to note that this is handled in a way which does not modify the actual ruleset, therefore no synchronisation is required. If the expected interface departs, the rule will simply not match. It will match once the interface has arrived. Hence, interface changes have minimum effect on NPF's packet processing capability.

The interface addresses can be dynamically handled by reusing NPF tables. However, there is a general need for a concurrent iteration (and lookup) of the interface addresses. Therefore, the objective is to add such mechanism at the network stack level and provide a general purpose API.

3 In development

3.1 Lockless state inspection

Unlike the ruleset inspection, the state inspection currently uses locks. Specifically, it uses a hash table (the MurmurHash2 function is used) where each bucket has a red-black tree and a read-write lock to protect it. Distributed locks reduce the lock contention and the trees ensure $O(\log_2 n)$ lookup complexity. While such a design provides a certain level of concurrency, it has important drawbacks: 1) read-write locks suffer from *cache-line bouncing effect* and do not scale in the long term, especially on many-core systems 2) the overhead

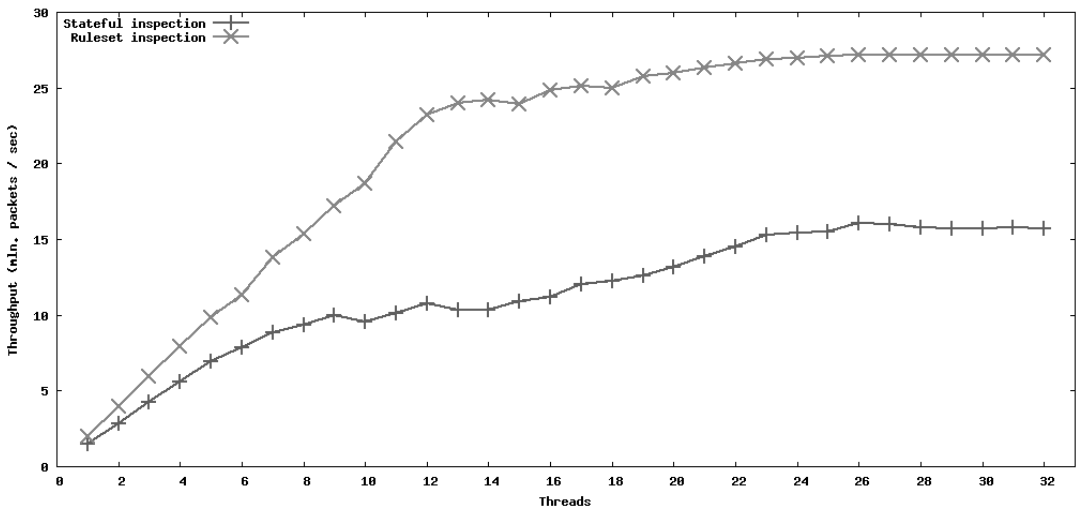


Figure 1: NPF scalability: npftest(8) benchmark using RUMP framework. 12 cores (24 threads), Intel(R) Xeon(R) CPU E5-2620, 2.00GHz.

of hashing, tree lookup and the read-write lock itself is not insignificant. As illustrated in Figure 1, the performance of the state inspection starts to degrade after 8 concurrent clients due to the described drawbacks, while the ruleset inspection has practically linear scalability on a 12-core processor with 2 threads per core (the lower growth after 12 clients is due to the CPU topology – hardware threads start to contend within the core). It should be noted that using a hash table alone is not suitable, as it suffers from DoS attacks exploiting collisions and worst case $O(n)$ behaviour if the bucket would have a linked list. Hence a better data structure is needed. The main requirements for the data structure are the following: 1) decent performance i.e. not only the algorithmic complexity, but also cache-awareness which is particularly important for modern CPUs 2) high concurrency 3) resistance to attacks exploiting worst case complexity.

The hash tables generally do not meet the third requirement unless combined with other data structures or additional techniques. Although there are concurrent hash tables which are capable of efficient and fully concurrent resizing, it may be quite difficult to perform this in a way which would be resistant to a sophisticated DoS attack. Hence we leave the discussion of such a possibility for a future research and focus on the existing solution scope – the lockless trees.

The packet filter performs 6-tuple state lookup where the key may be up to 40 bytes long. It is generally easier to implement concurrent radix trees, but given that our key is not small, they would have a higher lookup complexity. Also, radix trees which use a higher radix (arity) and sparse keys tend to have much higher memory usage. After some empirical research, the current data structure which is considered as a main candidate demonstrating good characteristics is Masstree – a lockless cache-aware B+ tree [MKM12]. However, one of the challenges is adapting it to the kernel environment. In particular, the code in the kernel cannot block while holding a spin-lock and it cannot block in the interrupt handler (e.g. when allocating the memory). Therefore, memory allocation failures while splitting the tree nodes must be handled in a graceful way. The current work in progress is to address these problems, integrate the data structure with NPF, benchmark the final solution and publish it. This work is expected to appear in the NetBSD 7.0 release.

4 Conclusion

Over the last few years, the core of NPF had some code refactoring and design adjustments. At the same time, the core functionality has gained a lot of testing and accumulated some user base. Upon the completion of state lookup and other improvements described in this paper, the core architecture will be highly optimised and generally solid ground for the further growth of features: high availability and quality of service.

References

[Kan09] Antti Kantee. Environmental Independence: BSD Kernel TCP/IP in Userspace. In *AsiaBSDcon 2009 proceedings*. Helsinki University of Technology, 2009.

[McH09] Patrick McHardy. [ANNOUNCE]: First release of nftables. `http://lwn.net/Articles/324251/`, March 2009.

[MKM12] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196. ACM, 2012.

[npf14] NPF documentation. `http://www.netbsd.org/~rmind/npf`, January 2014.

[Roo01] Guido Van Rooij. Real Stateful TCP Packet Filtering in IP Filter. In *10th USENIX Security Symposium invited talk*, August 2001.

[TMW11] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX-ATC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[WB11] Margaret Wasserman and Fred Baker. IPv6-to-IPv6 Network Prefix Translation. RFC 6296, June 2011.

OpenZFS: a Community of Open Source ZFS Developers

Matthew Ahrens
Delphix
San Francisco, CA
mahrens@delphix.com

***Abstract*—OpenZFS is a collaboration among open source ZFS developers on the FreeBSD, illumos, Linux, and Mac OSX platforms. OpenZFS helps these developers work together to create a consistent, reliable, performant implementation of ZFS. Several new features and performance enhancements have been developed for OpenZFS and are available in all open-source ZFS distributions.**

I. Introduction

In the past decade, ZFS has grown from a project managed by a single organization to a distributed, collaborative effort among many communities and companies. This paper will explain the motivation behind creating the OpenZFS project, the problems we aim to address with it, and the projects undertaken to accomplish these goals.

II. History of ZFS Development

A. Early Days

In the early 2000's, the state of the art in filesystems was not pretty. There was no defense from silent data corruption introduced by bit rot, disk and controller firmware bugs, flaky cables, etc. It was akin to running a server without ECC memory. Storage was difficult to manage, requiring different tools to manage files, blocks, labels, NFS, mountpoints, SMB shares, etc. It wasn't portable between different operating systems (e.g. Linux vs Solaris) or processor architectures (e.g. x86 vs SPARC vs ARM). Storage systems were slow and unscalable. They limited the number of files per filesystem, the size of volumes, etc. When available at all, snapshots were limited in number and performance. Backups were slow, and remote-replication software was extremely specialized and difficult to use. Filesystem performance was hampered by coarse-grained locking, fixed block sizes, naive prefetch, and ever-increasing fsck times. These scalability issues were mitigated only by increasingly complex administrative procedures.

There were incremental and isolated improvements to these problems in various systems. Copy-on-write filesystems (e.g. NetApp's WAFL) eliminated the need for fsck. High-end storage systems (e.g. EMC) used special disks with 520-byte sectors to store checksums of data. Extent-based filesystems (e.g. NTFS, XFS) worked better than fixed-block-size systems when used for non-homogenous workloads. But there was no serious attempt to tackle all of the above issues in a general-purpose filesystem.

In 2001, Matt Ahrens and Jeff Bonwick started the ZFS project at Sun Microsystems with one main goal: to end the suffering of system administrators who were struggling to manage complex and fallible storage systems. To do so, we needed to re-evaluate obsolete assumptions and design an integrated storage system from scratch. Over the next 4 years, they and a team of a dozen engineers implemented the fundamentals of ZFS, including pooled storage, copy-on-write, RAID-Z, snapshots, and send/receive. A simple administrative model based on hierarchical property inheritance made it easy for system administrators to express their intent, and made high-end storage features like checksums, snapshots, RAID, and transparent compression accessible to non-experts.

B. Open Source

As part of the OpenSolaris project, in 2005 Sun released the ZFS source code as open source software, under the CDDL license. This enabled the ports of ZFS to FreeBSD, Linux, and Mac OSX, and helped create a thriving community of ZFS users. Sun continued to enhance ZFS, bringing it to enterprise quality and making it part of the Solaris operating system and the foundation of the Sun Storage 7000 series (later renamed the Oracle ZFS Storage Appliance). The other platforms continually pulled changes from OpenSolaris, benefiting from Sun's continuing investment in ZFS. Other companies started creating storage products based OpenSolaris and FreeBSD, making open-source ZFS an integral part of their products.

However, the vast majority of ZFS development happened behind closed doors at Sun. At this time, very few core enhancements were made to ZFS by non-Sun contributors. Thus although ZFS was Open Source and multi-platform, it did not have an open development model. As long as Sun continued maintaining and enhancing ZFS, this was not necessarily an impediment to the continued success of products and community projects based on open-source ZFS – they could keep getting enhancements and bug fixes from Sun.

C. Turmoil

In 2010, Oracle acquired Sun Microsystems, stopped contributing source code changes to ZFS, and began dismantling the OpenSolaris community. This raised big concerns about the future of open-source ZFS – without its primary contributor, would it stagnate? Would companies creating products based on ZFS flounder without Sun's engineering resources behind them? To address this issue for both ZFS and OpenSolaris as a whole, the Illumos project was created. Illumos took the source code from OpenSolaris (including ZFS) and formed a new community around it. Where OpenSolaris development was controlled by one company, illumos creates common ground for many companies to contribute on equal footing. ZFS found a new home in Illumos, with several companies basing their products on it and contributing code changes. FreeBSD and Linux treated Illumos as their upstream for ZFS code. However, there was otherwise not much interaction between platform-specific communities. There continued to be duplicated efforts between platforms, and surprises when code changes made on one platform were not easily ported to others. As the pace of ZFS development on FreeBSD and Linux increased, fragmentation between the platforms became a real risk.

III. The OpenZFS Collaboration

A. Goals

The OpenZFS project was created to accomplish three goals:

1) Open communication: We want everyone working on ZFS to work together, regardless of what platform they are working on. By working together, we can reduce duplicated effort and identify common goals.

2) Consistent experience: We want users' experience with OpenZFS to be high-quality regardless of what platform they are using. Features should be available on all platforms, and all implementations of ZFS should have good performance and be free of bugs.

3) Public awareness: We want to make sure that people know that open-source ZFS is available on many platforms (e.g. illumos, FreeBSD, Linux, OSX), that it is widely used in some of the most demanding production environments, and that it continues to be enhanced.

B. Activities

We have undertaken several activities to accomplish these goals:

1) Website: The http://open-zfs.org website (don't forget the dash!) publicizes OpenZFS activities such as events, talks, and publications. It acts as the authoritative reference for technical work, documenting both usage and how ZFS is implemented (e.g. the on-disk format). The website is also used as a brainstorming and coordination area for work in progress. To facilitate collaboration, the website is a Wiki which can be edited by any registered user.

2) Mailing list: The OpenZFS developer mailing list[1] serves as common ground for developers working on all platforms to discuss work in progress, review code changes, and share knowledge of how ZFS is implemented. Before its existence, changes made on one platform often came as a surprise to developers on other platforms, and sometimes introduced platform compatibility issues or required new functions to be implemented in the Solaris Porting Layer. The OpenZFS mailing list allows these concerns to be raised and addressed during code review, when they can easily be addressed. Note that this mailing list is not a replacement for platform-specific mailing lists, which continue to serve their role primarily for end users and system administrators to discuss how to use ZFS, as well for developers to discuss platform-specific code changes.

3) Office hours: Experts in the OpenZFS community hold online office hours[2] approximately once a month. These question and answer sessions are hosted by a rotating cast of OpenZFS developers, using live audio/video/text conferencing tools. The recorded video is also available online.

4) Conferences: Since September 2013, 6 OpenZFS developers have presented at 8 conferences. These events serve both to increase awareness of OpenZFS, and also to network with other developers, coordinating work in person. Additionally, we held the first OpenZFS Developer Summit[3] in November 2013 in San Francisco. More than 30 individuals participated, representing 14 companies and all the major platforms. The two-day event consisted of a dozen presentations and a hackathon. Ten projects were started at the hackathon, including the "best in show": a team of 5 who ported the TestRunner test suite from illumos to Linux and FreeBSD. Slides from the talks and video recordings are available on the open-zfs.org website[3].

C. New features

In this section we will share some recent improvements to the OpenZFS code. These changes are available on all OpenZFS platforms (e.g. Illumos, FreeBSD, Linux, OSX, OSv).

1) Feature flags: The ZFS on-disk format was originally versioned with a linear version number, which was incremented whenever the on-disk format was changed. A ZFS release that supported a given version also must understand all prior versions.

This model was designed initially for the single-vendor model, and was copacetic with the OpenSolaris goals of community development while maintaining control over the essentials of the product. However, in the open development model of OpenZFS, we want different entities to be able to make on-disk format changes independently, and then later merge their changes together into one codebase that understands both features. In the version-number model, two companies or projects each working on their own new on-disk features would both use version N+1 to denote their new feature, but that number would mean different things to each company's software. This would make it very difficult for both companies to contribute their new features into a common codebase. The world would forever be divided into ZFS releases that interpreted version N+1 as company A intended, and those that interpreted it as company B intended.

To address this problem, we designed and implemented "feature flags" to replace the linear version number. Rather than having a simple version number, each storage pool has a list of features that it is using. Features are identified with strings, using a reverse-DNS naming convention (e.g. *com.delphix:background_destroy*). This enables on-disk format changes to be developed independently, and later be integrated into a common codebase.

With the old version numbers, once a pool was upgraded to a particular version, it couldn't be accessed by software that didn't understand that version number. This accomplishes the goal of on-disk versioning, but it is overly restrictive. With OpenZFS feature flags, if a feature is enabled but not actually used, the on-disk information reflects this, so software that doesn't understand the feature can still access the pool. Also, many features change the on-disk format in a way that older software can still safely read a storage pool using the new feature (e.g. because no existing data structures have been changed, only new structures added). OpenZFS feature flags also supports this use case.

2) LZ4 compression: ZFS supports transparent compression, using the LZJB and GZIP algorithms. Each block (e.g. 128KB) is compressed independently, and can be stored as any multiple of the disk's sector size (e.g. 68.5KB). LZJB is fairly fast and provides a decent compression ratio, while GZIP is slow but provides a very good compression ratio. In OpenZFS, we have also implemented the LZ4 compression algorithm, which is faster than LZJB (especially at decompression) and provides a somewhat better compression ratio (see Figure 1). For many workloads, using LZ4 compression

Fig. 1. Compression speeds and ratios compared (single core)

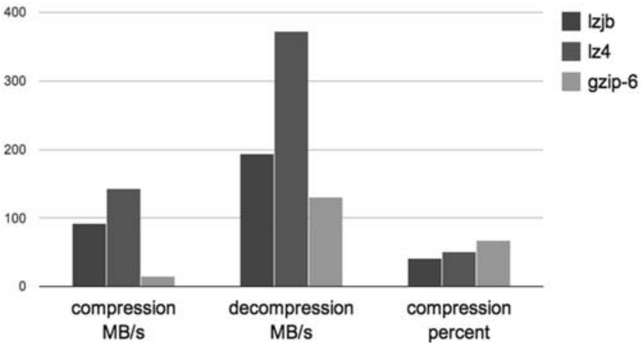
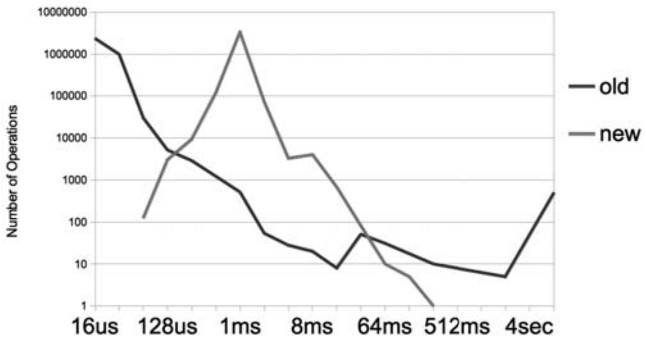


Fig. 2. Histogram of write latencies (log/log graph)



is actually faster than not compressing, because it reduces the amount of data that must be read and written.

3) Smooth write throttle: If the disks can't keep up with the application's write rate, then the filesystem must intervene by causing the application to block, delaying it so that it can't queue up an unbounded amount of dirty data.

ZFS batches several seconds worth of changes into a *transaction group*, or *TXG*. The dirty data that is part of each TXG is periodically synced to disk. Before OpenZFS, the throttle was implemented rather crudely: once the limit on dirty data was reached, all write system calls (and equivalent NFS, CIFS, and iSCSI commands) blocked until the currently syncing TXG completed. The effect is that ZFS performed writes with near-zero latency, until it got "stuck" and all writes blocked for several seconds.[4]

We rewrote the write throttle in OpenZFS to provide much smoother, more consistent latency, by delaying each write operation a little bit. The trick was to find a good way of computing how large the delay should be. The key was to measure the amount of dirty data in the system, incrementing it as write operations came in and decrementing it as write i/o to the storage completes. The delay is a function of the amount of dirty data (as a percentage of the overall dirty data limit). As more write operations come in, the amount of dirty data increases, thus increasing the delay. For a given workload, this algorithm will seek a stable amount of dirty data and thus a stable delay. Crucial for easy understanding of the system, this works without taking into account historical behavior or trying to predict the future. This makes the algorithm very responsive to changing workloads; it can't get "stuck" doing the wrong thing because of a temporary workload anomaly.

As a result of this work, we were able to reduce the latency outliers for a random write workload by 300x, from 10 seconds to 30 milliseconds (meaning that 99.9% of all operations completed in less than 30 milliseconds). (See Figure 2.)

IV. Further Work

Here we will outline some of the projects that are in progress.

A. *Platform-independent code repository*

Currently, code is shared between platforms on an ad-hoc basis. Generally, Linux and FreeBSD pull changes from illumos. This process is not as smooth as it could be. Linux and FreeBSD must maintain fairly tricky porting layers to translate the interfaces that the ZFS code uses on illumos to equivalent interfaces on Linux and FreeBSD. It is rare that changes developed on other platforms are integrated into illumos, in part because of the technical challenges that newcomers to this platform face in setting up a development environment, porting, building, etc.

We plan to create a platform-independent code repository of OpenZFS source code that will make it much easier to get changes developed on one platform onto every OpenZFS platform. The goal is that all platforms will be able to pull the exact code in the OpenZFS repo into their codebase, without having to apply any diffs.

We will define the interfaces that code in the OpenZFS repo will use, by explicitly wrapping all external interfaces. For example, instead of calling *cv_broadcast(kcondvar_t *)*, OpenZFS code would call *zk_cv_broadcast(zk_condvar_t *)*. Each platform would provide wrappers which translate from the OpenZFS *zk_* interfaces to platform-specific routines and data structures. This will allow the "Solaris Porting Layers" to be simplified.

The OpenZFS repo will only include code that is truly platform-independent, and which can be tested on any platform in userland (using the existing *libzpool.so* mechanism). Therefore it will include the DMU, DSL, ZIL, ZAP, most of the SPA, and userland components (/sbin/zfs, libzfs, etc). It will not include the ZPL, ZVOL, or vdev_disk.c, as these have extensive customizations for each platform. A longer-term goal is to split the ZPL into platform-independent and platform-dependent parts, and include the platform-independent part in the OpenZFS repo.

For more information, see the slides and video from the talk at the 2013 OpenZFS Developer Summit[3].

B. Conferences

Continuing the very successful 2012 ZFS Day and 2013 OpenZFS Developer Summit conferences, we plan to hold more OpenZFS-centered events. This will include annual OpenZFS Developer Summits, as well as more casual local meet-ups. We will also continue evangelizing OpenZFS at general technology conferences.

C. Resumable send and receive

ZFS send and receive is used to serialize and transmit filesystems between pools. It can quickly generate incremental changes between snapshots, making it an ideal basis for remote replication features. However, if the connection between send and receive processes is broken (e.g. by a network outage or one of the machines rebooting), then the send must re-start from the beginning, losing whatever data was already sent.

We are working on an enhancement to this that will allow a failed send to resume where it left off. This involves having the receiving system remember what data has been received. This is fairly simple, because data is sent in (object, offset) order. Therefore the receiving system need only remember the highest (object, offset) that has been received. This information will then be used to restart the send stream from that point.

The one tricky part is that we need to enhance the checksum that is stored in the send stream. Currently the checksum is only sent at the end of the entire send stream, so if the connection is lost, the data that was already received has not been verified by any checksum. We will enhance the send stream format to transmit the checksum after each record, so that we can verify each record as it is received. This will also provide better protection against transmission errors in the metadata of the send stream.

D. Large block support

ZFS currently supports up to 128KB blocks. This is large compared to traditional filesystems, which typically use 4KB or 8KB blocks, but we still see some circumstances where even larger blocks would increase performance. Therefore, we are planning to add support for blocks up to at least 1MB in OpenZFS.

We expect to see an especially large performance benefit when using RAID-Z, especially with very wide stripes (i.e. many devices in the RAID-Z group). RAID-Z breaks each block apart and spreads it out across all devices in the RAID-Z group. Therefore, under a random read workload, RAID-Z can deliver the IOPS of only a single device, regardless of the number of devices in the RAID-Z group. By increasing the block size, we increase the size of each IO, which increases the effective bandwidth of the random read workload.

This is especially important when scrubbing or resilvering, which in the worst case creates a random read workload. By increasing the block size, we raise the lower bound of the scrub or resilver time. For example, consider a RAID-Z group with eight 1-TB disks that can do 100 random reads per second. With 128KB block size, in the worst case we could resilver one drive in 186 hours (1TB * 8 drives / 128KB block size / 100 IOPS). Whereas with 8MB block size, in the worst case we could resilver a drive in 2.8 hours. This corresponds to a rate of 104MB/second, which is close to the typical maximum sequential transfer rate of hard drives, thus matching the performance of LBA-based resilver mechanisms.

V. Participation

OpenZFS exists because of contributions of every type. There are a number of ways you can get involved:

If you are working with ZFS source code, join the developer mailing list[1]. Post there to get design help and feedback on code changes.

If your company is making a product with OpenZFS, tell people about it. Contact *admin@open-zfs.org* to put your logo on the OpenZFS website. Consider sponsoring OpenZFS events, like the Developer Summit. If you have enhanced OpenZFS, work with the community to contribute your code changes upstream. Beside benefiting everyone using OpenZFS, this will make it much easier for you to sync up with the latest OpenZFS enhancements from other contributors, with a minimum of merge conflicts.

If you are using OpenZFS, help spread the word by writing about your experience on your blog or social media sites. Ask questions at the OpenZFS Office Hours events. And of course, keep sharing your suggestions for how OpenZFS can be even better (including bug reports).

VI. Conclusion

ZFS has survived many transitions, and now with OpenZFS we have the most diverse, and yet also the most unified, community of ZFS contributors. OpenZFS is available on many platforms: illumos, FreeBSD, Linux, OSX, and OSv. OpenZFS is an integral part of dozens of companies' products.[5] A diverse group of contributors continues to enhance OpenZFS, making it an excellent storage platform for a wide range of uses.

References

[1] Mailing list: *developer@open-zfs.org*, see http://www.open-zfs.org/wiki/Mailing_list to join.
[2] Office Hours, see http://www.open-zfs.org/wiki/OpenZFS_Office_Hours
[3] Developer Summit, see http://www.open-zfs.org/wiki/OpenZFS_Developer_Summit_2013
[4] Old ZFS write throttle, see http://blog.delphix.com/ahl/2013/zfs-fundamentals-write-throttle/
[5] Companies using OpenZFS, see http://www.open-zfs.org/wiki/Companies

Bambi Meets Godzilla: They Elope

Open Source Meets the Commercial World

Eric Allman
AsiaBSDcon
Tokyo, Japan
March 13–16, 2013

My Background

- Long time open source developer (started ~1975)
  - INGRES RDBMS (one of the first!)
  - syslog, –me (troff) macros, trek, other BSD utilities
  - sendmail
  - the guy who got U.C. Berkeley to start using SCCS
- Jobs in academia, commercial, and research
- Started Sendmail, Inc. in 1998
  - One of the early Open Source "hybrid" companies
  - Survived the tech crash (but that's another story)
- Now at U.C. Berkeley Swarm Lab
  - http://swarmlab.eecs.berkeley.edu

Sendmail's Background

- Sendmail started as an early open source project (as part of BSD), a classic “scratch your itch” example
- Like most Open Source of the era, it went through some growth spurts
  - Built to solve a single, local problem
  - Generalized due to community need
  - Got caught up in the Internet explosion
  - Remained community-supported, usually with the assistance of a small group of people (sendmail used the benevolent dictator model with trusted henchmen, same as Linux)
- O'Reilly book made a huge difference

The Onset of Success Disaster

- At some point, community scaling collapsed
  - I no longer had time to do coding due to support requirements
  - Some projects used the RTFM[1] approach (i.e., “you're on your own”), but that only works with sophisticated, dedicated users (and a FMTR[2])
  - Assertion: all successful large Open Source projects get outside support at some point
- I wanted to get time to do coding again, which meant ditching the day job
- So I started a company
  - All I really wanted was a support department

[1]Read the Fine Manual
[2]Fine Manual To Read

Open Source Does Not Exist in a Vacuum

Everyone who writes open source gets outside support

- You're a student (parents, grants, day jobs pay)
- You're university faculty/staff or at a research lab (university/grants/company pays)
- You work at an enlightened company that gives you "20% time" (company pays)
- You work at a company with a vested interest in Open Source
- You have no funding at all — you pay directly in the form of your leisure time/sleep/health/relationships

Models for Monetization

- Start a foundation, get donations (e.g., Mozilla, Eclipse, Apache, FreeBSD, ...)
- Find a patron who will shower you with money
  - Hard to do unless you are Bach or Mozart
- Sell yourself to a company with deep pockets
  - Note: they may not have your best interests in mind; may even just want to shut you down
  - Leverage limited if you are the only asset
- Start your own company (e.g., Sendmail, Red Hat)

A Note About Foundations

- Foundations insulate you from the day to day pressures of corporations
- Foundations *do not* prevent you from being pressured in other ways
- Foundations *do* take a lot of work to start and keep running
    - Especially not-for-profits
- You might lose some of the good things (e.g., good marketing input)
- Note: this doesn't count if you already have deep pockets (Rockefeller, Gates, etc.)

Assertion: Open Source Needs Commercial Input

- Developers seldom are also the customers
    - Was Open Source's traditional base; rare now
- Developer-designed consumer software usually "unimpressive" to "outright bad"
    - Developers don't think like normal humans (or communicate well with them on software design)
    - This is what Product Managers are supposed to do
- Examples of other benefits
    - "Soft" items such as user documentation
    - Front line support (unburden developers)
    - Overhead (e.g., system/network maintenance)

Deep Tension Between Open Source & Commercial

- Open source is about building, sharing, flexibility
  - Make the world a better place (give back)
  - Solve an interesting problem
  - Personal development (and perhaps fame?)
- Commercial is about making money
  - Sales guys do not understand how to make money by giving the product away ("you're a communist")
  - Immense pressure toward feature creep to keep a revenue stream going (e.g., Quicken, iTunes)
  - If you miss payroll, you're dead

Commercial Markets for Open Source

- Who's going to pay for product?
  - Folks who just want it free? Good luck with that
  - Businesses? What size? They buy trust, not just code
  - Consumers? Fickle, need polished product
- Most customers won't care about open source
  - Think like a customer. What are they buying?
- Open source tends to commoditize a market
  - Brings down the unit price
  - Suppliers have to move up the food chain

Commercial Models for Open Source

- Completely free, sell something else
  - Support, services, documentation, stability, etc.
  - Limited economies of scale
- Free, sell (often vertical) bundles (distro or appliance)
- Free basic technology, commercial non-open-source add-ons
  - Works best when you have a clean extension model or can “wrap” OSS in commercial software
  - Generally supersets “sell something else”
- Technology grab (close the software base)

Starting a Company

- Starting a company is not about technology
- It is about:
  - Finance (starting with Investors) ⇐ **Money**
  - Sales ⇐ **Money**
  - Marketing ⇐ **Money**
  - Support ⇐ **Money**
  - Services ⇐ **Money**
  - oh yeah, and some Engineering ⇐ **Expense**

A Word About Corporate Culture

- Engineering driven or Sales/Marketing driven?
  - Almost no large company is engineering driven (Google comes the closest, and it is an anomaly, and changing)
  - Investors prefer S/M driven, and they run the board
- Purely Sales/Marketing driven leads to aberrations, but it is very hard to avoid this
  - Sales always wins in a fiscal crisis
  - A fiscal crisis always comes along sooner or later
  - Possible exception: when you are sitting on a ton of cash (e.g., Apple, Google)
- Sales/Marketing/Finance want short term view, Engineering wants long(er) term view

Life Cycles: Open Source, Research, Companies

- A brief (and woefully imprecise) comparison of the lifecycle of an Open Source Project, a Research Project (non-proprietary, non-military), and a Company
- Note the similarities — and the differences

The Initial Inspiration

Open Source	"Scratch an itch"
Research Project	Ask a question
Company	See a revenue opportunity

Making It Possible

Open Source	See if it's already been done (optional) Do an architectural design (optional) Choose language/tools Start writing code
Research Project	Research the literature Get a grant / other funding Line up grad students
Company	Write a business plan Line up investors Figure out corporate culture (optional) Hire a team

Birthing the Baby

Open Source	Do early (0.x) releases Start building community
Research Project	Start writing code/researching Start writing “teasers”
Company	Start building product Line up early customers Start trade shows

Making it Real

Open Source	Release 1.0 Address support problem Got docs? Oops....
Research Project	Publish or Perish
Company	First release Scale out sales, support, services

Growing It

Open Source	No community? Hang it up Write the O'Reilly book Avoid Second System effect (optional) Release 2
Research Project	Thesis time ~~Slaves~~ Students graduate "Transactions" article(s)
Company	Second release Push to profitability First (second?) round of layoffs Second (3rd, 4th) investment round

Next Steps

Open Source	Throw it to the winds? Hand over to larger organization? Commercialize it? Just keep going?
Research Project	Ask a question (often suggested by previous cycle)
Company	"Liquidity Event" and continue "Liquidity Event" and assimilation Bankruptcy and die

Two Mistakes Founders Make

- Assuming they know everything and insisting on control. They don’t, and making other’s lives miserable is a good way to get forced out early
- Assuming everyone else is more knowledgeable and has no hidden agendas
  - Beware of people who tell you that their field is so arcane that you can’t possibly understand it. Sometimes it’s true, but not very often.
- Obviously, a happy medium is needed

Some Conclusions

Good News:

- Without a doubt, commercial input to open source has permitted it to take on far larger problems
- Similarly, good marketing input permits open source to take on different kinds of problems

Bad News:

- Open source has lost its innocence
- Corporations emphasize short-term survival (i.e., money) over technological beauty

Bonus Third Mistake Founders Make

- Believing that they will be able to get back to coding by starting a company

Thank You

Eric Allman

asiabsdcon-2014 (at) eric.allman.name

Bambi Meets Godzilla Video:
https://www.youtube.com/v/n-wUdetAAIY

Snapshots, Replication, and Boot-Environments - How new ZFS utilities are changing FreeBSD & PC-BSD.

by

Kris Moore
kris@pcbsd.org

Since the inclusion of ZFS into the FreeBSD base system it has revolutionized how enterprise users have managed their data. However due to higher memory requirements and the difficulty of the initial setup, it was often out of reach for less experienced system administrators and more modest system hardware. However, over the past several years ZFS on BSD has greatly matured, reducing the complexity of the initial setup and tuning required to perform optimally. In early 2013 this led the PC-BSD project to re-focus and fully embrace ZFS as its default and only file-system for both desktop and server deployments. This decision immediately spawned development of a new class of tools and utilities which assist users in unlocking the vast potential that ZFS brings to their system, in areas of data-integrity, instant backup and restore, fail-over, performance and more. In this paper we will take a look at ZFS Boot-Environments and the new Life-Preserver utility which assists users in ZFS management, including snapshots, replication, mirroring, monitoring and more.

The Initial Setup

Starting in PC-BSD 9.2 and later, any installation performed with its installer will automatically setup ZFS as the default root file-system. This includes installations of both a Desktop (PC-BSD) or Server (TrueOS). The default layout will be created with a SWAP partition, (mainly for crash-dumps) and the rest of the disk / partition allocated to a single ZFS zpool. During the install, options will be provided for the setupof multiple disk drives through ZFS mirroring or raidz levels 1-3. In addition the GRUB boot-manager will be installed, which will facilitate the usage of ZFS Boot-Environments.

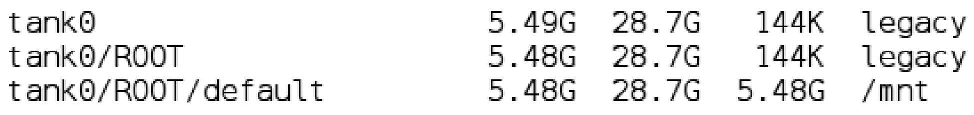
Boot-Environments

Included with the switch to ZFS as the default file-system in PC-BSD, the default boot-manager has also been switched to GRUB, largely due in part for its ability to boot ZFS snapshots. When bundled together with the "**beadm**" command they are more commonly referred to as Boot-Environments. Boot environments allow you to create an instant ZFS snapshot of your system environment, including the kernel, world and packages. Then if something goes wrong, such as a failed upgrade, it is possible to simply boot from the old BE and continue on.

While the traditional FreeBSD boot-loader does have the support for booting from ZFS, it isn't as well suited to ZFS boot-environments for a few reasons. First, the current FreeBSD loader menus provide no convenient method of listing and selecting the boot-environment to load. This means if you need to boot another environment, you will be stuck manually entering environment variables at the loader, assuming you remember the particular names of the BE you wish to load. In addition the boot-menus are only loaded after a successful load of the kernel. But what happens if during the course of an upgrade the kernel itself is missing or damaged? This would require that the user run some other boot-media to try and repair the damage by hand, rendering the whole point of a working boot-environment meaningless.

GRUB on the other-hand does not suffer from these drawbacks. Since it is a self-contained, scriptable boot-loader, it is possible to add boot-environments on the fly. Should something then go wrong, a quick reboot will provide a bootable list of environments on the system, allowing quick access to a working machine. So how does this work? Let us first take a look at the ZFS layout:

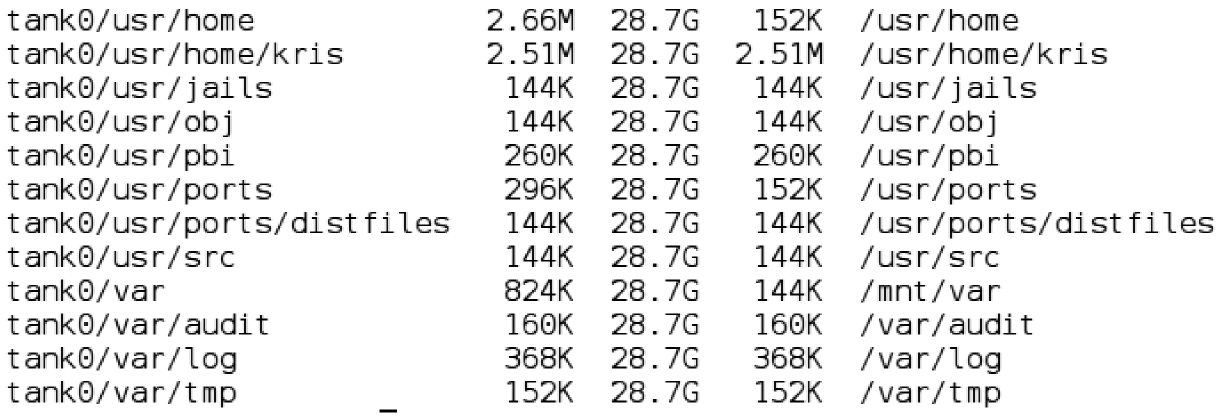
In order to make boot-environments contain the proper data for backup / restore, we have to start with a slightly different than traditional ZFS layout.



```
tank0                        5.49G  28.7G   144K  legacy
tank0/ROOT                   5.48G  28.7G   144K  legacy
tank0/ROOT/default           5.48G  28.7G  5.48G  /mnt
```

Illustration 1: ZFS Layout of the "default" boot-environment

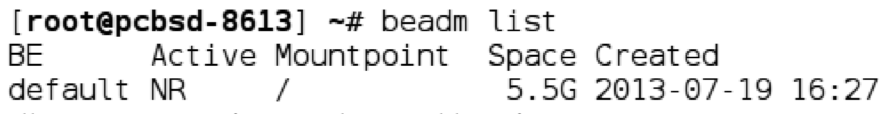
The first change is that we are creating a special <tank>/ROOT/default dataset. This dataset is the primary point of taking and cloning ZFS snapshots, which means anything outside of this dataset will **not** be included inside of a Boot-Environment. It will be mounted as '/' on your system. However, for a Boot-Environment to work, we will want to include /usr, /usr/local, and such within our snapshot, while still allowing child datasets such as /usr/home and /usr/jails to be created. To accomplish this, we create a /usr ZFS dataset and set the "**canmount=off**" flag. The same is done for the /var dataset, allowing us to include it in the snapshots, while creating /var/log, /var/tmp, and /var/audit to persist between Boot-Environments.



```
tank0/usr/home              2.66M  28.7G   152K  /usr/home
tank0/usr/home/kris         2.51M  28.7G  2.51M  /usr/home/kris
tank0/usr/jails              144K  28.7G   144K  /usr/jails
tank0/usr/obj                144K  28.7G   144K  /usr/obj
tank0/usr/pbi                260K  28.7G   260K  /usr/pbi
tank0/usr/ports              296K  28.7G   152K  /usr/ports
tank0/usr/ports/distfiles    144K  28.7G   144K  /usr/ports/distfiles
tank0/usr/src                144K  28.7G   144K  /usr/src
tank0/var                    824K  28.7G   144K  /mnt/var
tank0/var/audit              160K  28.7G   160K  /var/audit
tank0/var/log                368K  28.7G   368K  /var/log
tank0/var/tmp                152K  28.7G   152K  /var/tmp
```

Illustration 2: Additional ZFS file-systems excluded from a boot-environment

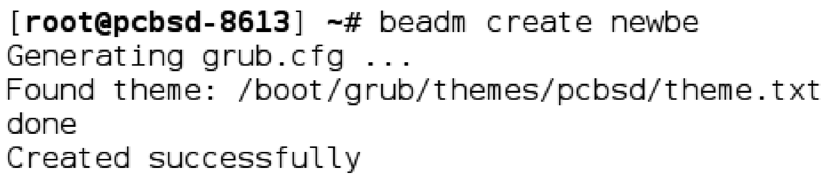
These layout choices ensure that when we take a snapshot of the system, we end up with all the system files and packages necessary to boot back up to a working desktop or server environment. With /usr/local being included, it makes Boot-Environments a good choice for backups before performing package updates, which are often just as critical to a user's workflow as the kernel and world are. With the default layout ready, we can now use the "**beadm**" command to create and delete new BE's.



```
[root@pcbsd-8613] ~# beadm list
BE      Active Mountpoint  Space Created
default NR     /            5.5G 2013-07-19 16:27
```

Illustration 3: Performing the initial list of Boot-Environments

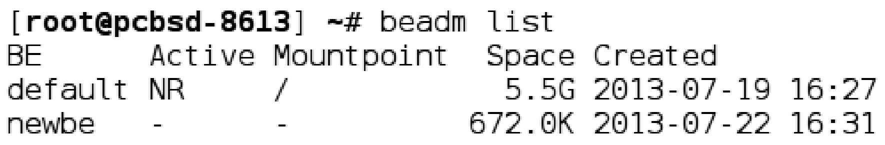
The “NR” flags indicate that this BE is currently active **N**ow, and will still be active on next **R**eboot. When we create a new BE, it is going to take a snapshot of the currently running BE at the current moment, so it would be wise to remember to do this **before** starting anything potentially dangerous. To add a new Boot-Environment, simply run the “beadm create <nickname>” command:



```
[root@pcbsd-8613] ~# beadm create newbe
Generating grub.cfg ...
Found theme: /boot/grub/themes/pcbsd/theme.txt
done
Created successfully
```

Illustration 4: Creating a new Boot-Environment

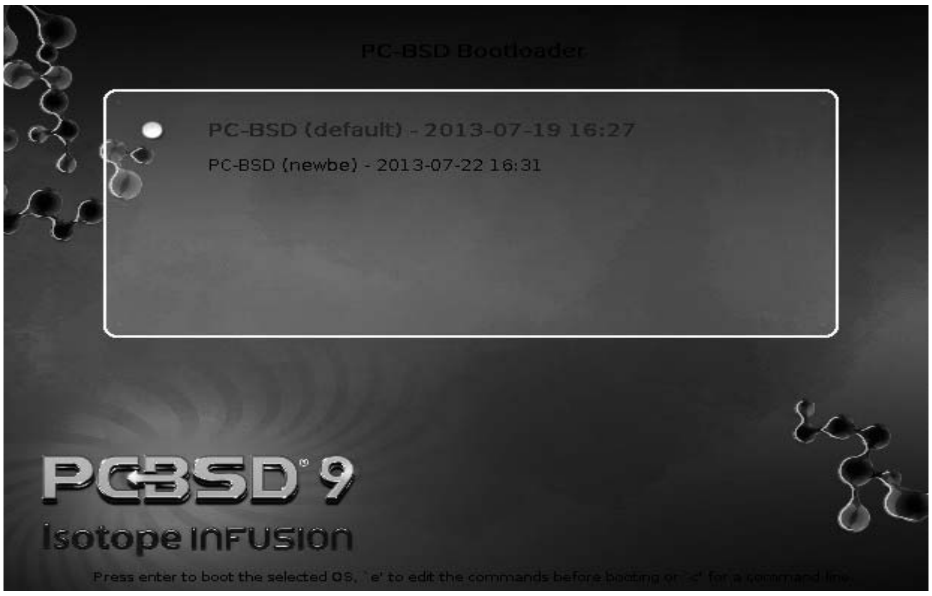
As you can see, we have created a new BE nicknamed “**newbe**”. A quick re-run of the list command will show the newly created environment, along with the time and current size cost.



```
[root@pcbsd-8613] ~# beadm list
BE      Active Mountpoint  Space Created
default NR     /            5.5G 2013-07-19 16:27
newbe   -      -          672.0K 2013-07-22 16:31
```

Illustration 5: Getting a listing of available Boot-Environments

During the creation of the new BE, you saw a notice about generating the grub.cfg file. This brings us to the point of putting the “**Boot**” into Boot-Environment. Similar to Solaris, PC-BSD includes the GRUB boot-loader by default and has it integrated into the “**beadm**” command in order to provide boot-time functionality. By default, anytime a new BE is created or destroyed, the GRUB menu configuration file will be refreshed using the “**grub-mkconfig -o /boot/grub/grub.cfg**” command. If we go ahead and reboot our system, we will now be presented with a new Boot-Environment menu:


PC-BSD Bootloader

PC-BSD (default) - 2013-07-19 16:27
PC-BSD (newbe) - 2013-07-22 16:31

Press enter to boot the selected OS, `e' to edit the commands before booting or `c' for a command line.

As you can see, our “**newbe**” Boot-Environment has been added to the menu, along with the creation date. By default, a countdown timer will boot the first environment after 5 seconds. However, if you press any key and interrupt this timer, you can arrow up and down to select the environment you wish to boot from. Once you have selected the BE you wish to boot, GRUB will load the kernel + modules from this BE and boot the system. Now that we have way to easily create / manage boot-environments, let us take a look at another utility which helps manage data-integrity.

Life-Preserver

Starting in PC-BSD 9.2 and later, a new “lpreserver” (Life-Preserver) CLI utility has been included. This utility provides a framework to easily manage other aspects of ZFS, such as its abilities to create snapshots, replicate data, mirror drives and more. We will first take a look at snapshot creation and scheduling.

Snapshots

Due to ZFS being a “copy-on-write” file-system, one huge benefit is the ability to instantly create “snapshots” of your data. A snapshot is simply a “meta-marker” of how your data appeared at any given moment in time. When a snapshot is created, ZFS will ensure that those portions of the disk which existed at the time of the snapshot will not be overwritten until the snapshot has been destroyed. Thus initially the snapshot will not consume any space on the disk, and grow only as the current data changes from the original snapshot state. Using the “**lpreserver**” utility we can easily setup a schedule of creating / pruning snapshots.

```
# lpreserver cronsnap tank1 start daily@22 10
```

The above command will use cron to schedule a set of daily snapshots to be created at 22:00 on the “tank1” zpool. The last number “10” is used to indicate the total number of snapshots to keep. When this number is exceeded, the oldest snapshot will be pruned automatically, reclaiming any disk space which may have been allocated to that snapshot. By default all snapshots created will be recursive, meaning that any additional ZFS datasets created on the “tank1” zpool will also have snapshots created / pruned automatically. While the recursive option can be disabled, most users will not want to do this, in order to keep the system intact when doing replication.

From an internals standpoint, the way this works is as follows. First, the “lpreserver” command will create the appropriate entry in /etc/crontab, scheduling your snapshots as specified:

```
0    14    *    *    *    root    /usr/local/share/lpreserver/backend/runsnap.sh tank1 10
```

This script being called by cron can be run manually, or via the “lpreserver mksnap” command. Once executed, it will perform the following steps:

- Confirm that the zpool / dataset exists.
- Create new snapshot, recursive by default.
- Auto-prune any snapshots beyond the 10 specified. (Only snapshots with ‘auto-’ prefix will be counted, preserving any manual ones created)
- Send out notification e-mail, if enabled.
- Start auto-replication, if enabled.

Replication

With our snapshots being created on a regular basis we in essence now have access to an instant on-disk backup. But what happens if the system disk dies, or we lose the entire computer due to theft or some natural catastrophe? Luckily ZFS also has the ability to replicate datasets and snapshots across the wire to a remote location. Using the “lpreserver” command it is easy to setup a replication target for your zpool.

```
# lpreserver replicate add backupsys backupuser 22 tank1 zpoolbackup/backups sync
```

The above command usage is fairly simple. In this example we have scheduled a replication to the “**backupsys**” host, connecting with “**backupuser**” via SSH on port 22. Our local “**tank1**” and its dependents will be replicated to the remote dataset “**zpoolbackup/backups**” and it will be kept in “**sync**” with local snapshot creation. Before this replication can occur a few things must take place on the remote side. First, the remote system must also have a valid ZFS zpool, which is compatible with the version of ZFS on your local machine. Second, you must ensure that the user you are connecting with has permissions to create the ZFS datasets on the remote side:

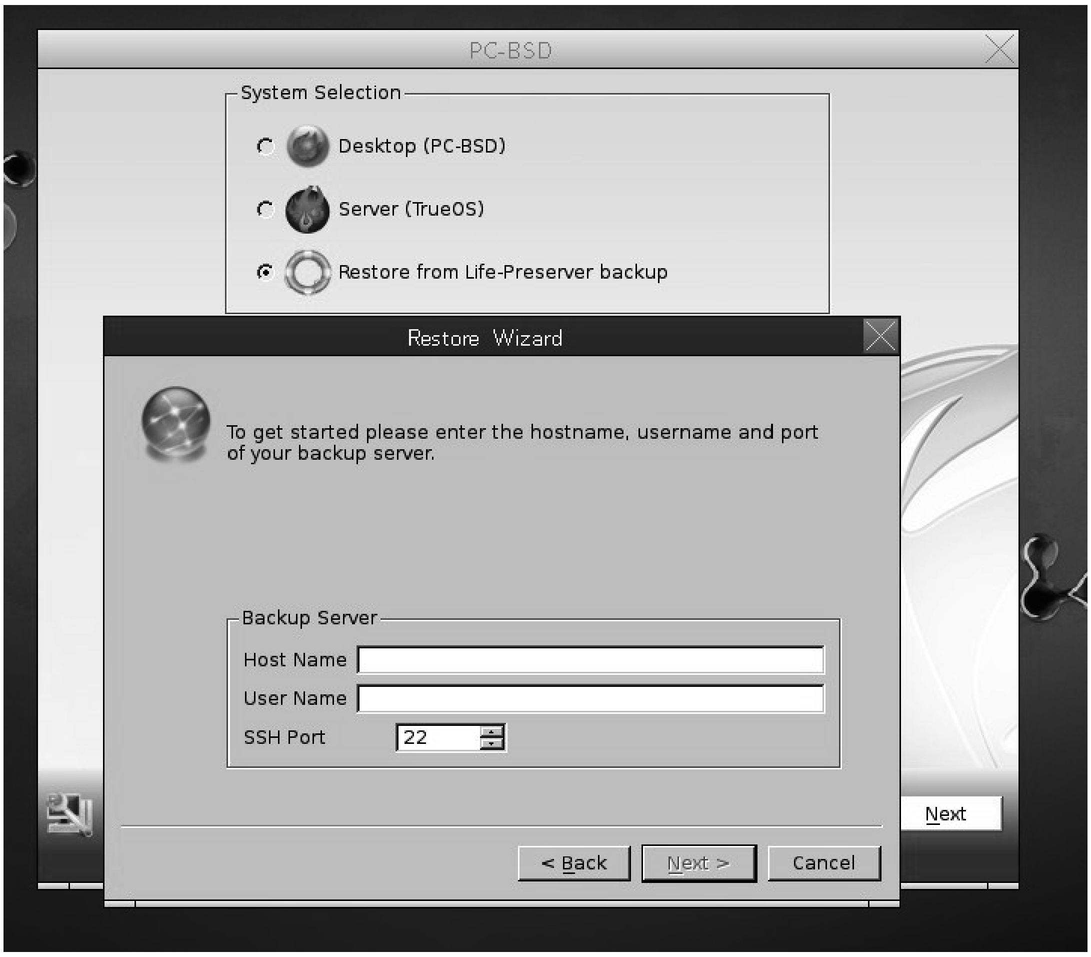
```
# zfs allow -u <user> create,receive,mount,userprop,destroy,send,hold <remotedataset>
```

Lastly we will need to setup authorized SSH keys between the local and remote system, allowing password-less logins to take place. With these steps followed the local system will now have access to “replicate” the data from your local machine to the target remote dataset. This ensures that even if something should happen to the local box, a copy of your data will still be retrievable from a different location. Replications are sent incrementally, and can be very quick, but the initial sync may take some time depending upon disk speed and network conditions. During the replication phase automatic snapshot pruning will be halted in order to prevent removing the replicating snapshots. The specific steps taken during a replication are as follows:

- First, check if the specific zpool / dataset exists and is flagged for replication.
- Check the 'backup:lpreserver' ZFS property to see when the last replication was performed.
- Check if we are doing a first-time or manually specified "full" replication, or else perform an incremental replication.
  - If doing a first-time replication, create the new target dataset on the remote system
- Begin a ZFS send / recv command, running across SSH.
- If the ZFS send / recv is successful, then set the 'backup:lpreserver' ZFS property marking the last successful replication.
- Save a list of the zpool dataset(s) and properties. This is used when doing bare-metal restores.
- If enabled, e-mail a summary of results.

Restoring

Booting a PC-BSD 9.2 or later install media gives you the option to not only install fresh, but restore from a Life-Preserver replication. During the installation you may select the "**Restore from Life-Preserver backup**" option to begin the process. The graphical wizard will then walk you through the process of connecting to the backup server, as well as setting options for restoration. A helpful feature is the ability to change your initial ZFS pool options during a restore. This can be used in the case of upgrading to a larger disk, or migrating from a single-disk system to a ZFS software raid, using all the replicated data you previously saved. Once the restore is finished, you will be able to reboot the system back to the state it was in during the last replication.



( ) Desktop (PC-BSD)
( ) Server (TrueOS)
[✓ Restore from Life-Preserver backup]

To get started please enter the hostname, username and port of your backup server.

Host Name
User Name
SSH Port 22

Illustration 6: Restoring from a Life-Preserver backup

The PC-BSD installer GUI is simply a front-end to the "pc-sysinstall" backend, which is scriptable for this kind of bare-metal restore. This allows users to customize their restore options, or even use it as a method of system deployment / cloning. The relevant portions of the pc-sysinstall script will look something like this:

```
installMode=zfsrestore

sshHost=mybackupserver
sshUser=sshbackupuser
sshPort=22
sshKey=/root/.ssh/backupkey_rsa
zfsProps=.lp-props-backups-oldhostname
zfsRemoteDataset=/backups/oldhostname
```
(pc-sysinstall configuration file)

(For further information about pc-sysinstall, refer to my previous talk from BSDCan 2010: http://www.bsdcan.org/2010/schedule/events/173.en.html)

Mirroring

In addition to being able to snapshot, replicate and restore your data, you may need a more "high-availability" solution. During the initial system installation you are given options to create a software raid, either in a direct mirror configuration, or using "raidz" levels 1-3. However if you only installed to a single disk, you can still take advantage of ZFS mirroring at any time. To get started you will need a second disk drive, the same size as the target disk or larger. This second disk can be connected internally or externally via USB 3.0, eSATA or any other method which presents a raw disk device to your system.

With the disk drive connected, it is now possible to "attach" it to your existing ZFS zpool using the following command:

```
# lpreserver zpool attach tank1 /dev/da0
```

This command will take the disk drive "**/dev/da0**" and attach it to the "**tank1**" ZFS zpool. During the setup, this disk will be wiped and new partitions created to match the first disk in your existing zpool configuration. Once the drive is attached, it will begin to sync up with the existing zpool data, known as the "resilvering" phase. This may take some time depending upon the speed and size of your disks. Additionally the disk will be stamped with GRUB, making it bootable. The advantage of this is that should your original disk in the array fail, you can continue to run from the second drive without any data loss. If you need to shut the system down and swap-out the bad disk, booting is still possible from the second drive. The replaced disk drive can then be attached in a similar manner, which will re-sync data from the secondary disk. This method can be used to attach as many disks as you require, giving you any level of extra redundancy that your system may need. As long as a single disk in the array remains intact your data will be safe and the system bootable.

In the case of using external disks, it is even possible to disconnect and re-connect at a later time using the "**lpreserver**" command:

```
# lpreserver zpool offline tank1 /dev/da0s1a
# lpreserver zpool online tank1 /dev/da0s1a
```

When the disk is re-connected and set in the "online" mode, the resilving process will begin again, re-syncing the external disk incrementally. Using the "offline/online" method is recommended when you plan on re-attaching the disk, otherwise you may need to resilver all the data over again. This can be used in the case of a portable machine, such as a laptop, which is connected to an external disk drive periodically for backups.

Monitoring

Last but not least, the "**lpreserver**" utility can also provide basic reporting about snapshots and your disk status. By using the "**lpreserver set email <address>**" command it is possible to enable reporting about the status of your system. Life-Preserver can be setup to send either all messages, warnings, or errors only using the "**lpreserver set emailopts ALL/WARN/ERRORS**" command. Currently Life-Preserver can report the results of automatic snapshots and replications, as well as provide warnings when a disk has failed, or if the zpool is running low on disk space. If your network only allows the usage of a smtp server, you can install the “ssmtp” utility to enable mailing through that gateway.

Conclusion

In this article we have taken a look how ZFS can provide support for Boot-Environments, as well as keeping your data safe using the Life-Preserver utility. By using these tools, users can better manage and plan for disasters on desktops, laptops or servers in a variety of unique situations. While the Life-Preserver utility is included in all installs of PC-BSD or TrueOS, it is also available to FreeBSD users in the public PC-BSD pkgng repo, or via source in GitHub.

References

PC-BSD pkgng repo
http://wiki.pcbsd.org/index.php/Turn_FreeBSD_into_PC-BSD%C2%AE

PC-BSD GitHub
https://github.com/pcbsd/

PC-BSD Mailing Lists, Forums and Bug-Tracker
http://lists.pcbsd.org
http://forums.pcbsd.org/
http://trac.pcbsd.org/

Netmap as a core networking technology [*]

Luigi Rizzo, Giuseppe Lettieri
Università di Pisa
{rizzo,g.lettieri}@iet.unipi.it

Michio Honda
NEC Europe Ltd
michio.honda@neclab.eu

ABSTRACT

netmap is a network I/O framework for FreeBSD and Linux that provides a 10-fold speedup over ordinary OS packet I/O mechanisms. netmap uses less than one core to saturate a 10 Gbit/s interface with minimum size frames (14.88 Mpps) or switch over 20 Mpps on virtual ports of a VALE switch (part of the netmap module).

In the past two years we have extended the framework in many ways, and it can now replace native in-kernel software switches, accelerate networking in virtual machines, and be used by unmodified applications based on libpcap.

In this paper we give an overview of the design principles used in netmap, present the current features of netmap and the VALE software switch, and present some applications where we have used these systems.

1. INTRODUCTION

In 2010, in response to the advent of software defined networking and the demand for high speed software packet processing, we set out to investigate how to fill the gap between the speed of OS mechanisms (sockets, bpf etc.) and the requirements of 10 Gbit/s and faster interfaces. Our goal was to come up with an I/O framework that was more portable and easier to use than the various custom solutions proposed at the time.

The initial result of our work, the netmap framework [9] that we designed in 2011, provided dramatic speedups to network I/O, over one order of magnitude for basic packet processing tasks: traffic sources and sinks, software packet forwarding, monitors. Our first demonstrators for the effectiveness of our approach were custom applications, or modifications of existing ones (such as the userspace version of OpenvSwitch [11]).

Following this initial result, we began to evaluate how the techniques used in netmap could be generalized to accelerate other network-related software functions. We then used netmap to implement a fast software switch called VALE [12]; enabled a seamless connection of physical devices and host stack to a VALE switch; extended the switch to act as a generic network dataplane [5]; and managed to reach bare metal speed with virtual machines [13] emulating ordinary NICs.

In parallel with our work, a large number of people have started to use netmap in research and production, often providing useful feedback on possible new functionalities.

Today, after a couple of large rewrites of its internals, netmap, VALE and related tools have become extremely powerful and simple to use, allowing users to concentrate on optimizing applications rather than having to focus on the low level details of packet I/O.

The goal of this paper is to present the current state of netmap and VALE, discuss the design decisions we made, and present their performance.

2. BACKGROUND

The cost of I/O processing, be it for network or storage, have a large per-transaction component, and a comparatively small per-bit one. Systems and protocols thus try to amortize the former over sufficiently large amounts of data. The original packet sizes (64-1518 bytes) chosen for ethernet networks were however chosen on different criteria. The lower bound was set to be large enough to allow collision detection on sufficiently long distances in presence of repeaters, but small enough to avoid too much overhead when sending short messages. The upper bound was chosen, somewhat arbitrarily, to avoid individual users monopolizing the link for too long intervals of time.

When moving from 10 Mbit/s to higher link speeds, the minimum packet size was kept unchanged, mostly for backward compatibility with existing software. Unfortunately this has stretched the maximum packet rates by three orders of magnitude, resulting in incredibly tight time budgets to process a packet on 10 Gbit/s links. Considering the framing overhead, these can carry up to 14.88 Mpps (million packets per second), or one every 67.2 μs.

[*]The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n.287581 - OPENLAB.

Even if rates are much lower (0.81 Mpps) with the largest frames, we cannot ignore such high packet rates: certain pieces of equipment (notably, switches, routers, firewalls and intrusion detection systems) will be subject to uncontrolled input streams and must be able to deal with worst case conditions.

A solution adopted in the past to cope with high packet rates is to have the network interfaces (NICs) split the traffic among multiple receive queues according to some criterion (RSS, see [8]), and then assign one core per queue. But an attacker (or even a legitimate user) can cause all traffic to go to a single queue, hence defeating this technique. The real solution to deal with these packet rates is to make systems more efficient and performant by reducing processing to the essential.

This contrasts with the approach followed in the design of network stacks, which normally try to support a huge number of options and functionalities, resulting in significant work at runtime to determine the actual processing requirements.

3. NETMAP

Netmap uses this minimalistic approach to achieve efficiency. We have presented extensively the netmap API in previous work [9, 10], so we only recall here the most important design principles as they relate to the content of this paper.

First and foremost, the netmap architecture relies heavily on I/O batching, to amortize certain costs (system calls, locking, I/O access, prefetching and pipeline stalls) at many different layers. In contrast, the socket API and other software interfaces for packet I/O commonly used in the OS tend to deal with one packet at a time.

Our second design principle is to use a very simple and uniform representation for packets: netmap normally uses one buffer per packet, large enough to hold a maximum sized segment. There are no flags or options to support specific hardware offloading features (checksums, segmentation, VLAN tag insertion/removal, etc.). In contrast, typical OS representations (`mbuf, skbuf, NDISpacket`) allow splitting a packet into an arbitrary number of buffers, share them, and delegate part of the processing to the hardware. The flexibility that comes from this approach carries significant runtime costs: at every software layer, even simply reading a block of data needs to locate the right buffer and make sure it is contiguous; writes also need to additionally check that the buffer is not shared, and duplicate it in case.

This bring us to our third design principle: no dynamic allocations. In common network frameworks, buffers tend to be allocated and freed at least once during the lifetime of a packet, thus adding significant costs to the processing. Conversely, in netmap, all transmit and receive buffers are allocated only once when the interface is first brought up. Applications then have full control over them, and it is their responsibility to make sure that the NIC does not run dry of receive buffers. Besides saving a recurring cost, this has the side effect of additional simplifications in the processing code (as an example, allocation failures cannot occur in the datapath).

As a final strategy, netmap uses memory mapping to share buffers and metadata between applications and the kernel. This is done for performance reasons (saving one data copy in some cases) and also reduces the length of certain critical paths within the kernel, improving the potential parallelism.

Memory mapping undeservedly appears in the name of our framework, but it is by no means the main contributor to its performance: the overall architecture that exploits batching is by far the most important factor.

3.1 Programming interface

A netmap application typically goes through three phases: create a file descriptor and bind it to an interface; move data between the application and the NIC; and synchronize with the OS to transmit or receive packets.

3.1.1 Initialization

In netmap, the initialization uses an `open()` and an `ioctl()` to make the binding, plus an `mmap()` to access packet buffers and descriptors. Its basic form is the following (the three calls are often hidden within a support library):

```
struct nmreq nmr = {};
void *mem;
int fd = open("/dev/netmap", O_RDWR);

strcpy(nmr.nr_name, "eth0");
nmr.nr_version = NETMAP_API;
ioctl(fd, NIOCREGIF, &nmr);
mem = mmap(0, nmr.nr_memsize,
    PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
```

Binding a file descriptor to a NIC gives access to replicas of the transmit and receive queues of the NIC itself (we call these replicas “netmap rings” or rings for brevity), and direct access to the data buffers, as described in Section 3.2. A file descriptor can be bound to a single TX/RX ring pair, or to all ring pairs associated with the NIC.

As a side effect, binding also disconnects the NIC from the host network stack. However the OS is unaware of the disconnection, and will still send traffic to that NIC (and expect to receive from it). These outgoing packets (and incoming ones) are made available

```
   netmap_if            netmap_ring          netmap buffers
+------------+      +-------------------+
|   ...      |  +-->|    ...            |
| ni_tx_rings|  |   |    head           |
| ni_rx_rings|  |   |    cur            |
|   ...      |  |   |    tail           |
+------------+  |   |    ...            |
| txring[0] ----+   +-------------------+     +----------+
| txring[1] --->... |len flags    idx ------->| buffer   |
|   ...      |      |       ptr         |     +----------+
| rxring[0] --->... +-------------------+
| rxring[1] --->... |len flags    idx ------>   ...
|   ...      |      |       ptr         |
+------------+      +-------------------+
                           .......            +----------+
                    |len flags    idx ------->| buffer   |
                    |       ptr         |     +----------+
                    +-------------------+
```

Figure 1: The data structures used by netmap to share buffers and descriptors between user applications and the system.

to netmap clients through another ring pair, called the "host" TX/RX rings.

3.2 Moving data

The `open()/ioctl()/mmap()` sequence gives access to the data structures in Figure 1, the most important one being the `netmap_ring` which includes head and tail pointers to a circular array of buffer descriptors. "RX" rings contain packets coming from NICs (or from other types of port, such as the host stack or VALE ports). "TX" rings are for sending packets out. Regardless of the direction, the ring should be thought as a container for a pool of buffers, whose producer is always the kernel, and the consumer is always the netmap client.

Rings and descriptors have changed slightly in recent versions, in response to user experience and feedback. In particular the ring now has three indexes (`head, cur, tail`) pointing into it.

`tail` is always modified by the kernel, which appends new buffers to the ring as it sees fit; this only occurs during the execution of a netmap system call.

`head` and `cur` can be modified by the consumer (the user process), but only when a netmap system call is not executing. Advancing `head` returns buffers to the kernel at the next system call.

The role of `cur`, is to acknowledge new data and indicate the next "wakeup point" without having to return buffers at the same time. This can be convenient at times, e.g. when an application needs more time to complete it processing.

Buffer descriptors ("`netmap_slot`") are 16 bytes each, and include a 16-bit length, 16-bit flags, a 32-bit buffer index, and a pointer.

Netmap buffers are statically allocated by the kernel and pinned in memory when the file descriptor is bound. They have indexes ranging from 2 to a given maximum value (indexes 0 and 1 are reserved); they are contiguous in the user (virtual) address space, but can be randomly scattered in the physical address space, the only constraint being that each individual buffer must belong to a single page. Indexes can be easily mapped to different addresses in the kernel (through an in-kernel lookup table) and in user processes (multiple clients can map the same netmap memory at different addresses).

The pointer is a recent addition that can help save one data copy when dealing with VALE switches and clients that expect to build outgoing packets in their own buffers (there are many such examples, the main one being virtual machines). To guarantee isolation among clients, VALE switches need to make a payload copy when moving data from source to destination. Since the data copy occurs in the context of the sending thread and without holding an exclusive lock, the source buffer does not need to be pinned, hence we can use an external buffer as data source (this is only valid on transmit rings).

Flags were originally used only to request a notification on transmission completion, or indicate that a buffer index had been modified (e.g. swapped with some other netmap buffer). Completion notifications are normally posted only lazily for efficiency, but there are cases (e.g. last transmission before terminating) where we do want to be notified. Information of buffer address changes is also important as it may be necessary to reprogram the IOMMU, and certainly to update the buffer's physical address in the NIC.

We have now extra flags to indicate whether this slot refers to a user-allocated buffer, and also support a few more features:

scatter-gather I/O Virtual machines and bulk data transfers use Transmit Segmentation Offloading (TSO) to achieve very high data rates. TSO sends out very large segments (up to 64 KB and more), which are too large to fit in a single buffer. We have then extended the API to support multisegment packets; the `NS_MOREFRAG` flag is set on all but the last segment in a packet.

transparent mode Netmap disconnects the NIC from the host stack, but many applications need to reestablish the connection for some or all traffic. This can be done by opening two file descriptors and manually moving buffers from one side to the other, but such a common task deserves explicit (and optimized) support.
`NIOCREGIF` can request a special mode of operation where, after netmap processing, all packets released by an RX ring and marked with the `NS_FORWARD` flag are passed to the transmit port

on the other side (from NIC to host port and vice-versa).

3.3 Synchronization

System calls are used to communicate the status of the rings between the kernel and the user process. Netmap uses `ioctl()` and `select()/poll()` for this. These system calls perform the following two actions:

- inform the kernel of buffers released by the user process (which has advanced the `head` index for this purpose). The kernel in turn queues new packets for transmission, or makes the buffers available for reception.
- report new buffers made available by the kernel (which advances the `tail` index). This makes available to the user process additional, empty TX buffers, or new incoming packets.

The netmap `ioctl()`s (`NIOCTXSYNC` and `NIOCRXSYNC`, separate for the two directions) are non blocking, and suitable for use in polling loops.

`select()/poll()` are instead potentially blocking, and they unblock when `tail` moves past the `cur` index in the ring. Their support makes netmap file descriptors are extremely easy to use in event loops.

Netmap system calls are heavily optimized for common usage patterns. As an example, event loops often only poll for input events (output tends to be synchronous in many APIs, but not in netmap); so `select()/poll()` by default handle TX rings (without blocking) even if output events are not specified. As another example, `poll()` is often followed by a `gettimeofday()`; to save the extra system call, netmap also writes a fresh timestamp in the netmap ring before returning from a poll.

4. NETMAP INTERNALS

Netmap is implemented as a kernel module and includes three main parts: the control path (device initialization and shutdown), the datapath (how packet payload is moved from and to the NIC), and synchronization (how transmit and receive notifications are passed around). All of them involve a generic OS component and a device specific one.

4.1 Source code organization

The netmap source code is split in multiple files according to the function: core functions, memory allocation, device-specific support (one file per driver), emulation over standard drivers, VALE switch, segmentation, etc. Additionally, we have three common headers: `net/netmap.h` contains definitions shared by user and kernel; `net/netmap_user.h` is for user-only definitions; `dev/netmap/netmap_kern.h` has the main kernel-only definitions.

Netmap is designed to support multiple operating systems (FreeBSD and Linux at the moment) and multiple versions of them, and we try to retain feature parity at all times. To ease the task, we use the same code base for both FreeBSD and Linux versions (and possibly other OSes we will port it to). One file per OS contains OS-specific functions, such as those to build modules.

In order to avoid an "esperanto" code style, we use a single family of identifiers and APIs when possible. We chose the FreeBSD APIs because this was the platform where netmap has been originally developed, but Linux would have been an equally reasonable choice. We carefully use only APIs that are expected to be available (or reasonably easy to emulate) on all platforms.

Kernel functions and data structures normally differ heavily between OSes, but we found that the network-related functions are reasonably similar. Naming differences can often be solved with macros or inline functions to do the remapping. In other cases, small wrapper functions do the job with little if any penalty at runtime.

4.2 Code distribution

Standard FreeBSD distributions already include the netmap/VALE code and necessary driver support. Linux distributions have not (yet ?) included our code, so we need to patch original drivers. Our distribution includes patches for Linux kernels from 2.6.30 to the most recent ones (3.13 at the time of this writing). The Makefile used to build netmap modules has code to fetch drivers and apply patches before compiling.

4.3 Control path

The OS-specific part of the control path relies on character devices and `ioctl()`'s to configure the device, and `mmap()` to share the data structures in Figure 1 between the kernel and the user process.

Memory is only managed by the OS, so the mmap support has OS-specific functions but no device dependencies. `NIOCREGIF` instead acts on the configuration of the NIC, hence its device specific part requires some programming information for the NIC involved. We minimize this problem by using the standard device driver for most of the initialization, with only minor tweaks (generally simplifications) to the code that initializes the transmit and receive data structures when entering netmap mode. The necessary (limited) amount of programming information for these modifications can be generally derived from the source code of the original driver.

4.4 Data path

Moving packets from and to a NIC is time-critical and must be done very efficiently. Here we cannot rely on existing driver routines, as they are too generic and inefficient for our purposes. Hence we extend drivers with two new methods, for the transmit (`*_txsync()`) and receive (`*_rxsync()`) side of the device.

Following the description in Section 3.2, each method has two sections: one to pass buffers to the NIC (outgoing packets for TX rings, empty buffers for RX rings), and one to pass buffers to the user process (new buffers for TX rings, newly received packets for RX rings).

The `*sync()` methods do require programming information for the NIC, but all they need to do is map between the netmap and internal packet representations, and access the relevant NIC registers to determine the evolution of I/O in the NIC. Due to our simple packet format and the absence of accelerations, also in this case it is often possible to infer the necessary programming information from the drivers' source code.

The `*sync()` methods are optimized for batching, and normally invoked in the context of the user process during an `ioctl(), select()` and `poll()`. Interrupts, in netmap mode, only need to wake up processes possibly sleeping on a `selinfo/wait_queue`. This has several useful consequences:

- the placement of interrupt threads and user processes has no influence on cache locality (though it may impact the scheduling costs);
- even under extreme input load the system does not livelock, as all processing occurs in user threads;
- system load is easy to control through scheduler priorities;
- interrupt moderation delays can be propagated to user threads.

4.5 Synchronization

Many systems with the same goal as netmap only support busy waiting on the NIC to detect transmit completions or new incoming packets. This avoids delays in notifications, which can be large due to interrupt moderation (up to 10..100 μs), and handoffs between interrupt handlers, threads and user processes (up to several microseconds, if not further delayed by other higher priority threads). Latency optimization however comes at high cost: the process doing busy wait causes full CPU occupation even with little or no traffic.

Netmap does support this option through the `ioctl()`s, which always call the corresponding `*sync()` method of the underlying device; however this is not the preferred mode of operation.

In netmap we were aiming at an efficient use of resources, and designed the system to use standard unix mechanisms for synchronization. This is actually completely trivial, and only requires to direct interrupt handlers to issue a `selwakeup()/wake_up()` on the `selinfo/wait_queue`.

The poll handler, `netmap_poll()`, which is part of the generic code, is heavily optimized for common usage patterns and programming idioms. We already mentioned the non-blocking handling of TX rings even in absence of requests for output events, or the generation of fresh timestamps on return. We also try to return efficiently when unnecessary calls are made (e.g. the rings already have buffers before the call), skipping the invocation of the `*sync()` functions in these cases.

4.6 Native support in device drivers

One of the key design principles in netmap is to make as few as possible modifications to the system. This especially applies to device drivers. We hook into drivers in three places:

device attach where we simply register with the OS the availability of native netmap support;

queue initialization called when the NIC switches in and out of netmap mode. Entering netmap mode, both TX and RX queued are pre-filled with netmap buffers instead of `mbuf/skbuf`s;

interrupt handlers where we only need to invoke a `selwakeup()/wake_up()`.

All these places are relatively simple to identify in the existing drivers' sources, and changes are only a handful of lines of code.

The `*xsync()` methods are brand new, but they have a common template, with the NIC-specific parts being limited to the parts that access NIC descriptors and queue pointers. To decouple the netmap code from vendor drivers, the `*xsync()` and any other new code is in an external file `#include`'d by the driver. Patches to the original driver and patches are clearly identified in `#ifdef DEV_NETMAP / #endif` sections.

4.7 Netmap emulation on standard drivers

We have only implemented native support for a small number of NICs. The benefits of netmap are mostly for fast devices (10 Gbit/s) or slower CPUs, and this reduces the number of candidate NICs. The number is further reduced because we found (after the fact!) that many "fast" NICs are barely able to reach 20-30% of the peak packet rate, due to their own hardware limitations. Hence providing native support for them was pointless. Likewise, we have not dealt with devices with undocumented architecture or programming information, as it would be difficult to provide a reliable implementation.

For devices without native netmap support, we have recently added a netmap emulation mode that works over unmodified drivers. This permits to experiment with the netmap API on any network device. Provided the hardware is not otherwise crippled, it also gives some performance advantage over other I/O methods, although much more limited than a the native netmap API.

The performance gains in emulation come mostly from batching, and from bypassing the whole socket layer. On the transmit path we also save the allocation overhead with a simple trick: we hold an extra (counted) reference to the the `mbufs/skbufs` queued for transmission, so when the driver frees them, no actual deallocation occurs. In order to receive notifications of transmission completions without hooking into the interrupt handler, we use another trick: we register a custom deallocator for the buffers, and once every few slots (say, every half ring) we leave the buffer with only one reference. On these buffers, the transmit completion causes a call to our deallocator, which we handle as an interrupt notification, and use to mark as free all preceding buffers in the ring.

As a final optimization, `mbuf/skbuf`s in the transmit ring can be made to point to the netmap buffers, thus removing the need for a data copy.

With these solutions, netmap emulation over an Intel 85299 NIC gives about 4 Mpps per core, peaking at around 12 Mpps with 3 cores. This is 2-3 times faster than ordinary sockets, although not as fast or efficient as native netmap mode on fast NICs.

On the receive side, we intercept the input handler that normally passes packets to the host stack. Our handler queues the mbufs in a temporary queue and acts as if an interrupt had been received, issuing a `selwakeup()/wake_up()`. We cannot unfortunately optimize allocations or copies as we do on the transmit side. Consequently, receive performance does not scale equally well. On the same Intel NIC, we measured up to 4..6 Mpps depending on the input patterns.

4.8 Extra buffers

Support for zero-copy is with no doubts one of the most appealing features of netmap, and the preallocation of transmit buffers suits well to systems where packets are processed to completion without intermediate queueing. Sometimes, though, incoming traffic must be held for some reason (slow output devices, waiting for additional information, shaping). A recent extension to netmap permits the request of additional buffers (even in large amounts) during the `NIOCREGIF` call. These are returned as a linked list of buffers, and applications can use them to replenish the receive queues if incoming packets cannot be used immediately.

5. VIRTUAL PORTS AND SWITCHES

The netmap API proved so effective that we soon wondered how it could be used as a generic interprocess communication mechanism. Since the initial abstraction was that of a network interface, our next step has been to build a software switch, called VALE (for VirtuAl Local Ethernet).

In its original version, VALE behaves as a learning ethernet bridge, and as such it distributes incoming packets to its output ports depending on the destination MAC address. The forwarding table is learned dynamically, and broadcast is used when the destination is unknown.

VALE is implemented by the same kernel module that implements netmap. Applications access a NIC or a port of a VALE switch exactly in the same way, the only difference being the name passed to the `NIOCREGIF` call. A name of the form `valeXX:YY` attaches the file descriptor to port YY on switch XX (both created dynamically if they do not exist), otherwise it refers to a NIC.

The only difference, from a user's point of view, is that each VALE port uses a separate memory region, whereas NICs (and host stack ports) all use the same common memory region. The practical implication is that one can do zero-copy forwarding between NICs and/or the host stack by simply swapping buffers in the rings, whereas connecting VALE ports requires an actual data copy.

The difference is not arbitrary and comes from two reasons. First and foremost, clients of a switch (virtual machines, typically) do not necessarily trust each other and should not be able to interfere or see others' traffic. Of course we use memory protection mechanisms for that, but underneath we have the choice between remapping pages from one space to another, or copying data. However, altering page tables is extremely expensive in multicore machines as it requires to notify all cores in the system.

The second reason for using copies is that when delivering data to multiple ports (multicast, broadcast or unknown destinations), we need to either copy or reference count the buffers. The latter would introduce huge complications in buffer management so we resort to copying also in this case.

Despite the cost of copying, VALE has a very high throughput: on a modern i7 CPU, we reach up to 20 Mpps for short frames, and 70 Gbit/s for 1500 byte frames, from one input to one output. Even higher throughput is achieved when using multiple rings on each port, or multiple sender to the same destination

5.1 VALE internals

To reach this level of performance, VALE is designed to exploit batching heavily, and to this purpose it processes packets in three stages: input prefetching (to amortize memory access delays); destination lookup (to help building output batches); and output queueing (to amortize locking overhead). The details are described in [12].

All the work in VALE forwarding is performed in the context of the sending thread, and copying data to the output port is one of the most expensive operations. To improve parallelism, even output queueing is further split in three stages. We first quickly reserve slots in the output ring, under lock. Then multiple senders can work in parallel, each one on its own portion of the ring, to copy packets from source to destination. Finally, the lock is re-acquired and notifications to the receiving process are sent in the proper order.

This organization has a significant impact on throughput: we can now move traffic from multiple sources to the same destination at over 70 Mpps, compared to the 20 Mpps we could achieve with the previous architecture.

But speed is not the only benefit of this new organization: the fact that data access is now lockless simplifies the overall locking design, and also allows the sender process to block (e.g. because of a page fault). As a result, we can now source data from buffers that are not wired in RAM, such as generic user-allocated memory. This new possibility is what led to the introduction of a pointer field into the netmap slot (see Section 3.2).

5.2 Netmap Pipes

Pipes are a widely used Unix IPC mechanism, and there are many variants of this concept, including AF_UNIX sockets, pseudo ttys, and the loopback and `epair` network interfaces.

A netmap *pipe* is identified by a *base name* and pipe ID, and is made of two netmap ports connected by a virtual crossover cable. A VALE switch with two ports is functionally almost completely equivalent to a netmap pipe.

A real netmap pipe, however, has a peculiarity: its two endpoints share the same memory space, which is a natural choice since all traffic is bound to be seen by the other endpoint. Not having to make copies, learning or forwarding decisions, a netmap pipe is incredibly fast, peaking at around 90 Mpps irrespective of packet sizes.

A netmap pipe also shares memory with all pipes and ports with the same basename. This is done specifically to enable one common task: software demultiplexing (Figure 2)

Many packet processing tasks in fact need to distribute input traffic to a number of different processing modules, either for scalability or for application specific requirements. Demultiplexing is sometimes supported by the hardware, as it happens with modern multi-queue NICs with Receive Side Scaling (RSS) support: these include filters that use exact match patterns or hash functions to split input traffic among queues.

The VALE switch also performs a form of demultiplexing, but the assumptions on which it operates (untrusted communicating parties) and the location within the kernel affect its efficiency and ease of use.

```
            +----------+
            |          |<--> worker
            |          |
   NIC <--->|  demux   |<--> worker
            |          |
            |          |    . . .
            |          |
            |          |<--> worker
            +----------+
```

Figure 2: The use of netmap pipes for software demultiplexing. All connections are bidirectional and can use the netmap API, possibly over the same memory region.

Netmap pipes sharing memory with an interface make it extremely easy and efficient to build such demultiplexers as a user process, and let them communicate with others. Depending on the requirements, the demux process can chose to use shared memory (and zero-copy) between the interfaces and the NIC, or different address spaces to retain isolation among same or all processes.

5.3 Attaching NICs and host stack to VALE

Software switches normally interconnect virtual ports, NICs and the host stack. Attaching the latter two to a VALE switch can be trivially done by userspace process, but that incurs additional thread handoffs and delays. Hence in recent versions of VALE we incorporate this functionality in the switch itself. When a NIC is attached to a VALE switch, it is put in netmap mode, and both the NIC and the host sides are connected to two ports on the switch. The traffic coming from NICs or from the host stack now may be processed within kernel threads (NAPI, ithreads, softirq handlers).

The use of netmap mode for the interconnection of the switches is fundamental to achieve good performance. While a native Linux or FreeBSD bridge barely reaches 2 Mpps a VALE switch using a single core and two NICs can forward traffic at over 12 Mpps (with further improvements being possible by removing the unnecessary memory copy that the code does at the moment).

5.4 Programmable VALE switches

By default a VALE switch implements a learning ethernet bridge: on each packet the source MAC address is looked up to update the forwarding table, and then the destination address is used to compute the output port. The dataplane architecture described in the previous sections makes it sure that relevant packet payload is close to L1 cache when needed, and that the actual queueing and copying is taken care of in an efficient way.

The forwarding decision, in the end, is just a simple and small function that does not need to know anything about the dataplane. As such, it was straightforward to add programmability to a VALE switch and let it run a different function. In the current implementation, kernel modules can register with a VALE switch and replace the default forwarding function with a custom one, while using the data plane for efficiently moving data around.

We have used this mechanism for several examples, including a port demultiplexer [5], and an accelerated in-kernel OpenvSwitch.

6. APPLICATIONS

For maximum performance, applications can use the native netmap API directly. This exploits batching and zero copy data access. The first applications we wrote to test netmap (a traffic source, a traffic sink, and a simple interconnection between two interfaces) used this approach. Results were fantastic, often 10-20 times faster than equivalent applications using conventional I/O systems.

This however happened because the applications we were comparing to were dominated by the cost of I/O, and our replacement were carefully designed to exploit batching and the other features that make netmap fast. Unfortunately, in many cases it is difficult to modify the inner structure of application code, and we can only replace the network I/O routines with direct access to the netmap API. When we started doing this, results were a mixed bag, in some cases achieving speedups of 5-10 times over the original, in others we were limited to much smaller increases (1.5-2 times). The latter numbers would be seen as a success in many other circumstances; but in this case they are an indication that network I/O is not the main bottleneck in the system.

If the application has other bottlenecks, the direct use of the netmap API is not worth the effort, and we are better off following a different approach, namely emulating some other standard API (such as `libpcap`) on top of netmap.

6.1 Netmap-based libpcap

Our initial implementation of netmap provided a very primitive library that implemented parts of the `libpcap` API. We could then use a subset of libpcap-based applications on top of our library without recompiling, and by simply pointing the dynamic linker to our library. This approach was however too limited and we recently implemented full netmap support for libpcap.

The set of functions that libpcap exposes to clients is extremely large and possibly redundant, but internally the interface with devices is relatively small, and it boils down to an open and close function, and two methods to read and write packets.

The read method (similar to `pcap_dispatch()`) takes as input a callback and a count, and applies the callback to incoming packets until the count is reached or a signal is received. This matches very well the way netmap works, because we can run callback directly on the netmap-supplied buffer, and exploit the batching that is inherent in netmap operation.

The write method (similar to `pcap_inject()`) takes a pointer to a use-specified buffer and is supposed to push the packet to the output, so that the buffer can be reused when the function returns. This is slightly less efficient for our purposes, because it forces a data copy and does not directly support batching (though it might be possible to operate lazily and actually send packets at the next netmap system call).

An interesting feature of this work is that `libpcap` provides filtering support through BPF [7] also for devices (such as netmap) that do not implement it natively.

6.2 Firewalling and dummynet

Firewalls can greatly benefit from high speed network I/O subsystems. The firewalls implemented within an OS are often limited in performance by the speed of the underlying device drivers. As a result, not only it is hard to tell how fast they could run on top of a better infrastructure, but it is also difficult to evaluate the effect of performance optimizations on the firewall itself.

To address this problem we have made some small modifications to the `ipfw` source code so that it can now be run in user space on top of netmap. This was done both as a proof of concept – to see how fast we could run the firewall and the `dummynet` network emulator – and also for possible use in production.

The resulting system is made of two parts (see Figure 3): the filtering function, which in the original implementation ran in the kernel, and the user interface, `/sbin/ipfw`. Communication between the two normally uses a control socket, but in this case both components run in userspace so we built some glue code to pass sockopt messages across an ordinary STREAM socket, running on top of the loopback interface.

The filtering function is normally invoked as part of the input/output code in the network stack. In our case we built a simple event loop which monitors two netmap file descriptors (and the control socket), wraps the netmap buffers into pseudo `mbufs`, and invokes the firewall code on them (Figure 3). Depending on the outcome the packet is then either dropped, passed to the output interface (which can be done without copying), or queued into a dummynet pipe. When this happens, we make an actual copy of the packet so that the netmap buffer can be returned to the input ring.

```
+-----------------+           +----------------+
|                 |           |                |
|   /sbin/ipfw    |           |   kipfw        |
|                 |           |                |
|                 |  TCP      +----------------+
|                 |<--------->|  glue code     |
+-----------------+           +----------------+
   |        |
  NIC      NIC
```

Figure 3: The architecture of the userspace ipfw. A shim layer is in charge of communicating with interfaces and passing packets to the former kernel component. The control socket is emulated over a regular TCP connection.

With this arrangement, and connecting the firewall between two VALE switches, we managed to achieve over 6 Mpps with a simple ALLOW rule, and about 2.5 Mpps when passing packets through a dummynet pipe. In both cases, this is about 5 times faster than the in-kernel implementation. Performance over physical interfaces should be even higher because we do not need to do the extra data copy that exists with VALE switches.

The use of extra buffers should also allow us to improve performance on dummynet pipes by a very large factor, possibly matching the speed of the pure firewall part.

7. RELATED WORK

Netmap is neither the only nor the fastest tool for network I/O. However it is, in our opinion, the most general and easiest to use solution to the problem of fast network I/O.

In addition to some proprietary options, there are three main alternatives to netmap.

Intel's DPDK [6] is a user library that executes the device driver in userspace. It provides direct access to the NIC, with rings and buffers in shared memory and libraries to manipulate them. DPDK only runs on Linux, and is essentially limited to Intel NICs, though commercial support exists for some other NIC. Despite being developed mostly by hardware vendors, DPDK is opensource and released under a BSD license. It is primarily focused on performance, and as such it makes extensive use of CPU and NIC optimizations, comes with libraries for NUMA-aware memory allocation, and data structures to support typical network applications. The optimizations included in DPDK make it extremely efficient (maybe 50% faster than netmap), but terribly difficult to use and modify. Also, DPDK mainly only supports busy wait as a synchronization mechanism, so applications need a core constantly spinning to monitor events.

Luca Deri's DNA [2] is another recent proposal for fast network I/O. Similar in principle to DPDK, DNA runs only on Linux and Intel NICs, and brings the device driver in user space supporting busy wait as the primary mode of operation. It is often used in network monitoring applications such as NTOP [3]. Performance is similar to that of netmap.

Another fast I/O solution has been proposed by KAIST with the PacketShader I/O engine [4], a custom Linux device driver for the Intel 82598 1O Gbit/s NIC, which uses mechanisms similar to netmap. This is mostly of historical interest because it is limited to one OS and one device.

Other older approaches are generally much more limited in terms of performance, being based on simpler APIs built on top of standard device drivers. Examples include Linux PF_PACKET and its predecessor, Luca Deri's PF_RING [1]. These frameworks are similar in terms of internal architecture to the netmap emulation mode presented in Section 4.7; note though that emulation is really a last resort and not the preferred mode of operation.

8. CONCLUSIONS

When we started our work on netmap, we only envisioned it as a general purpose tool to replace the existing slow (libpcap, sockets) or fast but proprietary or otherwise limited mechanisms for packet I/O.

The project grew beyond our best expectations, and evolved into a powerful switching infrastructure, enabling further research in different areas (such as virtual machines).

We believe that the part of our work related to I/O and software switching is mostly complete and stable, and we are actively working to include it in the relevant software distributions (FreeBSD, Qemu, libpcap, OpenvSwitch, etc.).

There is still significant work to do in applying the various techniques used in netmap to the acceleration of the host stack. This is going to be an incremental process, which will permit a gradual integration of our ideas without destabilizing the system too much and with a careful evaluation of the benefits.

9. REFERENCES

[1] L. Deri. Improving passive packet capture: Beyond device polling. In *SANE 2004, Amsterdam.*

[2] L. Deri, J. Gasparakis, P. Waskiewicz, and F. Fusco. Wire-speed hardware-assisted traffic filtering with mainstream network adapters. *Advances in Network-Embedded Management and Applications*, pages 71–86, 2011.

[3] L. Deri and S. Suin. Effective traffic measurement using ntop. *Communications Magazine, IEEE*, 38(5):138–143, 2000.

[4] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[5] M. Honda, F. Fuici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling Network Protocol Innovation with User-Level Stacks. *ACM CCR*, April 2014 (to appear).

[6] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.

[7] S. McCanne and V. Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX'93: Proc. of the USENIX Winter Conference*, pages 2–2. USENIX Association, 1993.

[8] Microsoft Corporation. Scalable networking: Eliminating the receive processing bottleneck - introducing rss. Technical report, Technical Report, 2004.

[9] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference ATC'12*, Boston, MA. USENIX Association, 2012.

[10] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Communications of the ACM*, 55(3):45–51, 2012.

[11] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Infocom 2012*. IEEE, 2012.

[12] L. Rizzo and G. Lettieri. VALE, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[13] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 47–58. IEEE Press, 2013.

ZFS for the Masses: Management Tools Provided by the PC-BSD and FreeNAS Projects

Dru Lavigne
dru@freebsd.org
iXsystems

Abstract

ZFS is a robust, copy-on-write filesystem designed to be self-healing and to overcome the inherent limitations of hardware RAID. While it has been available in FreeBSD since 7.0, a lack of integration with server and desktop management tools has slowed its adoption by many FreeBSD users.

This paper introduces some of the compelling features of ZFS from a user perspective and some of the PC-BSD and FreeNAS utilities for taking advantage of these features. The PC-BSD project (pcbsd.org) provides a suite of graphical tools, with command-line equivalents, for installing and managing a FreeBSD desktop or server. The FreeNAS project (freenas.org) provides a FreeBSD-based Network Attached Storage solution that provides a graphical, web-based configuration interface. The examples used in this paper are from PC-BSD 10.0 and FreeNAS 9.2.0.

ZFS Pools and Datasets

ZFS provides great flexibility for creating, managing, and growing the capacity of the filesystem as needed. Unlike traditional Unix filesystems, which create a fixed size at filesystem creation time, ZFS uses the concepts of a pool and datasets.

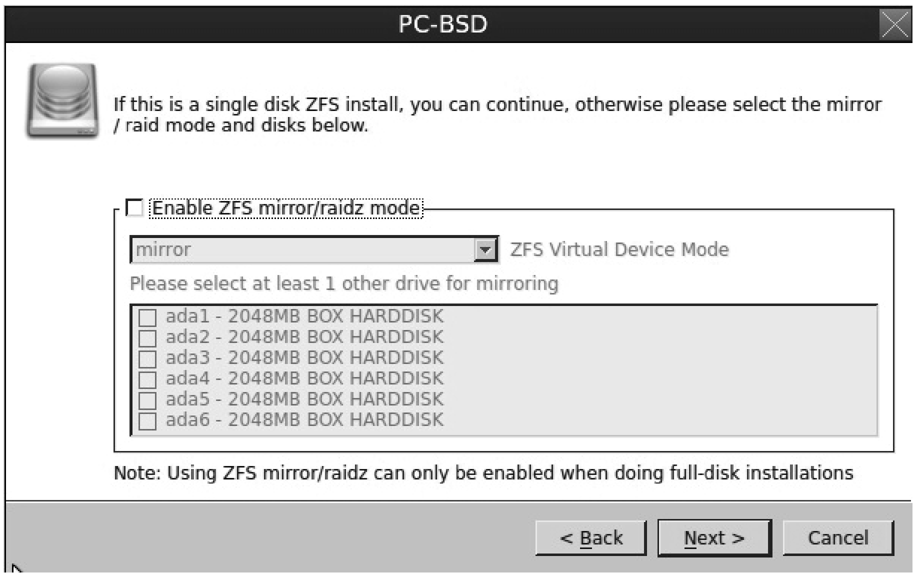
A ZFS *pool* is a disk or collection of disks that is formatted with ZFS. When adding multiple disks to a pool, a level of redundancy, known as a RAIDZ*, can be specified. The number following the RAIDZ indicates how many disks can fail without losing the pool. For example, one disk can fail in a RAIDZ1, two disks in a RAIDZ2, and three disks in a RAIDZ3. In theory, any number of disks can be added when creating a pool and any number of disks can be added to an existing pool to increase its capacity. In practice, specify the number of disks recommended for that RAIDZ level and plan to stripe that same number of disks should the storage capacity of the pool need to be increased in the future. Failure to do so can decrease performance significantly. The recommended number of disks for each RAIDZ configuration is explained in the ZFS Storage Pools Recommendation of the ZFS Best Practices Guide. 1

A ZFS *dataset* is similar to a folder in that it supports permissions. A dataset is also similar to a filesystem in that you can set properties such as quotas and compression. The full list of available properties are described in zpool(8). Datasets can be created at any time, allowing for great flexibility in managing data. Unless a quota is set on the dataset, the full remaining capacity of the pool is available to any dataset.

The PC-BSD installer makes it easy to create the ZFS pools and initial datasets. An example of the PC-BSD graphical installer is seen in Figures 1 and 2. An example of the PC-BSD ncurses installer is seen in Figure 3.

If multiple disks are available when installing PC-BSD, check the box “Enable ZFS mirror/raidz mode” shown in Figure 1, select the desired configuration from the drop-down menu, and check the number of disks. The installer will indicate the optimal number of disks for that configuration and will not let you select a configuration that does not contain the minimum required disks.

1 http://solarisinternals.com/wiki/index.php/ZFS_Best_Practices_Guide#ZFS_Storage_Pools_Recommendations



PC-BSD

If this is a single disk ZFS install, you can continue, otherwise please select the mirror / raid mode and disks below.

☐ Enable ZFS mirror/raidz mode

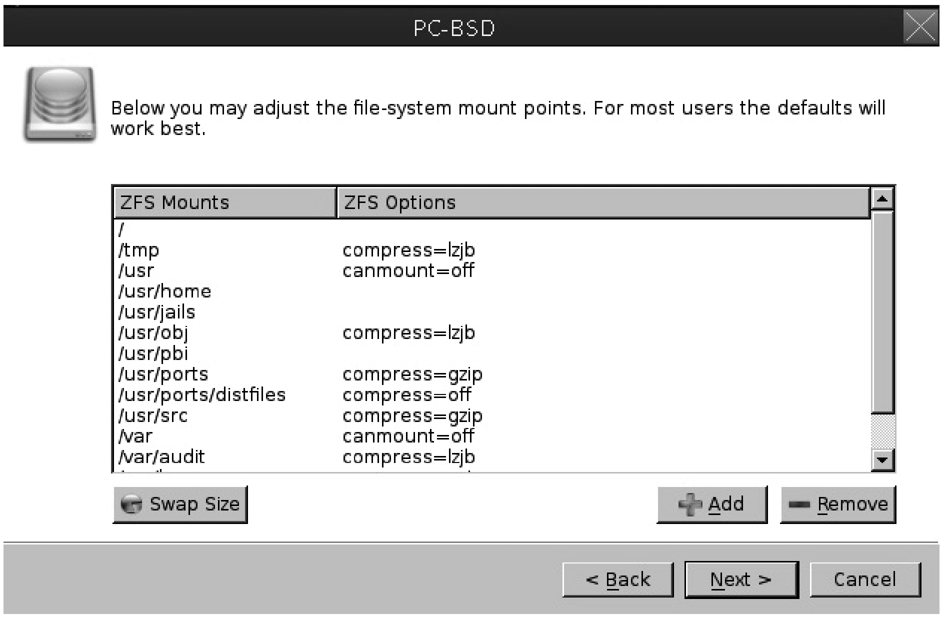
mirror — ZFS Virtual Device Mode

Please select at least 1 other drive for mirroring

☐ ada1 - 2048MB BOX HARDDISK
☐ ada2 - 2048MB BOX HARDDISK
☐ ada3 - 2048MB BOX HARDDISK
☐ ada4 - 2048MB BOX HARDDISK
☐ ada5 - 2048MB BOX HARDDISK
☐ ada6 - 2048MB BOX HARDDISK

Note: Using ZFS mirror/raidz can only be enabled when doing full-disk installations

Figure 1: Pool Creation in Graphical PC-BSD Installer

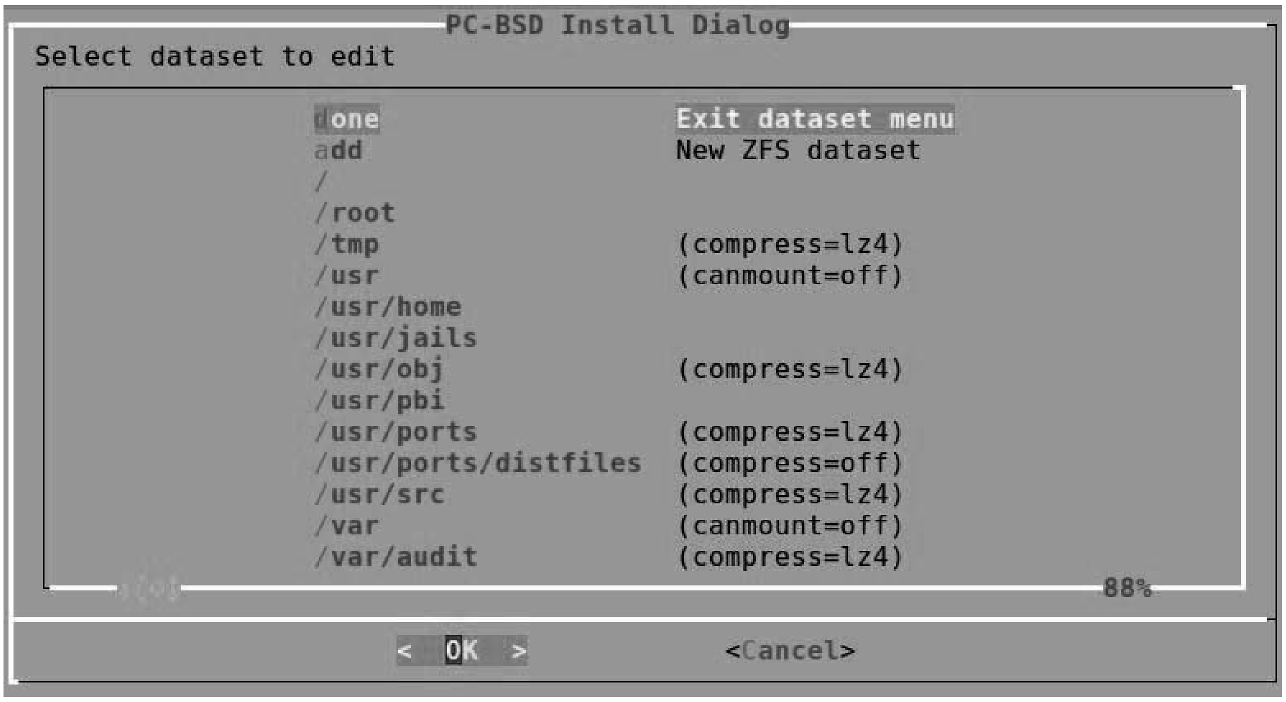


PC-BSD

Below you may adjust the file-system mount points. For most users the defaults will work best.

ZFS Mounts	ZFS Options
/	
/tmp	compress=lzjb
/usr	canmount=off
/usr/home	
/usr/jails	
/usr/obj	compress=lzjb
/usr/pbi	
/usr/ports	compress=gzip
/usr/ports/distfiles	compress=off
/usr/src	compress=gzip
/var	canmount=off
/var/audit	compress=lzjb

Figure 2: Dataset Creation in Graphical PC-BSD Installer



```
                    PC-BSD Install Dialog
 Select dataset to edit

                  done                     Exit dataset menu
                  add                      New ZFS dataset
                  /
                  /root
                  /tmp                     (compress=lz4)
                  /usr                     (canmount=off)
                  /usr/home
                  /usr/jails
                  /usr/obj                 (compress=lz4)
                  /usr/pbi
                  /usr/ports               (compress=lz4)
                  /usr/ports/distfiles     (compress=off)
                  /usr/src                 (compress=lz4)
                  /var                     (canmount=off)
                  /var/audit               (compress=lz4)
                                                              88%

                  <  OK  >              <Cancel>
```

Figure 3: Dataset Creation in ncurses PC-BSD Installer

To modify the ZFS options of a default dataset when installing PC-BSD, highlight an existing dataset in the screen shown in Figure 2. Use the "+Add" button to create additional datasets.

When using the ncurses installer shown in Figure 3, select an existing dataset with the up/down arrows and press enter to configure its options. Select "add" to create a new dataset and set its options.

In FreeNAS, the operating system is separate from the storage disks so pool creation occurs from the management interface. The ZFS Volume Manager, shown in Figure 4, will automatically display the optimal pool configuration and storage capacity for the number of selected disks. Once the pool is created, datasets can be created as needed using the screen shown in Figure 5.

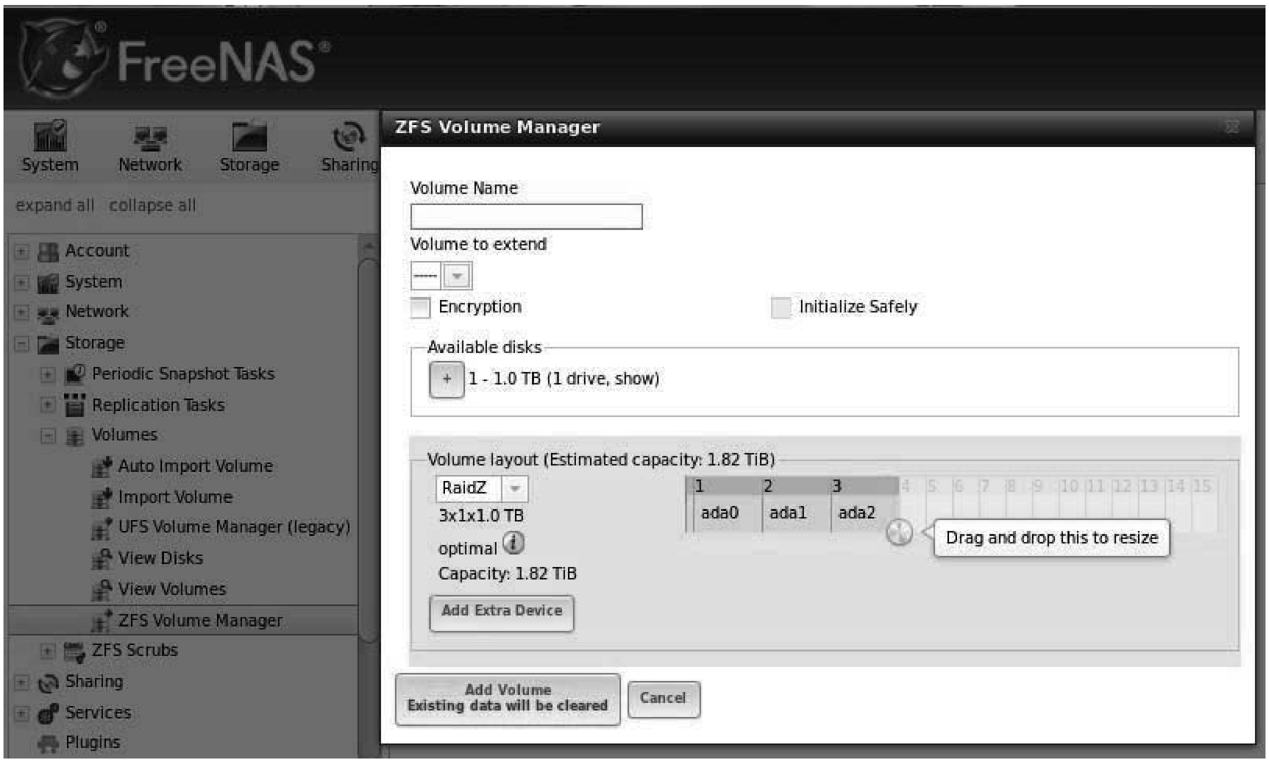


Figure 4: Using the FreeNAS ZFS Volume Manager to Create the Pool

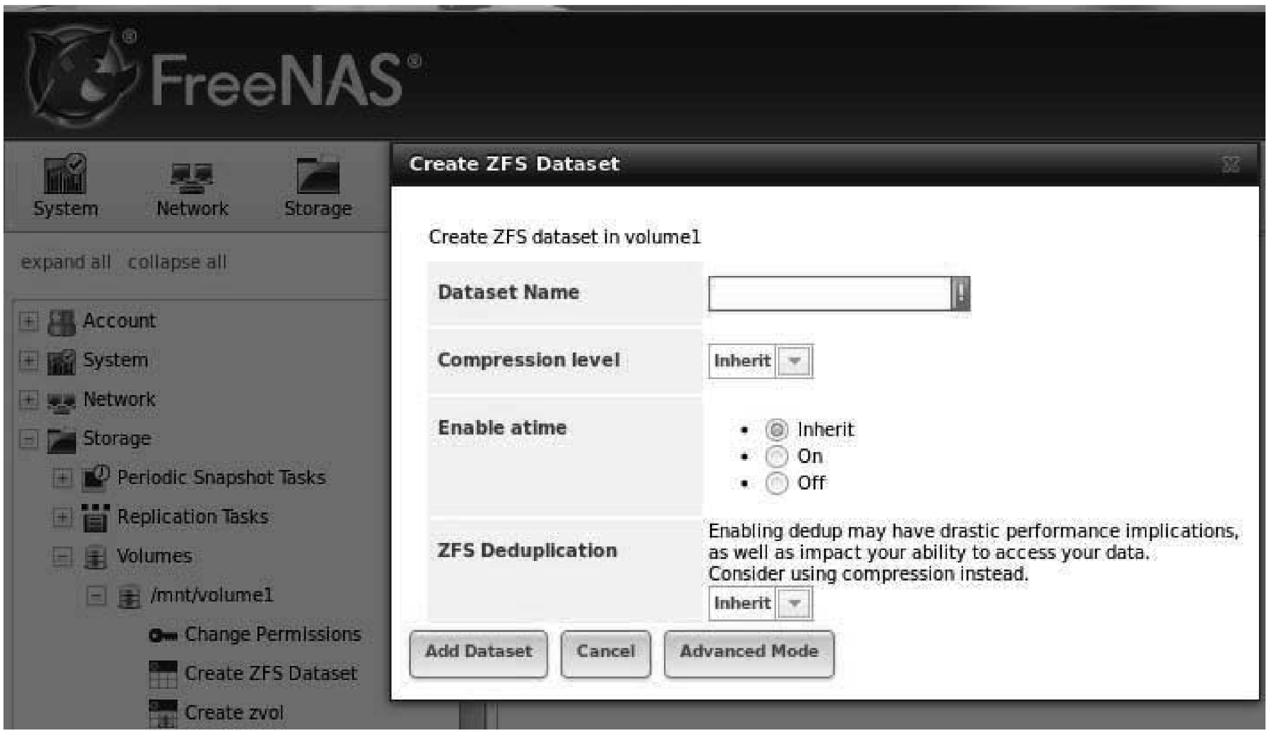
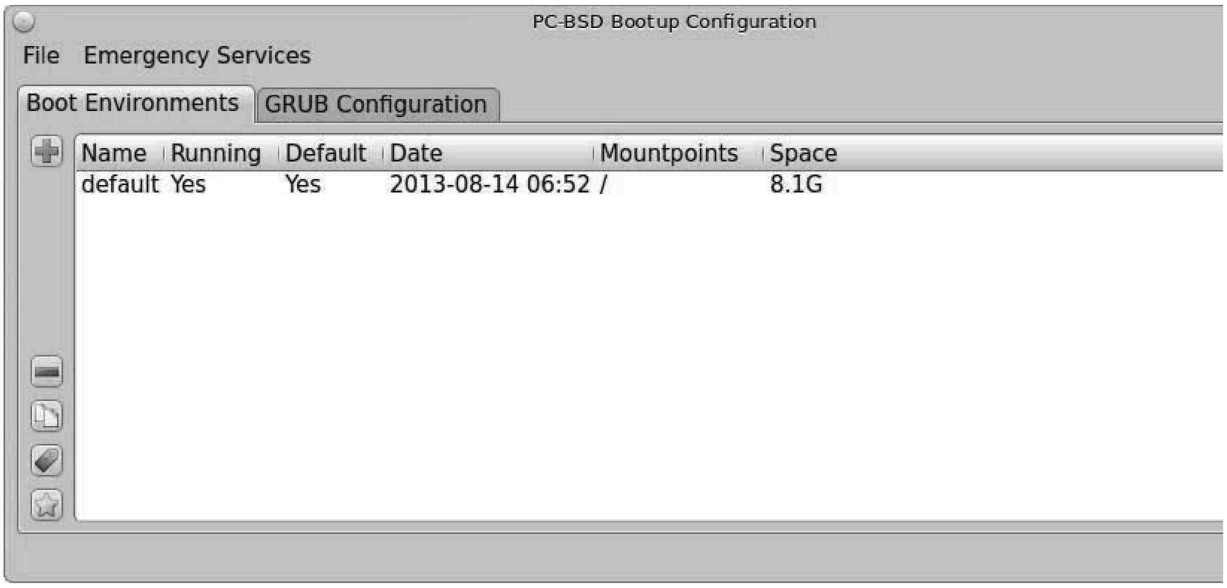


Figure 5: Creating a Dataset in FreeNAS

ZFS Multiple Boot Environments

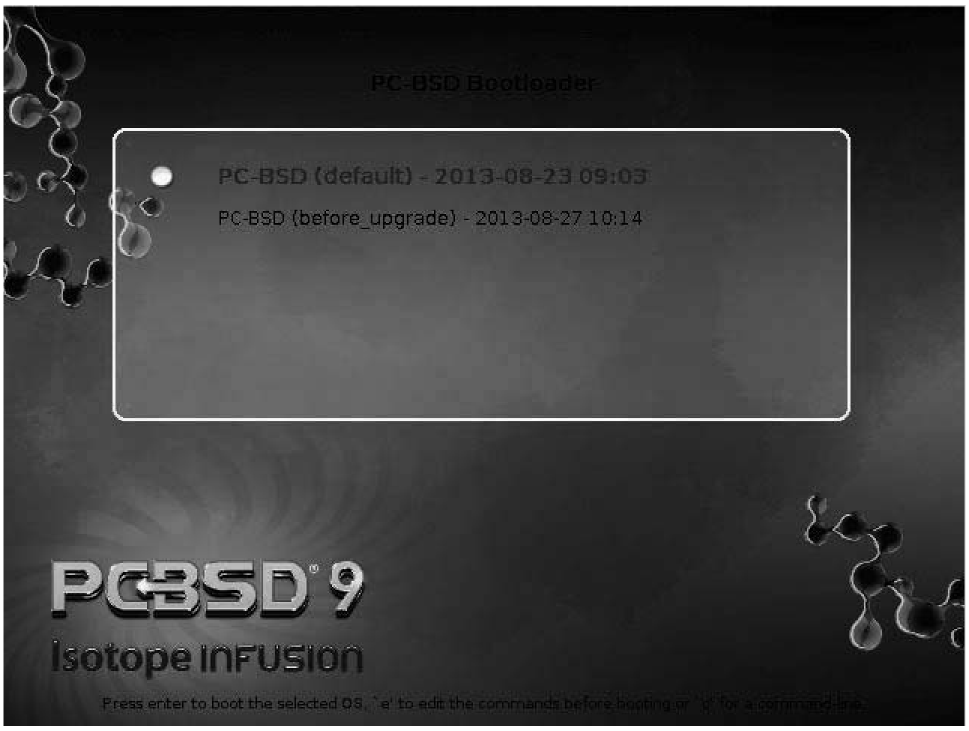
The default datasets provided by the PC-BSD installer are used to provide a ZFS feature known as *multiple boot environments*. Boot environments can be used to mitigate the risk associated with a failed operating system upgrade. They can also be used to test hardware compatibility on another system or to provide a staging area to test application upgrades. Before performing one of these operations, simply create a snapshot of the current boot environment. Should the upgrade fail or you wish to return to the previous boot environment, simply reboot and select the desired boot environment from the boot menu. Figure 6 shows the PC-BSD Boot Manager utility for creating and managing boot environments. This utility is a front-end to the built-in beadm CLI. Figure 7 shows an example boot menu after a boot environment has been created.

Boot environments do not include user home directories, meaning that any changes to a user's data will still be available if the system is booted into a previous boot environment. The Life Preserver tool, described in the next section, can be used by users to manage previous versions of their data.



Name	Running	Default	Date	Mountpoints	Space
default	Yes	Yes	2013-08-14 06:52	/	8.1G

Figure 6: Using Boot Manager to Manage Boot Environments



```
PC-BSD Bootloader

PC-BSD (default) - 2013-08-23 09:03
PC-BSD (before_upgrade) - 2013-08-27 10:14

Press enter to boot the selected OS, `e' to edit the commands before booting or `c' for a command-line.
```

Figure 7: Sample Boot Menu Containing a Created Boot Environment

Using Life Preserver for Snapshot Management in PC-BSD

In ZFS, a point-in-time *snapshot* can be taken at any time and can either be stored locally or replicated to another system. Snapshots are instantaneous and initially zero bytes in size. As the data changes between that point in time, the snapshot grows accordingly. Snapshots can be used to retrieve a version of a file from a previous point in time, or even to return the entire filesystem to a certain point in time.

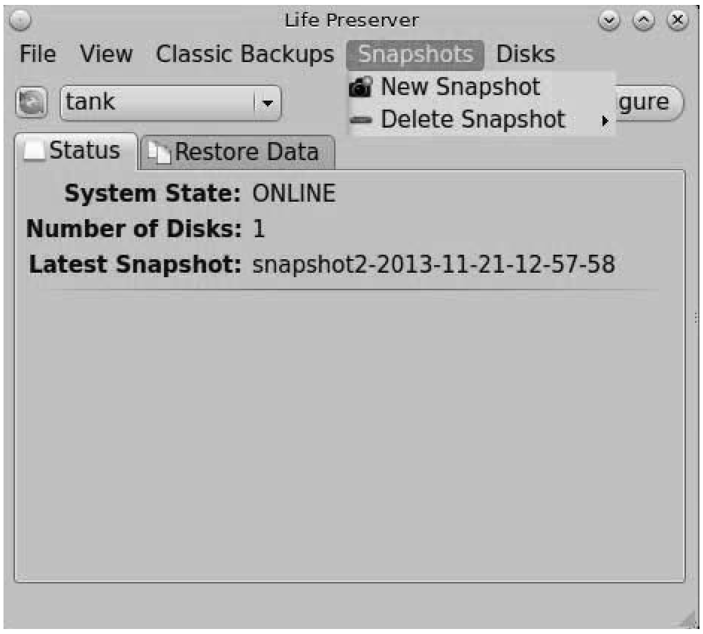
The Life Preserver utility, shown in Figure 8, can be used to create a snapshot at any time. For example, if a user is about to make edits to important files, they can first create a snapshot which will preserve the current versions of every file in the pool.

Alternately, Life Preserver provides a Wizard to schedule the automated creation of snapshots every 5, 10, 30, 60 minutes or once per day and to optionally send a copy of those snapshots to a remote system. The backup schedule and system to replicate to can be modified at any time by clicking the “Configure” button.

The “Restore Data” tab, shown in Figure 9, provides a time-slider for scrolling through snapshots. Once a snapshot is selected, simply browse to the location of a directory or file and click the “Restore” button to restore a copy of the selected item. Multiple versions can be restored, allowing the user to find a specific change.

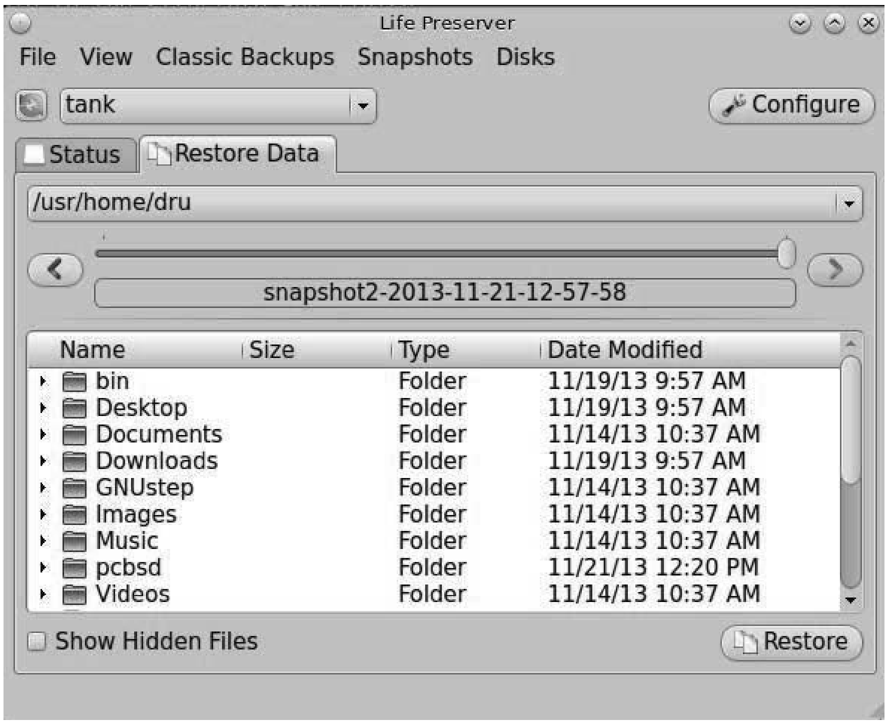
If snapshots are replicated to another system, a specified remote snapshot can be used to restore a damaged installation or to clone an existing installation to another system.To use this function, start a PC-BSD installation on the desired system, and select the “Restore from Life-preserver backup” option in the screen shown in Figure 10. A wizard will prompt you for the IP address and login information of the system hosting the snapshots. Once connected, you can select which snapshot (or point in time) to restore from.

Refer to Kris Moore's paper "Snapshots, Replication, and Boot-Environments - How new ZFS utilities are changing FreeBSD & PC-BSD" for more information about Life Preserver.



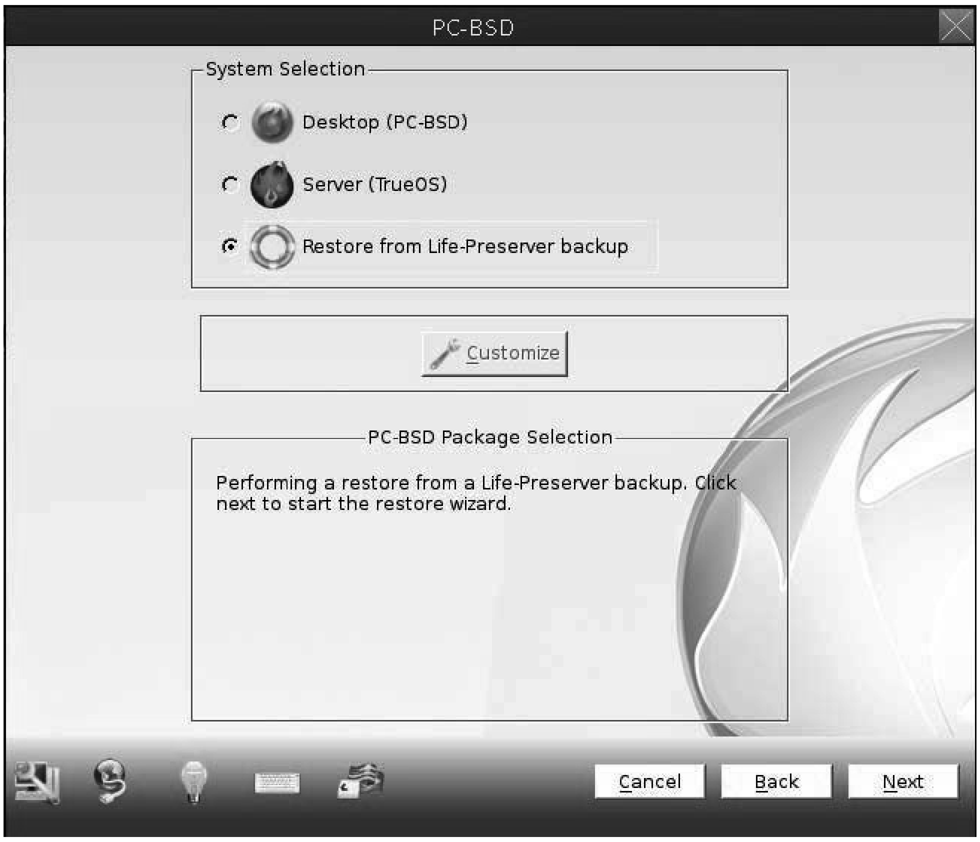
System State: ONLINE
Number of Disks: 1
Latest Snapshot: snapshot2-2013-11-21-12-57-58

Figure 8: Creating a Snapshot



Name	Size	Type	Date Modified
bin		Folder	11/19/13 9:57 AM
Desktop		Folder	11/19/13 9:57 AM
Documents		Folder	11/14/13 10:37 AM
Downloads		Folder	11/19/13 9:57 AM
GNUstep		Folder	11/14/13 10:37 AM
Images		Folder	11/14/13 10:37 AM
Music		Folder	11/14/13 10:37 AM
pcbsd		Folder	11/21/13 12:20 PM
Videos		Folder	11/14/13 10:37 AM

Figure 9: Restoring a Previous Version from a Snapshot



Performing a restore from a Life-Preserver backup. Click next to start the restore wizard.

Figure 10: Restoring/Cloning a System from a Replicated Snapshot

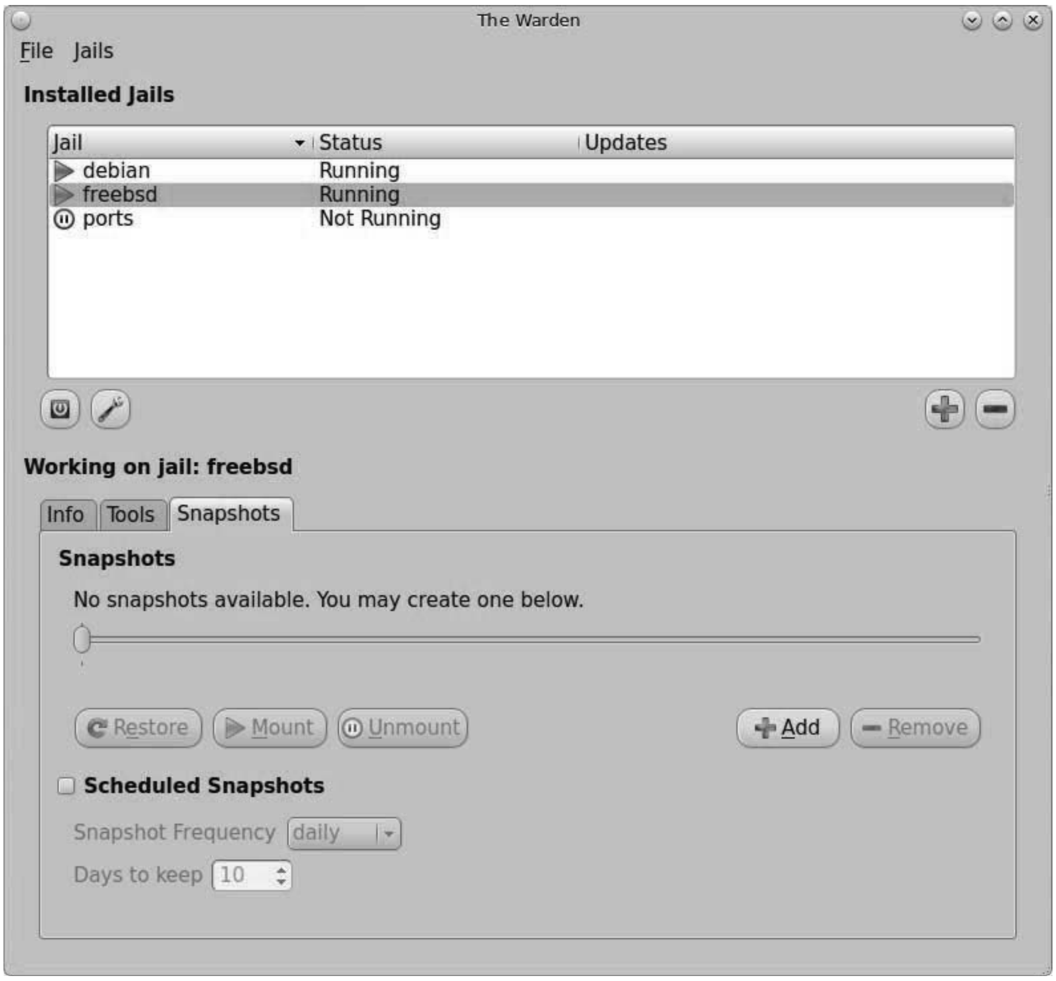
Using Warden to Manage Jail Snapshots in PC-BSD

Warden provides a graphical interface for quickly deploying and managing FreeBSD jails. It also has a built-in, per-jail, snapshot management utility. Figure 11 shows the “Snapshot” tab for a highlighted jail. To create a snapshot now, click the “+Add” button. To schedule the automatic creation of snapshots, check the “Scheduled Snapshots” box and select the frequency. Once snapshots are available, use the time-slider to select the desired snapshot.

It is worth noting that since Warden provides experimental Linux jail support, it provides a mechanism for experimenting with ZFS features on Linux without licensing concerns.

Snapshot Management in FreeNAS

FreeNAS provides Periodic Snapshot Tasks, shown in Figure 12, for scheduling snapshots. Snapshots can be scheduled on the filesystem (with or without its datasets) or on an individual dataset basis. Replication Tasks, shown in Figure 13, can optionally be used to schedule the replication of the resulting snapshots to another system running rsync. The replication can be optionally encrypted if the other system is running SSH. The replication task can be scheduled to occur immediately after snapshot creation or a replication window can be created to replicate queued snapshots after business hours.



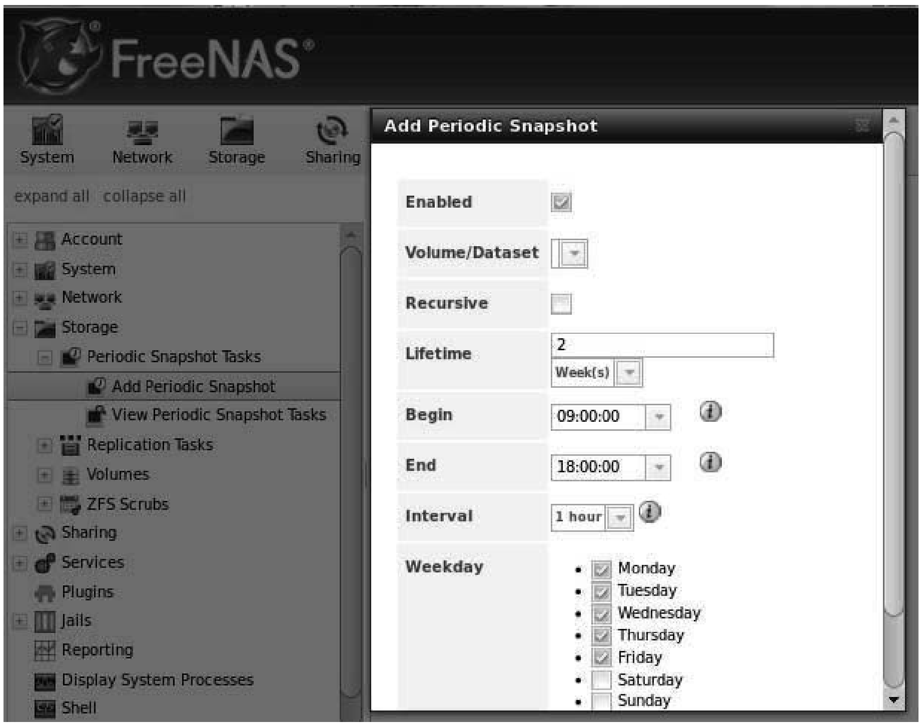
Jail	Status	Updates
debian	Running	
freebsd	Running	
ports	Not Running	

Working on jail: freebsd

No snapshots available. You may create one below.

Scheduled Snapshots: unchecked
Snapshot Frequency: daily
Days to keep: 10

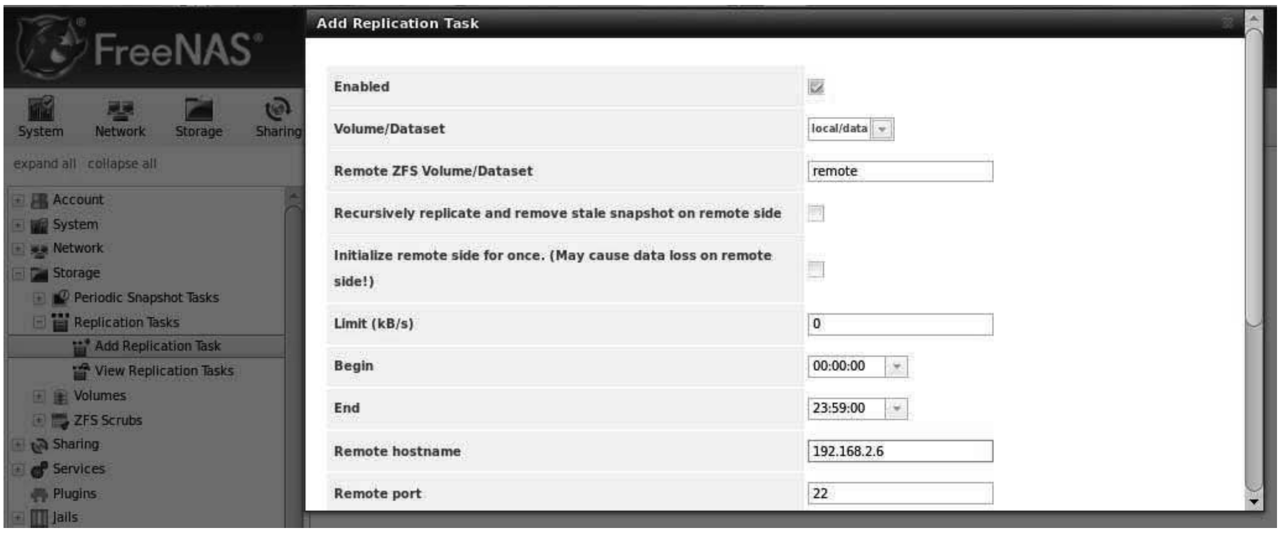
Figure 11: Managing a Jail's Snapshots



Add Periodic Snapshot

Enabled	✓
Volume/Dataset	
Recursive	
Lifetime	2 Week(s)
Begin	09:00:00
End	18:00:00
Interval	1 hour
Weekday	✓ Monday ✓ Tuesday ✓ Wednesday ✓ Thursday ✓ Friday Saturday Sunday

Figure 12: Scheduling a Periodic Snapshot Task in FreeNAS



Add Replication Task

Enabled	✓
Volume/Dataset	local/data
Remote ZFS Volume/Dataset	remote
Recursively replicate and remove stale snapshot on remote side	☐
Initialize remote side for once. (May cause data loss on remote side!)	☐
Limit (kB/s)	0
Begin	00:00:00
End	23:59:00
Remote hostname	192.168.2.6
Remote port	22

Figure 13: Scheduling the Replication of Snapshots to Another System

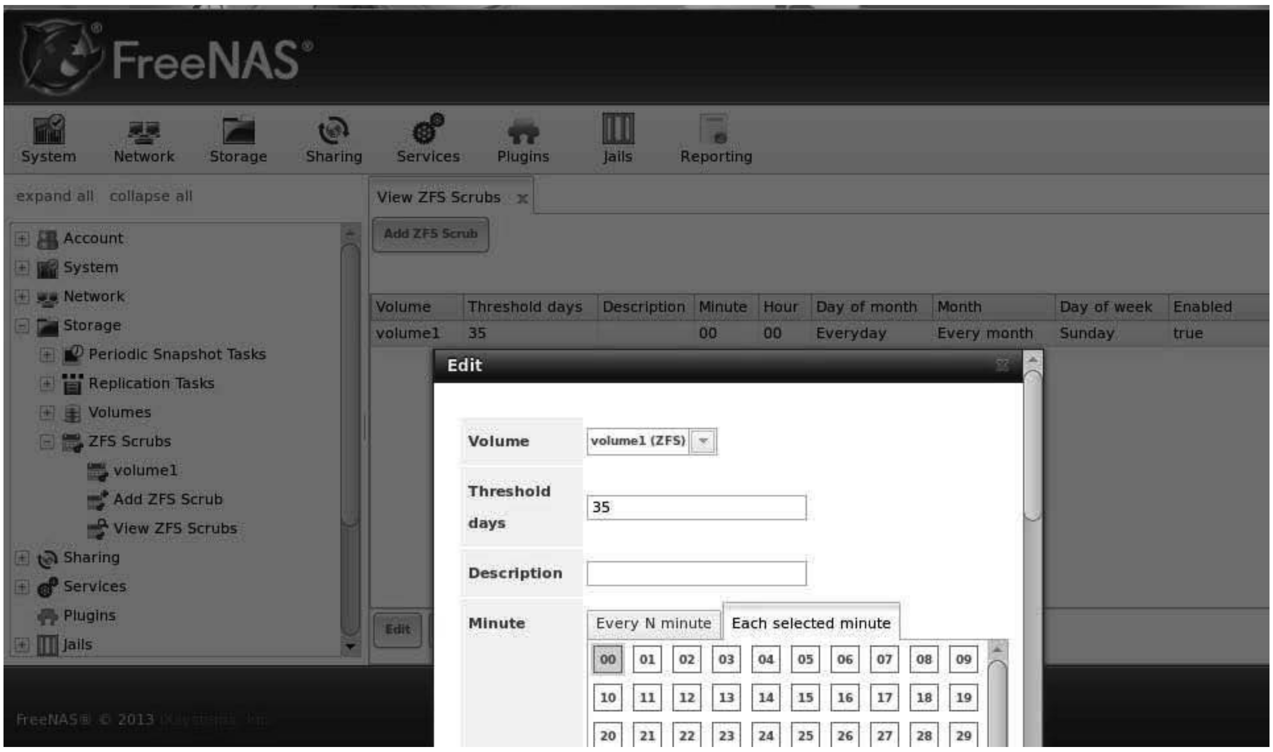
Scrub Management

Another ZFS feature is the *scrub*, a process which checks the checksum associated with each disk block, and if the checksum differs, attempts to correct the data stored in the block. Scheduling regular scrubs and viewing their results helps to identify data integrity problems, detect silent data corruptions caused by transient hardware issues, and provide an early indication of pending disk failures. This type of insight into the health of one's data is particularly useful in a device that is used for network storage, such as FreeNAS.

When a pool is created in FreeNAS, it automatically schedules a ZFS Scrub task to occur every Sunday evening at midnight. As seen in the example in Figure 14, the schedule can be modified to a time that least impacts users.

In PC-BSD 10.0, the ZFS Pools tab of Disk Manager can be used to start a scrub when needed. A future version of PC-BSD will add the ability to create a scrub schedule.

For any ZFS system, it is recommended to perform a scrub once per week for consumer-quality drives and once per month for datacenter-quality drives.



Volume	Threshold days	Description	Minute	Hour	Day of month	Month	Day of week	Enabled
volume1	35		00	00	Everyday	Every month	Sunday	true

Edit

Volume	volume1 (ZFS)
Threshold days	35
Description	
Minute	Each selected minute: 00

Figure 14: Editing the Scrub Schedule in FreeNAS

Additional Resources

The management utilities provided in both PC-BSD and FreeNAS make it easy to take advantage of the many features provided by ZFS. This paper has only scratched the surface of the available features.

Both the PC-BSD and FreeNAS Projects provide comprehensive Guides which detail the use of these and other tools. The Guides are version-specific and one should download the version that matches their operating system version:

• the PC-BSD Users Handbook can be downloaded from wiki.pcbsd.org

• the FreeNAS User Guide can be downloaded from doc.freenas.org

There are many good resources for learning more about ZFS. These include:

ZFS Best Practices Guide:
http://www.solarisinternals.com/wiki/index.php/ZFS_Best_Practices_Guide

Becoming a ZFS Ninja (video):
http://blogs.oracle.com/video/entry/becoming_a_zfs_ninja

ZFS Wikipedia entry:
http://en.wikipedia.org/wiki/Zfs

OpenBGPD turns 10 years - Design, Implementation, Lessons learned

Henning Brauer
BS Web Services GmbH

Abstract

The Border Gateway Protocol, BGP, is being used on almost all core routers at ISPs and some enterprises to announce reachability of own and downstream networks and learn about reachability of foreign networks. Thus, every of these routers builds a map of the internet from his point of view. The resulting routing table approaches 500000 entries now.

In 2004, I started working on a new BGP implementation which later became known as OpenBGPD. 2014 marks its 10th anniversary, a good opportunity to look at Design and Implementation as well as the lessons learned over the last 10 years.

1 Introduction

BGP4, the only BGP version in widespread use today, has been defined in RFC 1771 from 1995. Several extensions followed, the base protocol has been unchanged.

In 2004, the world looked quite a bit different than today. The ”full bgp table” had less than 200000 entries, compared to almost 500000 today. The router market was dominated by Cisco, with Juniper beginning to show up. If you wanted to run a router based on free software, you pretty much had to use Zebra - which lives on under the name quagga.

At work, I was running Zebra on OpenBSD - and had lots of problems with it. In the end it was just not reliable enough, despite me fixing several bugs in it. I had to realize that Zebra had fundamental design problems which weren't easily fixable, foremost a central event queue - when zebra was very busy, the ”send keepalive message” events could sit in the queue for so long that the neighbors would drop sessions due to lack of a keepalive. And it uses threads.

I started to think about implementing a new BGP daemon. And I made the mistake of talking about it to some other OpenBSD developers, which from that point on kept prodding me. Talking to them, foremost Theo de Raadt and Bob Beck, we quickly had a basic design and I eventually started coding.

2 Basic Design

Quite clearly we want a privilege seperated BGP daemon - principle of least privilege. One needs root permissions to open the tcp listeners on port 179, and one needs root to alter the kernel routing table. So we need a parent process running as root.

To talk BGP to the neighbor routers no special privileges are required. It was clear from the beginning that one thread per neighbor was not the way to go, or threads at all - they just make things very comlicated and hard to debug, while often being way slower than seperate processes using IPC - locking of shared memory regions is everything but simple and definately comes at a cost. One process per peer seemed to be overkill too, tho - so an event-based session engine, handling the BGP sessions to the neighbors, using non-blocking I/O it is.

Processing all the route information learned from neighbors, building up the map of the internet, should be seperated from the session handling. So the session engine (SE) handles session setup and teardown, keepalive handling and the various timers associated with the session, but doesn't process any routing information - it just hands it off to a third process, the Route Decision Engine (RDE).

The RDE processes all routing information, decides on eligibility and preferences of the routes it learned, to build the Routing Information Base (RIB), from which it builds up its view of the "best" (in terms of BGP) routing table (by selecting the "best" route from the RIB, aka the options it got) or, in BGP speak, Forward Information Base (FIB).

The "best"route per destination is then transmitted to the parent process, which talks to the kernel and modifies the kernel routing table accordingly.

In this model, the SE and the RDE can run chrooted and unprivileged.

3 Integrated Approach

Zebra and quagga follow a design approach where they have a central daemon, zebrad / quaggad, which talks to the kernel, and per-protocol daemons implementing the routing protocols. That is, last not least, a price for portability - the interface to the kernel routing tables isn't standarized, and the implementations differ widely.

I never realy believed in "take a generic Unix, add a BGP speaker, and you have a proper core router" - there's more to it. Instead of doing lots of magic in a central daemon - like handling preferences of OSPF versus BGP routes, we changed the kernel side.

OpenBSDs kernel routing table now has route priorities, so that we can have multiple routes to the same destination. The naming is slightly unfortunate since lower numbers mean higher priority. If there is a route to a given destination from OSPF and BGP, both ospfd and bgpd will insert their route to the kernel routing table with different priorities - by default, 32 for OSPF and 48 for BGP. The kernel picks the OSPF route because of the priority, and if that route gets removed or is marked down, it'll take the BGP route. Without that functionality in the kernel, by some means userland daemons need to decide which route to insert - i. e. if ospfd learns a route to a given destination and figures there already is a BGP route, it needs to overwrite it, and when ospfd removes that route, bgpd needs to re-insert its one - or a daemon all routing daemons talk to (like quaggad/zebrad) had to do it. We don't need that, fortunately.

4 Non-Blocking I/O

File descriptors, network connections are presented as such through the socket layer, are usually blocking in Unix. One calls write() on a file descriptor with a given amount of data to be written, and the write call will only return when the entire data has been written out or a permanent error has been encountered. If not all data can be written out at once, write() will sleep until it can go on, the process is blocked for that time.

While these semantics are really nice to work with for easy clients and servers that handle one socket/fd at a time, it is absoliutely not what we want in the session engine, since then a single slow peer could block the entire session engine and last not least can prevent keepalives to be sent out in time, which in turn makes the peers drop the connection. Thus, we need non-blocking sockets.

When switching a socket to non-blocking, a write (or read, or anything similar) call will never block, but immediately return as soon as it had to sleep. That can happen before it has written out all data, it signals the caller the amount of data written and it is the caller's duty to get the remaining data written out at a later point.

Instead of just implementing non-blocking I/O for bgpd I decided to abstract and wrote the buffer framework, which takes care of keeping unwritten data in a buffer per socket to be retried later, reads into a read buffer and generally abstracts the socket handling.

5 Messaging

We still need the 3 processes to communicate with each other. Some form of IPC, some internal messaging. Once again, instead of a quick implementation for bgpd, I went for a framework which I called imsg, building up on top of the buffer framework. It supplies pretty easy to use primitives to send and receive structured messages between the processes. It can work over almost any form of socket connection, be it a socketpair on the local machine or a tcp connection, potentially to another host.

Investing the time to make this a proper framework paid out. I used it again a bit later to implement ntpd - and now, 10 years later, the evolved imsg framework moved to libutil and is in use by no less than 25 daemons and programs in OpenBSD base.

6 kroute

A routing daemon obviously needs to interface with the kernel routing table. bgpd doesn't only need to insert routes, it also needs the kernel routing table to decide on eligability of routes it learns - foremost, it needs to make sure it can actually reach the nexthop (gateway) on that route.

Again, I went for a framework. Variants of it are being used by the other routing daemons that showed up after bgpd in OpenBSD: ospfd, ospf6d, ripd, ldpd, dvrmpd - and snmpd, which isn't a routing dameon.

kroute fetches the kernel routing table on startup using the sysctl(3) interface. It listens on the routing sockets to learn about any 3rd party updates and keeps its copy in sync. When bgpd itself wants to add, remove or change a route, it does so against the kroute copy of the kernel routing table, and kroute is responsible for promoting the changes to the kernel.

This design has a nice side effect: we can run "decoupled". That means that bgpd fetches the kernel routing table into kroute, but changes bgpd makes aren't promoted back into the kernel. And since the kroute view can be inspected using bgpctl, one can check what bgpd would do to the kernel routing table without actually touching it - once everything looks like it should, bgpd's kroute view and the kernel routing table can be coupled, which means that bgpd pushes the changes to the kernel routing table all at once. Coupling and decoupling can happen any time.

7 BGP messages

The BGP protocol itself is pretty simple, it only knows about 4 message types: OPEN, UPDATE, NOTIFICATION and KEEPALIVE. A fifth one, RREFRESH, was added later and is an extension.

OPEN initiates a new BGP session. UPDATE contains the actual routing data. A NOTIFICATION is an error message, upon reception of such a message bgpd must tear down the affected session. KEEPALIVEs are sent in regular intervals.

A BGP message consists of a marker - 16 bytes of all-ones, a length field and the type. The rest of the payload is message type dependant.

8 Session Engine: establishing new BGP sessions

To establish a bgp session, a BGP speaker must open a tcp connection to the neighbor/peer in question on port 179 and send an OPEN message. That message contains various parameters for the session, last not least the AS number of the BGP speaker. It also contains a list of (optional) capabilities of the BGP speaker. If the peer is fine with all the parameters in the OPEN message, it responds with an OPEN message itself, otherwise it sends a NOTIFICATION. That mechanism is also used for capability negotiation: if the peer doesn't like one of the capabilities we announce, it must send a NOTIFICATION back quoting the capability it doesn't like, we then disable that capability for that session and don't announce it again on the next attempt. The same scheme works the other way around, if we receive an OPEN message with an unsupported capability, an AS number that doesn't match the config or the like we must send a NOTIFICATION back.

The capability negotiation is one of the areas where real life surprises: some Cisco systems will send a "unsupported capability" NOTIFICATIONs back without actually including the capability it doesn't like, violating the RFCs and making it imossible for us to determine which of the announces capabilities is the culprit here. The only way to deal with such a broken system is to disable capability announcements alltogether upon reception of such a NOTIFICATION. It is then up to the admin to manually figure out the offending capability and manually disable it in the config file.

9 Session Engine: session handling

Once we have the BGP session established, route information is exchanged over it. When the session drops, the connection is assumed to have vanished - a link might have died. This in turn means that we must remove all routes we learned from the peer in question from our RIB and in turn the selected ("best") routes from that peer from the FIB and thus the kernel routing table.

To detect a dead session, BGP uses so called hold time. If we do not receive any message from the peer within the hold time, we must drop the session. The hold time is one of the parameters exchanged with the OPEN message, it's typically 90 or 180 seconds. We use timers per session, one for the hold time. It is reset to the negotiated hold time every time we receive a message from the peer, if it ever reaches 0 we must drop the connection.

To make sure we send a KEEPALIVE when needed we have another timer per session, the keepalive timer. It is reset to one third of the hold time every time we send a message to the peer, and when it expires, we send a KEEPALIVE. That also means that we don't need to send KEEPALIVEs if UPDATEs happen frequently enough.

To keep track of the session state BGP implements a Finite State Machine, FSM. A FSM has defined states and a set of actions, and state transitions in response to said actions. The FSM is defined in RFC 1771, but has a bug there - a missing transition. Despite me notifying the RFC author and him acknoledging the missing transition, the RFC has never been fixed.

A session can be in one of 6 states, plus a 7th, intermediate one only shortly there while setting up a new session. Initially, every session is in state IDLE, which means that we won't attempt to connect to the neighbor and won't accept a session from him either. Usually, that session goes into CONNECT quickly, which means we try to open a connection to the neighbor. If that fails or the session is configured to be passive, we go into the ACTIVE state, where we will accept a connection from the neighbor but won't try to establish one ourselves. Non-passive sessions will start yet another timer, the

ConnectRetry timer, when we enter the ACTIVE state, to go back into CONNECT once it expires. When the tcp connection is established we send our OPEN message and the session goes to OPENSENT, upong reception of the neighbor's OPEN message we continue to OPENCONFIRM, and once the first UPDATE or KEEPALIVE is exchanged the session is ESTABLISHED.

Any UPDATE message received is passed onto the RDE verbatim, minus the BGP header, but with our own little header. The same goes vice versa, routing updates from the RDE are passed verbatim, as far as the payload goes, to the peer in question - and resetting the keepalive timer upon sending, of course.

10 Route Decision Engine

The RDE parses the route update information it gets from the SE, applies filters which can modify attributes or cause the route to be dropped entirely. If the route is eligible, which foremost means its nexthop is reachable, the RDE adds, removes or updates it in the RIB.

Since changes to the filter would only affect newly learned routes, most BGP implementations, including ours, implement "soft reconfig", which means that we store the routing information learned from the peers verbatim so that we can re-apply filters.

The RIB stores the routes per destination, and decides which one is the best according to the algorithm RFC 1771 defines, which minor extensions (that pretty much every implementation has). This best route is then entered into the FIB and thus into the kernel routing table unless running decoupled, and is announced to downstream peers - if it passes the outbound filters toward the peer in question.

11 Route Servers

Especially on exchange points everybody having a BGP session to everybody doesn't scale all that nicely. While our bgpd has no problem with a lot of sessions, many of the commercial routers do.

In these scenarios one typically uses router servers. These do not actually route traffic and run bgpd in decoupled mode, there is no point to touch the kernel routing table at all. A route server learns routes from all its peers, decides on a "best" one per destination, and announces those to all peers again, typically without adding its own AS number to the AS path ("transparent AS"). That means that each participant at the exchange point just needs a session to the route server (or rather, one to each route server, there are typically two for redundancy) instead of each router to each router.

Since router operators typically want some more control over whom they learn routes from the route servers tag the learned routes with informatio where they learned them from etc using so-called communities, and the participants can filter based on these, and they can send communities with their own announcements, e. g. causing the route servers to not announce this route to specific ASes or the like. That leads to a problem: if the route server has a route from peer A selected as best route to the given destination, and the route is marked to not be announced towards peer B, peer B will receive no route to that destination at all instead of the second best one. The way around that is a seperate RIB per peer, which of course costs quite a bit of memory and processing power.

12 filters

I quickly implemented the filters in a proof-of-concept style when we had pretty much everything else done. I knwe they were slow, and I took a wrong decision - one global ruleset instead of filter blocks being applied to peers, which makes it even slower. In most scenarios the slow filters don't hurt too much, since one typically doesn't have a big ruleset. However, for route servers at big exchange points, things are different - the ruleset for DECIX, the world's biggest exchange point in Frankfurt, Germany, is several hundred thousand lines long.

I never considered the filters final, and we pretty quickly were clear that it's not just an implementation issues, but the global vs filter blocks decision was wrong, so reimplementing filters would require configuration changes. But even knowing all that, we didn't get around to rewrite the filters so far, which has cost us some installations at exchange points.

13 ToDo

OpenBGPD is a mature and very stable BGP implementations, used in a lot of places all over the world, including very critical ones. However, there is always room for improvement.

The filters finally need to be rewritten. Claudio and I know how, but keep not managing to actually get to it - lack of time, mostly.

I would love to see some autoaggregation to happen somewhere between bgpd and the kernel routing table. Half a million prefixes in the v4 table use up a fair amount of kernel memory, searches (route lookups) become more expensive, and due to the size of the table caches aren't as effective as we'd like. Aggregating at that level means that we only need to look at the nexthop, neighboring routes with the same nexthop can be aggregated, even if they are to different ASes - the kernel routing table doesn't care about that. In most scenarious such an autoaggregation would get the kernel routing table to less than 100000 entries. The culprit of course is with changes to a route covered by such an autoaggregated one, we might have to break up the aggregate if the nexthop changes. And the aggregation algorithm needs to be fast, since changes are very frequent.

14 Acknowledgments

As pretty much everything in OpenBSD, bgpd development is a group effort with many other developers involved. The majority of the code has been written by Claudio Jeker and myself.

My trip to AsiaBSDcon has been funded by the conference.

15 Availability

bgpd is part of OpenBSD since the 3.5 release.

This paper and the slides from my presentation will be availabe from the papers section on

`http://www.bulabula.org`

and be linked from OpenBSD's paper section on

`http://www.openbsd.org/papers`

Introduction to FreeNAS development

John Hixson
john@ixsystems.com
iXsystems, Inc.

Abstract

FreeNAS has been around for several years now but development on it has been by very few people. Even with corporate sponsorship and a team of full time developers, outside interest has been minimal. Not a week goes by when a bug report or feature request is not filed. Documentation on how to develop on FreeNAS simply does not exist. Currently, the only way to come up to speed on FreeNAS development is to obtain the source code, read through it, modify it and verify it works. The goal of this paper is to create a simple FreeNAS application to demonstrate some of the common methods used when dealing with FreeNAS development, as well as showcase some of the API.

1 Where to start

When learning to program on a new platform, where does one even begin? In the case of FreeNAS, the answer to this depends on what your needs are. If you want to modify a certain release, you can just grab an ISO image for said release and install it. If you would like to hack on new features, you will need to build your own image to work from. In either case you can checkout the source code for a release or the current development branch and build from it.

2 Where to get the code

FreeNAS source code can be obtained from github: http://github.com/freenas/freenas.git

3 How to build

The FreeNAS source code is built using the make(1) command. There are several targets available, however, most of the time the default “all” target is what you'll need. So let's look at a typical build:

```
# git clone http://github.com/freenas/freenas.git freenas
# cd freenas
# make
No git repo choice is set.  Please use "make git-external" to build as an
external developer or "make git-internal" to build as an iXsystems
internal developer.  You only need to do this once.
*** [git-verify] Error code 1

Stop in /usr/home/john/freenas.
```

At this point, the build will bomb out and tell you to either “make git-internal” or “make git-external”. A “git-external” build is what you will need, so:

```
# make git-external
# make
```

The build will now do a checkout of TrueOS source and a ports tree. What is TrueOS? TrueOS is FreeBSD with some of our local modifications in it. A frozen ports tree is also used so versions of the ports we use don't get bumped without us knowing. Once the checkout is complete, the build kicks off. Here is what happens:

1. buildworld
2. buildkernel
3. installworld
4. installkernel

5. debug kernel

Once the build has reached this point, a world chroot has been created and populated with FreeBSD. Now, ports need to be built. There are roughly 200 ports that get built. As each port is built, a package is also created and cached so that packages can be installed instead of building from source every build. After the ports are build, various customization functions run and then a raw disk image is created. When the disk image is completed, an ISO image is created as well as an GUI upgrade image. You are now ready to install FreeNAS!

4 Installing FreeNAS

Installing FreeNAS is easy. When developing for FreeNAS it is most convenient to install it in a virtual machine. FreeNAS will run fine in VMWare, VirtualBox, Parallels and even Bhyve. When you boot a FreeNAS ISO, the very first prompt is to install or upgrade. For this paper, we are installing. Upgrading is certainly an option and generally part of the development process though. Here is a rundown of a FreeNAS install:

1. Install
2. Pick the disk to install on
3. Proceed with installation
4. Eject ISO
5. Shutdown

Now, you are ready to boot FreeNAS!

Configuring FreeNAS:

When FreeNAS boots up, you will be dropped into a menu with several options. At the very minimum, a network interface must be configured along with DNS. Choose option 1 to configure the network interface and follow the instructions. Afterwards, choose option 6 to configure DNS and follow the instructions. Now you can reach FreeNAS in your web browser.

5 How to create a module

FreeNAS is primarily written in python using the django framework. So to make any kind of interface changes, you will be required to have some understanding of django. The UI for FreeNAS lives under /usr/local/www/freenasUI. To do any kind of work under this directory means you will have to mount the root file system read/write. So, let's create a module!

```
# mount -uw /
# cd /usr/local/www/freenasUI
# python manage.py startapp asiabsdcon
# ls asiabsdcon__init__.py
models.py
tests.py
views.py
```

You will see the following files:

- __init__.py
  - Not heavily used in FreeNAS but can be if using django apps as modules
- models.py
  - Represents data in the database
- tests.py
  - Not heavily used in FreeNAS, but can be used for unit tests
- views.py
  - Represents the “view” in the UI

These files are created when you run the manage.py command. FreeNAS has a few additional files that will need to be created for it to work with the UI:

- forms.py
  - “forms” that represent the model, often created in the view functions
- urls.py
  - List of URL's that get matched and what view to call
- nav.py
  - Placement of items in the navtree menu
- admin.py
  - How to display data in a datagrid (if using a datagrid).

Now that the files have been explained, we're going to create a very simple FreeNAS application. The application will display 'AsiaBSDCon' in the navtree. When you click on it, it will have an 'AsiaBSDCon 2014' subtree, under which messages can be created and displayed. This application is very silly, but will demonstrate how you create a FreeNAS application, display it in the navtree with various options, and how the UI interfaces with the applications.

```
# touch forms.py urls.py nav.py admin.py
```

First, a model must be created so that it can be represented in the database, so the models.py file is edited. In this example, the “AsiaBSDCon” class is created. It inherits from the freeadmin “Model” class which is the django “Model” class with some additional methods required by FreeNAS. This model will represent how data for this application is stored in the sqlite database used by FreeNAS. In this example, only a message with a length of 1024 bytes will be stored. Here are the meanings to the fields in the class:

a_msg – Name of the property, it is of the type models.CharField()
  max_length – Size of the field
  verbose_name – What gets displayed in front of the field on a form
  help_text – What gets displayed when the help icon is hovered over

The FreeAdmin class is used to set different meta-data used by the freeadmin app that handles must of the FreeNAS interface. In this case, icons are being set to be displayed in the navtree for the model.

```
from django.db import models
from django.utils.translation import ugettext_lazy as _
from freenasUI.freeadmin.models import Model

class AsiaBSDCon(Model):
   a_msg = models.CharField(
      max_length=1024,
      verbose_name=_("Message"),
      help_text=_("AsiaBSDCon message to display")
   )

   class Meta:
      verbose_name = _("AsiaBSDCon")
      verbose_name_plural = _("AsiaBSDCon")

   class FreeAdmin:
      icon_model = 'BobbleIcon'
      icon_object = 'BobbleIcon'
      icon_view = 'BobbleIcon'
      icon_add = 'BobbleIcon'
```

Once a FreeNAS model has been defined, it can be added to the database as a table. There are two steps to this process. The first is a schemamigration. If it is the first time doing a schemamigration for the application, django must be told so using the “--initial” flag. Otherwise, “--auto” can be used which will pick up any additional fields that may have been added to the model.

```
# python manage.py schemamigration asiabsdcon –initial
 + Added model asiabsdcon.AsiaBSDCon
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate asiabsdcon
```

What this does is verify that everything is okay and no default values need to be set. Once that is verified, a "migration" script is generated in migrations/0001_initial.py:

```
# -*- coding: utf-8 -*-
import datetime
from south.db import db
from south.v2 import SchemaMigration
from django.db import models


class Migration(SchemaMigration):

    def forwards(self, orm):
        # Adding model 'AsiaBSDCon'
        db.create_table(u'asiabsdcon_asiabsdcon', (
            (u'id', self.gf('django.db.models.fields.AutoField')(primary_key=True)),
            ('a_msg', self.gf('django.db.models.fields.CharField')(max_length=1024)),
        ))
        db.send_create_signal(u'asiabsdcon', ['AsiaBSDCon'])


    def backwards(self, orm):
        # Deleting model 'AsiaBSDCon'
        db.delete_table(u'asiabsdcon_asiabsdcon')


    models = {
        u'asiabsdcon.asiabsdcon': {
            'Meta': {'object_name': 'AsiaBSDCon'},
            'a_msg': ('django.db.models.fields.CharField', [], {'max_length': '1024'}),
            u'id': ('django.db.models.fields.AutoField', [], {'primary_key': 'True'})
        }
    }

    complete_apps = ['asiabsdcon']
```

The migration script provides to migrate to the new schema, and to migrate backwards if necessary as well. To perform the actual migration, the following command must be run:

```
# python manage.py migrate asiabsdcon
Running migrations for asiabsdcon:
 - Migrating forwards to 0001_initial.
 > asiabsdcon:0001_initial
 - Loading initial data for asiabsdcon.
Installed 0 object(s) from 0 fixture(s)
```

Now this model is represented as a table in the sqlite database. To confirm this, look at the database:

```
# sqlite3 /data/freenas-v1.db
SQLite version 3.8.0.2 2013-09-03 17:11:13
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables asiabsdcon_asiabsdcon
asiabsdcon_asiabsdcon
```

```
sqlite> .schema asiabsdcon_asiabsdcon
CREATE TABLE "asiabsdcon_asiabsdcon" ("id" integer NOT NULL PRIMARY KEY, "a_msg"
varchar(1024) NOT NULL);
```

Almost all models in FreeNAS need a form model as well. Most of the time, the form will have more fields then there are in the model along with additional method overrides to clean the passed in data and save it. In this example, we keep things simple and just tell FreeNAS to make the form the same as the model.

```
from freenasUI.common.forms import ModelForm
from freenasUI.asiabsdcon import models

class AsiaBSDConForm(ModelForm):
    class Meta:
        model = models.AsiaBSDCon
```

In order for FreeNAS to know where our application is located, we have to tell django where it resides. This is done in the urls.py file. The majority of URL patterns are just a regular expression, a name for it, and the name of the view that is called when the regular expression is matched. This example matches the FreeNAS URL + /home/", names the URL "asiabsdcon_home", and specifies the view function "asiabsdcon_home" get called. Naming of a URL pattern is used in django so that "asiabsdcon_home" can be used instead of a URL path.

```
from django.conf.urls import patterns, url

urlpatterns = patterns('freenasUI.asiabsdcon.views',
    url(r'^home/$', 'asiabsdcon_home', name="asiabsdcon_home"),
)
```

Here is the "asiabsdcon_home" view that was specified in urls.py. All views are in views.py. All view functions take a request object as a parameter. Additional parameters can be specified in the urls.py file. Often times, the unique ID for the object being referenced is matched in urls.py and a parameter to the view function.

This is a typical view for a form that handles GET and POST methods. If the request method is GET, an empty form is created and rendered. If the request method is POST, the form is checked to be valid, saved, and a message is returned indicating the operation was successful. The render() call relies on an HTML template directory for asiabsdcon to exist and that the home.html file exists.

```
from django.shortcuts import render
from django.utils.translation import ugettext_lazy as _
from freenasUI.freeadmin.views import JsonResp
from freenasUI.asiabsdcon import forms

def asiabsdcon_home(request):

    if request.method == 'POST':
        form = forms.AsiaBSDConForm(request.POST)
        if form.is_valid():
            form.save()
            return JsonResp(
                request,
                message=_("AsiaBSDCon successfully added.")
            )
    else:
        form = forms.AsiaBSDConForm()

    return render(request, "asiabsdcon/home.html", {
        'form': form
    })
```

Here is the contents of templates/asiabsdcon/home.html. This just "inherits" the freeadmin/generic_form.html file.

Since nothing is being added on here, it's simple. However, there is much flexibility with templates just as with classes. Templates can be inherited and properties within them can be overridden.

```
{% extends "freeadmin/generic_form.html" %}
```

The last part of this example is the navtree code. The job of the navtree is to display menus in a tree structure. Generally there will be a top level item with an icon with items or sub menus beneath it that contain operations to add, edit, configure and delete objects. Some items will just be a form that pops up, others a datagrid, and still others a tab in a multi pane window. All of the code goes into nav.py, which is a dynamically loaded TreeNode() object. So at the top level, a NAME and ICON will be defined. There are other properties available, but this is a simple example. For further reference, the freeadmin/tree/tree.py and freeadmin/navtree.py files can be checked out. What this does is show “AsiaBSDCon” with a beastie icon in the navtree. To show menus beneath it, more classes will need to be defined. So the “AsiaBSDConView” class is defined. It sets it's name to “AsiaBSDCon 2014”, which is what will show beneath the “AsiaBSDCon” menu. In it's init method, it creates a node where new messages can be added. This is displayed beneath “AsiaBSDCon 2014”. The init method then grabs all the messages (if any) that exist in the database and creates nodes for them and appends it beneath the “AsiaBSDCon 2014” menu. The additional fields in the for loop are as follows:

gname – Unique identifier for this TreeNode
name – Text displayed in the navtree
type – Tells the Javascript layer what function to call
view – The view to use
kwargs – Arguments to pass to the view
app_name – The name of the application

The magic that is occurring here is in the type, view and kwargs properties. The type is saying to use the Javascript editObject() function. The view 'freeadmin_asiabsdcon_asiabsdcon_edit' is a special view created for all models when FreeNAS starts. The syntax is “freeadmin_APPLICATION_MODEL_(add|edit|delete|datagrid)”. Whenever calling an edit or delete function, the unique ID needs to be passed as an argument so it's set in the kwargs field.

```
from django.utils.translation import ugettext_lazy as _
from freenasUI.asiabsdcon import models
from freenasUI.freeadmin.tree import TreeNode

NAME = _('AsiaBSDCon')
ICON = 'BeastieIcon'

class AsiaBSDConView(TreeNode):
    gname = 'AsiaBSDCon'
    name = _(u'AsiaBSDCon 2014')
    icon = 'BobbleIcon'

    def __init__(self, *args, **kwargs):
        super(AsiaBSDConView, self).__init__(*args, **kwargs)

        node = TreeNode()
        node.name = _(u'Add message')
        node.type = 'object'
        node.view = 'freeadmin_asiabsdcon_asiabsdcon_add'
        node.icon = 'SettingsIcon'
        self.append_child(node)

        msgs = models.AsiaBSDCon.objects.all()
        for m in msgs:
            node = TreeNode()
            node.gname = m.a_msg
            node.name = m.a_msg
            node.type = 'editobject'
            node.view = 'freeadmin_asiabsdcon_asiabsdcon_edit'
            node.kwargs = {'oid': m.id}
            node.model = 'AsiaBSDCon'
            node.icon = 'BobbleIcon'
        node.app_name = 'AsiaBSDCon'
        self.append_child(node)
```

Now that the application is complete, restart django so that the changes are visible in the UI.

```
# service django restart
```

6 Testing changes

Testing changes is no different than any other kind of web development. The only difference on FreeNAS is that if any changes are made to python code, django will need to be restarted. Changes made to HTML templates or Javascript do not require a restart. Javascript changes require a refresh whereas HTML template changes require no restart or refresh.

7 Debugging

Most debugging on FreeNAS comes from the python logging module. Just about every file has an import logging statement with it's on unique name. Logging for django is configured in django's settings.py file. DEBUG can also be set to TRUE so that when a crash occurs you get a python stack trace that helps isolate the issue. Other useful debugging methods are chrome and Firefox build in Javascript debuggers for when Ajax methods are crashing but it isn't immediately obvious from the UI. Lots of useful information is also written out to the following logs:

/var/log/messages
/var/log/nginx-access.log
/var/log/nginx-error.log

Usually, if django won't even start, a python stack trace indicating why can be found in the nginx error log. /var/log/debug.log can also be used but must be uncommented in /etc/syslog.conf and syslogd restarted.

How to get involved

FreeNAS is always looking for help! If you are interested in joining the project, helping out or submitting patches, go to http://www.freenas.org and go to 'Our community'. Source code can be obtained from http://github.com/freenas/freenas.git. If you have patches, please send a pull request.

8 Conclusion

FreeNAS is a very powerful operating system. It has become very popular. With popularity comes many more people using it and that means more bug reports and feature requests. The people who bring you FreeNAS are a small team and need help. It is my hope that by demonstrating this simple application that more developers will be interested in hacking on FreeNAS.

VXLAN and Cloud-based networking with *Open*BSD

Reyk Floeter (reyk@openbsd.org)

March 2014

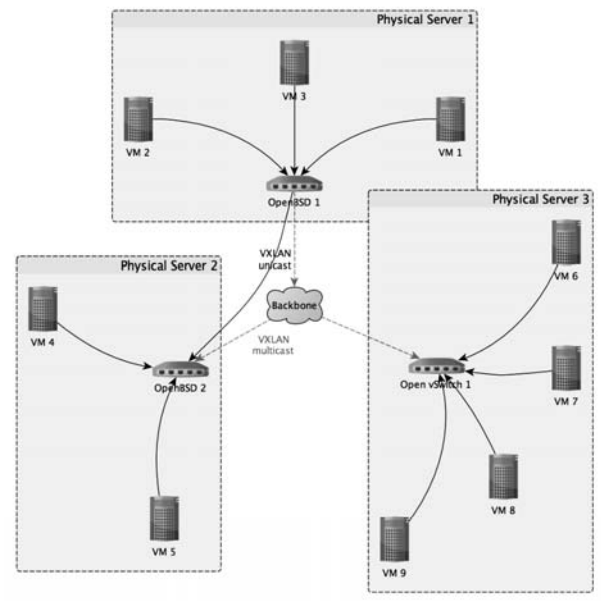
Abstract

This paper introduces the new vxlan(4) driver in the upcoming OpenBSD 5.5 release and illustrates some of the upcoming features in the area of Software Defined Networking (SDN). Today's dominance of VM-based infrastructures has heavily influenced the networking realm. The "Cloud" caused vendors to introduce a number of new protocols, technologies and methodologies. Despite the buzz, they had to adopt the paradigm shift to split physical and virtual infrastructures: the traditional network is used to interconnect physical hosts (hypervisors or Virtual Machine Monitor (VMM)) and the virtual network is used to interconnect virtual machines. The Virtual eXtensible LAN (VXLAN) protocol allows to run "virtual overlay networks" and eliminates some of the limitations of traditional IEEE 802.11Q Virtual LAN (VLAN). It is an IP/UDP-encapsulated tunnelling protocol for overlaying layer 2 networks over layer 3 networks which is used in the VMware world and between virtual switches like Open vSwitch or the Cisco Nexus 1000V. The vxlan(4) driver allows to create or join these networks without depending on such vswitches and it is very useful to run PF, relayd or OpenBSD's numerous VPN services in such environments.

1 Introduction

The first implementation of vxlan(4) for OpenBSD was written by David Gwynne (dlg@openbsd.org) who wanted to connect OpenBSD systems to a VMware infrastructure and the Cisco Nexus 1000V. The implementation, called `vxland`[2], is a userland daemon that uses the tun(4) network tunnel pseudo-device to exchange layer 2 packets with the kernel network stack.

Although this is a reasonable approach, a kernel-based implementation was a much more desirable solution. After a discussion with David Gwynne, it was concluded that OpenBSD should have a driver that works more like gif(4) or vlan(4). A driver that can be configured with ifconfig(8) without the need for an additional daemon. The design & implementation of the vxlan(4) driver was started and it turned out to be simpler task implementing it in the kernel because of the existing infrastructure of OpenBSD's network stack.



2 Design & Implementation

The VXLAN protocol is defined in an Internet Draft[4]. The standard is neither finished nor officially released but it is already widely deployed with current networking products. The initial implementation of the vxlan(4) driver was based on draft-mahalingam-dutt-dcops-vxlan-04, while the current version at the time of this writing is version draft-mahalingam-dutt-dcops-vxlan-08[4]. The differences between these versions include formatting and clarifications, but no changes to the protocol itself.

2.1 Protocol Overview

The VXLAN protocol provides layer 2 tunnelling over IP with UDP datagrams. Each vxlan interface uses a 24-bit VXLAN Network Identifier (VNI) that distinguishes multiple virtualized layer 2 networks and their tunnels between identical tunnel endpoints. Once configured, the interface encapsulates and decapsulates Ethernet frames in UDP datagrams that are exchanged with tunnel endpoints. The default UDP port for VXLAN traffic that has been assigned by IANA is 4789, but some older implementations use the port 8472.

The UDP datagrams include a header that carries the 24bit VNI, the "valid VNI" flag, and some reserved flags and fields that must be set to zero on the transmit side and ignored on the receiving side. The header is defined in the Internet Draft and implemented as the following C structure:

```
struct vxlan_header {
        u_int32_t                 vxlan_flags;
#define VXLAN_FLAGS_VNI           0x08000000
#define VXLAN_RESERVED1           0xf7ffffff
        u_int32_t                 vxlan_id;
#define VXLAN_VNI                 0xffffff00
#define VXLAN_VNI_S               8
#define VXLAN_RESERVED2           0x000000ff
} __packed;
```

As defined by the standard, the header structure defines two 32bit words with some masks and offsets for the bit operations. Please note that OpenBSD does not use bitfields in C structures, it is a general convention that exists for both historical and portability reasons.

In OpenBSD, the VNI is also called Virtual Network Identifier (vnetid) to use a term that is not specific to the VXLAN protocol in the general network stack and configuration parts of the system.

The complete IPv4 transport header is implemented with the C struct below. Each datagram includes such a header followed by an encapsulated layer 2 Ethernet frame of the inner protocol:

```
#ifdef INET
struct vxlanudpiphdr {
        struct ipovly           ui_i;
        struct udphdr           ui_u;
        struct vxlan_header     ui_v;
} __packed;
#endif
```

The current implementation only supports Internet Protocol Version 4 (IPv4) for the outer IP transport protocol. As VXLAN is normally used in internal datacenter backbone networks, and most vendors only properly do IPv4, the need for Internet Protocol Version 6 (IPv6) didn't have a priority in the initial implementation. IPv6 support will be added in the future and, of course, it is already possible to use any protocol or Ethernet type within the tunnel.

2.2 Network Interface

The vxlan(4) interface is implemented as a virtual Ethernet pseudo-interface using the "cloner" kernel API. The driver implementation was written from scratch, but inspired by techniques of the existing vlan(4) and vether(4) drivers.

The `vxlan_clone_create()` function is called when a new vxlan(4) interface is created, typically after running the `ifconfig vxlanX create` command. The function sets up a new Ethernet interface and registers it to the kernel. Because it is fully virtual interface that does not relate to a physical parent, it generates a random MAC address by calling OpenBSD's standard `ether_fakeaddr()` function. Note that the fake MAC address will change every time the interface is created, and after boot, but it is possible to use the `ifconfig vxlanX lladdr` command to specify a fixed address. The function also initializes different options and prepares the driver for multicast operation.

The Maximum Transmission Unit (MTU) is set to 1500 bytes and the "hardmtu", the hard MTU limit, to a maximum of 65535 (0xffff) bytes. The outer protocol headers will take at least 50 extra bytes resulting in an effective MTU of 1550 bytes. For this reason, most other implementations chose to lower the VXLAN MTU to 1450 bytes by default, to carry the encapsulated traffic over interfaces with the default Ethernet MTU of 1500 bytes. Lowering the MTU for tunnelling interfaces is the traditional approach that solved one problem but caused many others including path MTU discovery, fragmentation issues etc. When discussing it with Theo de Raadt (deraadt@openbsd.org), we decided to take a different approach with vxlan(4) in OpenBSD: instead of using a decreased MTU of just 1450 bytes, vxlan(4) defaults to the full Ethernet MTU. Users should configure a larger size on the transport interfaces of the VXLAN Tunnel End Point (VTEP) accordingly.

The latest update 08 of the Internet Draft[4] even confirmed this decision in section 4.3 of the document:

> *VTEPs MUST not fragment encapsulated VXLAN packets due to the larger frame size. The destination VTEP MAY silently discard such VXLAN fragments. To ensure end to end traffic delivery without fragmentation, it is RECOMMENDED that the MTUs (Maximum Transmission Units) across the physical network infrastructure be set to a value that accommodates the larger frame size due*

to the encapsulation. Other techniques like Path MTU discovery (see [RFC1191 and [RFC1981]) MAY be used to address this requirement as well.

2.3 Multicast Support

The VXLAN protocol uses unicast and multicast messaging for the communication between peers. If the configured tunnel endpoint destination is a multicast address, the `vxlan_multicast_join()` function configures IP multicast operation. It retrieves the transport interface that is associated with the source address and routing domain, and joins the configured multicast group on that interface. The `in_addmulti()` function is called to register the address and to program the hardware multicast filters of the specified network interface accordingly.

The driver additionally registers three interface hooks: an address hook, a link state hook, and a detach hook. These hooks are used to update the multicast association whenever a change happened on the transport interface. The link state hook is used to forcibly re-register the multicast address on link state changes - this is not strictly required but fixes an issue with network interface in VMware that can loose multicast associations after the VM was suspended.

2.4 Bridge Integration

By default, vxlan(4) can be configured with either a unicast or multicast tunnel endpoint address. If a multicast address is configured, the tunnel endpoint can consist of multiple peers that receive messages to the registered group with an identical VNI. This turns the VXLAN network into a shared medium - or a virtual Ethernet hub. Each encapsulated packet from the source VTEP is "broadcasted" to all other tunnel endpoints in the multicast group.

The VXLAN protocol draft additionally defines "Unicast VM to VM communication" where a VTEP should map the outer tunnel IP addresses of remote endpoints with inner destination MAC addresses. This way, only packets with an "unknown destination" are send to the multicast group and packets with a known destination are directly send to a mapped tunnel endpoint. It should dynamically learn the mapping from the outer source IP address and inner source MAC address of received packets.

Even if the "bridge" term is not mentioned in the document, the described behavior matches the functionality of a learning bridge. The BSD bridge(4) driver almost provides the desired functionality: it forwards packets between multiple endpoints and learns a mapping of network interfaces to destination MAC addresses. The vxlan(4) implementation utilizes this fact to implement the unicast mapping. The bridge code has been extended to learn the tunnel endpoint address in addition to the outgoing network interface.

The operation is as follows: The driver tags the received mbuf with the outer source IP address using the `PACKET_TAG_TUNNEL` tag. If the tag is present, the main bridge learning function `bridge_rtupdate()` retrieves the address from the mbuf and stores it in the bridge cache, in addition to the receiving network interface. When a packet is send, the bridge will look up the destination MAC address in the bridge cache and decides to either send it to a found unicast address or to broadcast it to all interfaces on the bridge. If the destination is a unicast address, and a tunnel endpoint address is found in the bridge cache, the `bridge_output()` function tags the mbuf with the address and sends it out through the mapped interface. Finally, the `vxlan_output()` function will look up the tag and the stored endpoint address to either send the packet to the configured multicast group or the retrieved endpoint IP address.

The configured VXLAN port is used as the UDP destination port unless the `IFF_LINK0` flag is set; the original source port of the learned tunnel endpoint is used in this case.

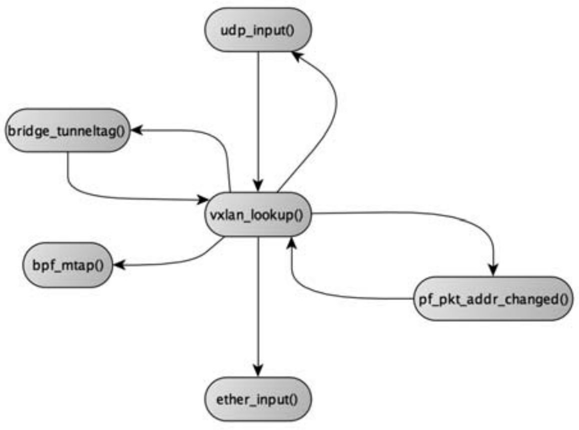
The described mechanism is enabled if the vxlan(4) interface is a member of a bridge(4) interface. The vxlan(4) driver will check if the interface is associated to a bridge and set or get the mbuf tag accordingly. The vxlan(4) interface can be the only interface that is configured on the bridge(4). But the trick also works in a "normal" bridge configuration with multiple local vxlan- or non-vxlan network interfaces. If vxlan(4) is not associated to a bridge and configured with a multicast destination address, it will fall back to the default multicast- or "Ethernet hub"-mode as described earlier.

The changes of the bridge(4) code have been designed in a way that is independent from vxlan(4) . The implementation provides a generic way to store tunnel endpoint addresses as an abitrary `sockaddr` by allowing a driver to set or get the `PACKET_TAG_TUNNEL` tag using the new `bridge_tunneltag()`, `bridge_tunneluntag()` and `bridge_tunnel()` API functions.

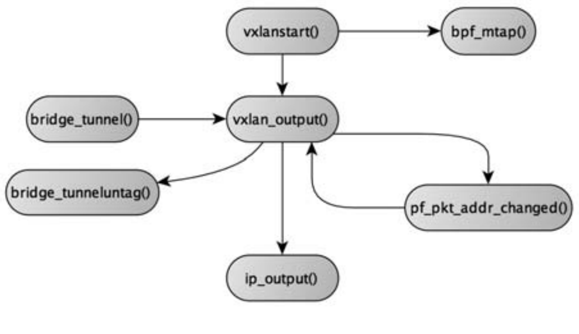
2.5 Send and Receive

In addition to the generic bridge integration, the vxlan(4) driver requires minimal changes in the network stack. Only two hooks are used for the send and receive functions `vxlan_output()` and `vxlan_lookup()`.

On the sending side, the registered Ethernet driver `vxlanstart()` callback sends every packet from the transmit queue with the `vxlan_output()` function. The function prepends the VXLAN, UDP and IP headers to the Ethernet frame, retrieves the tunnel endpoint address, updates the IP/UDP checksums and sends the UDP packet with `ip_output()` as `IP_RAWOUTPUT`.



The integration on the receiving side is a bit less elegant because OpenBSD does not provide a pluggable interface to register UDP servers in the kernel. The receive function `vxlan_lookup()` is directly called in `udp_input()` if at least one vxlan(4) interface is configured on the system; this hardcoded hook was inspired by the existing IPsec "UDP encapsulation" receiving path. The lookup function validates the VXLAN header and tries to find a configured interface that matches the UDP destination port, vnetid and routing domain of the received packet. If no vxlan(4) interface is found, the packet is returned to the stack for further processing, otherwise it is consumed by the vxlan(4) interface and the inner Ethernet packet is decapsulated and fed into `ether_input()`.



Calling `vxlan_lookup()` for every received UDP packet introduces an overhead for UDP traffic if at least one vxlan(4) interface is configured. An alternative implementation would call the receiving function after the PCB hash lookup of the UDP sockets and tie it into the socket code, similar to the `pipex_l2tp_*` code path. This approach has not been implemented; it would cause an increased complexity and more dependencies in the network stack. The implementation of a generic and abstracted interface for kernel-based UDP "servers" would provide another solution.

2.6 ioctl(2) Interface

The vxlan(4) interface is configured using the standard Ethernet ioctls and the tunnel configuration options. It uses the existing tunnel ioctls `SIOCSLIFPHYADDR`, `SIOCDIFPHYADDR`, `SIOCGLIFPHYADDR`, `SIOCSLIFPHYRTABLE` and `SIOCGLIFPHYRTABLE` from gif(4) and gre(4) to configure the tunnel endpoint addresses. The `IFPHYADDR` ioctls have been extended to include an optional port in the supplied `sockaddr`. The new ioctls `SIOCSVNETID`, `SIOCGVNETID`, `SIOCSLIFPHYTTL` and `SIOCGLIFPHYTTL` have been added to configure the vnetid and an optional non-standard multicast TTL.

3 Configuration Examples

The first example creates an interface in the non-standard unicast mode, which does not involve any multicast communication:

```
# ifconfig vxlan0 tunnel 192.168.1.100
192.168.1.200 vnetid 5
# ifconfig vxlan0 10.1.1.100/24
```

The next example creates a vxlan(4) interface in the default multicast mode with the local source IP address 192.168.1.100, the destination multicast group 239.1.1.100 and the vnetid 7395:

```
# ifconfig vxlan0 tunnel 192.168.1.100
239.1.1.100 vnetid 7395
# ifconfig vxlan0 10.1.2.100/24
```

Adding the previous interface to a bridge will enable the learning of tunnel endpoint addresses and enable the dynamic multicast and unicast mode:

```
# ifconfig bridge0 create
# ifconfig bridge0 add vxlan0 up
```

The VXLAN interface can be used like any other Ethernet interface in OpenBSD with network features like Packet Filter (PF), carp(4) or services like `dhcpd`.

4 Portability and other Implementations

Because of the minimal dependencies in the network stack, the vxlan(4) driver should be easily portable to other the BSDs. The author knows no porting efforts at present. There are other implementations of VXLAN that supposedly work under FreeBSD and NetBSD, but none of them was written as a "traditional" BSD network interface driver. In addition to the least-known `vxland`[2], there are Linux drivers that have been ported to these BSDs, including "upa/vxlan"[5] and "ovs-vxlan" of the Open vSwitch[7] stack. Interoperability between OpenBSD's vxlan(4) and Open vSwitch on Linux has been intensively tested during the development of the driver.

5 Future Work

In addition to vxlan(4) itself, future work includes improved Cloud-networking capabilities in OpenBSD.

5.1 The vxlan(4) driver

The vxlan(4) driver is enabled and usable in OpenBSD but can be improved for future releases. One important task is support for IPv6 as the IP transport protocol. Additionally, further improvements in the multicast support are possible. Ongoing changes in the Internet Draft[4] have to be adopted and the handling of IP fragmentation has to be updated according to the latest clarifications in draft 08.

A generic abstracted API for kernel-based UDP servers would allow to improve the receive-side of vxlan(4) , IPsec UDP encapsulation and potentially `pipex`.

5.2 NVGRE

An competing protocol is Network Virtualization using Generic Routing Encapsulation (NVGRE), a Microsoft-driven overlay network protocol that is using GRE instead of UDP as the transport protocol. NVGRE is not supported by OpenBSD at present, but the existing gre(4) driver, the applied changes to the bridge(4) code and the ifconfig(8) interface provide all of the requirements for NVGRE. The gre(4) driver would need additional support for GRE keys to hold the NVGRE header and vnetid, the additional virtual network segmentation and the required hooks for the bridge tunnel tag. The implementation of NVGRE would be trivial but has been abandoned because configuring it for interoperability testing on the PowerShell of Windows Server 2012 turned out to be a time-consuming task that seemed to be beyond my abilities.

5.3 SDN and NFV

SDN is a term that is used for many different protocols, even VXLAN, but is primarily related to the OpenFlow protocol. Integrating OpenBSD's network stack with OpenFlow, an SDN controller and additional features is part of the future work. The Network Functions Virtualization (NFV) approach might be either just another buzzword, or an approach that suits very well to OpenBSD's comprehensive network stack. The initial definition of NFV is not much more than a description of running network services in "software" instead of traditional "hardware" routers, but NFV evolves into a framework that combines several software-based networking techniques in virtualized environments. OpenBSD should be a choice for NFV and future work will evaluate the possibilities.

6 Appendix

6.1 About the Author

Reyk Floeter is the founder of Esdenera Networks GmbH[1], a company that develops OpenBSD-based networking and security products for cloud-based and software-defined networks. He is located in Hannover, Germany, but works with international customers like Internet Initiative Japan Inc. (IIJ) in Tokyo[3]. As a member of the OpenBSD[6] project, he contributed various features, fixes, networking drivers and daemons since 2004, like OpenBSD's ath, trunk (a.k.a. lagg), vic, hostapd, relayd, snmpd, and iked. For more than nine years and until mid-2011, he was the CTO & Co-Founder of .vantronix where he gained experience in building, selling and deploying enterprise-class network security appliances based on OpenBSD.

References

[1] Esdenera, *Esdenera Networks GmbH*, `http://www.esdenera.com/`.

[2] David Gwynne, *vxland*, `https://source.eait.uq.edu.au/svn/vxland/`.

[3] IIJ, *Internet Initiative Japan Inc.*, `http://www.iij.ad.jp/`.

[4] K. Duda P. Agarwal L. Kreeger T. Sridhar M. Bursell C. Wright M. Mahalingam, D. Dutt, *VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*, `http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-08`, February 2014.

[5] Ryo Nakamura, *upa/vxlan*, `https://github.com/upa/vxlan/`.

[6] OpenBSD, *The OpenBSD Project*, `http://www.openbsd.org/`.

[7] Open vSwitch, *An Open Virtual Switch*, `http://www.openvswitch.org/`.

Nested Paging in bhyve

Neel Natu
The FreeBSD Project
neel@freebsd.org

Peter Grehan
The FreeBSD Project
grehan@freebsd.org

Abstract

Nested paging is a hardware technique used to reduce the overhead of memory virtualization. Specifically, this refers to Intel EPT (Extended Page Tables) and AMD NPT (Nested Page Tables). Nested paging support is available in bhyve starting from FreeBSD [1] 10.0 and provides useful features such as transparent superpages and overprovisioning of guest memory. This paper describes the design and implementation of nested paging in bhyve.

1 Introduction

Intel and AMD have introduced extensions to the x86 architecture that provide hardware support for virtual machines, viz.

- Intel Virtual-Machine Extensions (VMX) [2, 3]
- AMD Secure Virtual Machine (SVM) [4]

The first generation of VMX and SVM did not have any hardware-assist for virtualizing access to the memory management unit (MMU). Hypervisors had to make do with existing paging hardware to protect themselves and to provide isolation between virtual machines. This was typically done with a technique referred to as *shadow paging* [5].

A hypervisor would examine the guest paging structures and generate a corresponding set of paging structures in memory called *shadow page tables*. The shadow page tables would be used to translate a guest-virtual address to a host-physical address. The hypervisor would be responsible for keeping the shadow page tables synchronized with the guest page tables. This included tracking modifications to the guest page tables, handling page faults and reflecting accessed/dirty (A/D) bits from the shadow page tables to guest page tables. It was estimated that in certain workloads shadow paging could account for up to 75% of the overall hypervisor overhead [5].

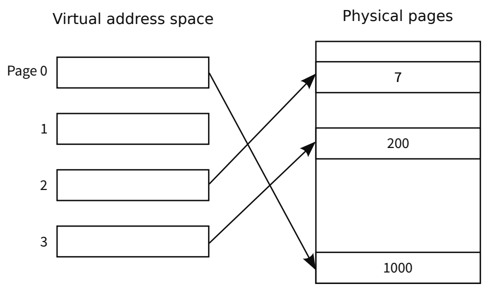
Nested page tables were introduced in the second generation of VMX and SVM to reduce the overhead in virtualizing the MMU. This feature has been shown to provide performance gains upwards of 40% for MMU-intensive benchmarks and upwards of 500% for micro-benchmarks [6].

With x86_64 page tables there are two types of addresses: virtual and physical. With nested page tables there are three types of addresses: guest-virtual, guest-physical and host-physical. The address used to access memory with x86_64 page tables is instead treated as a guest-physical address that must be translated to a host-physical address. The guest-physical to host-physical translation uses nested page tables which are similar in structure to x86_64 page tables.

bhyve has always relied on nested page tables to restrict guest access to memory, but until the nested paging work described in this paper it wasn't a well-behaved consumer of the virtual memory subsystem. All guest memory would be allocated upfront and not released until the guest was destroyed. Guest memory could not be swapped to stable storage nor was there a mechanism to track which pages had been accessed or modified[1].

The nested paging work described in this paper allows bhyve to leverage the FreeBSD/amd64 *pmap* to maintain nested paging structures, track A/D bits and maintain TLB consistency. It also allows bhyve to represent guest memory as a FreeBSD *vmspace* and handle nested page faults in the context of this *vmspace*.

[1]Modified and dirty are used interchangeably in this paper



MMU

Virt	Phys	Attr
0	1000	readonly
1		unmapped
2	7	readwrite
3	205	readwrite

Figure 1: Memory Management Unit

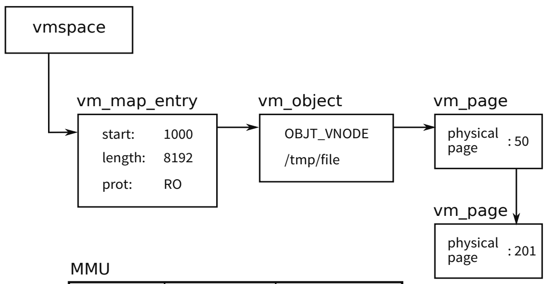
The rest of the paper is organized as follows: Section 2 provides an overview of virtual memory management in FreeBSD on x86_64 processors. Section 3 describes the virtualization extensions in Intel CPUs. Section 4 introduces Intel's implementation of nested page tables. Sections 5, 6 and 7 describe the design and implementation of nested paging in bhyve. Section 8 presents results of experimental evaluation of the overhead of nested paging. Section 9 looks at opportunities to leverage nested page tables for several useful features.

2 FreeBSD virtual memory management

The FreeBSD virtual memory (VM) subsystem provides each process with a virtual address space. All memory references made by a process are interpreted in the context of its virtual address space. These virtual addresses are translated into physical addresses by the MMU as shown in Figure 1.

The MMU performs address translation in fixed-sized units called pages. The size of a page is machine-dependent and for the x86_64 architecture this can be 4KB, 2MB or 1GB. The MMU also protects physical pages belonging to an address space from being written to or read from a different address space. All MMU implementations allow a translation to be marked as readonly while some implementations can keep track of which pages have been read or written by the process.

The process address space in FreeBSD is represented by a *vmspace* [7]. The address space is divided into contiguous ranges such that all addresses in a range are



MMU

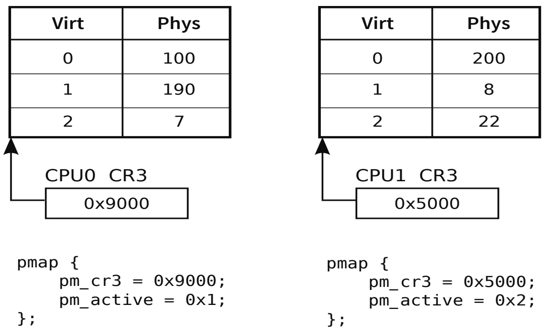
Virt	Phys	Attr
1000	50	readonly
1001	201	readonly

Figure 2: mmap(/tmp/file, 8192, readonly)

mapped with the same protection (e.g., readonly) and source data from the same backing object (e.g., a file on disk). Each range is represented by a *vm_map_entry*. The physical memory pages provided by the backing object are mapped into the address range represented by the *vm_map_entry*. The backing object is represented by a *vm_object*. The physical memory pages associated with the backing object are represented by a *vm_page*. A *vm_page* contains the physical address of the page in system memory. This address is used by the MMU in its translation tables. Figure 2 shows the data structures involved in a readonly mapping of */tmp/file* into a process's address space.

The physical-mapping (pmap) subsystem provides machine-dependent functionality to the VM subsystem, such as:

- Creating virtual to physical mappings
- Invalidating a mapping to a physical page
- Modifying protection attributes of a mapping



Virt	Phys
0	100
1	190
2	7

```
pmap {
    pm_cr3 = 0x9000;
    pm_active = 0x1;
};
```

Virt	Phys
0	200
1	8
2	22

```
pmap {
    pm_cr3 = 0x5000;
    pm_active = 0x2;
};
```

Figure 3: pmap

- Tracking page access and modification

Each *vmspace* has an embedded *pmap*. The *pmap* contains machine-dependent information such as a pointer to the root of the page table hierarchy.

For the x86_64 architecture the pmap subsystem maintains mappings in hierarchical address-translation structures commonly referred to as *page tables*. The page tables are the machine-dependent representation of the *vmspace*. The processor's control register CR3 points to the root of the page tables.

It is important to note that multiple processors may have an address space active simultaneously. This is tracked by the *pm_active* bitmap. Figure 3 depicts a dual-processor system with a different address space active on each processor.

2.1 x86_64 address translation

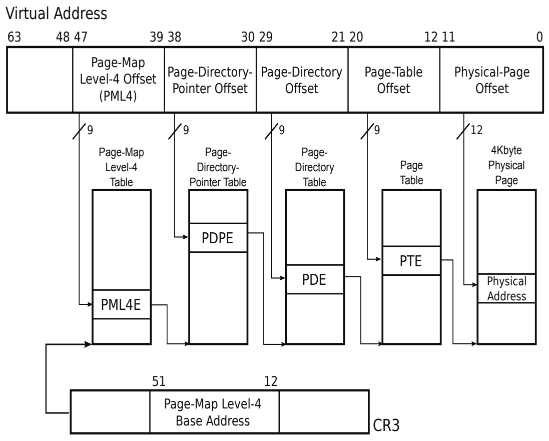


Figure 4: x86_64 address translation

A page-table-page is 4KB in size and contains 512 page-table-entries each of which is 64-bits wide. A page-table-entry (PTE) contains the physical address of the next level page-table-page or the page-frame. A page-table-entry also specifies the protection attributes, memory type information and A/D bits.

As shown in Figure 4, a 64-bit virtual address is divided into 4 fields with each field used to index into a page-table-page at different levels of the translation hierarchy:

- Bits 47:39 index into the page-map level4 table
- Bits 38:30 index into the page-directory pointer table
- Bits 29:21 index into the page-directory table
- Bits 20:12 index into the page table
- Bits 11:0 provide the offset into the page-frame

3 Intel Virtual-Machine Extensions

Intel Virtual-Machine Extensions (VMX) provide hardware support to simplify processor virtualization. This is done by introducing two new forms of processor operation: VMX root and VMX non-root.

A hypervisor runs in VMX root operation and has full control of the processor and platform resources. Guests run in VMX non-root operation which is a restricted environment.

A guest starts executing when the hypervisor executes the *vmlaunch* instruction to transition the processor into VMX non-root operation. The guest continues execution until a condition established by the hypervisor transitions the processor back into VMX root operation and resumes hypervisor execution. The hypervisor will examine the reason for the VM-exit, handle it appropriately, and resume guest execution.

The VMX transition from hypervisor to guest is a *VM-entry*. The VMX transition from guest to hypervisor is a *VM-exit*. VMX transitions and non-root operation are controlled by the *Virtual Machine Control Structure (VMCS)*. The VMCS is used to load guest processor state on VM-entry and save guest processor state on VM-exit. The VMCS also controls processor behavior in VMX non-root operation, for example to enable nested page tables. Of particular importance is the *Extended-Page-Table Pointer (EPTP)* field of the VMCS which holds the physical address of the root of the nested page tables.

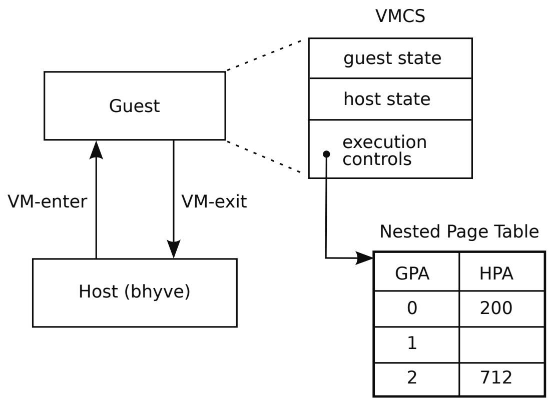


Figure 5: VMX operation

Figure 5 illustrates the VMX transitions and the nested page tables referenced from the VMCS.

4 Intel Extended Page Tables

The x86_64 page tables translate virtual addresses to physical addresses. This translation is done using page tables pointed to by CR3. In addition to mapping the virtual address to a physical address, the page tables also provide permissions and memory type associated with the mapping.

When the processor is operating in guest context and nested page tables are enabled, the physical address that is the output of x86_64 page tables is treated as a guest-physical-address. The nested page tables translate this guest-physical-address (GPA) to a host-physical-address (HPA). It is the HPA that is driven out on the processor's memory and I/O busses. This additional level of address translation allows the hypervisor to isolate the guest address space.

Note that with nested paging enabled there are two distinct page tables active simultaneously:

- x86_64 page tables pointed to by guest CR3
- nested page tables pointed to by the VMCS

Intel's implementation of nested page tables is called *Extended Page Tables (EPT)*. EPT is similar in structure and functionality to x86_64 page tables. It has same number of translation levels and it uses the the same bits to index into the page-table-pages. For example, bits 47:39 of the GPA index into the PML4 table. It also provides the same protection attributes as x86_64 page tables.

However, there are some differences.

4.1 Page-table-entry

The most obvious difference between the page-table-entries in Table 4.1 is that different bit positions are used to express the same functionality. For example, the dirty flag is bit 6 in the x86_64 PTE versus bit 9 in the EPT PTE.

Some differences arise when the x86_64 PTE has functionality that does not exist in the EPT PTE. Bit 8 in the x86_64 PTE is used to represent mappings that are global and are not flushed on an address space change. There is no corresponding bit in the EPT PTE because this functionality is not relevant in extended page tables.

The EPT PTE and x86_64 PTE also differ in their *default settings*. The execute permission must be explicitly granted in an EPT PTE whereas it must be explicitly revoked in a x86_64 PTE.

Bit	x86_64 PTE	EPT PTE
0	Valid	Read permission
1	Write permission	Write permission
2	User/Supervisor	Execute permission
3	Write-through cache	Memory type[0]
4	Cache disable	Memory type[1]
5	Accessed	Memory type[2]
6	Dirty	Ignore guest PAT
7	Page Attribute Table index	Ignored
8	Global	Accessed
9	Ignored	Dirty
61	Execute disable	Suppress #VE

Table 1: Differences between x86_64 and EPT PTEs

4.2 Capabilities

Table 4.2 highlights differences in the capabilities of x86_64 page tables and EPT page tables.

Capability	x86_64 PTE	EPT PTE
2MB mapping	Yes	Optional
A/D bits	Yes	Optional
Execute-only mapping	No	Optional

5 Design of nested paging in bhyve

The address space of a typical guest is shown in Figure 6. This guest is configured with 2GB of system memory split across two memory segments: the first segment starts at 0x00000000 and the second segment starts at 0x100000000. The region of the address space between 1GB and 4GB is called the *PCI hole* and is used for addressing *Memory-Mapped I/O (MMIO)* devices. The guest's system firmware[2] is mapped readonly in the address space just below 4GB.

The nested paging implementation in bhyve is based on the observation that the guest address space is similar to process address space:

- Guest memory segments are backed by a *vm_object* that supplies zero-filled, anonymous memory.
- Guest firmware is backed by a *vm_object* that is associated with the firmware file and is mapped readonly.

[2]The guest BIOS or UEFI image

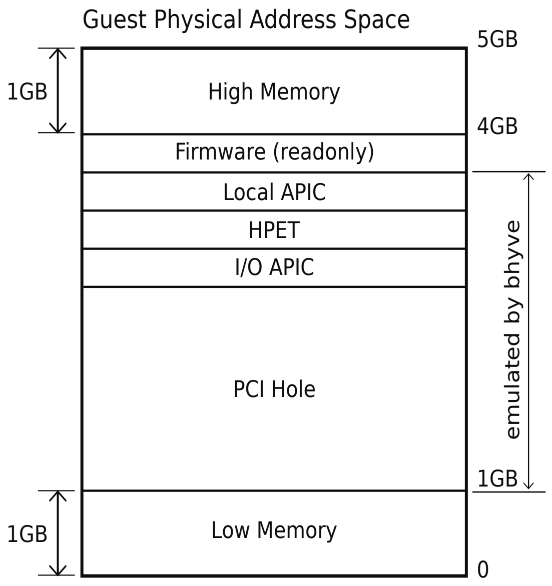


Figure 6: Guest address space

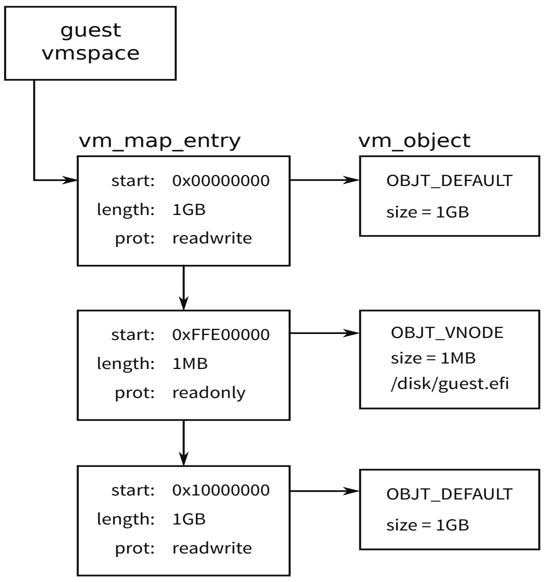


Figure 7: Guest vmspace

- The PCI hole is not mapped. Any access to it from the guest will cause a nested page fault.

Figure 7 shows the guest address space represented as a *vmspace*. The VM subsystem already had the primitives needed to represent the guest address space in this manner. However, the pmap needed modifications to support nested paging.

6 pmap modifications

The pmap subsystem is responsible for maintaining the page tables in a machine-dependent format. Given the differences between the x86_64 page tables and EPT, modifications were required to make the pmap EPT-aware.

6.1 pmap initialization

The *pmap* was identified as an x86_64 or EPT pmap by adding an enumeration for its type.

```
enum pmap_type {
    PT_X86,     /* regular x86 page tables */
    PT_EPT,     /* Intel's nested page tables */
    PT_RVI,     /* AMD's nested page tables */
};
struct pmap {
    ...
    enum pmap_type pm_type;
};
```

Prior to the nested paging changes *vmspace_alloc()* called *pmap_pinit()* to initialize the *pmap*. *vmspace_alloc()* was modified to accept a pointer to the pmap initialization function.

```
struct vmspace *
vmspace_alloc(min, max, pmap_pinit_t pinit)
{
    /* Use pmap_pinit() unless overridden by the caller */
    if (pinit == NULL)
            pinit = &pmap_pinit;
}
```

A new function *pmap_pinit_type* was added to initialize a *pmap* based on its type. In particular the pmap type is used to ensure that the kernel address space is not mapped into nested page tables.

```
int
pmap_pinit_type(pmap_t pmap, enum pmap_type type, int flags)
{
    pmap->pm_type = type;
    if (type == PT_EPT) {
        /* Initialize extended page tables */
    } else {
        /* Initialize x86_64 page tables */
    }
}

int
pmap_pinit(pmap_t pmap)
{
    return pmap_pinit_type(pmap, PT_X86, flags);
}
```

Finally the EPT pmap is created as follows.

```
int
ept_pinit(pmap_t pmap)
{
    return pmap_pinit_type(pmap, PT_EPT, flags);
}

struct vmspace *
ept_vmspace_alloc(vm_offset min, vm_offset max)
{
    return vmspace_alloc(min, max, ept_pinit);
}
```

6.2 EPT page-table-entries

Section 4.1 highlighted the differences between EPT PTEs and x86_64 PTEs. The pmap code was written to support the x86_64 page tables and used preprocessor macros to represent bit fields in the PTE.

```
#define PG_M    0x040   /* Dirty bit */
```

This would not work for nested page tables because the dirty flag is represented by bit 9 in the EPT PTE.

The bitmask is now computed at runtime depending on the pmap type.

```
#undef  PG_M
#define X86_PG_M  0x040
#define EPT_PG_M  0x200
pt_entry_t
pmap_modified_bit(pmap_t pmap)
{
    switch (pmap->pm_type) {
    case PT_X86:
        return (X86_PG_M);
    case PT_EPT:
        return (EPT_PG_M);
    }
}
```

Note that **PG_M** is now undefined to force compilation errors if used inadvertently. Functions that used **PG_M** were modified as follows:

```
void
some_pmap_func(pmap_t pmap)
{
    pt_entry_t PG_M = pmap_modified_bit(pmap);
    /* Rest of the function does not change */
}
```

The same technique was used for all fields in the EPT PTE that are different from the x86_64 PTE with the exception of **PG_U**. Section 6.3 discusses the special treatment given to **PG_U**.

6.3 EPT execute permission

bhyve has no visibility into how the guest uses its address space and therefore needs to map all guest memory with execute permission. An EPT mapping is executable if the **EPT_PG_EXECUTE** field at bit 2 is set in the PTE.

PG_U in the x86_64 PTE represents whether the mapping can be accessed in user-mode. **PG_U** is at bit 2 in the x86_64 PTE. The pmap sets **PG_U** if the address mapped by the PTE is in the range [0, VM_MAXUSER_-ADDRESS).

The guest address space is in the same numerical range as the user address space i.e., both address spaces start at 0 and grow upwards [3]. From pmap's perspective, mappings in the guest address space are considered user mappings and **PG_U** is set. However, bit 2 is interpreted as **EPT_PG_EXECUTE** in the EPT context. This has the desired effect of mapping guest memory with execute permission.

Note that the guest still retains the ability to make its mappings not executable by setting the **PG_NX** bit in its PTE.

6.4 EPT capabilities

The original pmap implementation assumed MMU support for 2MB superpages and A/D bits in the PTE. However these features are optional in an EPT implementation.

The *pm_flags* field was added to the *pmap* to record capabilities of the EPT implementation.

```
#define PMAP_PDE_SUPERPAGE      (1 << 0)
#define PMAP_EMULATE_AD_BITS    (1 << 1)
#define PMAP_SUPPORTS_EXEC_ONLY (1 << 2)
struct pmap {
    ...
    int    pm_flags;
}
```

A **PT_X86** pmap sets *pm_flags* to **PMAP_PDE_-SUPERPAGE** unconditionally. A **PT_EPT** pmap sets *pm_flags* based on EPT capabilities advertised by the processor in a model specific register.

The pmap already had code to disable superpage promotion globally and it was trivial to extend it to check for **PMAP_PDE_SUPERPAGE** in *pm_flags*.

6.5 EPT A/D bit emulation

The x86_64 page tables keep track of whether a mapping has been accessed or modified using the **PG_A** and **PG_M** bits in the PTE.

[3]VM_MAXUSER_ADDRESS implies an upper limit of 128TB on guest physical memory

The VM subsystem uses the accessed bit to maintain the activity count for the page. The dirty bit is used to determine whether the page needs to be committed to stable storage. It is important to faithfully emulate the A/D bits in EPT implementations that don't support them[4].

A straightforward approach would be to assign unused bits in the EPT PTE to represent the A/D bits. Dirty bit emulation was done by making the mapping read-only and setting the emulated **PG_M** bit on a write fault. Accessed bit emulation was done by removing the mapping and setting the emulated **PG_A** bit on a read fault.

Accessed bit emulation required the mapping to be entirely removed from the page tables with it being reinstated through *vm_fault()*. Dirty bit emulation required differentiating between true-readonly mappings versus pseudo-readonly mappings used to trigger write faults. The code to implement this scheme required extensive modifications to the pmap subsystem [8].

A simpler solution is to interpret the relevant bits in the EPT PTE as follows [9]: **PG_V** and **PG_RW** are now assigned to unused bits in the EPT PTE. On the other hand **PG_A** maps to **EPT_PG_READ** and **PG_M** maps to **EPT_PG_WRITE** which are interpreted by the MMU as permission bits.

	PTE bit	Interpreted by
PG_V	52	A/D emulation handler
PG_RW	53	A/D emulation handler
PG_A	0	MMU as **EPT_PG_READ**
PG_M	1	MMU as **EPT_PG_WRITE**

Clearing the accessed bit removes read permission to the page in hardware. Similarly, clearing the modified bit removes write permission to the page in hardware. In both cases the rest of the PTE remains intact. Thus, the A/D bit emulation handler can inspect **PG_V** and **PG_RW** in the PTE and handle the fault accordingly.

The A/D bit emulation handler can resolve the following types of faults:

- Read fault on 4KB and 2MB mappings
- Write fault on 4KB mappings

The handler will attempt to promote a 4KB mapping to a 2MB mapping. It does not handle write faults on 2MB mappings because the pmap enforces that if a superpage is writeable then its **PG_M** bit must also be set [10].

[4]Hardware support for A/D bits in EPT first appeared in the Haswell microarchitecture

6.5.1 EPT PTE restrictions:

There is an additional issue with clearing the emulated **PG_A**. Recall that clearing the emulated **PG_A** actually clears **EPT_PG_READ** and makes the mapping not readable.

The MMU requires that if the PTE is not readable then:

- it cannot be writeable
- it cannot be executable unless the MMU supports execute-only mappings

These restrictions cause pessimistic side-effects when the emulated **PG_A** is cleared. Writeable mappings will be removed entirely after superpage demotion if appropriate. Executable mappings suffer the same fate unless execute-only mappings are allowed.

6.6 EPT TLB invalidation

The *Translation Lookaside Buffer (TLB)* is used to cache frequently used address translations. The pmap subsystem is responsible for invalidating the TLB when mappings in the page tables are modified or removed.

To facilitate this a new field was added to *pmap* called *pm_eptgen*. This field is incremented for every TLB invalidation request. A copy of the generation number is also cached in the virtual machine context as *eptgen*. Just prior to entering the guest, *eptgen* is compared to *pm_eptgen*, and if they are not equal the EPT mappings are invalidated from the TLB.

The *pm_active* bitmap is used to track the cpus on which the guest address space is active. The bit corresponding to the physical cpu is set by bhyve on a VM-entry and cleared on a VM-exit. If the *pm_active* field indicates that the nested pmap is in use on other cpus, an *Inter-Processor Interrupt (IPI)* is issued to those cpus. The IPI will trigger a VM-exit and the next VM-entry will invalidate the TLB as previously described.

```
struct pmap {
    ...
    long pm_eptgen; /* EPT pmap generation */
};

struct vmx {
    ...
    long eptgen[MAXCPU]; /* cached pm_eptgen */
};
```

7 bhyve modifications

bhyve has always relied on nested page tables to assign memory to a guest. Prior to this work, guest memory was allocated when the virtual machine was created and released when it was destroyed. Guest memory could be accessed at all times without triggering a fault.

Representing guest memory as a *vm_object* meant that guest memory pages could be paged out or mapped read-only by the VM subsystem. This required bhyve to handle nested page faults. Additionally guest memory could be accessed only after ensuring that the underlying *vm_page* was resident.

7.1 Guest memory segments

A guest memory segment corresponds to a *vm_map_entry* backed by a *vm_object* of type **OBJT_DEFAULT** as depicted in Figure 7. Each memory segment is backed by zero-filled, anonymous memory that is allocated on-demand and can be paged out.

7.2 EPT-violation VM-exit

If the translation for a guest physical address is not present or has insufficient privileges then it triggers an EPT-violation VM-exit. The VMCS provides collateral information such as the GPA and the access type (e.g., read or write) associated with the EPT-violation.

If the GPA is contained within the virtual machine's memory segments then the VM-exit is a *nested page fault*, otherwise it is an *instruction emulation fault*.

7.2.1 Nested page fault

The nested page fault handler first calls into the pmap to do A/D bit emulation. If the fault was not triggered by A/D bit emulation, it is resolved by *vm_fault()* in the context of the guest vmspace.

Event re-injection A hypervisor can inject events (e.g., interrupts) into the guest using a VM-entry control in the VMCS. It is possible that the MMU could encounter a nested page fault when it is trying to inject the event. For example, the guest physical page containing the interrupt descriptor table (IDT) might be swapped out.

The hypervisor needs to recognize that a nested page fault occurred during event injection and re-inject the event on the subsequent VM entry.

It is now trivial to verify correct behavior of bhyve in this scenario by calling *pmap_remove()* to invalidate all guest physical mappings from the EPT.

7.2.2 Instruction emulation fault

An instruction emulation fault is triggered when the guest accesses a virtual MMIO device such as the local APIC. To handle this type of fault bhyve has to fetch the instruction that triggered the fault before it can be emulated. Fetching the instruction requires walking the guest's page tables. Thus bhyve needs to be able to access guest memory without triggering a page fault in kernel mode.

This requirement was satisfied by using an existing VM function: *vm_fault_quick_hold_pages()*. This function returns the *vm_page* associated with the GPA and also prevents the *vm_page* from being freed by the page daemon. *vm_gpa_hold()* and *vm_gpa_release()* in bhyve are the convenience wrappers on top of this.

vm_fault_hold() and superpages The original implementation of *vm_gpa_hold()* called *vm_fault_hold()*.

vm_fault_hold() resolved the GPA to a *vm_page* and called *pmap_enter()* to install it in the page tables. If there was already a superpage mapping for the GPA then it would get demoted, the *new* mapping would be installed and then get promoted *immediately*. This resulted in an inordinate number of superpage promotions and demotions.

7.3 PCI passthrough

bhyve supports PCI passthrough so a guest can directly access a physical PCI device. There were two memory-related issues that needed to be addressed with PCI passthrough.

7.3.1 MMIO BARs

Most PCI devices implement a set of registers to control operation and monitor status. These registers are mapped into MMIO space by programming the device's Base Address Register (BAR). The PCI device only responds to accesses that fall within the address range programmed in its BAR(s).

For the guest to get direct access to a PCI device the physical BAR has to be mapped into the guest's address space. This mapping is represented by a memory segment that is backed by a *vm_object* of type **OBJT_SG**. These mappings are *unmanaged* and they do not get paged out or promoted to superpages.

7.3.2 Direct memory access

A PCI device has the ability to access system memory independently of the processor. This is commonly referred to as Direct Memory Access (DMA). A PCI passthrough device is assigned to a guest and is programmed with guest physical addresses.

This implies that bhyve needs to install the GPA to HPA translation not just in the EPT but also in the I/O Memory Management Unit (IOMMU).

Additionally bhyve needs to ensure that the memory backing the guest address space is never paged out because the current generation of platforms cannot handle I/O page faults. This is implemented by calling *vm_map_wire()* on all memory segments.

7.4 Tunables, sysctls and counters

The following tunables can be used to influence the EPT features used by bhyve:

- hw.vmm.ept.use_superpages: 0 disables superpages
- hw.vmm.ept.use_hw_ad_bits: 0 forces A/D bit emulation
- hw.vmm.ept.use_exec_only: 0 disables execute-only mappings

The following sysctls provide nested pmap information:

- hw.vmm.ipinum: IPI vector used to trigger EPT TLB invalidation
- hw.vmm.ept_pmap_flags: *pm_flags* field in the pmap
- vm.pmap.num_dirty_emulations: count of dirty bit emulations
- vm.pmap.num_accessed_emulations: count of accessed bit emulations
- vm.pmap.num_superpage_accessed_emulations: count of accessed bit emulations for superpages
- vm.pmap.ad_emulation_superpage_promotions: superpage promotions attempted by the A/D bit emulation handler

The following bhyvectl counters[5] are available per vpcu:

- Number of nested pages faults
- Number of instruction emulation faults

8 Performance

The experiments described in this section were performed on a system with 32GB RAM and a Xeon E3-1220 v3 CPU at 3.10GHz. The host and the guest were both running FreeBSD/amd64 10.0-RELEASE.

[5] /usr/sbin/bhyvectl –get-stats

8.1 Nested paging overhead

In this experiment the guest was assigned 2 vcpus and 8GB memory. The vcpus were pinned to minimize scheduling artifacts. The host memory was overprovisioned to eliminate paging artifacts.

The user, system and wall-clock times for *make -j4 buildworld* in the guest were measured. The results are summarized in Table 2.

Guest memory	User	System	Wall-clock
Wired	3696	389	2207
Not wired	3704	409	2225
Not wired, A/D bit emulation	3784	432	2276

Table 2: Guest buildworld times in seconds

The buildworld time with guest memory wired established the baseline of 2207 seconds (i.e., nested paging disabled).

The buildworld took 18 seconds longer when guest memory was not wired and an additional 51 seconds with A/D bits emulated in software.

8.2 GUPS

Giga-updates per second (GUPS) is a measurement of how frequently a computer can issue updates to randomly generated memory locations [11].

In this experiment the guest was assigned 1 vcpu and 24GB memory. GUPS was configured with a 12GB working set and CPU time was measured.

Guest superpages	Host superpages	CPU time in seconds
Disabled	Disabled	500
Disabled	Enabled	258
Enabled	Disabled	267
Enabled	Enabled	102

Table 3: Effect of superpages on GUPS CPU time

Table 3 demonstrates the benefit of transparent superpage support in nested paging.

9 Future work

Nested paging is a foundational technology that will influence the design of upcoming features in bhyve.

The *vm_page* activity count and modified state may be used in live migration to compute the order in which guest memory is migrated.

Guest memory could be backed by a file which would be useful when suspending a virtual machine to disk.

10 Acknowledgements

Alan Cox and Konstantin Belousov provided guidance on nested page table support in pmap. John Baldwin reviewed the use of scatter-gather VM objects for PCI passthrough. Peter Holm tested the nested paging patch before it was committed to FreeBSD. Leon Dang did the artwork in this paper. We wish to thank them all.

References

[1] *The FreeBSD Project* `http://www.freebsd.org`

[2] *Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization* Intel Technology Journal, Volume 10, Issue 3

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual*

[4] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*

[5] *AMD-V Nested Paging White Paper* `http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf`

[6] *Performance Evaluation of AMD RVI Hardware Assist* `http://www.vmware.com/pdf/RVI_performance.pdf`

[7] *The Design and Implementation of the FreeBSD Operating System* Marshall Kirk McKusick, George V. Neville-Neil

[8] *FreeBSD svn revision 254317*

[9] *FreeBSD svn revision 255960*

[10] *Practical, transparent operating system support for superpages* Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox `http://www.usenix.org/events/osdi02/tech/full_papers/navarro/navarro.pdf`

[11] *GUPS* `http://en.wikipedia.org/wiki/Giga-updates_per_second`

Developing CPE Routers based on NetBSD: Fifteen Years of SEIL

Masanobu SAITOH(msaitoh@netbsd.org)* Hiroki SUENAGA(hsuenaga@iij.ad.jp)†

March 2014

Abstract

Typical ISPs use customized small routers to connect their network from customer's local networks. Such routers are called CPE, Customer Premises Equipment. We, Internet Initiative Japan (IIJ), also have own CPE named 'SEIL.' SEIL is a German word of rope. SEIL ropes IIJ and its customers together.

The firmware of SEIL is a customized NetBSD. IIJ has self-manufactured the firmware for 15 years, since March 1999. We describe about some implementation and enhancement for NetBSD during SEIL's 15 years history.

1 The Target environment of our CPE

Customer Premises Equipments (CPE) are communication devices, such as Internet access gateways and routers, deployed in customer's homes and offices. There are various customers, so we need to describe the target customers and environments before entering detailed discussion.

IIJ is an ISP company and most of its customers are corporations. Typical corporations use the Internet to communicate with their partners, satellite offices, and shops such as convenience stores.

Internal communications of corporations definitely include a lot of confidential information. So our CPE must have cryptographic functionalities such as IPsec and SSL, and their accelerators as much as possible. Supporting various secure tunneling protocols are also important. Our CPE supports PPTP, L2TP, L2TPv3, IPsec, SSTP, and so on, to satisfy many different requirements of our customers.

Most corporations don't have enough IP addresses and use NAPT to connect to the Internet. They also use IP filters to ensure minimum security. Since there are a lot of persons and computers in a office, performances of NAPT and IP filters are most important requirements of our CPE.

Such complicated requirements make CPE's configurations so difficult. IIJ have put efforts to simplify configuration syntax, but there are limitations to do so. Engineers of IIJ can write configurations, but most engineers in various corporations can't do enough. Thus, IIJ has found one more important requirement, simple and easy management of a number of CPEs. The word 'management' includes some concepts, configurations, operations, and monitoring. The name SEIL was selected to show this principle of management, it is abbreviation of 'Simple and Easy Internet Life.'

Table 1 shows past CPEs which made to achieve the mentioned requirements. Each of hardware architecture has been changed by the era of network environment, but core concepts of the CPEs are not changed. In this paper, we focus on the concepts and software implementations to realize it. We discuss about our management framework at first. It is an important start point of us, and an important difference between ISP's genuine CPE and other CPEs. In technical point of view, the management framework is not a BSD specific topic. But it is important

*The NetBSD Foundation
†Internet Initiative Japan Inc.

Table 1: Hardware architecture of SEILs

WAN Interfaces	LAN Interfaces	CPU(Model)	Released
128Kbps BRI	10Mbps Ethernet	Hitachi SH2(SH7604)@20MHz	Aug 1998
1.5Mbps PRI	10Mbps Ethernet	Hitachi SH3(SH7709A)@133MHz	Dec 1999
128Kbps BRI	100Mbps Ethernet	Hitachi SH4(SH7750)@200MHz	Oct 2001
1.5Mbps PRI	100Mbps Ethernet	Hitachi SH4(SH7750)@200MHz	Oct 2001
100Mbps Ethernet	100Mbps Ethernet	Hitachi SH4(SH7750)@200MHz	Nov 2001
25Mbps ATM	100Mbps Ethernet	Hitachi SH4(SH7750)@200MHz	Oct 2002
1Gbps Ethernet	1Gbps Ethernet	Freescale PowerPC G4(MPC7445)@600MHz	Jun 2003
100Mbps Ethernet	100Mbps Ethernet	Intel XScale(IXP425)@400MHz	Dec 2003
1Gbps Ethernet USB 3G/LTE Modem	1Gbps Ethernet	Cavium Octeon(CN3010)@300MHz	Feb 2008
1Gbps Ethernet USB 3G/LTE Modem	1Gbps Ethernet	Cavium Octeon(CN3120)@500Mhz	Feb 2008
100Mbps Ethernet USB 3G/LTE Modem 128Kbps BRI	100Mbps Ethernet	Intel XScale(IXP432)@400MHz	Oct 2008
1Gbps Ethernet USB 3G/LTE Modem	1Gbps Ethernet 802.11n Wireless LAN	Marvell Kirkwood(88F6281)@1.2GHz	Feb 2013

to understand why ISPs have made own CPEs from scratch. We discuss about extensions and modifications for NetBSD at second. And we discuss some knowledge and tweaks to make our daily development work flows easy and efficient.

2 Management of SEIL

2.1 Central management

Most important motivation of self-manufacturing CPE is to manage CPEs from ISP politely. The quality of managing CPEs is one of the quality of Internet connections. Most of CPEs are designed to be managed by local network managers of a customer. Of course, we can make a template configuration for them, we can advice what to do, and so on. But IIJ thinks that the work of customer should be almost nothing. Only work is to check reports from ISP and confirm there is no problem.

We create a management framework to achieve it. The framework is named SMF, SEIL Management Framework(Figure 1). It was released in 2003. The framework has the following behaviors:

1. Zero Configuration. Just power the CPE on, that's all.

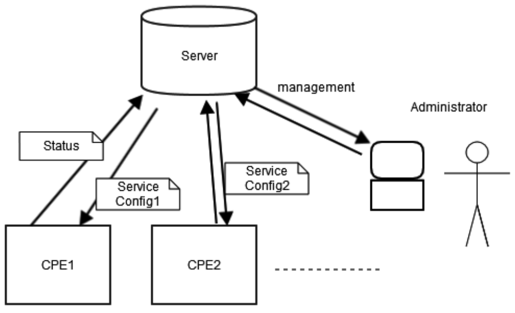


Figure 1: The SMF system

2. Watch the running status, logs of CPE by resources in ISP.

A lot of our customers use this system to built and manage complex networks they designed.

The SMF consists of server side system and CPE side system. IIJ uses NetBSD to create the intelligent CPE side system. The system is named 'recipe framework', and is developed using C language, libraries and scripting languages. We mainly have used C languages for many years. In 2013, we began to use mruby[1], a variant of Ruby scripting language, to control NetBSD based CPE itself. It's little difficult for C language to avoid buffer overflows. Using scripting language can avoid the problem, so we use it for performance independent part.

2.2 Manageability of UNIX like OS based CPE

For end users, this is not free UNIX like systems but CPE. So the following things are important:

easy to understand
: Easily and uniformly understandable without any knowledge of based OS. Easy to understand what happened. Easy to understand it's problem or not. Easy to understand what is a problem.

stability
: Not do panic. Strong against attack from others.

automation
: Automatically change setting if it can. Usually, some changes are done by editing files under /etc on UNIX like OS, and some functions don't consider changes of other functions. If a change is deterministic in system wide, it should be done automatically.

3 Development of SEIL

3.1 Extending device drivers

CPE has a number of network devices that is not common in desktop operating systems. CPE also has a naming scheme of the devices that is different from UNIX culture. IIJ has added some new H/W device drivers and special *if_net.if_xname* handling code.

Some CPE support networking port other than Ethernet. For example, SEIL supports ISDN BRI(IPAC-X on a simple device BUS), 3G MODEM(emulates USB serial port), LTE MDOEM(emulates USB Ethernet device). Most peoples weren't interested in IPAC-X, so we wrote the driver from scratch. Most people want the drivers for 3G/LTE modems, but there are no specification document. IIJ haven't had enough document in the fact, but we have tried to write the device driver and hold down the ugly BUGs of the 3G/LTE modems. The modems are managed by userland daemon 'connmgrd'. The daemon manipulates various type of P2P connections such as ISDN, 3G modem, L2TP, and so on. 802.11 wireless networking device that supports AP-mode is also a topic on CPE. NetBSD has its own 802.11 stack, but IIJ has ported vendor genuine, but buggy, 802.11 wireless support to NetBSD.

IIJ also has modified basic functionalities of NetBSD's network device. We can change the device name via ifconfig. Because port name of the CPE such as lan1, lan2, are far different from BSD's device name such as *wm*0, *em*0, *bge*0, and so on. Of course, we can translate CPE's configuration name to NetBSD device name. If we think about logging, using existing daemons, to change the device name is most cost effective way.

There are Ethernet switching devices in CPE. NetBSD has no framework to manage Ethernet switching functions such as Port based VLAN, per-port link status detection, learning table separation. IIJ wrote simple framework to do this and configuration command named *swconfig* [2].

We also change queuing strategy of network devices to work with link status detection. For example, to queue packets to link-downed network device wastes mbuf resources though such old packets are useless. And, it's sometime dangerous because some old packet might cause the network trouble. So our implementation drops the packet as soon as possible if there were some problems on link state or protocol state.

There are some pseudo networking device implemented by IIJ. For example, IPsec tunneling device[2], paravirtualized Ethernet driver for Hyper-V. FreeBSD also has a Hyper-V driver. There is no functional difference between FreeBSD's one and IIJ's one. The reason why we implemented a driver is simple, there was no FreeBSD driver when IIJ needed it. These are such duplicated implementation, and IIJ's implementation is not so special, but some of these can be useful.

3.2 Extending IP Networking stack

The IP networking stack is the most important part of CPEs. We need both of the routing speed and additional functionality. CPE is placed on a border between the ISP and the customer's LAN. CPE doesn't require very high performance such as 10G Ethernet, but require to absorb characteristics of each customer and to maximize the benefits of the ISP services.

Several years ago, IIJ implemented own IP filter and NAT/NAPT functions named *iipf* and *iipfnat*. NetBSD had an IP filter implementation *ipf*, however it didn't satisfy our requirements. *pf* and *npf* wasn't born yet. *iipf* had implemented some ideas to improve throughput. It will be described in another paper[2].

Our IPsec stack also has a caching layer on Security Policy Database (SPD) and Security Association Database (SAD). IPsec tunneling is also important for VPN; many customers prefer Route-based VPN to Policy-based VPN. This topic will be described in another paper[2].

A CPE typically uses a very cheap CPU, thus cryptographic accelerators are very important components. IIJ has done many efforts to use the accelerators effectively, and implemented a framework to use the accelerators. Using accelerators in a C function (i.e., single kernel context) was possible, but the resulting performance was very slow. So IIJ decided to separate IP stack into two parts, "before IPsec" and "after IPsec". This strategy is same as *opencrypt*(9) subsystem and *FAST_IPSEC*(4) stack that stems from OpenBSD. This framework works fine today, so IIJ has decided to use it. (Though there were some minor problems fixed by IIJ, the performance of the framework is fairly good now.)

A number of network interfaces can be causes of many problems. For a desktop machine, there are just a few network interfaces. But for a CPE, there can be many pseudo network interfaces which provide various tunnel connections. If some code uses a simple list to manage the interfaces, it becomes very slow, and consumes a large amount of memory. For example, *getifaddrs*() function uses a large memory footprint in both of the kernel and userland processes if there are a lot of interfaces. IIJ has added selector and cache layers on *getifaddrs*(). We can get a list of interfaces which link-state is up by using *getifaddrs_up*() for example.

There are some common hacks on CPEs, such as TCP-MSS clumping to avoid the Path MTU Discovery problem and Session Hijacking to create transparent proxy. IIJ has own implementations against them to satisfy requirements from customers.

3.3 Implementing New Network Protocols

IIJ has implemented some network tunneling protocols on NetBSD. PPTP and L2TP protocols are implemented in NetBSD userland. There is also an in-kernel cut-through forwarding mechanism named PIPEX. These functions are already merged to OpenBSD [1].

We has an implementation of L2TPv3. The L2TPv3 is a kind of Ethernet encapsulation and tunneling protocol described in RFC3931. The L2TPv3 network device acts as a kind of Ethernet device, and can be added to an Ethernet bridging group. Our CPE can bridge separated Ethernet segments via an L2TPv3 network device. If multiple L2TPv3 network devices are added to one bridging group, the CPE acts as a virtual Ethernet HUB.

There are also some experimental implementations of new Internet Drafts. For example, IIJ has a MAP (`draft-ietf-softwire-map-xx`) implementation. Because IIJ is an ISP company, so we are so interested in new protocols. They are not a standard protocol yet, but experimental implementations are important to standardize good protocols. The development of L2TPv3 is one of successful efforts. It had been started with a project that develops Internet Draft of L2TPv3 (`draft-ietf-l2tpext-l2tp-base-xx`).

Most CPEs support the UPnP protocol. IIJ implements UPnP services from scratch. They are completely different from *libupnp* based implementations. They are designed to cooperate with *iipfnat* and control *iipfnat* rules by the UPnP protocol.

[1]The merge has been done by `yasuoka@openbsd.org`

IIJ implements '*sockfromto*' which is a collection of extended socket API used in some operating systems. The typical functions of sockfromto are *sendfromto*() and *recvfromto*(). These functions enable to reduce a complicated usage of *bind*(). A sending socket which is bound the source port can cause a unexpected packet arrival to the sending socket. If you used *INADDR_ANY* to receiving socket, and forgot that the sending socket can receive packets, some arrival packets may lost during sending packets. *sendfromto*() can send packet with specified source port without calling *bind*().

3.4 Standing for heavy Ethernet rx interrupts

Traditional BSD system used simple *spl* mechanism. A high priority event always obstructs lower priority events. In CPE case, Ethernet rx interrupt always obstructs IP Networking stack, routing daemons, user interface, and so on. Especially, live-lock of IP Networking stack is serious problem for CPE. IIJ did some efforts to reduce such live-lock. It was serious problem for IIJ, because the live-lock can break our centralized management framework.

At first we tried to control interrupt enable/disable bits, rate control bit of Ethernet devices. What is the trigger to throttle the interrupts? We tried to add some probes that detect stall of IP Networking stack. Checking IP input queue(*ipintrq*) length, checking system load(*kern.cp_time*), checking callout timers, etc, etc..

OpenBSD tell us to control rx buffer works fine, instead of to control the interrupts directly. The idea is implemented as *MCLGETI* API of OpenBSD [2]. IIJ has ported the *MCLGETI* API to NetBSD and does some performance test. We confirm the *MCLGETI* works fine enough by various inspection.

[2]The API has other motivation that reduce memory usage on supporting jumbo frames.

4 Daily workflow

4.1 Creating new products

IIJ has created many products. Here is a list of our common works to create a new product.

- Create plain new port of NetBSD like evbxxx.
- Create customized ramdisk of the product like install kernel.
- Launch an NTP daemon and measure clock jitter/drifts, and tune a clock parameter if needed.
- Send/Receive various sizes of Ethernet frames. Frame with 802.1q VLAN tag often reveals MTU handling problem of Ethernet drivers.
- Check if dmesg buffer (*kern.msgbuf*) is cleared after software reboot. If it is cleared on reboot, fix it not to clear. The buffer is important for debugging.
- Measure primitive performances such as memory access. CPU benchmark(INT, FLOAT), cryptographic benchmark(DES, AES, DH, ..), system calls benchmark. The performance of system calls tell us performance of VM(large copyin/copyout), performance of exception handlers. We often reveal a architecture dependent problem by system call benchmarks.
- Measure IP routing performances using various major commercial measuring equipments. Such measuring equipments are also useful to apply high network load to the product. The load often reveals *spl* bugs.
- Check counters. If an value isn't visible, add it. If an counter is not incremented on some cases, fix it.
- Modify log facility, level and the message. Some logs's level are inadequate for users, so change it. Some log messages might be misunderstood by users, so modify it or remove it. Some event is important. If not log message is generated by the event, add it.

- Throttling log. Some logs might be frequently generated. If it occurred, stability of the system will be bad.

Some of the works are hard to be done by non-commercial hackers due to lack of environments, equipments, and time. If bugs are found, BSD hackers in IIJ sometimes merge the fix for the bugs.

4.2 Debugging NetBSD on small embedded hardware

On developing commercial product, debugging is very important, and we pay very much costs for it. To minimize the costs, IIJ has implemented debugging functionalities for small, embedded devices such as CPE.

IIJ has customized the syslog daemon. The log rotation mechanism on filesystem is works fine in desktop, but it is not always useful in restricted CPE. To minimize memory usage, our syslogd has internal ring buffer to remember logs, and user interface process can get logs via IPC. There are multiple ring buffer per facilities, and user can configure the size of each ring buffer. Most important facility is different for each customers.

A CPE often has no storage device, so there is no room to dump core files. So our kernel report the core information of the core files to dmesg buffer. For example, process name, program counter that causes an exception, back-trace of the userland process. The back-trace doesn't include symbol information, but is useful enough. MIPS processor has no frame pointer so the back-trace is not so trusted.

IIJ extended ddb to debugging networking stack. To print or list socket structure in kernel, To print the last function who touches a mbuf structure. Due to NetBSD-current modifies pool framework to support cache-coloring, some of these function are not working now. We need to re-design these.

Watch dog timer(wdog) is very important component in commercial product. IIJ has implemented wdog framework, and there are many point to kick the wdog. There is genuine wdog framework recent NetBSD, we are surveying it. Configuring wdog is not difficult, but kick the wdog is difficult. Especially to live-lock situation requires very sensitive design. *panic*() is also difficult situation. We do want to see the information from *panic*() and ddb stack dump, but we do avoid the infinite loop in dump. We kick the wdog during dump, but there is limit in the depth of stack. The dump can cause a exception and start new stack dump. We force cpu reboot in such situation.

4.3 Following the evolution of NetBSD

IIJ currently uses NetBSD-6.x as it's base system. Past products used NetBSD-1.x and NetBSD-3.x. Because the evolution of NetBSD is faster than life cycle of our product lines, leaping into a new NetBSD become a hard work for us. Though it's easy to generate a diffs of our implementation, it's sometimes difficult to apply the diffs to a new NetBSD.

Unfortunately the last fifteen years was so tough years, IIJ has not contributed to BSD community enough. Few BSD developers in IIJ have contributed to the community. For example, yasuoka@openbsd.org contributed and has developed PPP implementation '*npppd*' and in kernel PPP cut through mechanism '$PIPEX$' now.

As we wrote above, we have implemented some new functions and have enhanced some current functions, but a lot of have not merged yet.

5 Conclusion

IIJ has developed own CPE named 'SEIL' for long years. The name SEIL was often appeared in NetBSD developers community in the past, but IIJ didn't say much about it. This and [2] are 1st public articles about IIJ's past works and knowledges. IIJ hopes that the articles become some good lessons for BSD communities.

References

[1] mruby https://github.com/mruby/mruby

[2] Masanobu SAITOH and Hiroki SUENAGA, "Implementation and modification for CPE: filter rule optimization, IPsec interface and Ethernet switch" In proceedings of AsiaBSDCon2014, March 2014.

Deploying FreeBSD systems with Foreman and mfsBSD

Martin Matuška (mm@FreeBSD.org)

AsiaBSDCon 2014
March 15-16, 2014

Abstract

Foreman is an open source host life cycle management tool that covers the whole deployment process for production-ready physical and virtual systems in multiple datacenters. This includes creating and destroying virtual instances, BMC control of physical machines, PXE boot over TFTP and embedded Puppet configuration management. Foreman uses Ruby on Rails and is highly extensible, including the UI. Even though its development is mainly driven by Red Hat developers, Foreman is by far not just Linux. Combined with mfsBSD, a toolset for creation of diskless FreeBSD systems, Foreman is capable to deploy and manage FreeBSD systems.

1 Foreman

Foreman[6] is an open source host life cycle management tool designed to perform unattended system installations using DHCP, DNS, and PXE with TFTP. It is capable to provide automated deployment, configuration and basic monitoring of physical and virtual servers. For configuration management, embedded Puppet is part of the package and basic support for integration with Chef is provided.

Foreman is a relatively new project with its 1.0 release dating back to July 2012. The project is under intense development and the current release 1.4 integrates many new features[2].

1.1 Architecture

The core of Foreman is a Ruby on Rails engine with a RESTful API and a web-based graphical user interface. The core engine itself communicates with Compute Resources and Smart Proxies.

Compute Resources are interfaces to external providers of virtual systems. Communication with these providers is provided by the Ruby Cloud Services library fog[1].

Foreman currently (version 1.4) supports the following Compute Resources providers:

- Amazon EC2
- Google Compute Engine
- Libvirt
- OpenStack Nova
- oVirt/RHEV
- Rackspace
- VMWare

Using these providers Foreman is capable to automatically create, destroy, inspect and configure virtual servers and therefore control their complete life cycle. For physical sys-

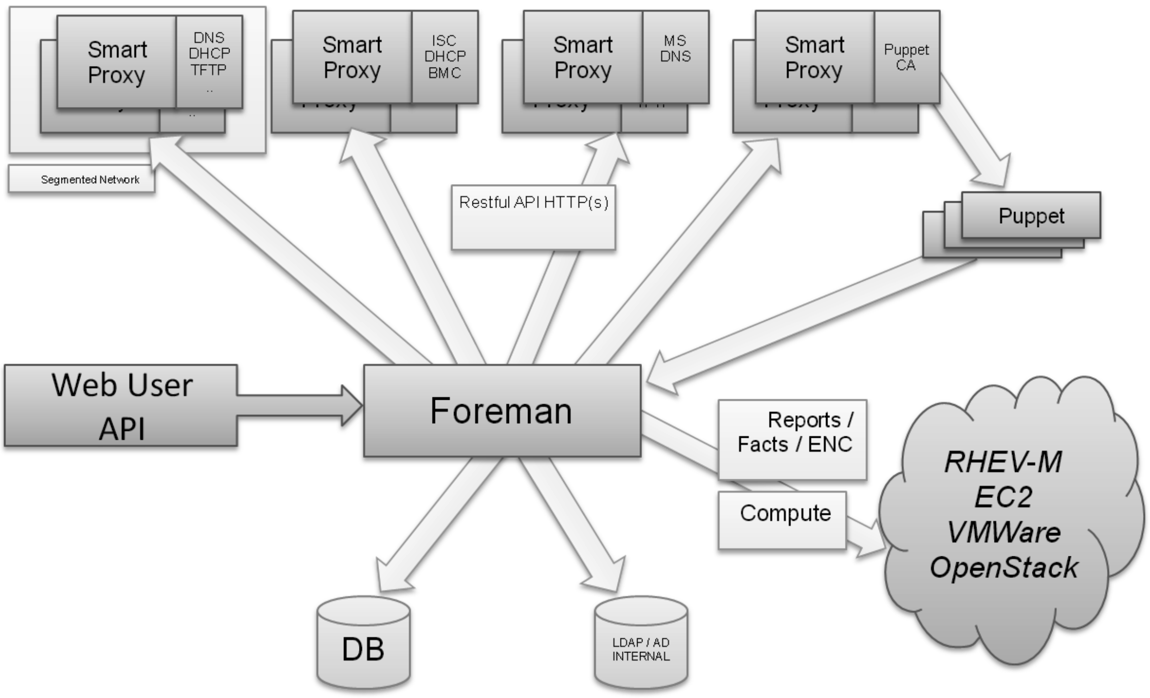


Figure 1: Foreman Architecture

tems a limited set of management tasks is supported vi BMC (e.g. power on/off if BMC is available).

A Smart Proxy is a standalone external Ruby daemon that is reachable by Foreman and manages one or more of DHCP, DNS, TFTP, BMC and Puppet Certification Authority (proxy) services. Smart proxies are Foreman's single point of contact with systems in target networks. Foreman requires at least one Smart Proxy for operation. If necessary, it is possible to operate a Smart Proxy on a Windows system for communication with the MS DHCP and/or DNS Service.

Foreman and Smart Proxy do not yet support automated installation on other than Linux host platforms. A manual installation of Smart Proxy on a MS Windows system is supported. The Linux installation is simplified using a Puppet-based auto-configuration tool. The author of this document intends to integrate the Foreman and Smart-Proxy services to the FreeBSD ports tree.

1.2 Configuration

Configuration of Foreman is web-based including a REST API to read and modify configuration entries. It provides delegated user administration with LDAP authentication support. An extensive set of access rights can be bundled in ”Roles” that are assigned to individual users. A basic support for user groups is provided, but no roles can be assigned to a user group yet.

Network data is configured in ”Subnets”, where references to (different) Smart Proxies are provided for each configured subnet.

A physical or virtual machine in Foreman is represented by an entity called ”Host”. Each host is a member of zero or exactly one ”Host Group”. Host groups can be nested and bundle configuration common to a set of hosts. The main feature of Foreman are freely configurable inheritable parameters, that can be defined and/or overridden on host, host group, operating system (version) or global levels. These parameters are used to render installation scripts (e.g. kickstart) and are accessible from Puppet, too. For virtual systems, Foreman supports creating and destroying virtual instances and access to these functions is provided via the web-interface.

1.3 Host Deployment Process

To be able to successfully deploy a system, we need a properly configured host with at least one assigned DHCP and TFTP proxy. The host is put into build mode and Foreman instructs the responsible DHCP server to PXE boot based on host's MAC address. The system boots an bootable image downloaded from a Smart Proxy's TFTP server and passes to it a required minimal set of parameters - the most important one is the URL of the Foreman server. Operating system boots and contacts Foreman via HTTPS and requests a provisioning script that is rendered from a template. The provisioning script is usually responsible for the OS installation, OS configuration and the installation of configuration tools (e.g. Puppet or Chef). The configuration tools then take over and perform post-installation tasks. The installed host reports success back to Foreman by triggering a special URL causing Foreman to consider this host "built" and deactivate it's boot via DHCP.

The result of a Puppet (or Chef) run is submitted to Foreman separately as a "report". Foreman provides web-based access to the report, indicating success or failure with an icon in the hosts screen. Access to submitted Puppet facts is provided, too. It is possible to trigger a Puppet run on a host via the web interface (using mcollective).

1.4 Extendability

The Foreman software is customizable and extensible by various plugins. A very useful plugin is the "Foreman Hooks" plugin, which enables extension of various events (e.g. host creation, deletion or update) by custom tasks. There are many other plugins available for download that extend Foreman's UI or add other features. It is possible to write individual plugins using the plugin API.

Recently introduced subproject Foreman Chef Handler[3] improves Foreman support for the Chef configuration tool by providing integration with Foreman's facts and reports features. Combined with Foreman Hooks it is possible to integrate the Chef configuration management tool with Foreman.

1.5 Development and Community

The project development is located on github[5]. As Foreman is a Red Hat community project, most of Foreman developers are Red Had employees. As Foreman is going to be one of the key components of the new Red Hat Satellite 6 product, most of the development is focused on RHEL-related issues. Nevertheless Foreman provides basic support for other operating systems.

Bug reports and feature request can be submitted on the project's Redmine issue tracker[4]. Pull requests submitted on the project github page[5] should reference these issues.

There is ongoing work on providing better FreeBSD support with key contributions from Nils Domrose[1] and the author of this article[2]. Their recent changes have been reviewed and accepted by the project.

2 Deploying FreeBSD

Since version 1.4 Foreman is able to deploy FreeBSD systems. The current requirement is a bootable image (e.g. mfsBSD) loaded by the Syslinux's memdisk feature. This memdisk image needs to be configured to download

[1]Nils Domrose's Github page: `https://github.com/endyman`

[2]Martin Matuška's Github page: `https://github.com/mmatuska`

and process the provisioning script provided by Foreman during the startup phase. The name of the image must be: **FreeBSD-[arch]-[release]-mfs.img** and it must be located in the **boot/** subdirectory of the TFTP root (provided by a Smart Proxy). The provisioning script is downloaded from the default URL[3] and may be e.g. a pc-sysinstall configuration file or a simple shell script - this depends on the image. The task of this rendered script is to install FreeBSD (with or without Puppet/Chef) on the underlying physical or virtual system. Puppet/Chef in intended to post-configure the installed system.

2.1 mfsBSD Image

mfsBSD[8] is a toolset to create small-sized but full-featured mfsroot based distributions of FreeBSD that store all files in memory (MFS) and load from hard drive, usb storage device, optical media or network.

A customized mfsBSD memdisk image can be built for this purpose. The only not yet resolved drawback is the requirement of hardcoding of the Foreman URL to the mfsBSD image. In future Foreman releases this issue may be resolved e.g. via custom DHCP options.

The image can be created e.g from a FreeBSD 10 ISO image by following the standard procedures from documentation files and the mfsBSD Homepage[7]. The only difference is the requirement to add a custom startup script (e.g. /etc/rc.local) to the image. This script downloads and processes further provisioning data from Foreman.

3 Conclusion

Foreman is a relatively new tool trying to establish a market share in the field of open source system deployment and configuration management. It is still under intense development and its main focus lies on integration with RHEL Linux. Nevertheless one of Foreman's goals is to be an universal tool and non-Linux OS support contributions are welcome.

Starting with Foreman 1.4 it is possible to deploy FreeBSD systems using mfsBSD with some limitations[4]. This article author's future goal is to add Foreman and Smart Proxy server installation to the FreeBSD ports tree.

References

[1] Fog. The Ruby cloud services library homepage. `http://fog.io`.

[2] Foreman Project. Foreman 1.4 Manual. `http://theforeman.org/manuals/1.4/index.html`.

[3] Foreman Project. Foreman Chef Handler. `https://github.com/theforeman/chef-handler-foreman`.

[4] Foreman Project. Foreman Issue Tracker. `http://projects.theforeman.org/projects/foreman/issues`.

[5] Foreman Project. Foreman Project Github Page. `https://github.com/theforeman`.

[6] Foreman Project. Foreman Project Homepage. `http://www.theforeman.org`.

[7] M. Matuška. mfsBSD Homepage. `http://mfsbsd.vx.sk`.

[8] M. Matuška. mfsBSD - The Swiss Army Knife for FreeBSD system administrators. *BSD Magazine*, 4(8):16–20, August 2011.

[3] default Foreman provisioning URL: `"http://foreman/unattended/provision"`

[4] missing ability to pass parameters on boot (e.g. Foreman provisioning URL)

Implementation and Modification for CPE Routers: Filter Rule Optimization, IPsec Interface and Ethernet Switch

Masanobu SAITOH(msaitoh@netbsd.org)[*] Hiroki SUENAGA(hsuenaga@iij.ad.jp)[†]

March 2014

Abstract

Internet Initiative Japan Inc. (IIJ) has developed its own Customer Premises Equipment (CPE), called *SEIL* , for 15 years. The firmware of *SEIL* is based on NetBSD and IIJ has modified NetBSD to optimize for the use as a CPE.

A CPE is one of special use cases, so we don't say all of our modifications is worth to merge. Nevertheless, we think some of them are worth to merge and there are some considerable ideas. We mainly describes about three things: filter rule optimization, IPsec interface and Ethernet switch.

1 Implementation and modification for CPE

IIJ has modified the some parts of NetBSD to improve performance and functionalities of our CPE.

Several years ago, IIJ implemented own IP filter and NAT/NAPT functions named *iipf* and *iipfnat*. NetBSD had an IP filter implementation *ipf*, however it didn't satisfy our requirements. *pf* and *npf* wasn't born yet. *iipf* had implemented some ideas to improve throughput. It has a hash-based flow-caching layer. Even if cache-miss occurs, *iipf* keeps reasonable throughput thanks to flow rules that are stored in an optimized tree.

Our IPsec stack also has a caching layer on Security Policy Database (SPD) and Security Association Database (SAD). Because NetBSD's *PF_KEY* API uses list structures for SPD and SAD, throughput will drop if there are a number of SP or SA. A CPE is often used to create VPNs, so the number of SP and SA can be very large. IPsec tunneling is also important for VPN; many customers prefer Route-based VPN to Policy-based VPN. (This topic will be described in another article.)

For small office, Ethernet switch is required. Ethernet switch chip is not expensive and it's easy to integrate into CPE. Integrating Ethernet switch into CPE is better than nothing because both router function and Ethernet switch function can be managed comprehensively.

2 Filter Rule Scan Optimization

In this section, we describe the new optimization of our packet filer.

2.1 Filter rule, state and result cache

On the implementations of general packet filter and old filter implementation on SEIL, the processing speed is proportional to the number of filter rules. If the number of the rules is 100, in the worst case, all 100 rules are checked.

To avoid this problem, the state mechanism is used. When a packet was passed, the interface, source/destination address, source/destination port and so on were saved into an entry of a hash table. And then, a hash value is calculated in each packet and the value is looked up. If the entry was found,

[*]The NetBSD Foundation

[†]Internet Initiative Japan Inc.

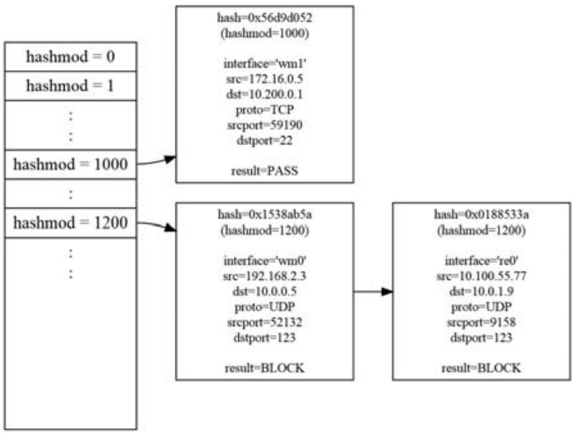


Figure 1: Filter Result Cache

the packet is passed. This mechanism is good because it doesn't scan rules if it is in a hash table. If a lot of data is processed with the same state, the hit ratio is very high. But, if a hash miss occurred a rule scan is done. It's heavy task.

In iipf, it caches a packet's information(interface, source, dest, protocol), the hash value and the result(block or pass) independently of the result. When rule is scanned, at first, a packet's hash value is calculated from the address, protocol and port number, and the value is used to check the result cache. If it exists, the result(block or pass) is returned immediately. If it doesn't exist, rule scan is done, and the packet information, hash value and the result are saved into the result cache for the next check (Figure 1). The entries are managed by LRU algorithm.

2.2 Optimizing rule scan itself

An fundamental way to speedup filtering is to improve the performance of rule scanning which is done when a cache miss happened.

Figure 2 is an example of iipf's filter rules. The top entry of the rules is evaluated first and the bottom entry is evaluated at last. The third column is an identifier string name to specify each rule. With the old implementation, the rule is evaluated like Figure 3. So, the more number of rules increased, the longer the processing speed becomes. One of optimization way is to change such evaluations like Figure 4. New implementation does such optimization when rules are set.

2.3 Implementation

One of the way to implement optimization described above is to use compiler technique. Rules are decomposed into small statements and then those statements are optimized with composer technique. It's possible but the implementation is not easy.

Our solution is splitting filter rule lists using with special condition statements to reduce the number of rules that are scanned. The merit of this way is that it's unnecessary to change the evaluation of each filter rule.

1. Make a list of conditions which are used for splitting rules into two groups. Interface, address, protocol an port are used for the conditions.

2. Select one of conditions.

3. Split filer rules into two groups by checking whether a rule matches or not. If a rule can't be identified whether it matches or not. The rule is put into both groups.

4. For each group, goto step 2 and retry.

5. Stop when the number of rules was decreased a specified limit.

Try this algorithm to Figure 2's rules. Figure 5 is the first try. pppoe0 is selected. An rule that the interface "any" matches both pppoe0 and others, so the rule is put into both groups. Figure 6 is the second try. The next condition is "protocol number is less than 17". The rule of "protocol any" belongs to both groups. Figure 7 is the third try. The next condition is "protocol number is less than 6". The rule of "protocol any" belongs to both groups. The result means that three rules are checked on the worst case. The number was decreased from 6 to 3.

```
filter add LAN        interface lan0   direction in/out                           action pass
filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
filter add DNS_PASS   interface pppoe0 direction in/out protocol udp  dstport 53  action pass
filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
filter add BLOCK_RULE interface any    direction in/out protocol any              action block
```

Figure 2: iipf's filter rules example 1

```
/* LAN */
 if (pkt.interface == "lan0")
   return PASS;

 /* PING_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.direction == in) &&
     (pkt.protocol == icmp) &&
     (pkt.icmp.type == 8))
        return PASS;

 /* ICMP_BLOCK */
 if ((pkt.interface == "pppoe0") &&
     (pkt.direction == in) &&
     (pkt.protocol == icmp))
        return BLOCK;

 /* DNS_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.protocol == udp) &&
     (pkt.udp.dstport == 53))
        return PASS;

 /* TCP_PASS */
 if ((pkt.interface == "pppoe0") &&
     (pkt.protocol == tcp) &&
        return PASS;

 /* BLOCK_RULE */
 return BLOCK;
```

Figure 3: Normal processing example of Figure 2 rules

```
if (pkt.interface == "pppoe0") {
  if (pkt.direction == in) {
    if (pkt.protocol == icmp) {
      if (pkt.icmp.type == 8) {
        return PASS;
      } else {
        return BLOCK;
      }
    } else if (pkt.protocol == udp) {
      if (pkt.udp.dstport == 53) {
        return PASS;
      }
    } else if (pkt.protocol == tcp) {
      return PASS;
    } else {
      return BLOCK;
    }
  } else {
    if (pkt.protocol == udp) {
      if (pkt.udp.dstport == 53) {
        return PASS;
      }
    } else if (pkt.protocol == tcp) {
      return PASS;
    } else {
      return BLOCK;
    }
  }
} else {
  if (pkt.interface == "lan0") {
    return PASS;
  }
  return BLOCK;
}
```

Figure 4: Optimized processing example of Figure 2 rules

```
COND_INTERFACE("pppoe0")
  filter add PING_PASS    interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
  filter add ICMP_BLOCK   interface pppoe0 direction in     protocol icmp             action block
  filter add DNS_PASS     interface pppoe0 direction in/out protocol udp  dstport 53  action pass
  filter add TCP_PASS     interface pppoe0 direction in/out protocol tcp              action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block

!COND_INTERFACE("pppoe0")
  filter add LAN          interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
```

Figure 5: Try 1. Split with pppoe0.

```
COND_INTERFACE("pppoe0")
  COND_PROTOCOL(<17)
    filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
    filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
    filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
    filter add BLOCK_RULE interface any    direction in/out protocol any              action block
  !COND_PROTOCOL(<17)
    filter add DNS_PASS   interface pppoe0 direction in/out protocol udp  dstport 53  action pass
    filter add BLOCK      interface any    direction in/out protocol any              action block
!COND_INTERFACE("pppoe0")
  filter add LAN          interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
```

Figure 6: Try 2. Split with protocol number is less than 17.

```
COND_INTERFACE(‘‘pppoe0’’)
  COND_PROTOCOL(<17)
    COND_PROTOCOL(<6)
      filter add PING_PASS  interface pppoe0 direction in     protocol icmp icmp-type 8 action pass
      filter add ICMP_BLOCK interface pppoe0 direction in     protocol icmp             action block
      filter add BLOCK      interface any    direction in/out protocol any              action block
    !COND_PROTOCOL(<6)
      filter add TCP_PASS   interface pppoe0 direction in/out protocol tcp              action pass
      filter add BLOCK      interface any    direction in/out protocol any              action block
  !COND_PROTOCOL(<17)
    filter add DNS_PASS     interface pppoe0 direction in/out protocol udp  dstport 53  action pass
    filter add BLOCK_RULE   interface any    direction in/out protocol any              action block
!COND_INTERFACE(‘‘pppoe0’’)
  filter add LAN            interface lan0   direction in/out                           action pass
  filter add BLOCK_RULE     interface any    direction in/out protocol any              action block
```

Figure 7: Try 3. Split with protocol number is less than 6.

2.4 Selection of condition value

In above example, condition values were selected a little intentionally. In real program, all values which appeared in the rules are tried and the best balanced condition is used.

Next filter rule example is Figure 8. At first step, make a list of conditions which are used for splitting rules into two groups. The list is as follows:

```
INTERFACE = "pppoe0"
SRC < 10.0.0.0
SRC < 11.0.0.0
SRC < 172.16.0.0
SRC < 172.32.0.0
SRC < 192.168.0.0
SRC < 192.169.0.0
PROTOCOL < TCP
DSTPORT < 22
DSTPORT < 24
DSTPORT < 80
DSTPORT < 443
DSTPORT < 512
DSTPORT < 514
```

And then, split rules using with the all conditions. The result is Table 1. The best balanced condition is COND_DSTPORT(<24), so the condition is selected. Then, apply the algorithm with top half of Figure 9. The next candidates are:

```
INTERFACE = pppoe0
SRC < 10.0.0.0
SRC < 11.0.0.0
SRC < 172.16.0.0
SRC < 172.32.0.0
SRC < 192.168.0.0
SRC < 192.169.0.0
PROTOCOL < TCP
DSTPORT < 22
DSTPORT < 24
```

And, the next split candidates and the results are in Figure 2 The best balanced condition is COND_SRC(<11.0.0.0), so the condition is selected. By repeating this, the final result is in Figure 10. This rules can't be split anymore. The result means that three rules are checked on the worst case. The number was decreased from 7 to 2.

2.5 Performance result

The performance result is shown in Figure 11. Test environment is:

- SEIL/B1 (Intel IXP432 400MHz, RAM 128MB)

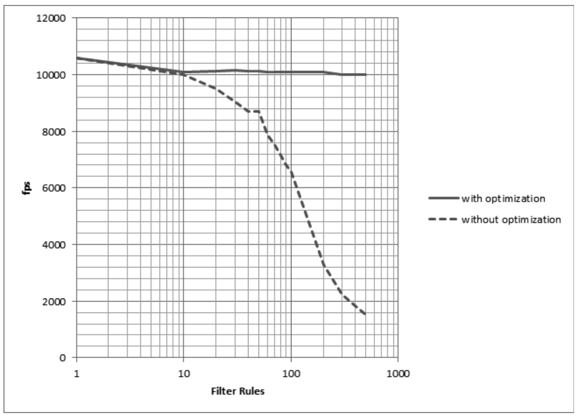


Figure 11: Performance comparison of filter rule optimization

- Packet length 512bytes
- One direction.
- Incrementing source address in the range of 10.0.0.0/8.

The result showed that the new algorithm works fine and the effect of the number of filter rule is very small.

3 Route packets to IPsec tunnel using routing table

IPsec based VPN is one of important functionalites of IIJ's CPE. Many corporations use IPsec VPN for internal communications. Some corporations have a large number of satellite offices, and redundant data center networks. Each of satellite office has redundant VPN connections to each data center network. So the CPE on a satellite network needs to select one from the redundant connections somehow.

A typical IPsec implementation uses 'Security Association Database(SAD)' to create VPN connections, and uses 'Security Policy Database(SPD)' to select one from the VPN connections. On NetBSD, the SPD is implemented as strictly ordered lists like

```
filter add RULE1 interface pppoe0 src 10.0.0.0/24    protocol any                action pass
filter add RULE2 interface pppoe0 src 172.16.0.0/12  protocol any                action pass
filter add RULE3 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
filter add RULE4 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
filter add RULE5 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
filter add RULE6 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
filter add RULEB interface any                       protocol any                action block
```

Figure 8: iipf's filter rules example 2

condition hogehoge	rules
COND_INTERFACE("pppoe0")	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_INTERFACE("pppoe0")	RULEB
COND_SRC(<10.0.0.0)	RULEB
!COND_SRC(<10.0.0.0)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<11.0.0.0)	RULE1,RULEB
!COND_SRC(<11.0.0.0)	RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<172.16.0.0)	RULE1,RULEB
!COND_SRC(<172.16.0.0)	RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<172.32.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<172.32.0.0)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<192.168.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<192.168.0.0)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_SRC(<192.169.0.0)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_SRC(<192.169.0.0)	RULEB
COND_PROTOCOL(<tcp)	RULE1,RULE2,RULEB
!COND_PROTOCOL(<tcp)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<22)	RULE1,RULE2,RULEB
!COND_DSTPORT(<22)	RULE3,RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<24)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<24)	RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<80)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<80)	RULE4,RULE5,RULE6,RULEB
COND_DSTPORT(<443)	RULE1,RULE2,RULE3,RULE4,RULEB
!COND_DSTPORT(<443)	RULE5,RULE6,RULEB
COND_DSTPORT(<512)	RULE1,RULE2,RULE3,RULE4,RULE5,RULEB
!COND_DSTPORT(<512)	RULE6,RULEB
COND_DSTPORT(<514)	RULE1,RULE2,RULE3,RULE4,RULE5,RULE6,RULEB
!COND_DSTPORT(<514)	RULEB

Table 1: First Split candidates and the results

```
COND_PROTOCOL(<24)
        filter add RULE1 interface pppoe0 src 10.0.0.0/24    protocol any                action pass
        filter add RULE2 interface pppoe0 src 172.16.0.0/12  protocol any                action pass
        filter add RULE3 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
        filter add BLOCK interface any                       protocol any action block
!COND_PROTOCOL(<24)
        filter add RULE4 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
        filter add RULE5 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
        filter add RULE6 interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
        filter add RULEB interface any                       protocol any                action block
```

Figure 9: 1st split result

condition	rules
COND_INTERFACE("pppoe0")	RULE1,RULE2,RULE3,RULEB
!COND_INTERFACE("pppoe0")	RULEB
COND_SRC(<10.0.0.0)	RULEB
!COND_SRC(<10.0.0.0)	RULE1,RULE2,RULE3,RULEB
COND_SRC(<11.0.0.0)	RULE1,RULEB
!COND_SRC(<11.0.0.0)	RULE2,RULE3,RULEB
COND_SRC(<172.16.0.0)	RULE1,RULEB
!COND_SRC(<172.16.0.0)	RULE2,RULE3,RULEB
COND_SRC(<172.32.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<172.32.0.0)	RULE3,RULEB
COND_SRC(<192.168.0.0)	RULE1,RULE2,RULEB
!COND_SRC(<192.168.0.0)	RULE3,RULEB
COND_SRC(<192.169.0.0)	RULE1,RULE2,RULE3,RULEB
!COND_SRC(<192.169.0.0)	RULEB
COND_PROTOCOL(<tcp)	RULE1,RULE2,RULEB
!COND_PROTOCOL(<tcp)	RULE3,RULEB
COND_DSTPORT(<22)	RULE1,RULE2,RULEB
!COND_DSTPORT(<22)	RULE3,RULEB
COND_DSTPORT(<24)	RULE1,RULE2,RULE3,RULEB
!COND_DSTPORT(<24)	RULEB

Table 2: Second Split candidates and the results

```
COND_PROTOCOL(<24)
   COND_SRC(<11.0.0.0)
      filter add RULE1       interface pppoe0 src 10.0.0.0/24    protocol any                 action pass
      filter add BLOCK       interface any                       protocol any action block
   !COND_SRC(<11.0.0.0)
      COND_SRC(<172.32.0.0)
         filter add RULE2    interface pppoe0 src 172.16.0.0/12  protocol any                 action pass
         filter add BLOCK    interface any                       protocol any action block
      !COND_SRC(<172.32.0.0)
         filter add RULE3    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 22-23   action pass
         filter add BLOCK    interface any                       protocol any action block
!COND_PROTOCOL(<24)
   COND_DSTPORT(<512)
      COND_DSTPORT(<443)
         filter add RULE4    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 80      action pass
         filter add BLOCK    interface any                       protocol any action block
      !COND_DSTPORT(<443)
         filter add RULE5    interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 443     action pass
         filter add BLOCK    interface any                       protocol any action block
   !COND_ DSTPORT(<512)
      filter add RULE6       interface pppoe0 src 192.168.0.0/16 protocol tcp dstport 512-513 action pass
      filter add BLOCK       interface any                       protocol any action block
```

Figure 10: final split

filter rules. Each entry of SPD describes a packet to be secured.

IIJ added some modifications to the NetBSD's implementation. We discuss about the modifications in this section. Figure 12 shows the outline of IIJ's modifications. Gray colored components in the figure are extended by IIJ.

3.1 Problem of typical IPsec implementation

We have 2 problem to use NetBSD as a CPE.

One problem is performance. The implementation of SPD is simple and secure, but we must execute $LIST_FOREACH()$ to each packet. Encryption throughput of our CPE is about 100 - 200 Mbps. If average packet length is 1000 bytes, the packet arriving rate is about 12 kpps to 25 kpps. This means *LIST_FOREACH*() will be executed 25,000 times a seconds. And if the SPD has 100 entries, *memcmp*() will be executed 2,500,000 times a seconds(2.5 MHz!). Of course, the SPD is much smaller on many workstations, but the SPD of VPN devices often have hundreds of entries. SPD grows lager due to number of offices and data centers, and due to number of net-

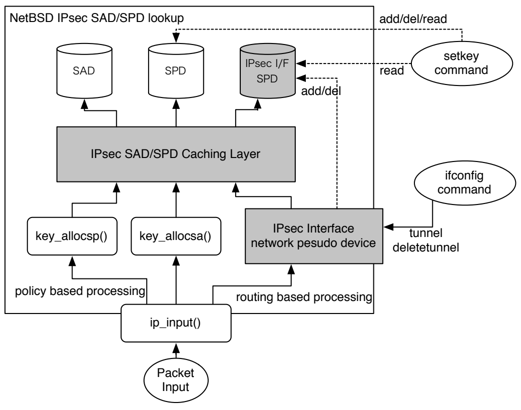


Figure 12: IPsec modifications

work segments in each of networks. It is very easy to grow the SPD.

The other problem is redundancy. SPD is a strictly ordered list, and there is no same order(priority). Each of entry just has a single actions, and there is no way to select multiple connection. It is hard to have a benefit of redundant connections. Some VPN devices can use a routing table instead of SPD. Because there are many existing redundant routing techniques, it easy to have a benefit of the redundant VPN connections. IIJ's CPE supports such routing based IPsec VPN. Here is a example loop in *netipsec/key.c*.

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
    u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        LIST_FOREACH(sp, &sptree[dir], chain) {
...
                if (sp->state == IPSEC_SPSTATE_DEAD)
                        continue;
                if (key_cmpspidx_withmask(&sp->spidx,
                    spidx))
                        goto found;
        }
...
```

3.2 Improve SAD/SPD lookup performance

IIJ implements software based caching layer to SPD and SAD. The caching code takes a packet header, hash it, then lookup the cache table. The table has a pointer to a SAD/SPD entry. If there is no entry for the packet, scan the SAD/SPD and write it to cache table. This strategy works fine for the CPE. Because the number of node in the corporation's network is much smaller than the real Internet, the flow table doesn't become so large. Our implementation uses 2048 entries for the cache table and it works fine to connect to 100 - 200 satellite networks. Of course, there are some exceptions. For example, random traffics generated by malwares are pollutes the cache table.

There are 13 API functions for SPD/SAD caching layer management, 1 initialization, 4 lookups for each of structure, and 8 invalidates.

```
void key_cache_init();

struct secpolicy *sp_cache_lookup();
struct secashead *sah_cache_lookup();
struct secasvar *sav_cache_lookup();
struct secacq *acq_cache_lookup();

void sp_cache_inval(void);
void sp_cache_inval1(struct secpolicy *);
void sah_cache_inval(void);
void sah_cache_inval1(struct secashead *)
void sav_cache_inval(void);
void sav_cache_inval1(struct secasvar *);
void acq_cache_inval(void);
void acq_cache_inval1(struct secacq *);
```

Cache lookup code is simply inserted before the *LIST_FOREACH*().

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
    u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        if (key_cache_enable > 0) {
                /* IIJ Extension: lookup cache */
                sp = sp_cache_lookup(spidx, dir);
                goto skip;
        }

        LIST_FOREACH(sp, &sptree[dir], chain) {
```

The hashing algorithm is very important component. The algorithm must be fast enough and must have enough distribution. Unfortunately, there is no specialist of mathematics in IIJ's CPE team, the algorithm should not be a best. Here is our hashing code for your interest.

```
if (src->ss_family == AF_INET) {
        u_int32_t *saddr, *daddr;
        u_int32_t sport, dport;

        saddr = (u_int32_t *)&satosin(src)->sin_addr;
        daddr = (u_int32_t *)&satosin(dst)->sin_addr;
        sport = (u_int32_t)satosin(src)->sin_port;
        dport = (u_int32_t)satosin(dst)->sin_port;

        hash = *saddr ^ bswap32(*daddr) ^
               (sport << 16) ^ dport;
        hash = (hash >> 16) ^ hash;
        hash = (hash >> 4) ^ hash;
}
else if (src->ss_family == AF_INET6) {
        struct in6_addr *saddr, *daddr;
        u_int32_t sport, dport;
        u_int32_t hash128[4];

        saddr = &satosin6(src)->sin6_addr;
        daddr = &satosin6(dst)->sin6_addr;
        sport = (u_int32_t)satosin6(src)->sin6_port;
        dport = (u_int32_t)satosin6(dst)->sin6_port;

        /* stage 1 */
        hash128[0] =
            saddr->s6_addr32[0] ^ daddr->s6_addr32[3];
        hash128[1] =
            saddr->s6_addr32[1] ^ daddr->s6_addr32[2];
        hash128[2] =
            saddr->s6_addr32[2] ^ daddr->s6_addr32[1];
        hash128[3] =
            saddr->s6_addr32[3] ^ daddr->s6_addr32[0];

        /* stage 2 */
        hash128[0] = hash128[0] ^ hash128[3];
        hash128[1] = hash128[1] ^ hash128[2];

        /* stage 3 */
        hash = hash128[0] ^ hash128[1] ^
               (sport << 16) ^ dport;
}
```

3.3 VPN tunnel network device

IIJ implements a VPN tunnel network device named IPsec Interface. The device has BSD name *ipsec*0, *ipsec*1, ..., *ipsecN*. It is a kind of pseudo network device like a IP-IP tunneling device like gif, gre. If a packet is routed into the IPsec interface, the kernel apply IPsec tunnel encryption. There is no need to write a SPD.

The device is controlled by *ifconfig* command as same as *gif* device. When tunnel address is configured, the device create Security Policies automatically. The Security Policies are registered to a SPD other than NetBSD's genuine SPD. i.e. IIJ's kernel has 2 separated SPDs. SP lookup code always looks for genuine SPD 1st, then the IPsec Interface's SPD 2nd. The generated entry is fully compatible with Security Policies of transport mode IPsec to secure tunnel end-point address. Thus, there is no modification for crypto subsystem. And IPsec Interfaces can share the NetBSD's genuine SAD. The code snippet is here. A *LIST_FOREACH* is just added.

```
struct secpolicy *
key_allocsp(const struct secpolicyindex *spidx,
    u_int dir, const char* where, int tag)
{
        struct secpolicy *sp;
        int s;
...
        LIST_FOREACH(sp, &sptree[dir], chain) {
...
        }
#if NIPSECIF > 0
        LIST_FOREACH(sp, &ipsecif_sptree[dir], chain) {
...
        }
#endif
```

setkey command can add or delete entries in genuine SPD but it cannot add or delete entries in IPsec Interface's SPD. But the *setkey* command can read the entries in IPsec Interface's SPD. An IKE server can also read the entries in IPsec Interface's SPD, and create a SA for a entry in IPsec Interface's SPD. We don't need to modify IKE server and most of management services. In kernel IPsec stack also read a entry in IPsec Interface's SPD via APIs in key.c, so we don't need to modify existing IPsec stack. We just modified DB lookup code in key.c. Here is simple example of SPD behavior.

Example 1, configure the interface. IPv6 traffic is dropped by default. Lack of awareness of IPv6 is security risk.

```
# setkey -DP
No SPD entries.
# ifconfig ipsec0
ipsec0: flags=8010<POINTOPOINT,MULTICAST>
        inet6 fe80::2e0:4dff:fe30:28%ipsec0
          ->  prefixlen 64 scopeid 0xf
# ifconfig ipsec0 tunnel 203.0.113.1 203.0.113.2
# ifconfig ipsec0 inet 192.0.2.1
# ifconfig ipsec0
ipsec0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST>
        tunnel inet 203.0.113.1 --> 203.0.113.2
        inet 192.0.2.1 ->  netmask 0xffffff00
        inet6 fe80::2e0:4dff:fe30:28%ipsec0
          ->  prefixlen 64 scopeid 0xf
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in discard
        spid=36 seq=3 pid=1807
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16402
        spid=34 seq=2 pid=1807
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out discard
        spid=35 seq=1 pid=1807
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16401
        spid=33 seq=0 pid=1807
        refcnt=1
#
```

Example 2, setkey cannot delete SP entries for IPsec Interfaces.

```
# setkey -FP
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16410
        spid=44 seq=3 pid=2229
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16408
        spid=42 seq=2 pid=2229
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16409
        spid=43 seq=1 pid=2229
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16407
        spid=41 seq=0 pid=2229
        refcnt=1
```

Example 3, accept IPv6 traffic. It is controlled by link2 option.

```
# ifconfig ipsec0 link2
# setkey -DP
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16406
        spid=40 seq=3 pid=13654
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 4(ipv4)
        in ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16404
        spid=38 seq=2 pid=13654
        refcnt=1
203.0.113.2[any] 203.0.113.1[any] 41(ipv6)
        out ipsec
        esp/transport/203.0.113.2-203.0.113.1/unique#16405
        spid=39 seq=1 pid=13654
        refcnt=1
203.0.113.1[any] 203.0.113.2[any] 4(ipv4)
        out ipsec
        esp/transport/203.0.113.1-203.0.113.2/unique#16403
        spid=37 seq=0 pid=13654
        refcnt=1
#
```

Example 4, unconfigure tunnel.

```
# ifconfig ipsec0 deletetunnel
# setkey -DP
No SPD entries.
#
```

Once IPsec Interface is configured, and IKE server creates SAs for it, we can use the interface as common P2P network interface like gif, ppp, pppoe, and so on. We can manage VPN traffic by RIP, OSPF, floating stack routes, other common routing techniques. We can also use IP Filter on the IPsec Interface. It very easy to have a benefit of redundant VPN connections.

4 Ethernet Switch Framework

One of previous product named SEIL/X2 has an Ethernet switch. The function is almost the same as SA-W1's Ethernet switch chip, but the old code had not enough functions and it's difficult to reuse. At that time, FreeBSD has Ethernet switch function but it's little hardware dependent, so we designed new Ethernet switch framework from scratch.

4.1 Design

The main concept on design is separating code into Ethernet switch common function part and hardware

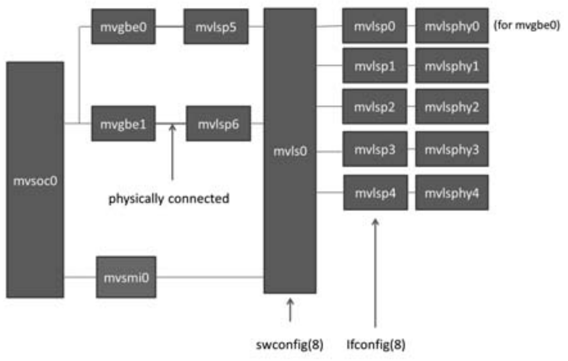
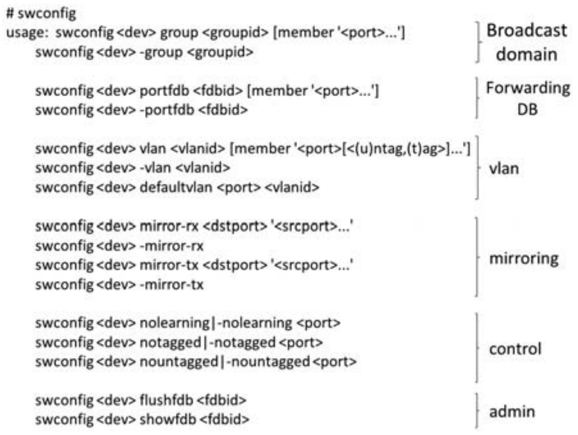


Figure 13: block diagram of SA-W1



```
# swconfig
usage:  swconfig <dev> group <groupid> [member '<port>...']               Broadcast
        swconfig <dev> -group <groupid>                                    domain

        swconfig <dev> portfdb <fdbid> [member '<port>...']               Forwarding
        swconfig <dev> -portfdb <fdbid>                                       DB

        swconfig <dev> vlan <vlanid> [member '<port>[<(u)ntag,(t)ag>]...']
        swconfig <dev> -vlan <vlanid>                                      vlan
        swconfig <dev> defaultvlan <port> <vlanid>

        swconfig <dev> mirror-rx <dstport> '<srcport>...'
        swconfig <dev> -mirror-rx
        swconfig <dev> mirror-tx <dstport> '<srcport>...'                  mirroring
        swconfig <dev> -mirror-tx

        swconfig <dev> nolearning|-nolearning <port>
        swconfig <dev> notagged|-notagged <port>                           control
        swconfig <dev> nountagged|-nountagged <port>

        swconfig <dev> flushfdb <fdbid>                                    admin
        swconfig <dev> showfdb <fdbid>
```

Figure 14: swconfig(4)

specific part. For example, Ethernet function is separated into if_ethersubr.c and if_xxx.c (e.g. if_bge.c). Like that, Ethernet Switch framework is separated into Ethernet switch common part and hardware specific part. Former is if_etherswsubr.c and latter is mvls.c for SA-W1.

To control Ethernet switch function, we made new command swconfig(8). The main purpose of this command was to hide hardware dependent part. The current function of swconfig is similar to brconfig(4). We think swconfig(8) and brconfig(8) can be integrated into one command.

The driver is separated into two parts. One is driver for controlling switch function(mvls(4)) and another is driver for each port (mvlsp(4)). The ifnet structure is used for those drivers. To control each PHY with mii(4) layer, mvlsply(4) was made and is attached from mvlsp(4) via mii_attach(). Figure 13 is the block diagram of SA-W1's Ethernet switch part.

With this design, ifconfig, netstat, snmp can be used without any modification. the media status and each port's counter can be checked with those programs.

See Figure 14 for the detail of the function of swconfig(4). swconfig(4) calls MI ioctls to control switch functions.

4.2 Current problem

Currently we have some problem. First, though this is not specific to Ethernet switch, there is no best way to know what mechanism is used between Ethernet MAC and switch (or MII PHY), e.g. GMII, RGMII, I2C or something else. So we sometimes have to write it by hard-coding.

Another problem is the relation between the framework and vlan(4). It's little difficult to cooperate with each other.

4.3 Future work

We implemented this framework only for Marvell 88E6171R. We are planning to port this framework to other chips to check whether our design is appropriate or not.

5 Conclusion

Some of implementation can be merged to NetBSD and other *BSD's. For filter rule optimization, the idea can be useful for some other filter implementations.

An Overview of Security in the FreeBSD Kernel

Brought to you by

Dr. Marshall Kirk McKusick

AsiaBSD Conference
15th–16th March 2014

Tokyo University of Science
Tokyo, Japan

Security Mindset

Security is part of the design, not added later

From its beginning UNIX identified users and used those identities for:

- access control to files
- manipulation control of processes
- access control to devices
- limited privilege expansion using *setuid* ( ) and *setgid* ( )

Over time these basic controls have been refined though still remain intact more than 40 years later

Trusted Computing Base

Set of things that have to be secure for system to be secure

- Kernel
- Boot scripts
- Core utilities (shell, login, ifconfig, etc)
- Libraries used by core utilities

Solid crypto support

- OpenSSH
- OpenSSL
- IPSEC
- GBDE
- GELI
- Hardware crypto
- **/dev/random**

Overview

Immutable and Append-only Flags

- Tamperproof critical files and logs

Jails

- Lightweight FreeBSD virtual machine

Random numbers (**/dev/random**)

- Needed for strong crypto

Access control lists (ACL)

- Discretionary access control to files and directories

Mandatory access control (MAC)

- Systemwide controlled information flow between files and programs

Privilege

- Subdivision of root privileges

Auditing

- Accountability and intrusion detection

Capsicum

- Sandboxing of process rights

Immutable and Append-only Flags

- Immutable file may not be changed, moved, or deleted
- Append-only file is immutable except that it may be appended
- User append-only and immutable flags may be toggled by owner or root
- Root append-only and immutable flags may not be cleared when system is secure
- System secure levels:

-1	always insecure (must be compiled into kernel)
0	insecure mode (normally single user)
1	secure mode (normally multiuser)
2	very secure mode (at system admin discretion)

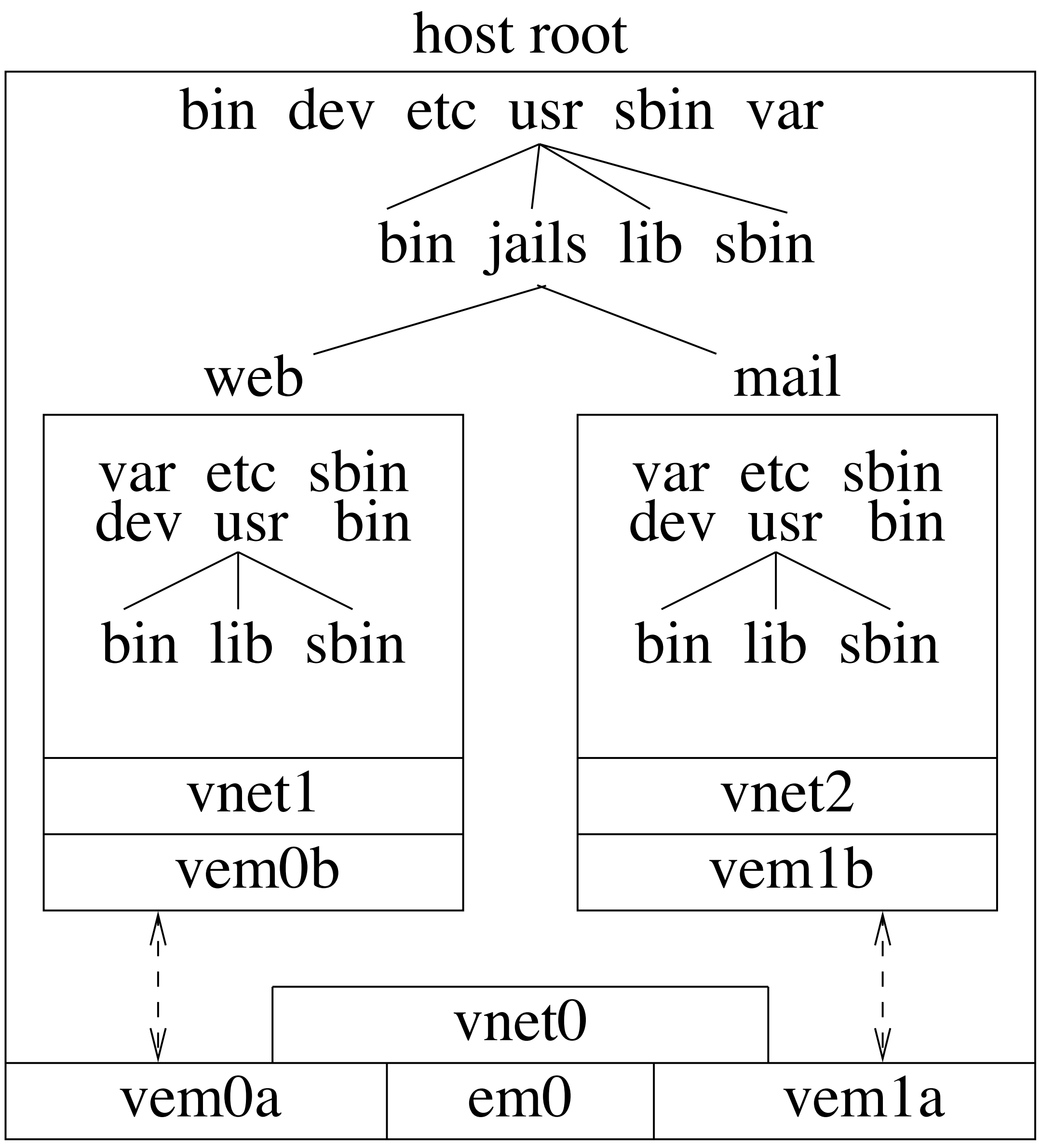
- Secure mode prevents writing /dev/kmem, /dev/mem, and mounted disks
- Very secure mode additionally prevents writing any disk or rebooting

Immutable Limitations

- Immutable files can only be updated when system is single-user
- Append-only files can only be rotated when system is single-user
- Direct hardware access is restricted
- All startup activities must be protected
  - Startup scripts and their containing directories
  - All binaries executed during startup
  - All libraries used during startup
  - Configuration files used during startup

Jails

Create a group of processes with their own root-administered environment



Jail Rules

Permitted

- running or signalling processes within jail
- changes to files within jail
- binding ports to jail’s IP addresses
- accessing raw, divert, or routing sockets on jail’s virtual network interfaces

Not permitted

- getting information on processes outside of the jail
- changing kernel variables
- mounting or unmounting filesystems
- modifying physical network interfaces or configurations
- rebooting

Random Number Generation

Application access to random number using **/dev/random**

- introduced in FreeBSD 5.3 in 2004 by Mark Murray
- uses Yarrow, a cryptographic pseudo-random number generator (PRNG)
- Yarrow reuses existing cryptographic primitives such as cryptographic hashes and counter-mode block encryption

Yarrow operational parts

- a flexible framework for entropy acquisition from various types of sources (e.g., interrupt timing, hardware RNG)
- an entropy accumulator based on a cryptographic hash
- a reseed mechanism by which entropy is converted into keying material,
- a generation mechanism using a counter-mode encryption function (SHA256 and AES) to generate a pseudo-random sequence

Random Numbers in FreeBSD

Many CPUs implement built-in hardware random number generators using oscillator loops to generate difficult-to-predict output

- FreeBSD 5.3+ use the VIA generator directly
- FreeBSD 9.2+ use Intel’s rdrand directly
- FreeBSD 10.0+ incorporate Intel’s rdrand through Yarrow since hard to tell if rdrand is working correctly or has been trojaned by NSA, GCHQ, or anyone else.

For FreeBSD 11, Yarrow will be replaced by Fortuna which automates the estimation of how/when to use alternate entropy sources

Ongoing work for **/dev/random**

- boot-time integration
- good sources of entropy
- adaptations to Fortuna to improve performance
- meeting application needs

Access Control Lists

File permission bits

- file permission bits are three entries in the ACL itself
- permits full backward compatibility with historical implementations

ACL capabilities:

- read, write, execute, lookup, and admin permissions
- list of users each with own permissions
- list of groups each with own permissions
- permissions for all others

Default/inheritable ACL's that propagate down the file hierarchy

Two user-level commands:

- *getfacl* - get file ACL permissions
- *setfacl* - set file ACL permissions

Access Control List Semantics

Support for POSIX.1e and NFSv4 semantics

- By design, NFSv4 semantics are very similar to Windows filesystem ACL semantics
- UFS implements both POSIX.1e and NFSv4 semantics (specified at boot time)
- ZFS implements only NFSv4 semantics
- NFSv4 uses inheritable ACLs rather than the default ACL in POSIX.1e
- FreeBSD uses the same command-line tools and APIs for both ACL types

Privilege

Each superuser privilege is identified and treated separately

Nearly 200 defined in **/sys/sys/priv.h**, some examples:

- PRIV_ACCT – Manage process accounting.
- PRIV_MAXPROC – Exceed system processes limit.
- PRIV_SETDUMPER – Configure dump device.
- PRIV_REBOOT – Can reboot system.
- PRIV_SWAPON – Add swap space.
- PRIV_MSGBUF – Read kernel message buffer.
- PRIV_KLD_LOAD – Load a kernel module.
- PRIV_ADJTIME – Set time adjustment.
- PRIV_SETTIMEOFDAY– Can set time of day.
- PRIV_VFS_WRITE – Override vnode write permission.

Priviledge Applied

Privilege checks cover all areas of the system

- network configuration and filtering
- filesystem mounting, unmounting, and exporting
- accessing or modifying kernel data and modules
- many others

Each privilege has three properties applied to a process or a file

- permitted: whether the process or file may ever have the privilege
- inheritable: whether the process or file may grant the privilege
- effective: whether the process or file can currently use the privilege

Access to privilege is done with MAC modules via the *priv_check* ( ) function.

Mandatory Access Control

Allows arbitrary security policies to be added to the system using labels and an expansion of traditional root access controls

Controls access/use of:

- files, pipes, and sockets
- kernel load-modules
- network interface configuration
- packet filtering
- process execution, visibility, signalling, and tracing
- file mapping
- kernel data
- accounting information
- NFS exports
- swapping

Auditing

Accountability and intrusion detection

Based on Open Basic Security Module (OpenBSM)

Generate records for kernel events involving

- access control
- authentication
- security management
- audit management
- user-level audit reports

Volume of audit trail is controllable

- audit preselection policy
- **auditreduce** to thin audit logs

User credentials can be augmented with an audit identifier (AUID)

- Holds terminal and session to be added to each audit record
- audit mask to subset global audit preselection policy

Audit Handling

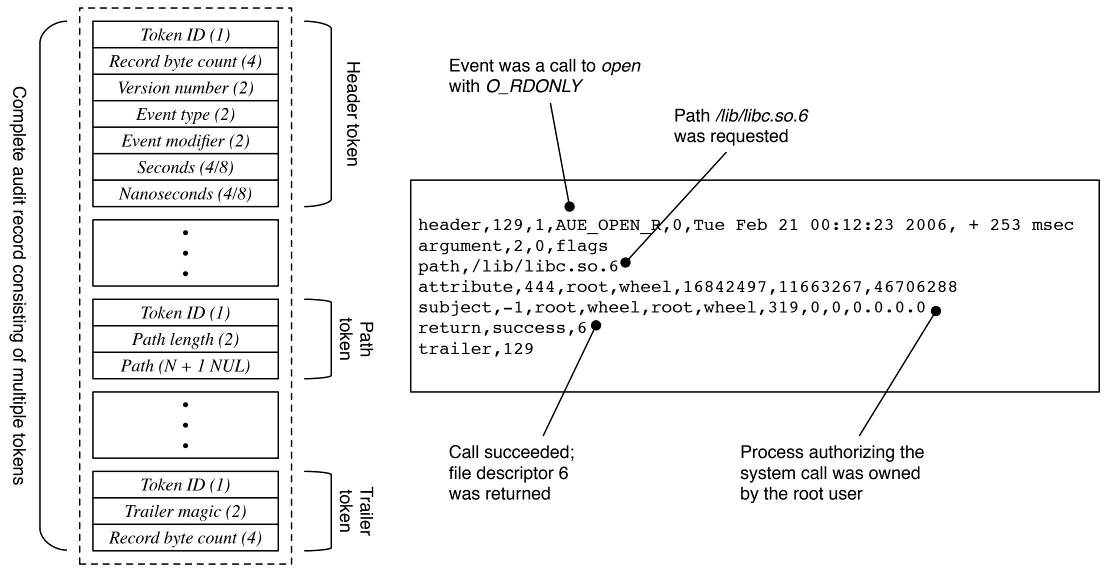
auditd daemon

- manages data collection
- content selection including selection of records collected
- responds to events such as running low on disk space

auditd daemon starts a kernel thread that manages record distribution

- stored in local filesystem
- sent elsewhere for storage
- sent to intrusion detection daemon

Example audit record

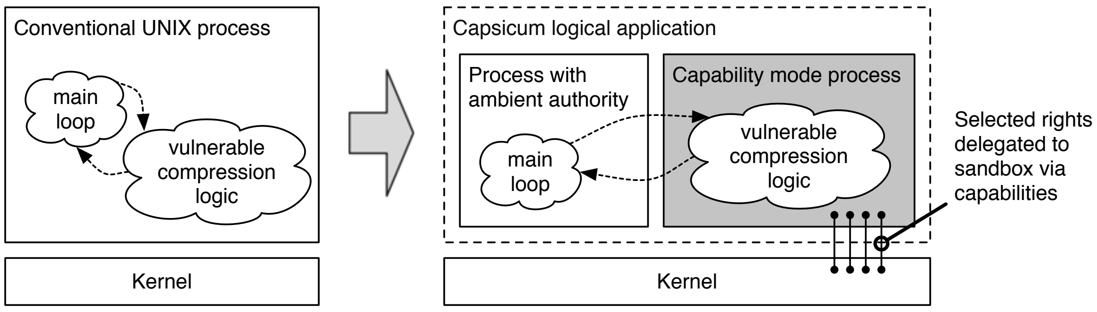


```
header,129,1,AUE_OPEN_R,0,Tue Feb 21 00:12:23 2006, + 253 msec
argument,2,0,flags
path,/lib/libc.so.6
attribute,444,root,wheel,16842497,11663267,46706288
subject,-1,root,wheel,root,wheel,319,0,0,0.0.0.0
return,success,6
trailer,129
```

Capsicum

Sandboxing of limited trust modules

- A small process with full privileges
- Untrusted libraries/modules run in separate process with access limited to minimal set of things that they need


Conventional UNIX process
main loop
vulnerable compression logic
Kernel
Capsicum logical application
Process with ambient authority
main loop
Capability mode process
vulnerable compression logic
Selected rights delegated to sandbox via capabilities
Kernel

Using Capsicum

- Process put into capability mode with *cap_enter* ( )
- Once in capability mode, cannot exit
- Can only work with its own file descriptors
- No access to filesystem namespace (e.g., *open* ( ) will fail but *openat* ( ) will work if given a descriptor open on a directory from which to start.

Sample Capsicum Capabilities

A set of rights is delegated to each descriptor

Sixty defined in **/sys/sys/capability.h**, some examples:

- CAP_READ – Read or receive
- CAP_WRITE – Write or send
- CAP_SEEK – Modify file descriptor offset
- CAP_FCHFLAGS – Set file flags
- CAP_FCHDIR – Set working directory
- CAP_FCHMOD – Change file mode
- CAP_FCHOWN – Change file owner
- CAP_LOOKUP – Use as starting directory for at operations
- CAP_POLL_EVENT – Test for events using select, poll, kqueue
- CAP_POST_EVENT – Post an event to kqueue
- CAP_ACCEPT – Accept sockets
- CAP_LISTEN – Set up a listen socket

Questions

Marshall Kirk McKusick
<mckusick@mckusick.com>
http://www.mckusick.com
http://www.freebsd.org
http://www.freebsdfoundation.org

Transparent Superpages for FreeBSD on ARM

Zbigniew Bodek
Semihalf, The FreeBSD Project
zbb@{semihalf.com, freebsd.org}

Abstract

This paper covers recent work on providing transparent superpages support for the FreeBSD operating system on ARM. The concept of superpages mechanism is a virtual memory optimization, which allows for efficient use of the TLB translations, effectively reducing overhead related to the memory management. This technique can significantly increase system's performance at the interface between CPU and main memory, thus affecting its overall efficiency.
The primary goal of this work is to elaborate on how the superpages functionality has been implemented on the FreeBSD/arm and what are the results of its application. The paper presents real-life measurements and benchmarks performed on a modern, multiprocessor ARM platform. Actual performance achievements and areas of application are shown. Finally, the article summarizes the possibilities of future work and further improvements.

1 Introduction

ARM technology becomes more and more prevailing, not only in the mobile and embedded space. Contemporary ARM architecture (ARMv7 and the upcoming ARMv8) is already on a par with the traditional PC industry standards in terms of advanced CPU features like:

- MMU (with TLB)
- Multi-level Cache
- Multi-core
- Hardware coherency

Performance and scalability of the ARM-based machine is largely dependent of these functionalities. Majority of the modern ARM chips is capable of running complex software and handle multiple demanding tasks simultaneously. In fact, general purpose operating systems have become the default choice for these devices.
The operating system (*kernel*) is an essential component of many modern computer systems. The main goal of the kernel operations is to provide runtime environment for user applications and manage available hardware resources in an efficient and reasonable way. Memory handling is one of the top priority kernel services. Growing requirements of the contemporary applications result in a significant memory pressure and increasing access overhead. Performance impact related to the memory management is likely to be at the level of 30% up to 60% [1]. This can be a serious issue, especially for the system that operates under heavy load.

Today's ARM hardware is designed to improve handling of contemporary memory management challenges. The key to FreeBSD success on this architecture is a combination of sophisticated techniques that will allow to take full advantage of the hardware capabilities and hence, provide better performance in many applications. One of such techniques is transparent superpages mechanism.
Superpages mechanism is a virtual memory system feature, whose aim is to reduce memory access overhead by making a better use of the CPU's Memory Management Unit hardware capabilities. In particular, this mechanism provides runtime enlargement of the TLB (translation cache) coverage and results in less overhead

related to memory accesses. This technique had already been applied on i386 and amd64 architectures and brought excellent results.

FreeBSD incorporates verified and mature, high-level methods to handle superpages. Work presented in this paper introduces machine-dependent portion of the superpages support for ARMv6 and ARMv7 on the mentioned OS.

To summarize, in this paper the following contributions have been made:

- Problem analysis and explanation
- Introduction to possible problem solutions
- Implementation of the presented solution
- Validation (benchmarks and measurements)
- Code upstream to the mainline FreeBSD 10.0-CURRENT

The project was sponsored by Semihalf and The FreeBSD Foundation. The code is publicly available beginning with FreeBSD 10.0.

2 Problem Analysis

In a typical computer system, memory is divided into few, general levels:

- CPU cache
- DRAM (main memory)
- Non-volatile backing storage (Hard Drive, SSD, Flash memory)

Each level in the hierarchy has significantly greater capacity and lower cost per storage unit but also longer access time. This kind of design provides best compromise between speed, price and capabilities of the contemporary electronics. However, the same architecture poses a number of challenges for the memory management system.

User applications stored in the external, non-volatile memory need to be copied to the main memory so that CPU can access them. The operating system is expected to handle all physical memory allocations, segments transitions between DRAM and external storage as well as protection of the memory chunks belonging to the concurrently running jobs. Virtual memory system carries these tasks without any user intervention. The concept allows to implement various, favorable memory management techniques such as on-demand paging, copy-on-write, shared memory and other.

2.1 Virtual Memory

Processor core uses so called *Virtual Address* (VA) to refer to the particular memory location. Therefore, the set of addresses that are 'visible' to the CPU is often called a *Virtual Address Space*. On the other hand there is a real or *Physical Address Space* (PA) which can incorporate all system bus agents such as DRAM, SoC registers, I/O.

Virtual memory introduces additional layer of translation between those spaces, effectively separating them and providing artificial private work environment for each application. This mechanism, however, requires some portion of hardware support to operate. Most application processors incorporate special hardware entity for managing address translations called *Memory Management Unit* (MMU). Address translation is performed with the *page* granulation. Page defines VA$\longrightarrow$PA translation for a subset of addresses within that page. Hence, for each resident page in the VA space exists exactly one frame in the physical memory. For the CPU access to the virtual address to succeed MMU has to provide the valid translation to the corresponding physical frame. The translations are stored in the main memory in the form of virtually indexed arrays, so called *Translation Tables* or *Page Tables*.

To speed up the translation procedure *Memory Management Unit* maintains a table of recently used translations called *Translation Lookaside Buffer* (TLB).

2.1.1 TLB Translations

Access to the pages that still have their translations cached in the TLB is performed immediately and implies minimal overhead related to the access completion itself. Other scenarios result in a necessity to search for a proper translation in the Translation Tables (presented in the Figure 1) or, in case of failure, handling the time consuming exception. TLB is therefore in the critical path of every memory access and for that reason it is desired to be as fast as possible. In practice, TLBs are fully associative arrays of size limited to several dozens of entries. In addition, operating systems usually configure TLB entries to cover the smallest available page size so that dense page granulation, thus low memory fragmentation could be maintained. Mentioned factors form the concept of *TLB coverage*, which can be described as the amount of memory that can be accessed directly, without TLB miss. Another substantial TLB behavior can be observed during frequent, numerous accesses to different pages in the memory (such situation can occur when a large set of data is being computed). Because a lot of pages is being touched in the process, free TLB entries become occupied fast. In order to make room for subsequent translations some entries need to be evicted. TLB evictions are made according to the eviction algorithm which is implementation defined. However, regardless of the eviction algorithm, significant paging traffic can cause recently used translations to be evicted even though they will need to be restored in a moment. This phenomenon is called *TLB trashing.* It is associated directly with the TLB coverage factor and can seriously impact system's performance.

2.1.2 Constraints and opportunities

It is estimated that performance degradation caused by the TLB misses is at 30-60%.

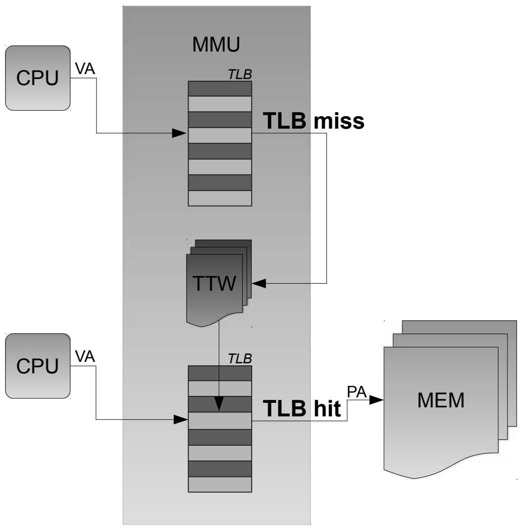


Figure 1: Memory access with TLB miss.

That is at least 20%, up to 50% more than in 1980's and 1990's [1]. TLB miss reduction is therefore expected to improve memory bandwidth and hence overall system performance, especially for resource-hungry processes. Reducing the number of TLB misses is equivalent to TLB coverage enhancement. Obvious solutions to achieve that would be to:

◦ Enlarge the TLB itself.
However, bigger translation cache means more logic, higher complexity and greater energy consumption that still may result in a little improvement. To sustain satisfying TLB characteristics with the currently available technologies, translation buffers can usually hold tens up to few hundreds of entries.

◦ Increase the base page size.
Majority of the microprocessor architectures support more than one page size. This gives the opportunity to cover larger memory areas consuming only a single entry in the TLB. However, this solution has a major drawback in the form of increased fragmentation and hence, inefficient memory utilization. The application may need to access very limited amount of memory but placed in a few, distinct locations. If the small pages were used as a base allocation

unit, less memory is reserved and more physical frames are available for other agents. On the other hand using superpages as a main allocation unit results in a rapid exhaustion of available memory for new allocations. In addition, single page descriptor contains only one set of access permissions and page attributes including *dirty* and *referenced* bits. For that reason, the whole dirty superpages needs to be written back to the external storage on page-out since there is no way to determine which fraction of the superpage has been actually written. This may cause serious disk traffic that can surpass the benefit from reducing TLB misses.

◦ Allow user to choose the page size.
In that case, the user would have to be aware of the memory layout and requirements of the running applications. That approach could be as much effective for some cases as it will be ineffective for any other. In fact, this method contradicts the idea of the virtual memory that should be a fully transparent layer.

2.1.3 Universal Solution

Reduction of the TLB miss factor has proven to be a complex task that requires support from both hardware and operating system sides. OS software is expected to provide low-latency methods for memory layout control, superpage allocation policy, efficient paging and more.

FreeBSD operating system offers the generic and machine independent framework for transparent superpages management. Superpages mechanism is a well elaborated technology on FreeBSD, which allow for runtime page size adjustment based on the actual needs of the running processes. This feature is already being successfully utilized on i386 and amd64 platforms. The observed memory performance boost for those architectures is at 30%. These promising numbers encouraged to apply superpages technique on another, recently popular ARM architecture. Modern ARM revisions (ARMv6, ARMv7 and upcoming ARMv8) are capable of using various page sizes allowing for superpages mechanism utilization.

3 Principles of Operation

Virtual memory system consists of two main components. The machine-independent VM manages the abstract entities such as address spaces, objects in the memory or software representations of the physical frames. The architecture-dependent `pmap(9)`, on the other hand, operates on the memory management hardware, page tables and all low-level structures. Superpages framework affects both aspects of the virtual memory system. Therefore, in order to illustrate the main principles of superpages mechanism, relevant VM operations are described. Then the specification of the Virtual Memory System Architecture (VMSA) introduced in ARMv6/v7-compliant processors is provided along with the opportunities to take advantage of the superpages technique on that architectures.

3.1 Reservation-based Allocation

VM uses `vm_page` structure to represent physical frame in the memory. In fact, the physical space is managed on page-by-page basis through this structure [2]. In the context of superpages, `vm_page` can be called the base page since it usually represents the smallest translation unit available (in most cases 4 KB page). Operating system needs to track the state and attributes of all resident pages in the memory. This knowledge is a necessity for a pager program to maintain an effective page replacement policy and decide which pages should be kept in the main memory and which ought to be discarded or written back to the external disk.

Files or any areas of anonymous memory are represented by virtual objects. `vm_object` stores the information about related `vm_pages` that are currently resident in the main memory, size of the area described by this object, pointer to shadow objects that hold private copies of modified pages and other information [3]. At system boot time, kernel detects the number of free pages in the memory and assigns them `vm_page` structures (except for pages occupied

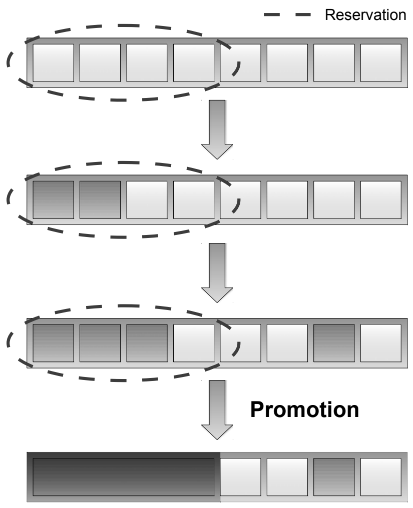


Figure 2: Basic overview of the reservation-based allocation.

by the kernel itself). When the processes begin to execute and touch memory areas they generate page faults since no pages from the free list have been filled with relevant contents and assigned to the corresponding object. This mechanism is a part of the on-demand paging and implies that only requested (and further utilized) pages of any object are cached in the main memory. Superpages technique relies on this virtual memory feature and is in a way its extension. When the reservation-based allocation is enabled (`VM_NRESERVLEVEL` set to non-zero value) and the referenced object is of superpage size or greater, VM will reserve a continuous physical area in memory for that object. This is justified by the fact that superpage mapping can translate a continuous range of virtual addresses to the range of physical addresses within a single memory frame. Pages within the created area are grouped in a population map. If the process that refers to the object will keep touching subsequent pages inside the allocated area, the population map will eventually get filled up. In that case, the related memory chunk will become a candidate for promotion to a superpage. The mechanism is briefly visualized in the Figure 2.

Not all reservations can be promoted even though the underlying pages satisfy the continuity requirements. That is because the single superpage translation has only one set of attributes and access permissions for the entire area covered by the mapping. Therefore, it is obvious that all base pages within the population map must be consistent in terms of all settings and state for promotion to succeed. In addition, superpages are preferred to be promoted read-only unless all base pages have already been modified and are marked 'dirty'. The intention is to avoid increased paging traffic to the disk. Since there is only one modification indicator for the whole superpage, there is no way to determine which portion of the corresponding memory has been actually written. Hence, the entire superpage area needs to be written back to the external storage. Demotion of the read-only superpage on write attempt is proven to be a more effective solution [1]. Summarizing, to allow for the superpage promotion, the following requirements must be met:

- The area under the superpage has to be continuous in both virtual and physical address spaces
- All base mappings within the superpage need to have identical attributes, state and access permissions

Not all reservations can always be completed. If the process is not using pages within the population map then the reservation is just holding free space for nothing. In that case VM can evict the reserved area in favor of another process. This proves that the superpages mechanism truly adapts to the current system needs as only active pages participate in the page promotion.

3.2 ARM VMSA

Virtual Memory System Architecture introduced in ARMv7 is an extension of the definition presented in ARMv6. Differences between those revisions are not relevant to this work since backward compatibility with ARMv6 has to be preserved (ARMv6 and ARMv7 share the the same `pmap(9)` module).

ARMv6/v7-compliant processors use Virtual Addresses to describe a memory location in their 32-bit Virtual Address Space. If the CPU's Memory Management Unit is disabled, all Virtual Addresses refer directly to the corresponding locations in the Physical Address Space. However, when MMU is enabled, CPU needs additional information about which physical frame to access when some virtual address is used. Both, logical and physical address spaces are divided into chunks - pages and frames respectively. Appropriate translations are provided in form of memory resident Translation Tables. Single entry in the translation table can hold either invalid data that will cause Data/Prefetch abort on access, valid translation virtual⟶physical or pointer to the next level of translation. ARMv7 (without Large Physical Address Extension) defines two-level translation tables.

L1 table consists of 4096 word sized entries each of which can:

- Cause an abort exception
- Translate a 1 MB page to 1 MB physical frame (section mapping)
- Point to a second level translation table

In addition, a group of 16 L1 entries can translate a 16 MB chunk of virtual space using just one, supersection mapping.
L1 translation table occupies 16 KB of memory and needs to be aligned to that boundary.

L2 translation table incorporates 256 word sized entries that can:

- Cause an abort exception
- Provide mapping for a 4 KB page (small page)

Similarly to L1 entries, 16 L2 descriptors can be used to translate 64 KB large page by a single TLB entry. L2 translation table takes 1 KB of memory and has to be stored with the same alignment.

Recently used translations are cached in the unified TLB. Most of the modern ARM processors have additional, 'shadow' TLBs for instructions and data. These are designed to speed-up the translation process even more and are fully transparent to the programmer. Usually, TLBs in ARMv6/v7 CPUs can hold tens of entries so the momentary TLB coverage is rather small. An exceptional situation is when pages bigger than 4 KB are used.

3.2.1 Translation Process

When a TLB miss occurs MMU is expected to find a mapping for the referenced page. The process of fetching translations from page tables to TLB is called a Translation Table Walk (TTW) and on ARM it is performed by hardware.

For a short page descriptor format (LPAE disabled), translation table walk logic may need to access both L1 and L2 tables to acquire proper mapping. TTW starts with L1 page directory whose address in the memory is passed to the MMU via Translation Table Base Register (TTBR0/TTBR1). First, 12 most significant bits of the virtual address (VA[31:20]) are used as an index to the L1 translation table (page directory). If the L1 descriptor's encoding does not indicate otherwise the section (1 MB) or supersection (16 MB) mapping is inserted to the TLB and translation table walk is over. However, if L1 entry points to the L2 table then 8 subsequent bits of the virtual address (VA[19:12]) serve as an index to the destination L2 descriptor in that table. Finally the information from L2 entry can be used to insert small (4 KB) or large (64 KB) mapping to the TLB. Of course, invalid L1 or L2 descriptor format results in data or prefetch abort depending on the access type.

3.2.2 Page Table Entry

Both L1 and L2 page descriptors hold not only physical address and size for the related pages but also a set of encoded attributes that can define access permissions, memory type, cache mode and other. Page descriptor format is programmable to some extent, depending on enabled features and overall CPU/MMU settings (access permissions model, type extension, etc.). In general, every aspect of any memory access is fully described by the page table entry. This also indicates that any attempt to reference a page in a different manner than allowed will cause an exception.

4 Superpages Implementation for ARM

The paragraph elaborates on how the superpages mechanism has been implemented and operates on ARM. Main modifications to the virtual memory system have been described along with the explanation of the applied solutions.

4.1 Superpage size selection

First step to support superpages on a new architecture is to perform VM parameters tuning. In particular, reservation-based allocation needs to be enabled and configured according to the chosen superpages sizes.

Machine independent layer requires two parameters declared in `sys/arm/include/vmparam.h`:

- `VM_NRESERVLEVEL` - specifies a number of promotion levels enabled for the architecture. Effectively this indicates how many superpage sizes are used simultaneously.
- `VM_LEVEL_{X}_ORDER` - for each reservation level this parameter determines how many base pages fully populate the related reservation level.

At this stage a decision regarding supported superpage sizes had to be made. 1 MB section mapping has been chosen for a superpage whereas 4 KB small mapping has remained a base page. This approach has a twofold advantage:

1. Shorter translation table walk when TLB miss on the area covered by a section mapping.

   In that scenario, TTW penalty will be limited to one memory access only (L1 table) instead of two (L1 and L2 tables).
2. Better comparison with other architectures.

   i386 and amd64 can operate on just one superpage size of 2/4 MB. Similar performance impact was expected when using complementary page sizes on ARM.

Summarizing, VM parameters have been configured as follows:

`VM_NRESERVLEVEL` set to 1 - indicates one reservation level and therefore one superpage size in use.
`VM_LEVEL_0_ORDER` set to 8 - level 0 reservation consists of 256 (1 « 8) base pages.

4.2 `pmap(9)` extensions

The core part of the machine dependent portion of superpages support is focused on the `pmap` module. From a high-level point of view, VM ”informs” lower layer when the particular reservation is fully populated. This event implies a chance to promote a range of mappings to a superpage but promotion itself still may not succeed for various reasons. There are no explicit directives from VM that would influence superpages management. `pmap` module is therefore expected to handle:

- promotion of base pages to a superpage
- explicit superpage creation
- superpage demotion
- superpage removal

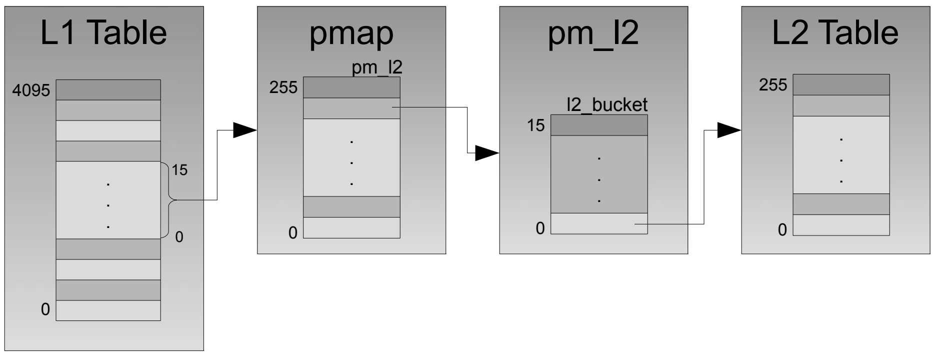


Figure 3: Page tables and kernel structures organization.

4.2.1 Basic Concepts

pmap(9) module is responsible for managing real mappings that are recognizable by the MMU hardware. In addition it has to control the state of all physical maps and pass relevant bits to the VM. Main module file is located at sys/arm/arm/pmap-v6.c and is supplemented by the appropriate structure definitions from sys/arm/include/pmap.h. Core structure representing physical map is struct pmap.

During virtual memory system initialization pmap module allocates one L1 translation table for each fifteen user processes out of maximum pool of *maxproc*. L1 entries sharing can be achieved by marking all L1 descriptors with the appropriate domain ID. Architecture defines 16 domains of which 15 are used for user processes and one is reserved for the kernel. This design can reduce KVM occupancy as each L1 table requires 16 KB of memory which is never freed. Each pmap structure holds pm_l1 pointer to the corresponding L1 translation table meta-data (l1_ttable) which provides table's physical address to move to the TTBR on context switch as well as other information used to allocate and free L1 table on process creation and exit.

Figure 3 shows the page tables organization and their relation with the corresponding kernel structures. L1 page table entry points to the L2 table which collects up to 256 L2 descriptors. Each L2 entry can map 4 KB of memory. L2 table is allocated on demand and can be freed when unused. This technique effectively saves 1 KB of KVA per each unused L2 table.
pmap's L2 management is performed via pm_l2 array of type struct l2_dtable. Each of pm_l2 fields holds enough L2 descriptors to cover 16 MB of data. Hence, for each 16 L1 table entries, exists one pm_l2 entry. l2_dtable structure incorporates 16 elements of type struct l2_bucket each of which describes single L2 table in memory. In the current pmap-v6.c implementation, both l2_dtable and L2 translation table are allocated in runtime using UMA(9) zone allocator. l2_occupancy and l2b_occupancy track the number of allocated buckets and L2 descriptors accordingly. l2_bucket can be deallocated if none of 256 L2 entries within the L2 table is in use. Similarly, l2_dtable can be freed as soon as all 16 l2_buckets within the structure are deallocated.

Additional challenge for the pmap module is to track multiple mappings of the same physical page. Different mappings can have different states even if they point to the same physical frame. When modifying physical layout (page-out, etc.) it is necessary to take into account wired, dirty and other attributes of all pages related to a particular physical frame. The described functionality is provided by using pv_entry structures organized in chunks and maintained for each pmap in the system. When a new mapping is created for any pmap, the corresponding pv_entry is allocated and put into the PV list of the related vm_page.

Superpages support required to provide extensions for the mentioned mechanisms and techniques. Apart from implementing routines for explicit superpage management the objective was to make the existing code superpages aware.

4.2.2 Promotion to a Superpage

The decision whether to attempt promotion is based on two main conditions:

- vm_reserv_level_iffullpop() - indicates that physical reservation map is fully populated
- l2b_occupancy - implies that (aligned) virtual region of superpage size is fully mapped using base pages

Both events will most likely occur during new mapping insertion to the address space of the process. Therefore the promotion attempt is performed right after successful pmap_enter() call.

The page promotion routine (pmap_promote_section()) starts with the preliminary classification of the page table entries within the potential superpage. At this point the decision had to be made which pages to promote and which of them should be excluded from the promotion. In the presented implementation, promotion to a superpage is discontinued for the following cases:

- *VA belongs to a vectors page*
  Access to a page containing exception vectors must never abort and should be excluded from any kind of manipulation for safety reasons. Every abort in this case would result in nested exception and fatal system error.
- *Page is not under PV management*
  With Type Extension (TEX) disabled, page table entry has not enough room to store all the necessary status bits. For that reason pv_flags field from the pv_entry structure holds the additional data including bits relevant for the promotion to a superpage.
- *Mapping is within the kernel address space*
  On ARM, kernel pages are already mapped using as much section mappings as possible. The mappings are then replicated in each pmap.

Page table entry in the L2 under promotion is also tested for reference and modification bits as well as permission to write. Superpage is preferred to be a read-only mapping to avoid expensive, superpage-size transitions to a disk on page-out. Therefore it is convenient to clear the permission to write for a base page if it has not been marked dirty already. All of the mentioned tests apply to the first base page descriptor in the set. This approach can reduce overhead related to the unsuccessful promotion attempt since it allows to quickly disregard invalid mappings and exit. However if the first descriptor is suitable for the promotion then the remaining 255 entries from the L2 table still need to be checked

Apart from the above mentioned criteria the area under superpage must satisfy the following conditions:

1. *Continuity in the VA space*
2. *Continuity in the PA space*
   Physical addresses stored in the subsequent L2 descriptors must differ by the size of the base page (4 KB).
3. *Consistency of the pages' attributes and states*

When all requirements are met then it is possible to create single 1 MB section mapping for a given area. It is important that during promotion process L2 table zone is not being deallocated. Corresponding l2_bucket is rather stashed to speed-up the superpage demotion in the future.

The actual page promotion can be divided into two stages:

- pmap_pv_promote_section()
  At this point pv_entry related to the first vm_page in a superpage is moved to another list of PV associated with the 1 MB physical frame. The remaining PV entries can be deallocated.
- pmap_map_section()
  The routine constructs the final section mapping and inserts it to the L1 page descriptor. Mapping attributes, access permissions and cache mode are identical with all the base pages.

Successful promotion ends with the TLB invalidation which flushes old translations and allows MMU to put newly created superpage to the TLB.

4.2.3 Explicit Superpage Creation

Incremental reservation map population is not always a necessity. In case of a mapping insertion for the entire virtual object it is possible to determine the object's size and its physical alignment. The described situation can take place when pmap_enter_object() is called. If the object is at least of superpage size and VM has performed the proper alignment it is possible to explicitly map the object using section mappings.

pmap_enter_section() has been implemented to create a direct superpage mappings. The routine has to perform preliminary page classification similar to the one in pmap_promote_section(). This time however, it is not necessary to check any of the base pages within the potential superpage since they do not exist yet. Bits that still need to be tested are:

- *PV management status*
- *L1 descriptor status*
  The given L1 descriptor cannot be used for a section mapping if it is already a valid section or it is already serving as a page directory for a L2 table.

Direct insertion of the mapping involves a necessity to allocate new pv_entry for a 1 MB frame. This task is performed by pmap_pv_insert_section() which may not succeed. In case of failure the superpage cannot be mapped, otherwise section mapping is created immediately.

4.2.4 Superpage Demotion and Removal

When there is a need to page-out or modify one of the base pages within the superpage it is required to destroy a corresponding section mapping. Lack of any mapping for a memory region that is currently in use would cause a chain of expensive vm_fault() calls. Demotion procedure (pmap_demote_section()) is designed to overcome this issue by recreating L2 translation table in place of the removed L1 section.

There are two possible scenarios of the superpage demotion:

1. Demotion of the page created as a result of promotion.
   In that case it is possible to reuse the already allocated l2_bucket that has been stashed after the promotion. This scenario has got two major advantages:
   - No need for any memory allocation for L2 directory and L2 table.
   - If the superpage attributes have not changed then there is no need to modify or fill the L2 descriptors

2. Demotion of the page that was directly inserted as a superpage.
   This implies that there is no stashed L2 table and it needs to be allocated and created from scratch. Any allocation failure results in an immediate exit due to speed restrictions. Sleeping is not an option.

The demotion routine has to check if the superpage has exactly the same attributes and status bits as the stashed (or newly created) L2 table entries. If not then the L2 entries need to be recreated using current L1 descriptor. PV entries also need to be allocated and recreated using `pv_entry` linked with the 1 MB page. Finally when the L2 table is in place again, the L1 section mapping can be fixed-up with the proper L1 page directory entry and the corresponding translation in the TLB ought to be flushed.

The last function used for superpage deletion is `pmap_remove_section()`. It is used to completely unmap any given section mapping. Calling this function can speed-up `pmap_remove()` routine if the removed area is mapped with a superpage and the size of the space to unmap is at least of superpage size.

4.2.5 Configuration and control

At the time when this work is written, superpages support is disabled by default in `pmap-v6.c`. It can be enabled in runtime during system boot by setting a loader variable:

```
vm.pmap.sp_enabled=1
```

in `loader.conf` or it can be turned on during compilation time by setting:

```
sp_enabled
```

variable from `sys/arm/arm/pmap-v6.c` to a non-zero value.

System statistics related to the superpages utilization can be displayed by invoking:

```
sysctl vm.pmap
```

command in the terminal. The exemplary output can be seen below:

```
vm.pmap.sp_enabled: 1
vm.pmap.section.demotions: 258
vm.pmap.section.mappings: 0
vm.pmap.section.p_failures: 301
vm.pmap.section.promotions: 1037
```

`demotions`	– number of demoted superpages
`mappings`	– explicit superpage mappings
`p_failures`	– promotion attempts that failed
`promotions`	– number of successful promotions

5 Results and benchmarks

The functionality has been extensively tested using various benchmarks and techniques. The performance improvement depends to a large extent on the application behavior, usage scenarios and amount of available memory in the system. Processes allocating large areas of consistent memory or operating on big sets of data will benefit more from superpages than those using small, independent chunks.

Presented measurements and benchmarks have been performed on Marvell Armada XP (quad-core ARMv7-compliant chip).

5.1 GUPS

The most significant results can be observed using the *Giga Updates Per Second* (GUPS) benchmark. GUPS measures how frequently system can issue updates to randomly generated memory locations. In particular it measures both memory latency and bandwidth. On multi-core ARMv7 platform, measured CPU time usage and real time duration dropped by 34%. Number of updates performed in the same amount of time has increased by 52%.

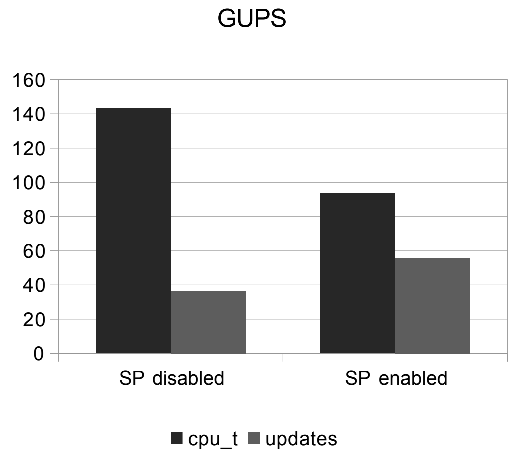


Figure 4: GUPS results.CPU time used [s], number of updates performed [100000/s].

5.2 LMbench

LMbench is a popular suite of system performance benchmarks. It is equipped with the memory testing program and can be used to examine memory latency and bandwidth. Measured memory latency has dropped by 37,85% with *superpages* enabled. Memory bandwidth improvement varied depending on the type of operation and was in the range from 2,26% for `mmap` reread to 8,44% for memory write. It is worth noting that *LMbench* uses *STREAM* benchmark to measure memory bandwidth which uses floating point arithmetic to perform the operations on memory. Currently FreeBSD does not yet support FPU on ARM what had a negative impact on the results.

Mmap reread `[MB/s]`	*Bcopy (libc)* `[MB/s]`	*Bcopy (hand)* `[MB/s]`	*superpages*
645,4	305,4	432,3	
660,0	312,4	446,9	✓

Table 1: LMbench. Memory bandwidth measured on various system calls.

Mem read `[MB/s]`	*Mem write* `[MB/s]`	*Mem latency* `[ns]`	*superpages*
681	3043	238,8	
696	3300	148,4	✓

Table 2: LMbench. Memory bandwidth and latency measured on memory operations.

The results summary is shown in Tables 1 and 2. Table 3 on the other hand shows the the percentage improvement of the parameters with the best test results.

Mem write %	*Rand mem latency %*
8,44	37,85

Table 3: LMbench. Percentage improvement of the selected parameters.

5.3 Self-hosted world build

Using superpages helped to reduce self-hosted world build when using GCC. The results are summarized in Table 4. The time needed for building the whole set of user applications comprising to the root file system has dropped by 1 hour 22 minutes (20% shorter). No significant change has been noted when using CLANG.

GCC	*CLANG*	*superpages*
6h 36min	6h 16min	
5h 14min	6h 15min	✓

Table 4: Self-hosted `make buildworld` completion time.

5.4 Memory stress tests

Presented functionality has been also tested in terms of overall stability and reliability. For that purpose two popular stress benchmarks have been used:

- *forkbomb:* `forkbomb -M`
  Application can allocate entire available memory using `realloc()` and access this memory.

- *stress:* `stress -vm 4 -vm-bytes 400M` Benchmark imposes certain types of compute stress on the system. In this case 4 processes were spinning on `malloc()`/`free()` calls, each of which working on 400 MB of memory.

No anomalies or instabilities were detected even during long runs.

6 Future work

The presented functionality has significant impact on system's performance but does not cover all of the hardware and OS capabilities. There are possible ways of improvement.

Adding support for additional 64 KB page size will further increase the amount of created superpages, enabling a smoother and more efficient process for the promotion from 4 KB small page to 1 MB section. In addition, a larger number of processes will be capable of taking advantage from superpages if the required population map size is smaller.

In addition, current `pmap(9)` implementation uses PV entries to store some information about the mapping type and status. This implies the necessity to search through PV lists on each promotion attempt. `TEX` (Type Extension) support would allow to move those additional bits to the page table entry descriptors and lead to reduction of the promotion failure penalty.

7 Conclusions

Presented work has brought the transparent superpages support to the ARM architecture on FreeBSD. The paper described virtual memory system from both OS and hardware points of view. System's bottle-necks and design constrains have been carefully described. In particular the work has elaborated on the TLB miss penalty and its influence on the overall system performance.

Mechanisms implemented during the project met their objectives and provided performance gain on the interface between CPU and memory. This statement has been supported by various tests and benchmarks performed on a real ARM hardware. Test results vary between different benchmarks but improvement can be observed in all cases and is at 20%.

Introduced superpages support has been committed to the official FreeBSD SVN repository and is available starting from revision 254918.

8 Acknowledgments

Special thanks go to the following people:

Grzegorz Bernacki and Alan Cox, for all the help and mentorship.
Rafał Jaworowski, mentor of this project.

Work on this project was sponsored by Semihalf and The FreeBSD Foundation.

9 Availability

The support has been integrated into the mainline FreeBSD 10.0-CURRENT and is available with the FreeBSD 10.0-RELEASE. The code can also be downloaded from the FreeBSD main SVN repository.

References

[1] Juan E. Navarro, *Transparent operating system support for superpages*, 2004

[2] The FreeBSD Documentation Project, *FreeBSD Architecture Handbook*, 2000-2006, 2012-2013

[3] Marshall Kirk McKusick, *The Design and Implementation of the FreeBSD Operating System*, 2004

Carve your NetBSD

Pierre Pronchery

Freelance IT-Security Consultant
DUEKIN Consulting
<pierre.pronchery@duekin.com>

Guillaume Lasmayous

<guigui2@guigui2.net>

Abstract

After over 20 years of active development [1], NetBSD [2] proves to be a resilient, attractive, featureful and stable platform for industrial products and research projects alike [3]. The reasons behind the technical and practical merits of the system will not be explored or debated here; however, there is always space for improvement. This paper (and associated talk) attempts to identify areas in which gaps may be determined, and presents ways and ongoing work to address them. The topics covered range from the development model to a more user-oriented release strategy, through the adoption of key industrial processes. The EdgeBSD Project is introduced [4] as a platform to experiment with these propositions. Additionally, user interfaces for both desktop and embedded environments are demonstrated, thanks to the DeforaOS Project [5].

[1] Initial revision for `src/Makefile`, *http://cvsweb.netbsd.org/bsdweb.cgi /src/Makefile?rev=1.1&content-type=text/x-cvsweb-markup*

[2] The NetBSD Project, *http://www.netbsd.org/*

[3] « What is NetBSD? » in the NetBSD Guide, *https://www.netbsd.org/docs/guide/en/chap-intro.html*

[4] The EdgeBSD Project, *https://www.edgebsd.org/*

[5] The DeforaOS Project, *https://www.defora.org/*

Introduction

NetBSD is one of the first Open Source projects to have adopted a community-based development model [6]. Established and first released as far back as 1993, it is developed using a "global" approach to the system, with the kernel and essential user-land components integrated together as a whole [7]. This is very different from the approach observed in the omnipresent (GNU/)Linux community, where hundreds of software distributions respectively collect and integrate myriads of components together, as found within the global Open Source Software "bazaar" [8].

The development workflow for NetBSD reflects this desire of a clean, integrated platform for the base system and packages. While generally praised for the quality of the designs and solutions adopted, it can also be harmful for the growth and renewal of its developer base [9]. Moreover, the centralized source code management tool behind development, CVS, no longer fits the distributed approach in increasing use within the industry [10].

The audience met by the NetBSD Project in its current form is largely composed of software developers and experienced users. While a number of software and hardware companies ship products based on NetBSD [11], this is usually not advertised as such, and few actively and officially take part of the development effort (as opposed to FreeBSD [12] and OpenBSD [13] for instance). Besides the development model itself, the reasons identified here include the arguably rough, "old-school" aspect of the project releases for prospective users and developers.

A number of ideas and ways to address the issues listed are available, with various costs. Among them, extending releases to ready-to-flash software images for a range of devices and purposes is already an unofficial trend among the developers [14] (for the Raspberry Pi and BeagleBone for instance). An alternative installer with a graphical interface (as opposed to text-based) may also be welcome. Then, developers might expect more guidance as to their work environment, for which new possibilities are demonstrated. These goals led to the creation of the EdgeBSD Project [15], where there is complete freedom to experiment with these propositions.

Finally, functional devices with a modern, featureful user interface running on NetBSD in a variety of ways and purposes are introduced. They are based on the DeforaOS Project [16], which desktop environment can already be found packaged in the latest releases of the pkgsrc project [17]. They range from a typical desktop environment for users or developers, to embedded use as an e-book reader, a tablet [18], or as a telephony platform [19]. Software and hardware for the latter two projects was presented at BSD conferences in 2012 and 2013, and they are both being actively maintained [20], with significant progress to be demonstrated.

Development workflow

The development workflow of The NetBSD Project is handled separately when it comes to the base system and to package management. Although being subject to very different release-engineering policies and portability constraints, they are very closely related and a number of issues can be identified as a result.

Base system

Release-engineering

The NetBSD Project releases new major versions of the base system based on a given set of features desired for the release. A branch is forked from the main development tree once these features implemented, for heavier testing and preparation for the new release. This means that there is usually no particular deadline set with regard to when a new version should be released: it is done when considered ready.

There are three different kinds of releases [21]:

- major releases: like NetBSD 5.0 or 6.0, as described above;
- stable releases: like NetBSD 5.1 or 6.2, containing new features backported to the release branch;
- minor releases: like NetBSD 6.0.1 or 6.1.2, containing only essential fixes and security patches relevant to a stable release.

Stable branches for the two latest major releases are always maintained for essential fixes and security issues. In practice, this means that a stable branch for an old major release may be released after a new major release (for instance, NetBSD 5.2 was released after NetBSD 6.0). Although a common practice in the software industry, this can be confusing to some users.

More importantly, as of January 22nd 2014, there are at least five releases of the base system which are officially supported: 5.0.2, 5.1.3, 5.2.1, 6.0.3 and 6.1.2. While potentially useful for system administrators (who may then choose to follow minor release updates for stability) the number of releases available and supported at any given time can be overwhelming for the developers of the project and for users alike.

System sets

The NetBSD system is primarily shipped and distributed as a collection of binary sets. They simply consist of compressed tar archives. Two of them are essential for a working initial setup ("base" and "etc"), while the others provide additional functionality or documentation accordingly. Instructions and additional tools (where applicable) to assist the installation process are provided for each architecture supported.

Centralized development

The source code for NetBSD has always been maintained thanks to the CVS Source Code Management tool (SCM) [22]. CVS is a client-server based tool, which requires an active connection to the server to be able to commit to the repository. In other words it is a centralized system, requiring developers to be online to be able to track their own set of changes. Every developer is allowed to create and manage an own set of branches [23]. All branches are public and their use in CVS is resource-intensive, as it typically requires modifying the entire remote repository; as a result, their use is very limited.

More generally speaking, CVS is no

longer on par with more modern SCM tools, and regularly criticized for its current limitations [24]. Moving or renaming files is a cumbersome process in CVS for instance. As another example, while in theory it is perfectly possible to use CVS in a decentralized fashion (using the "cvs import" command), in practice this is error-prone, resource intensive and inelegant, therefore seldom performed.

Package management

In NetBSD, packages are provided by the pkgsrc project [25]. pkgsrc is a cross-platform packaging system. Born in 1997 as a fork of the FreeBSD ports for NetBSD [26], it has largely diverged since then and supports over twenty platforms nowadays [27]. Although NetBSD remains the primary target platform for pkgsrc, the project is meant to be maintained separately and subject to distinct release-engineering policies.

Source-based approach

Contrary to NetBSD's own release management, pkgsrc is primarily meant as a source-based package deployment tool. A number of reasons behind this choice are regularly mentioned in the release notes [28]. Among them, emphasis is placed on the security aspects of this choice, which include the verification of the provenance and integrity of the source code, as well as that of the compilation environment. We do not believe these reasons to be accurate, given the possibility of the presence of backdoors in the original source code archives [29] or within the compilers used [30].

Quarterly releases

Again, the release management for pkgsrc is opposed to that of the NetBSD Project. pkgsrc is under constant development, with stable releases being tagged exactly four times per year (hence quarterly releases). After a short freeze period, used to work on the most essential build fixes prior to release, the main development tree is released as-is.

Centralized development

Just like for NetBSD, development of the pkgsrc project is performed with CVS as the SCM tool. It is therefore centralized as well, and hosted by The NetBSD Foundation on the same servers as NetBSD [31]. Importantly, being an official pkgsrc developer requires membership to The NetBSD Foundation too.

An additional repository for pkgsrc is available, called "pkgsrc-wip" for "Work In Progress". Managed by a prominent NetBSD and pkgsrc developer, Thomas Klausner (wiz@), this repository is hosted on SourceForge [32]. Also using CVS as the SCM tool, it is meant as a staging area for pkgsrc. Unlike its counterpart, it does not require membership to The NetBSD Foundation to be able to contribute. Packages from this repository are not included in official releases for pkgsrc though.

Issues identified

CVS is deprecated

The CVS SCM tool is no longer actively maintained, with the last preview version released about seven years ago. While mature, stable and functional, it is obsolete by today's standards and often dismissed by the new generation of software developers. We believe using CVS today to be a showstopper for the integration of new contributors to the NetBSD and pkgsrc projects for this reason.

Also, because of its centralized approach, it is very difficult for prospective developers to work efficiently without being official developers, and therefore gain the experience and confidence required to fulfill the integration process.

Migration to a newer SCM tool is a regular topic on the corresponding NetBSD mailing-list [33]. While no clear consensus has been made for a potential new tool, Git is regularly mentioned in such posts. The NetBSD wiki proposes a summary of these passionate discussions [34].

Lost contributions to the system

An indirect consequence of the centralized contribution management is the difficulty for external contributors to either submit significant contributions, or provide patches conveniently. In the first case in particular, such contributions have to be hosted on separate servers and are easily forgotten or worse, can be definitely lost [35].

Decentralized SCM tools like Git allow trivial ways to mirror external contributions locally, or handle patches conveniently.

Conflicting deployment and security policies between the base system and packages

The release policies of the base system and that of the packages are totally opposed on the following aspects:

- they are not synchronized (for either releases or end of support)
- the base system relies on binaries while packages are typically built from source.

As a result, it is difficult to run a stable system or to follow the latest packages. In the former case, packages are out of date after three to six months, and then are no longer supported for security. In the latter case, it is necessary to either build packages from source continuously (potentially re-building all of them regularly because of dependencies), or to switch releases of pkgsrc every three to six months, with many risks of regressions.

Both scenarios are commonplace, and the situation is worsened for every additional system (and architecture) the user is running. The significant amount of versions available for each major release of the base system also puts additional burden on the users.

We believe this situation to be counter-productive, harmful for the vitality of the project and dangerous for the security of its users.

Lack of quality assurance on pkgsrc releases

Packages in pkgsrc are maintained by individual developers, each potentially running a different system with a specific installed base. There is no way to ensure that any package will function as expected, even when installed on official, pristine binary releases. Pre-release periods for the pkgsrc project (called "freezes") are focused on build tests rather than functionality tests.

Another negative impact of the "freeze" periods is that it prevents developers to work on changes forbidden in these periods. This can also be seen as a consequence of the technical limitations of the centralized SCM tool in use, CVS (see the section called “Centralized development”).

Possibly insecure distribution of sets and packages

There are additional possibilities for external attacks, notably while distributing the repository of patches and checksums via insecure means. The most common way to distribute binary sets and packages is currently via HTTP, BitTorrent, anonymous FTP or rsync, with most protocols being unencrypted and easy to tamper with [36].

Source code is distributed in source sets, or optionally via separate means. While SSH is supported, the CVS pserver protocol is still available and also vulnerable to such attacks.

Fortunately, there is a way to obtain a list of checksums applicable to the files officially available, that is also PGP-signed by NetBSD's Security Officer [37]. It is however neither straightforward to actually verify single downloads through this file, nor is this file actually signing the files being downloaded. Partial attacks against the hashing algorithms used might be enough to fool some users.

Binary signatures long broken

pkgsrc has been claiming to support signed binary packages since 2001. While being among the first software distributions to implement this feature, it was dysfunctional until recently: an issue with uninitialized variables let the integrity checks fail silently (and the package would then not install, with no error message). This issue was found and fixed while working on the EdgeBSD Project [38].

This illustrates the lack of interest of the project (and its current users) in the secure binary distribution of packages. Even though signed packages do not solve every issue in this regard, we believe they are a necessary step forward.

Relation to The NetBSD Project

Contributing to the pkgsrc project currently requires membership to The NetBSD Foundation. This has a number of consequences, including legal and political ones since The NetBSD Foundation is a legal entity in the United States [39].

Potential contributors may very well not desire (or not be allowed) to be part of The NetBSD Foundation, or to be associated with the NetBSD Project directly.

Conflicts with respect to other platforms

pkgsrc supports a significant number of platforms. These platforms do not all have the same features and API. This is generally not an issue: not every packages is required to build or work on each and every platform, and many issues can be patched and reported upstream.

However, there are cases where significant packages evolve at a faster pace than NetBSD does, and where they only support APIs which are not yet available in NetBSD. This situation leads to a dilemma:

- either packages are kept in their older versions, and still support NetBSD while being obsolete on other platforms,
- or on the contrary, the troublesome packages are updated but they no longer work on NetBSD, in spite of being the primary target platform.

This situation is unfortunately happening already: Xorg as packaged within pkgsrc (called "modular Xorg") was kept to an old version for some time, before being updated to a version requiring availability of the KMS API (Kernel Mode Switching). As a consequence, modular Xorg no longer works on NetBSD, and running Xorg on NetBSD requires obtaining X from the base system.

Package options are not binary-friendly

A number of pkgsrc packages support build-time options, of which default values depend on pkgsrc's default configuration, the current platform, and pkgsrc's global configuration file (`/etc/mk.conf`) if available. Packages depending on libraries built with different options are likely to be different and incompatible with each other. This can be troublesome when mixing official binary packages with packages built from sources, since the options chosen may as well differ. The list of options used for the binary packages may not even be available publicly. This is a commonplace scenario for users (like when building packages from wip) and we believe this harms reliable distribution of binary packages as well.

Moreover, some options may be essential to some users while being disabled by default (like LDAP support for instance). This also can easily discourage users from adopting pkgsrc (notably in corporate environments) since it makes it then necessary to maintain in-house binary package repositories. Again, we believe this to be harming the popularity of the pkgsrc project, especially in industrial environments.

This issue was solved differently in the Debian Project for instance. There, multiple versions of such packages are available, and all built automatically from the same source package. This design has been ruled out in pkgsrc because of its original source-based nature, but also because of the additional amount of work it would be expected to create while maintaining such packages [40][41].

Packages are heavy to download and install

Packages in pkgsrc are typically built directly from the source archive of a project, with only one package being built each time. These packages must therefore contain the development interface for any library they may provide for instance. This usually involves static libraries, development files, extensive documentation and additional files and binaries that may not be relevant in most cases. This means that packages are often bigger than they may just have to be.

While this issue can easily be dismissed on regular desktop and workstation systems (where the extra resources required may be negligible), it is not the case on embedded platforms. This, again, may discourage some industrial users to consider using pkgsrc for the software distribution of their packages.

Redistributable packages do not easily build unprivileged

While it is absolutely possible (and easy) to bootstrap pkgsrc to build packages for unprivileged users, this is not true of regular redistributable packages. They are currently expected to be built (or at least created) with root privileges, which almost always largely exceeds the privileges actually necessary for this task.

A workaround for this issue was found while working on the EdgeBSD Project, thanks to the **fakeroot** command from the sysutils/fakeroot package [42].

First list of suggestions

Switching to a decentralized SCM

As mentioned above in the section called “CVS is deprecated”, rather than CVS itself we believe the actual issue when it comes to the SCM tool to use is whether it allows external developers to use the repositories from a project. While a number of decentralized tools exist and provide their respective list of features and advantages, Git is by far the most popular tool today. Both GitHub and Gitorious are immensely popular, hosting millions of repositories [43]. Joyent, one of the largest industrial users of pkgsrc [44], also has its pkgsrc repositories hosted on GitHub.

A number of attempts have been made at providing functional repositories for The

NetBSD Project using different SCM tools. Joerg Sonnenberger, a prominent NetBSD developer, is heavily involved in this task and evaluating alternatives to CVS [45]. His initial work was based on Git (in 2008) and then Fossil [46]. Since July 2011, he publishes mirrors for both the src and pkgsrc source trees on the GitHub platform.

These mirrors have a significant number of users, with the pkgsrc repository forked about 50 times on GitHub alone [47].

Provide a public SCM service for any potential contributor

It should be possible to avoid losing contributions (as mentioned above in the section called “Lost contributions to the system”) by providing a mirror of the source code repositories, where just about anyone could publish code as well.

This may cause indirect legal or security issues, as there would be a possibility for contributors to upload tainted code, malicious files or otherwise inappropriate content (which caused a premature end to no less than the Google Code Download Service [48]).

However, the popularity of this platform should also attract moderators in sufficient proportions. Additionally, some SCM tools allow for the efficient access control and removal of branches (like Git when combined with Gitolite [49]).

In any case, this service (or the external branches) should be handled separately from the main repositories, or clearly advertized as such.

Long-Term-Support (LTS) branches for pkgsrc

It appears necessary to maintain a branch of pkgsrc for both stability and security. A proposal has already been made (and declined) to the PMC to keep maintaining the release corresponding to the first quarter of every (second) year in this fashion.

This process is commonplace in major Open Source projects and software distributions (like Mozilla ESR and Ubuntu LTS) but does involve an amount of extra work, on which ground it was not adopted. One such additional task is that of maintaining a separate security vulnerability list for each LTS branch.

Rolling-release for the stable branches

It would likely help both reduce the maintenance work, please every type of user and help acceptance of minor issues to maintain only two stable releases per major release. They would consist of:

- a branch remaining as close as possible to the original major release as possible (ie accepting security and essential fixes only)
- a branch with every security issue, fix or backported addition deemed fit by the developer requesting the pull-up and the release-engineering team, in a rolling-release fashion (e.g. with updated binaries always available) and an official release-worthy tag and binaries for the latest essential change pushed.

The first choice is mandatory, because it is the safest way to allow building packages compatible with every later addition to the release. It is therefore meant for bulk builders, and for administrators requiring absolute stability to the system while also tracking security fixes.

The second choice would also have the advantage to more easily provide an update channel for the major release, and probably encourage users to use and provide feedback about the same, updated version. NetBSD already provides such a channel [50], but it is not officially available and usually not mirrored on other servers. There is no official in-place upgrade tool either, although a few are available already (like sysupgrade in pkgsrc [51]).

Towards industry standards: EdgeBSD

EdgeBSD is a young project started in the second half of 2013 [52] by Pierre Pronchery, already a NetBSD developer at the time. It was started as a way to experiment freely with changing some of the usual aspects of working with The NetBSD Project, and hopefully attracting new developers to its ecosystem.

From portability to usability

The main strength of NetBSD is certainly how much its developers care for a clean and intelligible design. This has eventually allowed the system to be easily portable, and gained the project a reputation for portability. Among these capacities, we would like to emphasize on the following:

- the system is cross-compiled by default,
- internal frameworks are carefully integrated and documented,
- hardware-level bus access is abstracted away for driver developers.

Unfortunately, in spite of these unique capabilities, improving the system for use on desktop environments has never gained much attraction (or, rather, acceptation [53]).

One of the main reasons behind this situation is certainly the amount of work required to track the latest developments of the broader Open Source desktop community, as typically driven by the GNU/Linux class of systems. Most of the recent developments require major architectural updates to the system in order to work (or even compile) optimally, like KMS (Kernel Mode Switching) with Xorg. Desktop environments take time to fully port as well (GNOME 3, XFCE), while some changes may not be desirable in the first place (like systemd, another init system).

It was demonstrated that it is indeed possible to provide a modern and stable desktop environment on top of NetBSD, on desktop and embedded environments regardless (including tablets [54] or possibly smartphones [55]). Projects like the DeforaOS desktop [56] aim at running and integrating on more systems than just GNU/Linux.

In fact, EdgeBSD may go as far as adopting a default desktop environment, providing both a controlled and maintained user experience and a reference implementation for other projects to work with. Another very important reason for this is the availability of an Integrated Development Environment within this desktop, providing developers with a known, stable and featureful work environment that is more easily supported.

One way to help this happen is to provide prospective users and developers with ready-to-use system images for a number of devices and contexts; this is also a goal of the EdgeBSD Project.

Decentralized development workflow

Given the increasing popularity of Git within the industry (as can be seen in the Android ecosystem) it was decided to use this SCM tool to host its repositories. As of today, the initial fork was obtained from Joerg Sonnenberger's work directly on GitHub. A recurring issue was found with this approach: the Git chain of commits is regularly rewritten.

Incremental updates to the Git repository format sometimes break the existing chain of commits and are published using "forced pushes". In Git, it is typically impossible to go back in time. This is however common practice among NetBSD developers through the use of the "cvs admin" command, usually to improve commit messages after the fact. In turn, this may force Git users to perform tedious manual operation.

This issue is currently being addressed: given the popularity of the Git mirrors, NetBSD's PMC - pkgsrc Management Committee - has disallowed the use of the "cvs admin" command in the pkgsrc repository. This should tremendously ease conversions from now on, although issues with the "cvs import" command may still occur.

---

[1] Initial revision for `src/Makefile`, *http://cvsweb.netbsd.org/bsdweb.cgi /src/Makefile?rev=1.1&content-type=text/x-cvsweb-markup*

[2] The NetBSD Project, *http://www.netbsd.org/*

[3] « What is NetBSD? » in the NetBSD Guide, *https://www.netbsd.org/docs/guide/en/chap-intro.html*

[4] The EdgeBSD Project, *https://www.edgebsd.org/*

[5] The DeforaOS Project, *https://www.defora.org/*

[6] Open Sources: Voices from the Open Source Revolution, *http://oreilly.com/catalog /opensources/book/kirkmck.html*

[7] The NetBSD system, *https://www.netbsd.org/about/system.html*

[8] Linux distribution, *https://en.wikipedia.org/wiki/Linux_ distribution*

[9] « NetBSD development model » in EdgeBSD at FrOSCon 2013, *http://people.defora.org /~khorben/papers/froscon2013/EdgeBSD.pdf*

[10] « Criticism » in Concurrent Versions System, *https://en.wikipedia.org /wiki/Concurrent_ Versions_ System#Criticism*

[11] Products based on NetBSD, *http://www.netbsd.org/gallery/products.html*

[12] « Who Uses FreeBSD? » in the FreeBSD Handbook, *http://www.freebsd.org /doc/en_ US.ISO8859-1/books/handbook/nutshell.html#introduction-nutshell-users*

[13] « Commercial Users » in OpenBSD Users, *http://www.openbsd.org/users.html*

[14] Jared D. McNeill's misc repository on NetBSD's FTP server, *http://ftp.netbsd.org /pub/NetBSD/misc/jmcneill/rpi/*

[15] The EdgeBSD Project, *https://www.edgebsd.org/*

[16] The DeforaOS Project, *https://www.defora.org/*

[17] The deforaos-desktop meta-package for pkgsrc, *http://cvsweb.netbsd.org/bsdweb.cgi/pkgsrc*

/meta-pkgs/deforaos-desktop/

[18] Touch your NetBSD at EHSM 2012, *http://people.defora.org/~khorben/papers/ehsm2012 /Touch%20your%20NetBSD.pdf*

[19] Call your NetBSD at BSDCan 2013, *http://www.bsdcan.org/2013/schedule/events /381.en.html*

[20] The DeforaOS Open Source Project on Ohloh, *http://www.ohloh.net/p/DeforaOS*

[21] NetBSD release glossary and graphs, *http://www.netbsd.org/releases/release-map.html*

[22] Concurrent Versions System on Wikipedia, *http://en.wikipedia.org /wiki/Concurrent_ Versions_ System*

[23] src/doc/BRANCHES, *http://cvsweb.netbsd.org/bsdweb.cgi/~checkout~/src/doc /BRANCHES?content-type=text/plain*

[24] Concurrent Versions System on Wikipedia: Criticism, *http://en.wikipedia.org /wiki/Concurrent_ Versions_ System#Criticism*

[25] pkgsrc, *http://www.pkgsrc.org/*

[26] 10 years of pkgsrc - pkgsrc and the concepts of package management 1997-2007, *https://www.netbsd.org/gallery/10years.html*

[27] pkgsrc: The NetBSD Packages Collection, Supported platforms, *http://www.netbsd.org /docs/software/packages.html#platforms*

[28] pkgsrc-2013Q4 branched, *http://mail-index.netbsd.org/pkgsrc-users/2013/12/31 /msg019107.html*

[29] BitchX Trojan Horse Vulnerability, *https://www.juniper.net/security/auto/vulnerabilities /vuln7333.html*

[30] Strange Loops: Ken Thompson and the Self-referencing C Compiler, *http://scienceblogs.com/goodmath/2007/04/15/strange-loops-dennis-ritchie-a/*

[31] NetBSD CVS Repositories, *http://cvsweb.netbsd.org/bsdweb.cgi/*

[32] The pkgsrc-wip project, *http://pkgsrc-wip.sourceforge.net/*

[33] The first step away from CVS on tech-repository, *http://mail-index.netbsd.org/tech-repository/2010/01/06/msg000204.html*

[34] tech-repository on the NetBSD Wiki, *http://wiki.netbsd.org/mailing-lists/tech-repository/*

[35] Rubberhose mirror (at the Internet Archive), *http://web.archive.org/web/20110726185300 /http://iq.org/~proff/rubberhose.org/*

[36] NetBSD Mirror Sites, *http://www.netbsd.org/mirrors/*

[37] NetBSD-6.1.3_hashes.asc, *ftp://ftp.netbsd.org/pub/NetBSD/security/hashes/NetBSD-6.1.3_hashes.asc*

[38] NetBSD Problem Report #48194, *http://gnats.netbsd.org/48194*

[39] The NetBSD Foundation, Inc., *http://www.netbsd.org/foundation/*

[40] "New options for freeswitch" thread on the "tech-pkg" mailing-list, *http://mail-index.netbsd.org/tech-pkg/2012/10/17/msg010217.html*

[41] "Package split or package options?" thread on the "tech-pkg" mailing-list, *http://mail-index.netbsd.org/tech-pkg/2013/12/04/msg012303.html*

[42] "pkgsrc/mksandbox.sh" from the EdgeBSD Project, *http://git.edgebsd.org/gitweb/?p=edgebsd-infrastructure.git;a=blob;f=pkgsrc/mksandbox.sh*

[43] 10 million repositories, GitHub, *https://github.com/blog/1724-10-million-repositories*

[44] Joyent Packages Documentation, *http://pkgsrc.joyent.com/*

[45] Fossil and NetBSD, *http://www.sonnenberger.org/2010/10/24/fossil-and-netbsd/*

[46] Fossil, *http://www.fossil-scm.org/*

[47] The pkgsrc network graph on GitHub, *https://github.com/jsonn/pkgsrc/network*

[48] A Change to Google Code Download Service, *http://google-opensource.blogspot.de/2013/05/a-change-to-google-code-download-service.html*

[49] Gitolite: Hosting git repositories, *http://gitolite.com/gitolite/*

[50] Daily builds of the netbsd-6 branch on `ftp.netbsd.org`, *http://nyftp.netbsd.org/pub/NetBSD-daily/netbsd-6/*

[51] Introducing sysupgrade for NetBSD, *http://julipedia.meroh.net/2012/08/introducing-sysupgrade.html*

[52] "EdgeBSD was introduced at FrOSCon" 2013, *https://www.edgebsd.org/edgebsd/news/6/edgebsd%20was%20introduced%20at%20froscon*

[53] Archives for the netbsd-desktop mailing-list, *http://mail-index.netbsd.org/netbsd-desktop/*

[54] "Touch your NetBSD" presentation at EHSM 2012, *http://mail-index.netbsd.org/netbsd-advocacy/2013/01/13/msg000512.html*

[55] "Call your NetBSD" presentation at BSDCan 2013, *http://www.bsdcan.org/2013/schedule/events/381.en.html*

[56] "Graphical environment" from DeforaOS, *http://www.defora.org/os/wiki/3426/graphical%20environment*

How FreeBSD Boots: a soft-core MIPS perspective

Brooks Davis, Robert Norton, Jonathan Woodruff, Robert N. M. Watson

Abstract

We have implemented an FPGA soft-core, multithreaded, 64-bit MIPS R4000-style CPU called BERI to support research on the hardware/software interface. We have ported FreeBSD to this platform including support for multithreaded and soon multicore CPUs. This paper describes the process by which a BERI system boots from CPU startup through the boot loaders, hand off to the kernel, and enabling secondary CPU threads. Historically, the process of booting FreeBSD has been documented from a user perspective or at a fairly high level. This paper aims to improve the documentation of the low level boot process for developers aiming to port FreeBSD to new targets.

1. Introduction

From its modest origins as a fork of 386BSD targeting Intel i386 class CPUs, FreeBSD has been ported to a range of architectures including DEC Alpha[1], AMD x86_64 (aka amd64), ARM, Intel IA64, MIPS, PC98, PowerPC, and Sparc64. While the x86 and Alpha are fairly homogeneous targets with mechanics for detecting and adapting to specific board and peripheral configurations, embedded systems platforms like ARM, MIPS, and PowerPC are much more diverse. Porting to a new MIPS board often requires adding support for a new System on Chip (SoC) or CPU type with different interrupt controllers, buses, and peripherals. Even if the CPU is supported, boot loaders and associated kernel calling conventions differ significantly between boards.

We have ported FreeBSD/MIPS to BERI, an open-source MIPS R4000-style[1] FPGA-based soft-core processor that we have developed. This required a range of work including boot loader support, platform startup code, a suite of device drivers (including the PIC), but also adapting FreeBSD's existing FDT support to FreeBSD/MIPS. We currently run FreeBSD/BERI under simulation, on an Altera Stratix IV FPGA on a Terasic DE4 FPGA board, and on an Xilinx Virtex-5 FPGA on the NetFPGA-10G platform. The majority of our peripheral work has been on simulation and the DE4 platform. FreeBSD BERI CPU support is derived from the MALTA port with some inspiration from the sibyte port.

Based on our experiences bringing up FreeBSD on BERI we have documented the way we boot FreeBSD from the firmware embedded in the CPU to userspace to provide a new view on the boot process. FreeBSD is generally very well documented between project documentation and books like the Design and Implementation of the FreeBSD Operating System [3], but detailed documentation of the boot process has remained a gap. We believe this paper well help porters gain a high level understanding of the boot process and go allow interested users to understand the overall process without the need to create an new port.

[1]Removed in 2006.

Figure 1: BERIpad with application launcher

The rest of this paper narrates the boot process with a special focus on the places customization was required for BERI. We begin by describing the BERI platform (Section 2), and then in detail documents the kernel architecture-specific boot process for FreeBSD on BERI: boot loader (Section 3) and kernel boot process (Section 4). In the interest of brevity many aspects of boot are skipped and most that are not platform or port-specific are ignored. Some platform-specific components such as the MIPS pmap are not covered. The goal is to provide a guide to those pieces someone porting to a new, but relatively conventional MIPS CPU would need to fill in. Porters interested in less conventional CPUs will probably want to examine the NLM and RMI ports in `mips/nlm` and `mips/rmi` for examples requiring more extensive modifications.

2. The BERIpad platform

We have developed BERI as a platform to enable experiments on the hardware-software interface such as our ongoing work on hardware supported capabilities in the CHERI CPU[5]. Our primary hardware target has been a tablet based on the Terasic DE4 FPGA board with a Terasic MTL touch screen and integrated battery pack. The design for the tablet has been released as open source at `http://beri-cpu.org/`. The CPU design will be released in the near future. The modifications to FreeBSD—except for MP—support have been merged to FreeBSD 10.0. The tablet and the internal architecture of BERI are described in detail in the paper *The BERIpad Tablet* [2] The following excerpt provides a brief overview of BERI and the drivers we have developed.

> The Bluespec Extensible RISC Implementation (BERI) is currently an in-order core with a 6-stage pipeline which implements the 64-bit MIPS

> instruction set used in the classic MIPS R4000. Some 32-bit compatibility features are missing and floating point support is experimental. Achievable clock speed is above 100MHz on the Altera Stratix IV and average cycles per instruction is close to 1.2 when booting the FreeBSD operating system. In summary, the high-level design and performance of BERI is comparable to the MIPS R4000 design of 1991, though the design tends toward extensibility and clarity over efficiency in the micro-architecture.
>
> ...
>
> We developed device drivers for three Altera IP cores: the JTAG UART (altera jtag uart), triple-speed MAC (atse), and SD Card (altera sdcard), which implement low-level console/tty, Ethernet interface, and block storage classes. In addition, we have implemented a generic driver for Avalon-attached devices (avgen), which allows memory mapping of arbitrary bus-attached devices without interrupt sources, such as the DE4 LED block, BERI configuration ROM, and DE4 fan and temperature control block.
>
> Finally, we have developed a device driver for the Terasic multitouch display (terasic mtl), which implements a memory-mapped pixel buffer, system console interface for the text frame buffer, and memory-mapped touchscreen input FIFO. Using this driver, UNIX can present a terminal interface, but applications can also overlay graphics and accept touch input.

In addition to the drivers described above, made extensive modifications to the exiting `cfi(4)` (Common Flash Interface) driver to fully support Intel NOR flash and improve write performance.

2.1. Flat Device Tree

Most aspects of BERI board configuration is described in a Flat Device Trees (FDT) which are commonly used on PowerPC and ARM-based systems [4]. Currently a Device Tree Blob (DTB) is built into each FreeBSD kernel and describes a specific hardware configuration. Each DTB is built from a device tree syntax (DTS) file by the device tree compiler[2] before being embedded in the kernel. Figure 2 exerpts the DTS file `boot/fdt/dts/beripad-de4.dts` and includes the BERI CPU, 1GB DRAM, programmable interrupt controller (PIC), hardware serial port, JTAG UART, SD card reader, flash partition table, gigabit Ethernet, and touchscreen.

3. The early boot sequence

The common FreeBSD boot sequence begins with CPU firmware arranging to run the FreeBSD `boot2` second-stage boot loader which in turn loads `/boot/loader` which loads

[2]`dtc(1)`

```
model = "SRI/Cambridge BeriPad (DE4)";
compatible = "sri-cambridge,beripad-de4";
cpus {
  cpu@0 {
    device-type = "cpu";
    compatible = "sri-cambridge,beri";
  };
};
soc {
  memory {
    device_type = "memory";
    reg = <0x0 0x40000000>;
  };
  beripic: beripic@7f804000 {
    compatible = "sri-cambridge,beri-pic";
    interrupt-controller;
    reg = <0x7f804000 0x400 0x7f806000 0x10
      0x7f806080 0x10 0x7f806100 0x10>;
  }
  serial@7f002100 {
    compatible = "ns16550";
    reg = <0x7f002100 0x20>;
  };
  serial@7f000000 {
    compatible = "altera,jtag_uart-11_0";
    reg = <0x7f000000 0x40>;
  };
  sdcard@7f008000 {
    compatible = "altera,sdcard_11_2011";
    reg = <0x7f008000 0x400>;
  };
  flash@74000000 {
    partition@20000 {
      reg = <0x20000 0xc00000>;
      label = "fpga0";
    };
    partition@1820000 {
      reg = <0x1820000 0x027c0000>;
      label = "os";
    };
  };
  ethernet@7f007000 {
    compatible = "altera,atse";
    reg = <0x7f007000 0x400 0x7f007500 0x8
      0x7f007520 0x20 0x7f007400 0x8
      0x7f007420 0x20>;
  };
  touchscreen@70400000 {
    compatible = "sri-cambridge,mtl";
    reg = <0x70400000 0x1000
      0x70000000 0x177000 0x70177000 0x2000>;
  };
};
```

Figure 2: Excerpt from Flat Device Tree (FDT) description of the DE4-based BERI tablet.

the kernel and kernel modules. Finally the kernel boots which is described in Section 4.

3.1. Miniboot

At power on or after reset, the CPU sets the program counter of at least one thread to the address of a valid program. From the programmer perspective the process by which this occurs is essentially magic and of no particular importance. Typically the start address is some form of read-only or flash upgradable firmware that allows for early CPU setup and may handle details such as resetting cache state or pausing threads other than the primary thread until the operating system is ready to handle them. In many systems, this firmware is responsible for working around CPU bugs.

On BERI this code is known as miniboot for physical hardware and simboot for simulation. Miniboot is compiled with the CPU as a read-only BRAM. It is responsible for settings registers to initial values, setting up an initial stack, initializing the cache by invalidating the contents, setting up a spin table for MP boot, running code to initialize the HDMI output port on the DE4 tablet, and loading a kernel from flash or waiting for the next bit of code to be loaded by the debug unit and executing that. With BERI we are fortunate to not need to work around CPU bugs in firmware since we can simply fix the hardware.

Miniboot's kernel loading and boot behavior is controlled by two DIP switches on the DE4. If DIP0 is off or miniboot with compiled with `-DALWAYS_WAIT` then we spin in a loop waiting for the general-purpose register `t1` to be set to `0` using JTAG. This allows the user to control when the board starts and given them an opportunity to load a kernel directly to DRAM before boot proceeds. DIP1 controls the relocation of a kernel from flash. If the DIP switch is set, the kernel is loaded from a flash at offset of `0x2000000` to `0x100000` in DRAM. Otherwise, the user is responsible for loading a kernel to DRAM by some other method. Currently supported mechanisms are described in the BERI Software Reference [7].

The kernel loading functionality occurs only on hardware thread 0. In other hardware threads, miniboot skips this step and enter a loop waiting for the operating system to send them a kernel entry point via the spin-table. Multithread and multicore boot is discussed in more detail in section 4.3.

Before miniboot enters the kernel it clears most registers and sets `a0` to `argc`, `a1` to `argv`, `a2` to `env`, and `3` to the size of system memory. In practice `argc` is `0` and `argv` and `env` are `NULL`. It then assumes that an ELF64 object is located at `0x100000`, loads the entry point from the ELF header, and jumps to it.

We intend that miniboot be minimal, but sufficiently flexible support debugging of various boot layouts as well as loading alternative code such as self contained binaries. This allows maximum flexibility for software developers who may not be equipped to generate new hardware images.

3.2. boot2

On most FreeBSD systems two more boot stages are interposed between the architecture dependent boot code and the kernel. The first of these is `boot2`[3], the second stage bootstrap, which has a mechanism for accessing local storage and has code for read-only access to a limited set of file systems (usually one of UFS or ZFS). Its primary job is to load the loader and to pass arguments to it. By default it loads `/boot/loader`, but the user can specify an alternative disk, partition, and path.

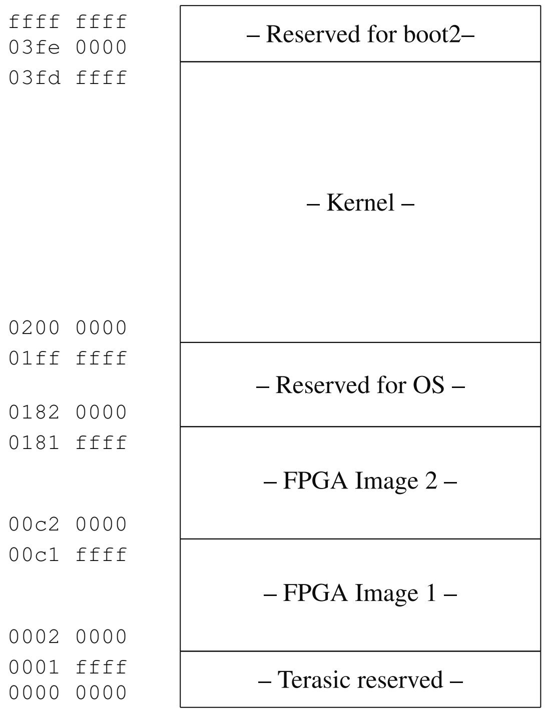
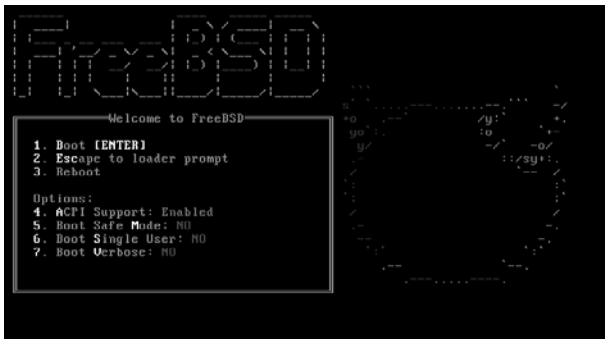


Figure 3: Layout of the DE4 flash

We have ported `boot2` to BERI, creating three 'microdrivers' allowing JTAG UART console access, and use of CFI or the SD card to load `/boot/loader` or the kernel. These microdrivers substitute for boot device drivers provided by the BIOS on x86 or OpenFirmware on SPARC. It also supports jumping to an instance of `/boot/loader` loaded via JTAG. In our current implementation, `boot2` is linked to execute at 0x100000 and loaded from CFI flash as the kernel currently is allowing it to be used with an unmodified miniboot. In the future, we plan to place a similar version of `boot2` at `0x03fe0000`, a 128K area reserved for its use. This will allow a normal filesystem to be placed in CFI flash from `0x1820000`, which might contain the full boot loader, a kernel, etc. Currently, we use `boot2` to load `/boot/loader` from the SD card, which offers an experience more like conventional desktop/server platforms than a conventional embedded target.

Many versions of boot2 exist, targeted at different architectures. The version of boot2 in BERI is derived from the x86 boot2, and is hence (marginally) more feature-rich than ones targeted at more space-constrained embedded architectures.

[3] `boot(8)`



```
Welcome to FreeBSD

1. Boot [ENTER]
2. Escape to loader prompt
3. Reboot

Options:
4. ACPI Support: Enabled
5. Boot Safe Mode: NO
6. Boot Single User: NO
7. Boot Verbose: NO
```

Figure 4: FreeBSD loader boot menu

3.3. loader

The third common boot stage is the `loader(8)`. The loader is in effect a small kernel whose main job is to set up the environment for the kernel and load the kernel and any configured modules from the disk or network. The loader contains a Forth interpreter based on FICL[4]. This interpreter it used to provide the boot menu shown in Figure 4, parses configuration files like `/boot/loader.conf`, and implements functionality like `nextboot(8)`. In order to do this, the loader also contains drivers to access platform-specific devices and contains implementations of UFS and ZFS with read and limited write support. On x86 systems that means BIOS disk access and with the pxeloader network access via PXE. On BERI this currently includes a basic driver for access to the CFI flash found on the DE4.

We have ported the loader to FreeBSD/MIPS and share the SD card and CFI microdrivers with `boot2` to allow kernels to be loaded from CFI flash or SD card. We currently load the kernel from the SD card. We hope to eventually add a driver for the onboard Ethernet device to allow us to load kernels from the network.

The loader's transition to the kernel is much the same as miniboot. The kernel is loaded to the expected location in the memory, the ELF header is parsed, arguments are loaded into registers, and the loader jumps into the kernel.

3.4. The bootinfo structure

In order to facilitate passing information between `boot2`, `/boot/loader`, and the kernel a pointer to a `bootinfo` structure is between them allowing information such as memory size, boot media type, and the locations of preloaded modules to be shared. In the future we will add support for passing a pointer to the FDT device database that will be embedded in the CPU or stored separately in flash.

4. The path to usermode

This section narrates the interesting parts of the FreeBSD boot process from a MIPS porter's perspective. In the electronic version of this document most of the paths, function names, and symbols are links to appropriate parts of `http://fxr.watson.org` to enable further exploration.

[4]`http://ficl.sourceforge.net`

4.1. Early kernel boot

The FreeBSD MIPS kernel enters at `_start` in the `_locore` function defined in `mips/mips/locore.S`. `_locore` performs some early initialization of the CP0 registers, sets up an initial stack and calls the platform-specific startup code in `platform_start`.

On BERI `platform_start` saves the argument list, environment, and pointer to `struct bootinfo` passed by the loader. BERI kernels also support an older boot interface, in which memory size is passed as the fourth argument (direct from miniboot). It then calls the common mips function `mips_postboot_fixup` which provides kernel module information for manually loaded kernels and corrects `kernel_kseg0_end` (the first usable address in kernel space) if required. Per CPU storage is then initialized for the boot CPU by `mips_pcpu0_init`. Since BERI uses Flat Device Tree (FDT) to allow us to configure otherwise non-discoverable devices `platform_start` then the locates the DTB and initializes FDT. This is the norm for ARM and PowerPC ports, but is currently uncommon on MIPS ports. We expect it to become more popular over time. The `platform_start` function then calls `mips_timer_early_init` to set system timer constants, currently to a hardcoded 100MHz, eventually this will come from FDT. The console is set up by `cninit` and some debugging information is printed. The number of pages of real memory is stored in the global variable `realmem`[5]. The BERI-specific `mips_init`[6] function is then called to do the bulk of remaining early setup.

BERI's `mips_init` is fairly typical. First, memory related parameters are configured including laying out the physical memory range and setting a number of automatically tuned parameters in the general functions `init_param1` and `init_param2`. The MIPS function `mips_cpu_init` performs some optional per-platform setup (nothing on BERI), identifies the CPU, configures the cache, and clears the TLB. The MIPS version of `pmap_bootstrap` is called to initialize the pmap. Thread 0 is instantiated by `mips_proc0_init` which also allocates space for dynamic per CPU variables. Early mutexs including the legacy Giant lock are initialized in `mutex_init` and the debugger is initialized in `kdb_init`. If so configured the kernel may now drop into the debugger or, much more commonly, return and continue booting.

Finally `mips_timer_init_params` is called to finish setting up the timer infrastructure before `platform_start` returns to `_locore`. `_locore` switches to the now configured `thread0` stack and calls `mi_startup` never to return.

4.2. Calling all SYSINITS

The job of `mi_startup` is to initialize all the kernel's subsystems in the right order. Historically `mi_startup` was called `main` and the order of initialization was hard coded.

[5]The `btoc` macro converts bytes to clicks which in FreeBSD are single pages. Mach allowed multiple pages to be managed as a virtual page.

[6]Most ports have one of these, but it seems to be misnamed as it is not MIPS generic code.

```
static void
print_caddr_t(void *data)
{
        printf("%s", (char *)data);
}
SYSINIT(announce, SI_SUB_COPYRIGHT,
    SI_ORDER_FIRST, print_caddr_t,
    copyright);
```

Figure 5: Implementation of copyright message printing on FreeBSD boot.

This was obviously not scalable so a more dynamic registration mechanism called `SYSINIT(9)` was created. Any code that needs to be run which at startup can use the `SYSINIT` macro to cause a function to be called in a sorted order to boot or on module load. The sysinit implementation relies on the 'linker set' feature, in which constructors/destructors for kernel subsystems and modules are tagged in the ELF binary so that the kernel linker can find them during boot, module load, module unload, and kernel shutdown.

The implementation of `mi_startup` is simple. It sorts the set of sysinits and then runs each in turn marking each done when it is complete. If any modules are loaded by a sysinit, it resorts the set and starts from the beginning skipping previous run entries. The end of `mi_startup` contains code to call `swapper`, this code is never reached as the last sysinit never return. One implementation detail of note in `mi_startup` is the use of bubble sort to sort the sysinits due to the fact that allocators are initialized via sysinits and thus not yet available.

Figure 5 shows a simple example of a sysinit. In this example `announce` is the name of the individual sysinit, `SI_SUB_COPYRIGHT` is the subsystem, `SI_ORDER_FIRST` is the order within the subsystem, `print_caddr_t` is the function to call, and `copyright` is an argument to be passed to the function. A complete list of subsystems and orders within subsystems can be found in `sys/kernel.h`. As of this writing there are more than 80 of them. Most are have little or no port-specific function and thus are beyond the scope of this paper. We will highlight sysinits with significant port-specific content.

The first sysinit of interest is `SI_SUB_COPYRIGHT`. It does not require porting specifically, but reaching it and seeing the output is a sign of a port nearing completion since it means low level consoles work and the initial boot described above is complete. The MIPS port has some debugging output earlier in boot, but on mature platforms the copyright message is the first output from the kernel. Figure 6 shows the three messages printed at `SI_SUB_COPYRIGHT`.

The next sysinit of interest to porters is `SI_SUB_VM`. The MIPS `bus_dma(9)` implementation starts with a set of statically allocated maps to allow it to be used early in boot. The function `mips_dmamap_freelist_init` adds the static maps to the free list at `SI_SUB_VM`. The ARM platform does similar work, but does require `malloc` and thus runs `busdma_init` at `SI_SUB_KMEM` instead.

Further `bus_dma(9)` initialization takes place at `SI_SUB_LOCK` in the platform-specific, but often identical, `init_bounce_pages` function. It initializes some counters, lists, and the bounce page lock.

All ports call a platform-specific `cpu_startup` function at `SI_SUB_CPU` set up kernel address space and perform some initial buffer setup. Many ports also perform board, SoC, or CPU-specific setup such as initializing integrated USB controllers. Ports typically print details of physical and virtual memory, initialize the kernel virtual address space with `vm_ksubmap_init`, the VFS buffer system with `bufinit`, and the swap buffer list with `vm_pager_bufferinit`. On MIPS the platform-specific `cpu_init_interrupts` is also called to initialize interrupt counters.

Most platforms have their own `sf_buf_init` routine to allocate `sendfile(2)` buffers and initialize related locks. Most of these implementations are identical.

The bus hierarchy is established and device probing is performed at the `SI_SUB_CONFIGURE` stage (aka autoconfiguration). The platform-specific portions of this stage are the `configure_first` function called at `SI_ORDER_FIRST` which attaches the nexus bus to the root of the device tree, `configure` which runs at `SI_ORDER_THIRD` and calls `root_bus_configure` to probe and attach all devices, and `configure_final` which runs at `SI_ORDER_ANY` `cninit_finish` to finish setting up the console with `cninit_finish`, and clear the `cold` flag. On MIPS and some other platforms `configure` also calls `intr_enable` to enable interrupts, A number of console drivers complete their setup with explicit sysinits at `SI_SUB_CONFIGURE` and many subsystems like CAM and `acpi(4)` perform their initialization there.

Each platform registers the binary types it supports at `SI_SUB_EXEC`. The primarily consists of registering the expected ELF header values. On a uniprocessor MIPS this is the last platform-specific sysinit.

The final sysinit is an invocation of the `scheduler` function at `SI_SUB_RUN_SCHEDULER` which attempts to swap in processes. Since `init(8)` was previously created by `create_init` at `SI_SUB_CREATE_INIT` and made runnable by `kick_init` at `SI_SUB_KTHREAD_INIT` starting the scheduler results in entering userland.

4.3. Multiprocessor Support

Multiprocessor systems follow the same boot process as uniprocessor systems with a few added sysinits to enable and start scheduling the other hardware threads. These threads are known as application processors (APs).

The first MP-specific sysinit is a call to `mp_setmaxid` at `SI_SUB_TUNABLES` to initialize the `mp_ncpus` and `mp_maxid` variables. The generic `mp_setmaxid` function calls the platform-specific `cpu_mp_setmaxid`. On MIPS `cpu_mp_setmaxid` calls the port-specific `platform_cpu_mask` to fill a `cpuset_t` with a mask of all available cores or threads. BERI's implementation extracts a list of cores from the DTB and verifies that they support the spin-table enable method. It further verifies that the spin-table entry is properly initialized or the thread is

```
Copyright (c) 1992-2013 The FreeBSD Project.
Copyright (c) 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994
        The Regents of the University of California. All rights reserved.
FreeBSD is a registered trademark of The FreeBSD Foundation.
```

Figure 6: Copyright and trademark messages in FreeBSD 10

```
struct spin_entry {
        uint64_t        entry_addr;
        uint64_t        a0;
        uint32_t        rsvd1;
        uint32_t        pir;
        uint64_t        rsvd2;
};
```

Figure 7: Definition of a spin_entry with explicit padding and the argument variables renamed to match MIPS conventions.

ignored.

The initialization of APs is accomplished by the `mp_start` function called at `SI_SUB_CPU` after `cpu_startup`. If there are multiple CPUs it calls the platform-specific `cpu_mp_start` and upon return prints some information about the CPUs. The MIPS implementation of `cpu_mp_start` iterates through the list of valid CPU IDs as reported by `platform_cpu_mask` and attempts to start each one except it self as determined by `platform_processor_id`[7] with the platform-specific `start_ap`. The port-specific `platform_start_ap`'s job is to cause the AP to run the platform-specific `mpentry`. When runs successfully, it increments the `mp_naps` variable and `start_ap` waits up to five seconds per AP for this to happen before giving up on it.

A number of mechanisms has been implemented to instruct a CPU to start running a particular piece of code. On BERI we have chosen to implement the spin-table method described in the ePAPR 1.0 specification[4] because it is extremely simple. The spin-table method requires that each AP have an associated `spin_entry` structure located somewhere in the address space and for that address to be recorded in the DTB. The BERI specific definition of `struct spin_entry` can be found in Figure 7. At boot the `entry_addr` member of each AP is initialized to `1` and the AP waits for the LSB to be set to `0` at which time it jumps to the address loaded in `entry_addr` passing `a0` in register `a0`. We implement waiting for `entry_addr` to change with a loop in miniboot. In BERI's `platform_cpu_mask` we look up the `spin_entry` associated with the requested AP, set the `pir` member to the CPU id and then assign the address of `mpentry` to the `entry_addr` member.

The MIPS implementation of `mpentry` is assembly in `mips/mips/mpboot.S`. It disables interrupts, sets up a stack, and calls the port-specific `platform_init_ap` to set up the AP before entering the MIPS-specific `smp_init_secondary` to complete per-CPU setup and await the end of the boot process. A typical MIPS implementation of `platform_init_ap` sets up interrupts on the AP and enables the clock and IPI interrupts. On BERI we defer IPI setup until after device probe because our programmable interrupt controller (PIC) is configured as an ordinary device and thus can not be configured until after `SI_SUB_CONFIGURE`.

The MIPS-specific `smp_init_secondary` function initializes the TLB, setups up the cache, and initializes per-CPU areas before incrementing `mp_naps` to let `start_ap` know that it has finished initialization. It then spins waiting for the flag `aps_ready` to be incremented indicating that the boot CPU has reached `SI_SUB_SMP` as described below. On BERI it then calls `platform_init_secondary` to route IPIs to the AP and set up the IPI handler. The AP then sets its thread to the per-CPU idle thread, increment's `smp_cpus`, announces it self on the console, and if it is the last AP to boot, sets `smp_started` to inform `release_aps` that all APs have booted and the `smp_active` flag to inform a few subsystems that we are running with multiple CPUs. Unless it was the last AP to boot it spins waiting for `smp_started` before starting per-CPU event timers and entering the scheduler.

The final platform-specific sysinit subsystem is `SI_SUB_SMP` which platform-specific `release_aps` functions are called to enable IPIs on the boot CPU, inform previously initialized APs that they can start operating, and spin until they do so as described above. In the MIPS case this means atomically setting the `aps_ready` flag to `1` and spinning until `smp_started` is non-zero.

4.4. A word on IPIs

In multiprocessor (MP) systems CPUs communicate with each other via Inter-Processor Interrupts (IPIs). A number of IPI mechanisms exist, with FreeBSD MIPS using the simplest model, a per-CPU integer bitmask of pending IPIs and a port-specific mechanism for sending an interrupt, almost always to hardware interrupt 4. This is implemented by the `ipi_send` which is used by the public `ips_all_but_self`, `ipi_selected`, and `ipi_cpu` functions. MIPS IPIs are handled by `mips_ipi_handler` which clears the interrupt with a call to `platform_ipi_clear`, reads the set of pending IPIs, and handles each of them.

On BERI IPIs are implemented using the BERI PIC's soft interrupt sources. IPIs are routed by `beripic_setup_ipi`, sent by `beripic_send_ipi`, and cleared by `beripic_clear_ipi`. These functions are accessed via `kobj(9)` through the FDT_IC interface defined in `dev/fdt/fdt_ic_if.m`. The internals of BERI PIC are described in the BERI Hardware Reference[6].

[7]Implemented in `mips/beri/beri_asm.S` on BERI.

5. Conclusion

Porting FreeBSD to a new CPU, even within a previously supported family, is a significant undertaking. We hope this paper will help prospective porters orient themselves before they begin the process. While we have focused on a MIPS ports, the code structure in other platforms–especially ARM–is quite similar.

5.1. Acknowledgments

We would like to thank our colleagues - especially Jonathan Anderson, David Chisnall, Nirav Dave, Wojciech Koszek, Ben Laurie, A Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Peter G. Neumann, Philip Paeps, Michael Roe, and Bjoern Zeeb.

This work is part of the CTSRD Project that is sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] J. Heinrich, *MIPS R4000 Microprocessor User's Manual*, 1994, second Edition.

[2] A. T. Markettos, J. Woodruff, R. N. M. Watson, B. A. Zeeb, B. Davis, and S. W. Moore, "The BERIpad tablet: Open-source construction, CPU, OS and applications," 2013.

[3] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[4] Power.org, *Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR)*, 2008.

[5] R. Watson, P. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. Moore, S. Murdoch, P. Paeps *et al.*, "CHERI: A Research Platform Deconflating Hardware Virtualization and Protection," in *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.

[6] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, R. Norton, and J. Woodruff, "BERI bluespec extensible RISC implementation: Hardware reference," 2014, forthcoming publication.

[7] R. N. M. Watson, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, and J. Woodruff, "BERI bluespec extensible RISC implementation: Software reference," 2014, forthcoming publication.

Adapting OSX to the enterprise

Jos Jansen
Deputy Director of Engineering,
Snow B.V., Geldermalsen, The Netherlands

Abstract

How we try to manage OSX desktops while maintaining security and preserving our sanity

In this paper I examine ways and means of deploying and managing OSX in a business environment while taking account of the first five of the Twenty Critical Security Controls for Effective Cyber Defense[1] in a cost-effective manner.

I will describe efforts to steadily and gradually improve the quality of our desktop and server infrastructure with respect to ease of administration and security, leveraging well-understood tools and avoiding or mitigating excessive contraints on user-friendliness.

1. About Snow

Snow B.V. was established in 1997 and has grown into the largest UNIX contractor in The Netherlands, employing approximately 110 engineers and consultants and some 30 office staff. Among our customers are government institutions, major Dutch corporations and multinational enterprises and a number of NGO. We specialise in UNIX and network administration, storage and security.

2. The network

The network is quite simple and consists of VLANs for consultants and engineers, the DMZ and wireless guests. In addition, there is a "no man's LAN" for services which need to be accessible from all other VLANs and an office VLAN behind its own packet filter. 802.1x with FreeRadius and an OpenLDAP-backend is active on all points of access to the LANs.

3. The server side

We run Linux on a number of (mostly virtualized) servers. There are good reasons for this; a majority of our engineers are (to my regret) much more familiar with Linux than with the BSD family and receive assignments for improvement projects when between jobs.

Most servers which I truly care about — *i.e.* those providing mission-critical services in the protected LAN — run FreeBSD or, in the case of boundary control, OpenBSD. For general usage I prefer FreeBSD as the *ports* tree is rather more verdant and I can have ZFS. These servers are managed with the community edition of CFEngine[2] with respect to file and directory permissions, checksums of important files, restarting processes (which, of course, should never happen, as FreeBSD processes aren't supposed to crash), the propagation of public keys and the maintenance of certain elementary configuration files. This is simple and essentially foolproof, provided that all amendments to the CFEngine configuration files have been properly tested prior to deployment.

3.1. Storage

We are not a very large company; our office document history and databases since 1997 occupy less

1. Twenty Critical Security Controls for Effective Cyber Defense, http://www.sans.org/critical-security-controls/

2. CFEngine, http://cfengine.com

than 100GB of storage. However, we are required by law to maintain a mail archive which takes another 300GB. We are very concerned with the mail archive as office staff are apt to permanently delete emails accidentally and we may want it back months or years later.[3]

As for the mail store for the IMAP server, 270 GB sufficed per September 2013 for all user accounts but this grew to 307 GB at the beginning of this year. There is some redundancy as I provide separate PDA accounts upon request to office employees.

3.1.1. ***Why PDA accounts?.*** Very early on, when I used primitive smartphones over 2G connections, I found that it was not wise to keep a full cache of an IMAP mail store on a mobile device, and the benefits of using an Apple iPhone have not convinced me of the opposite. Therefore I use *ports/procmail* to copy incoming mail to a PDA mailbox where I could delete mail to my satisfaction.

Users now are entitled to an optional PDA mail account in the form of pdaUSER@snow.nl and are free to delete mail from this account; what they think of as their "desktop" mail store is not affected.

In theory, therefore, I could run the entire company off a machine with a 1 TB disk.

In practice, however, I need a huge amount of storage to serve all LANs in a secure and redundant manner; OSX home directories in particular are horrendous in size.

I started out with 2 RAID cabinets with a SuperMicro main board and 48G of RAM (ZFS is greedy), triple-mirrored 2TB disks, an Intel SSD as L2ARC device and two small Intel SSD as mirrored log devices. For historical reasons I ran the business directories on a UFS-volume on ZFS as ZFS ACL are not quite compatible with POSIX ACL; this has since been corrected as the POSIX ACL were too labour-intensive to maintain.

It has been established that a local LDAP replica is indispensable; just try to run a recursive *stat(1)* on a directory tree without one. I run replicas on all servers which require access to LDAP user data (and, of course, have very tight ACL on machines where only subsets of LDAP user data are needed).

3. You are advised to excercise great caution when searching mail archives. Privacy laws may apply in your jurisdiction and a protocol should most likely be observed when performing searches for either discovery or recovery.

You are advised, by the way, to cut down on OpenLDAP log level; the default log level may make your loghost very unhappy (see below under 3.3).

But there is a lot more to be done in order to create a resilient, scalable and secure infrastructure:

- backup and recovery
- logging
- monitoring and event management
- configuration management

3.2. Backup and recovery

This not discussed in great detail in this paper as it is a well-trodden path. All backups are encrypted (and decrypted in a test pass) and the backup media is stored off-site. By necessity the encryption passphrase is stored on the backup server and I intend to improve on this a bit. Disaster recovery scenarios exist (these include the quick configuration of spare Juniper switches from backups) but are, for various reasons, out of scope.

3.3. Logging

Logging is an essential element of security, and therefore you may desire to log all you can get. This is surprisingly expensive; collecting all logging from all OpenLDAP replicas, *bind* replicas with query logging enabled and FreeRadius can suffocate your loghost very quickly unless log levels are sanitized.

At the very least, collect the following:

- *dhcpd*
- *named* queries — we want to know who's contacting a command and control server
- *pflog*
- *mail server* logs.[4]

4. Not so much for security as for being able to trace misdirected mail. Mail address autocompletion is evil, as is the practice of some major enterprises to let MS Exchange expose internal host names which, of course, are unresolvable.

What to do with the log files?

grep, *awk* and *sed* are always available and are the tools of first resort.
We recommend Splunk[5] to customers but this tool is not inexpensive and I have no business case for the procurement of a license.

However, *sysutils/logstash*[6] in combination with Kibana[7] looks promising if you're prepared to live with Java. A *grep* filter in *logstash* suffices to hide events of low relevance.

```
grep {
match => [ "%{syslog_program}", \
    "nfsen" ]
negate => true
}
```

It does no harm, by the way, to be aware that not all ports provide a proper *newsyslog* rule. This is also true for many Linux packages.

Your loghost should be protected to an extreme degree and the compressed logs should be written to read-only media once a month. [8]

3.4. Monitoring and event management

We use Zabbix[9] to monitor all servers and essential services provided by these machines, plus certain environmental values such as the temperature in our server room. Zabbix has been configured to send alerts to a couple of mail addresses including an SMS server.

3.4.1. UPS. A power failure is, of course, the supreme event. We suffered an extended power failure a few years ago at our old premises when a Royal Dutch Army Apache helicopter knocked over a power pylon near Neerijnen, NL.[10]

sysutils/upsmon[11] is configured to shut down all clients after 5 minutes and storage servers after 10 minutes as I do not desire a fully drained UPS when the power comes back on. Your mileage may vary; it may take more than 5 minutes to shut down all virtual machines in your LAN. If you install UPS then do not forget to connect *all* your core switches and *all* devices needed to send alerts, otherwise the shutdown signals will never be communicated to the clients and administrators.

We have no business case for a backup generator; keeping a couple of servers alive is meaningless if there is no way to heat and light part of the office and to power a couple of desktops and a coffeemaker. Essential services such as billing and banking are SAAS or web-based, anyway.

3.5. Configuration Management

All servers are managed with CFEngine[12] with respect to:

- the verification and correction of file and directory permissions;
- the verification of checksums of important files;
- the restarting of processes (should never happen);
- the propagation of certain public keys;
- the editing of configuration files,

and other minor adjustments. You are advised to mount file systems on SSD-devices with a *noatime* flag if the tripwire function is used. More about CFEngine in 5.7.

5. Splunk, http://www.splunk.com
6. *sysutils/logstash*, http://logstash.net
7. Kibana, http://kibana.org
8. A protocol should exist with regard to the destruction of old media. I dare not suggest a rule of thumb here and I suggest that you merely ensure that a protocol with regard to the preservation and destruction of log files exists and has been endorsed (signed) by very senior management.
9. Zabbix, http://www.zabbix.com

10. No injuries except to some careers after court martial.
11. *sysutils/upsmon*, http://www.networkupstools.org
12. CFEngine, http://cfengine.com

4. Security

4.1. Inventory of Authorized and Unauthorized Devices

This is SANS-20 Critical Control #1. At our office, IEEE 802.1x[13] is the first line of defence.

I run the occasional *security/nmap* scan but mainly rely on *net-mgmt/arpalert* for the detection of unknown devices.[14]

As a rule, any MAC address which is not in my well-protected dhcpd.conf and whose IP address cannot be resolved is regarded as an anomaly deserving investigation. All ARP-detection utilities require tweaking in environments where jails are run on aliased interfaces. If you use *arpalert* the *ip_change* flag is indispensable. I am aware, as you should be, that this is far from perfect.

For the detection of rogue DHCP-servers, *net-mgmt/dhcdrop* is useful as it will effectively disable them. Do not play around with this tool in a production environment as it works very well indeed.

4.2. Inventory of Authorized and Unauthorized Software

This is SANS-20 Critical Control #2.

Creating software inventories and enforcing compliance with a baseline is surprisingly difficult in real life, unless a system with just the distribution binaries is good enough for you. Such a system may, indeed, give you much more than you desire — Postfix on RHEL, for instance, brings in a MySQL runtime library.

Few systems are built to be exactly the same, but all should be built from a certain baseline – an xxBSD-distribution or a Linux kickstart of a minimal installation for a given flavour. If you document a baseline your systems are close to compliance with this control but there is no way you will be fully compliant.

13. IEEE 802.1x, http://www.ieee802.org/1/pages/802.1x-2004.html

14. MAC addresses can be spoofed and therefore 802.1x is a must-have.

4.3. Continuous Vulnerability Assessment and Remediation

This is SANS-20 Critical Control #4.

In a nutshell, this refers to the periodic scanning of hosts for vulnerabilities, preferaby using an SCAP[15]-validated tool. Such tools are generally quite expensive; we've investigated some of the offerings and software from Tenable, Saint and Tripwire (enterprise version) look promising. Finding code-based vulnerabilities (CVE) is not something most of us are good at, so read your mailing lists.

Fixing configuration-based vulnerabilities (CCE) is less complicated if a solid configuration management infrastructure is in place.

4.4. Malware Defenses

This is SANS-20 Critical Control #5; for reasons of space and because this is a relatively well-understood topic, I treat this topic as out of scope in the server ecology.

4.5. Boundary Defense

This is SANS-20 Critical Control #13, *Boundary Defense* and, strictly speaking, out of scope. But we have regarded packet filters as kind of very important since 1997.

We maintain two packet filters: between our gateway and the rest of the world and between our main gateway and the office LAN. At the moment both are OpenBSD 5.4 on USB drives built with flashrd[16]. I may eventually replace one or both with a second-generation firewall.

15. SCAP, http://scap.nist.gov

16. flashrd, http://www.nmedia.net/flashrd/

4.5.1. Block and log.
Block all inbound traffic except when permitted and *log*. It is not an error to be too restrictive.

Since mid-2012 I've blocked all traffic from countries which are not credibly of interest to Snow and are major sources of malware and cyber attacks,[17] and a table with known bad hosts is updated frequently from Emerging Threats[18] by *cron(1)*.[19]

As we publish an LPIC-2[20] Exam Prep book this has caused a few issues. For instance, I block all traffic from Brazil, a notorious source of SPAM, abuse and other evil. Someone from that country wanted to read our book but was blocked; her e-mail was also blocked until she mailed us from *gmail.com* and I unblocked the *brazil* range for port 80.

Block all outbound traffic out except when permitted and *log*.[21]

This, again, increases the load on your loghost (and your packet filter) but the information gathered may be of immense value.

> *Why*
> At a point in time I received warnings, through our ISP, from law enforcement that connections were made from our gateway to a well-known C&C. Only by logging all DNS queries and all outgoing connections was I able to track down the originating workstation. This did not take very long as the search could quickly be narrowed down to our three Windows virtual machines.

17. Note that the Country IP Block tables available on the internet are not ncessarily reliable. I have good reason to believe that at least in Europe there is some informal commerce in IPv4 addresses.
18. Emerging Threats, http://rules.emergingthreats.net/fwrules/emerging-Block-IPs.txt
19. By the way, there are subtle but lethal differences between *bash* and *ksh*. As I regard portability quite highly, the Korn shell is used as standard scripting shell on all xxBSD-systems. On FreeBSD this requires a custom install and a minor edit of */etc/shells* which is handled by *CFEngine*, discussed below.
20. LPIC-2, http://lpic2.unix.nl
21. Privacy legislation in your jurisdiction should be most carefully read and understood.

5. Desktops

5.1. A brief history of desktops

In 1997 all office staff were familiar with a terminal interface, *emacs* and LaTeX, and were given XDM and a simple desktop menu the next year.

This is an instance of *simplicity*; most configuration was handled by a tree of Lisp files somewhere in */usr/local* and a longish */etc/profile* and all complexity was hidden from the user.

The employment of new salespersons and office staff resulted in growth but also proved that *XDM*, *mh-mail* and *Gnumeric* had to go; the time to train a salesperson off the street to productivity was growing unacceptable. As the company grew, performance deteriorated and in 2005 new HP-51xx personal computers were procured and a simple but effeective Gnome desktop was built based on CentOS 4. Performance was excellent and updates or the deployment of new applications were handled by CFEngine.

OpenOffice proved to be less than usable and in 2007 we deployed Windows XP with Samba homes, a volume license for Microsoft Office and OpenLDAP as directory server.[22] In essence, this was a very robust setup and Nitrobit Policy Extensions[23] allowed me to manage these machines with Microsoft Group Policy Objects, which are actually quite good. Zarafa[24] with Outlook plugins handled mail and calendar services for a few years to general satisfaction.[25].

In 2010 I finished building a proof-of-concept netinstall of Windows 7 but the nice and shiny aluminum Intel 12″ iMacs were a very attractive alternative.

5.1.1. Why Apple.
We're a UNIX company. To confront our guests with Windows desktops at the reception desk is not what we had in mind. Almost

22. I had no intention to be locked in by Active Directory.
23. Nitrobit Policy Extensions, http://www.nitrobit.com
24. Zarafa, http://www.zarafa.com
25. Zarafa is a very good tool for UNIX or Linux users who require Outlook-compatibility; the only reason I stopped using Zarafa was that I did not need the MS Outlook plugins any more and switched to Courier for IMAP and OSX Server for CalDav.

all senior staff were using MacBooks to their great satisfaction, and I believed that Snow Leopard was about good enough for corporate use.

The iMacs look good, are silent, have excellent displays and comfortable keyboards, and require just one power outlet.

Operating a contracting business has its advantages; in 2010 I commissioned a colleague who had been Apple-trained earlier in his career to build 2 MacMini with Snow Leopard Sever and FireWire failover while I built an AFP home directory server on FreeBSD 8 with ZFS and took care of OpenLDAP and the like, taking pains to avoid vendor lock-in as much as possible; I had already established that OSX Server wasn't good for much except iCal, software updates and a certain amount of desktop management.

Migrating the users was quite simple. Only a few essential attributes were added to the user account in LDAP by generating a few LDIFs: an objectClass: *apple-user* and a few atributes of this class (*authAuthority, apple-generateduid, apple-user-homeDirectory and apple-user-homeurl*).

In addition, a few mappings were made in Directory Utility and incorporated in the disk image.[26]

A *dscl /LDAPv3/ldap1.snow.nl -read /Users/zkonijn* shows a very minimal entry which preserves compatibility with UNIX (and Windows):

```
dsAttrTypeNative:apple-user-homeDirectory:\
<home_dir><url>smb://server/zkonijn\
</url><path></path></home_dir>
dsAttrTypeNative:authAuthority: ;basic;
dsAttrTypeNative:givenName: Zus
dsAttrTypeNative:mail: zkonijn@snow.nl
dsAttrTypeNative:maildrop: zus.konijn
dsAttrTypeNative:objectClass: inetOrgPerson \
    posixAccount shadowAccount \
    sambaSamAccount CourierMailAlias \
    apple-user SnowPerson top
dsAttrTypeNative:sambaAcctFlags: [UX    ]
dsAttrTypeNative:sambaPrimaryGroupSID: 513
dsAttrTypeNative:sambaSID: \
    S-1-5-21-3227326526-2509306901-whatever
dsAttrTypeNative:sn: Konijn
dsAttrTypeNative:SnowCanonicalName: \
    zus.konijn@snow.nl
AppleMetaNodeLocation: /LDAPv3/ldap0.snow.nl
AppleMetaRecordName:  cn=Zus \
    Konijn,ou=techniek,ou=intern,ou=people, \
    dc=snow,dc=nl
```

26. Adding mappings by injecting a *template.plist* is not allowed in Lion and higher. Relevant information is stored under /Library/Preferences/OpenDirectory/Configurations/LDAPv3, one plist per directory server.

```
GeneratedUID: \
    3ef6f677-2c85-4e38-8efd-438a30d67d53
HomeDirectory:
  <home_dir><url>smb://server/zkonijn\
</url><path></path></home_dir>
NFSHomeDirectory: /home/zkonijn
Password: ********
PrimaryGroupID: 1159
RealName:
 Zus Konijn
RecordName: zkonijn
RecordType: dsRecTypeStandard:Users
UniqueID: 11234
UserShell: /bin/bash
```

The deployment of Snow Leopard was a hugely successful enterprise; staff loved the iMacs and most got used to the UI in a remarkably short time with few complaints and no serious issues.

> As a side note: at the outset it was intended to provide salespersons with an iPad and keyboard for mail, web and so on and an iPhone for voice, while deploying one iMac for two salespersons. For various reasons, this turned out to be less successful than expected.

5.2. Upgrade Path

Immediately after deployment I began to make plans for the future as Apple's life cycle management policy is quite clear.

Lion was a no-go; upon first installation I was annoyed by the Directory Service login bug (any password would do) and by the amazing time it took Apple to fix this issue. We never did deploy 10.7, anticipating that Lion would quickly go wherever Vista went.

Mountain Lion was better and eventually 10.8.1 was deployed. Mountain Lion server, on the other hand, was, to put it mildly, verminous and crash-prone, and I eventually gave up on this software altogether.

OSX server's fail-over mode was removed in 10.7 which affected availability and increased the urgency of further investigations into vendor-independence.

Eventually proxy management on the 10.6 calendar server slowly disintegrated all by itself, resulting in lots of tickets (for which, by the way, a few generations of interns have built an excellent system based on OTRS[27], including a CMDB.

SOGo[28] replaced Calendar Service. It has some minor issues but runs off PostgreSQL in a very straightforward and perfectly manageable way, and Reposado handles software updates.

This is a sequence of small steps but these actions eliminated a number of more or less dysfunctional infrastructure components, enhancing *simplicity* (less SPOF) while improving *flexibility* (a change to a CFEngine file results in changes all over the network.[29])

5.3. Images

DeployStudio[30] is used to create an image of a machine which looks the way I want it to, including current versions of applications such as Microsoft Office, printer drivers and the like. Clients are "netbooted", for which OSX Server is needed; ours runs on an old Mac Pro.

We maintain one general-purpose image and a special one for the finance department which contains a fully configured Windows 7 virtual machine. Some minor post-installation must be done manually.

5.4. Performance

AFP performance has always been atrocious. NFS performance is very good but Spotlight will not work. SMB is also quite good and Spotlight does work, but there is another issue: caches in /var/folders/xxxxx are used by Mail and AddressBook and many other things. And many user processes do not really quit when a user logs out.

This is a non-issue as long as a user sticks to her own workstation, as is true for the great majority of users.

I could probably have done something in RADIUS to block double logins but the cache issue would not have gone away.

Last August I decided to create local home directories for non-roaming users and all complaints about performance have gone away. This also gives them Time Machine (home folder only). There is a very clear policy that business files must be stored on business shares, and as these files are shared this policy is generally adhered to.

5.5. Printing

We lease a couple of quite expensive Canon i5330 printers with Department ID, not for security but for accounting (color pages are expensive). This is fine, but Microsoft Office and Adobe whatever have a print icon, and printing by clicking on this icon results in stuck jobs as no department ID is passed on. Out of the box, "Regular users" are not allowed to remove queued jobs even if member of lpadmin and I therefore have CFEngine distribute a customized *cupsd.conf*.

5.6. Managed Preferences

Apple, as a consumer-oriented enterprise, do not offer a decent tool to manage clients. The 10.7 Profile Manager was so crash-prone that I stopped using it. Workgroup Manager allows the administration of certain aspects of a client but is not very convincing either. It at least allowed me to "grey out" certain icons in System Preferences, set screen saver defaults and some other minor stuff – but even that can be done from the command line:

```
defaults write com.apple.systempreferences  \
    HiddenPreferencePanes -array-add \
    com.apple.preferences.Bluetooth
```

Deleting part of the array is currently not sup-

27. OTRS, http://www.otrs.com
28. SOGo, http://www.sogo.nu
29. It is, of course, advised to have a DTAP setup with a separate subnet for testing CFEngine in order to prevent accidents happening. CFEngine clients will run off-line if the master fails *and/or* until the time upon which they will attempt to go fail-safe, if you have provided for such a configuration, but accidents may happen and with the immense performance at low license fees provided by VMWare's Fusion tool, there is no excuse for not dedicating a Mac to DTAP.
30. DeployStudio, http://www.deploystudio.com

ported, however, so the entire HiddenPreferencePanes object must be deleted and reassembled minus the item which you wished to unhide. This isn't very convenient.

There are ways of storing MCX settings (machine and user preferences) in OpenLDAP[31]. The problem here is that such items are Base64-encoded.

However, most settings are stored in plists, and plists are XML or a binary format which can be translated into XML with *plutil* (or *ports/converters/p5-plutil* on FreeBSD). And XML files are easily edited. To this end, I experimented with grabbing all preferences as stored in */Library/Managed Prefences* on a running client with a logged-in user, and caused the login script to modify certain settings them upon login.[32]

As distributed profiles shall obviously the way to go in future, I have no intention to spent too much time on this. Profiles are just XML files which can be edited (though, because they are three-liners, a pass through *xmllint* is needed), stored in a repository and deployed in a number of ways. When we eventually deploy Mavericks this will probably be the way to manage preferences.

The 10.9 Profile Manager is less crash-prone than before, it just needs to be restarted often. If you deploy a "Settings for Everyone" basic profile manually, do not forget to create a separate profile for *admin* without restrictions.

5.7. CFEngine

All desktops are subject to management by CFEngine 3. On a master machine I do a *"port mpkg cfengine3"* which brings in dependencies such als TokyoCabinet *etc.* and builds a metapackage. Distribution follows using Apple's not entirely useless Remote Desktop tool; if necessary, a shell script handles postinstall. The first run of *cf-agent* installs the supporting *launchd* plists.

31. http://www.siriusopensource.com/articles/osx-and-openldap-taming-leopard,*i.a.*

32. There is a LoginHook in *com.apple.loginwindow*. Its use has been deprecated since 10.4 but it still works, is convenient *and is intrinsically insecure*. It is easy to write a launchd script which causes the same effects; this is described in the next section.

```
#!/bin/ksh
export PATH=$PATH:/opt/local/sbin
sudo cf-agent --bootstrap --policy-server \
    cfengine.snow.nl
cd /opt/local/var/lib/cfengine
sudo ln -s /opt/local/sbin/cf-* bin
sudo cf-agent --bootstrap --policy-server \
    cfengine.snow.nl
sleep 1
sudo cf-agent -K
sudo killall cf-execd
sudo launchctl load \
    /Library/LaunchDaemons/nl.snow.*.plist
sudo launchctl list | grep snow
```

The line

```
sudo cf-agent --bootstrap --policy-server \
    cfengine.snow.nl
```

acquaints the client with the server and involves a key exchange (which is reversible on the server side). I next run *cf-agent*, which pulls a number of configuration files off the server including a plist for launching *cf-execd* in the canonical way and starts *cf-execd*, which is not what I want but has been marked as a "must have" process in CFEngine. This process is subsecquently replaced with a *cf-execd* instance started through *launchctl*.

5.7.1. Actions on the client side. Rules are processed if a class match is found. The simplest class is "any", but that is too simplistic.

On the Macs, rules are processed if the built-in class 'darwin' is matched. Upon a match, certain actions are carried out: files such as *cupsd.conf*, profiles, login scripts are copied if they have changed on the master; files may be edited, permissions are verified and corrected if necessary and so on. If a built-in ("hard") class provides insufficient granularity, you may define your own, e.g.:

```
  vars:
   "sysctl_cores" string => \
       execresult("/usr/sbin//sysctl -n \
       machdep.cpu.core_count", "noshell");

  classes:
   "slowmac" expression => \
       strcmp("$(sysctl_cores)", "2");
   "imac" and => { regcmp("imac.*", \
       "$(sys.uqhost)") };

 reports:
   slowmac::
    "$(sys.uqhost) is an old mac";
   imac::
        "This is $(sys.uqhost)";
```

which will output

```
R: imac14 is an old mac
R: This is imac14
```

There is no space to discuss CFEngine in detail and a few very simple and perhaps slightly contrived examples should suffice.

```
#
# $Id$ left out
#
bundle common macs {
 classes:
   "imacs" expression => "darwin";
}

bundle agent imac_files {

vars:
 "master_location" string => \
     "/var/cfengine/masterfiles/osx";
 "policyhost" string => "cfengine.snow.nl";

files:
 imacs::
  "/etc/profile"
  comment => "ensure that shell history is \
      timestamped",
  edit_line => AppendIfNoLine('export \
      HISTTIMEFORMAT="%F %T> "');

 imacs::
  "/Library/LaunchDaemons/nl.snow.cfexecd.plist",
  comment => "LaunchDaemon to start cf-execd \
      at boot",
  perms => mog("u+rw,og+r","root","wheel"),
  copy_from =>
   secure_cp("$(master_location)/nl.snow.\
       cfexecd.plist", "$(policyhost)");
}
```

This is the plist:

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST \
    1.0//EN" \
    "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>nl.snow.cfexecd</string>
    <key>ProgramArguments</key>
    <array>
        <string>/opt/local/sbin/cf-execd</string>
        <string>-F</string>
    </array>
    <key>KeepAlive</key>
    <true/>
</dict>
</plist>
```

Files which are routinely copied (but only if necessary) include:

```
admindotprofile
auto_master
cupsd.conf
loginscript.sh
nl.snow.cfexecd.plist
nl.snow.cffailsafe.plist
nl.snow.failsafe.plist
nl.snow.swupdate.plist
nl.snow.loginscript.plist
sudoers
swupdate.sh
sysctl.conf
```

There is some trickery in the schdule for upates *nl.snow.swupdate.plist*, One would not want all 35 iMacs to assault the update server at 17:00 every Monday. CFEngine allows me to randomize day and minute, however. This plist is only copied from the server if it does not yet exist, and is only edited if there is a zero to replace.

```
vars:
....
 "run_minute" int => randomint(1,59);
 "run_day" int => randomint(1,5);
....

"/Library/LaunchDaemons/nl.snow.swupdate.plist"
   comment => "minute after 17:00 to start \
       swupdate",
   edit_line => my_replace( \
       "<integer>0</integer>",
        "<integer>$(run_minute)</integer>");

"/Library/LaunchDaemons/nl.snow.swupdate.plist"
   comment => "day of week to run swupdate",
   edit_line => \
       my_replace("<integer>1</integer>",
        "<integer>$(run_day)</integer>");
```

A copy action in CFEngine can be made dependent upon another action; it is, for instance, pointless to try and copy an `authorized_keys` unless the `.ssh` directory exists.

```
imacs::
 "/Users/admin/.ssh/."
 handle => "make_ssh_dir",
 perms => mog("u+rwx,og-rwx","admin","staff"),
 create => "true";

imacs::
 "/Users/admin/.ssh/authorized_keys"
 depends_on => { "make_ssh_dir" },
 perms => mog("u+rw,og-rwx","admin","staff"),
 copy_from =>
  secure_cp("$(master_keys)/authorized_keys", \
      \
  "$(policyhost)");
```

As a final example, I don't want desktops to try to use wireless accidentally or otherwise, as there are at least two wireless devices per user already:

```
commands:
 ``/usr/sbin/networksetup -setairportpower \
     airport off"
  comment => "disable wireless interfaces";
```

By default, such commands are executed without lauching a shell. Networksetup, by the way, looks like a very versatile tool on first appearance, but a subset of its flags is not supported on 10.8 and above. In a similar way, items may be added to the dock using, for instance, dockutil[33].

A rather important file is the login script in */usr/local/libexec* but that is too long to include in this paper (and quite impossible to display in two-column format). It causes certain settings to be forced to the desired default if the user has changed them or I have changed my mind. For instance, mail in HTML format is deprecated:

```
...

# no HMTL
defaults write com.apple.mail \
   SendFormat Plain ||
   error "$LINENO: cannot write default"
```

The login script is started by a script in /Library/LaunchAgents, which means that it is run by the user who has logged in and not by root.

33. dockutil, https://raw.github.com/kcrawford/dockutil/master/scripts/dockutil

5.8. Inventory of Authorized and Unauthorized Software

This is SANS-20 Critical Control #2.

Creating software inventories and enforcing compliance with a baseline is essentially impossible on OSX.

Spotlight helps a bit:

```
mdfind 'kMDItemKind == "Application"'
```

which at least finds applications in non-standard locations.

```
 mdls /Applications/\$.app -name \
     kMDItemVersion}
```

will show a version string, and

```
mdfind 'kMDItemKind == "Unix Executable File"'
```

should deliver all binaries and executable shell scripts in non-App directories. Unfortunately you will get much more than you asked for:

```
-rwxr-xr-x (...) \
    /Library/Printers/Canon/UFR2/Profile \
    /Device/iPR C7010VP+
```

which is an XML document in DOS format but for weird reasons has the x bits, and there are many more files like this.

I'm in two minds on this issue. I appreciate the value of a proper inventory but it requires a lot of work and the results are of questionable value on ephemeral machines such as desktops.[34]

5.9. Secure Configurations for Hardware and Software on Mobile Devices, Laptops, Workstations and Servers

This is SANS-20 Critical Control #3, and part of it is slightly easier to implement on workstations and servers than on mobile devices. I exclude unmanaged notebooks and PDA as I have no control over these

34. Do not waste time on desktops; re-install or replace if unable to diagnose or repair the issue within 15 minutes!

things and therefore deny all access to internal networks to such devices.

/usr/libexec/ApplicationFirewall/socketfilterfw provides an interface to the standard firewall which is very basic indeed. Selecting 'block all' disallows ARD connections which is inconvenient.

FreeBSD's *pf* replaced *ipfw* in OSX in 10.7 but is not actually used, as Scott Lowe[35] wrote some time ago.

```
sudo pfctl -s rules

No ALTQ support in kernel
ALTQ related functions disabled
scrub-anchor "com.apple/*" all fragment \
    reassemble
anchor "com.apple/*" all
```

A colleague is looking into this matter as we may wish more tightly to lock down the notebooks which we plan to issue, as alluded to in section 6.

6. BYOD

Bring your own device is is a new and creepy phenomenon, and one which most sysadmins have trouble dealing with. It works like this: Alice obtains a MacBook Air but has no idea how it works. Alice asks her neighbour Bob, who naturally knows nothing about your company's policy, to help her with the setup. Bob installs some games, Adobe Flash Player, Transmission and other undesirable software. In order to be able to help Alice, Bob creates an admin account for himself. Alice arrives at the office and wants to access the Sales directory and also entrust her collection of pirated movies to the home server. It would therefore be foolish to give Alice's MacBook an IP-address in the office LAN.

Now, how to deal with Alice. The simple answer is *don't*. However, IT people are unpopular enough as things are, and users often do stuff which generates money to pay our salaries, so we should be kind. A more politically correct approach would be "OK, but…". BYOD is here to stay so we ought to shape appropriate policies and provide a modicum of support. As such machines have been installed and configured in an unapproved manner, support can only be "best effort" and access to sensitive resources should be denied. Management co-operation is essential; staff should be made to sign a code of conduct. As in all matters concerning security, *awareness* is paramount.

35. Scott Lowe, http://blog.scottlowe.org/2013/05/15/using-pf-on-os-x-mountain-lion/

Providing locked-down notebooks is not necessarily a good alternative from the point of view of support staff; they will be expected to help people connect to their home WLAN, print on home printers and so on. However, we will roll out 13″ MBA for our sales staff and provide Thunderbolt displays for power and network.

6.1. The Cloud

6.1.1. Google. The European Union has rather clear laws in regard of the storage of privacy-sensitive data. Google declined to give me an assurance that our data would only be stored on servers inside the European Union and that did not really surprise me in view of Google's storage infrastructure. And as Google is a US corporation the Patriot Act applies to our data. So Google Mail and Google Apps are out.

The same applies to iCloud, DropBox and the like. However, there is no way we can (or would wish to) prevent people from obtaining Apple IDs and storing stuff in iCloud.

6.1.2. Your own cloud. It may be wise to provide an alternative for dropbox-type services so as to give users what they want while staying on top of things – it is not like I'm short of disk space. For the past six months I've run OwnCloud[36] on my own web server without major disappointments, but I will not deploy this service at Snow without a code review. AeroFS[37] may fit your bill.

7. Lessons Learned

- Try to be vendor-independent. I have chosen not to run Active Directory or Open Directory and I am still in business;

36. OwnCloud, http://owncloud.org
37. AeroFS, https://aerofs.com

- do not be too confident that Apple will fix bugs and features OSX within a time window that suits your needs. Their core business is selling high-margin PDA's and notebooks. OSX is a sideline and they've evidently accepted that the corporate desktop market belongs to Microsoft.
- use CFEngine or similar (and store the configuration files in SVN, GIT or similar);
- verify backups and encrypt offsite backups;
- have a disaster recovery scenario and *do* excercise this at regular intervals;
- leverage existing well-understood tools.

8. Resources

Apart from the resources provided in the footnotes, the Mac Developer Library[38] contains some rather good stuff if you are patient and dig down deeply.[39]

9. About the author

I've been with Snow since 1997 and my employee number is 4. I have been a FreeBSD user since 1994. I made my acquaintance with OSX in 2001 at the first EuroBSDCon in Brighton (U.K.).

38. Mac Developer Library, https://developer.apple.com/library/mac/navigation

39. This site still contains with mind-boggling titles such as "Deploying OS X Server for High Performance Computing".

Analysis of BSD Associate Exam Results

By Jim Brown, BSDCG
jimbyg@gmail.com

Abstract

The BSD Certification Group (BSDCG) was chartered in 2005 by a group of educators, system administrators, writers, and BSD enthusiasts who believed that a high quality certification for BSD systems was needed in the computing industry. In a little over two years, the gr oup designed and developed its first certification exam, the BSD Associate (BSDA) Exam. Since that time approx imately 250 candidates have taken the exam . This paper analyzes the results of the exam over the period 2009 – 2013 to determine how candidate scores have changed.

Background

From the be ginning, the BSDCG det ermined that the exams produced by the grou p should be managed by a professional psy cho-metrician. Psychometrics is the science of analyzing cognitive and a pplied performance testing. In consultation with the p sycho-metrician, the BSDCG created a Job Task Analysis (JTA) Survey. The purpose of a JTA is to determ ine what activities are considered most important and what t asks are performed most frequently by system administrators in their day-to-day work. The results were analyzed by the psy chometrician and were eventually embedded into the BSD Associate Exam Objectives Document which was released to the public in October 2005. (The current version was revi sed in Novem ber, 2012.)

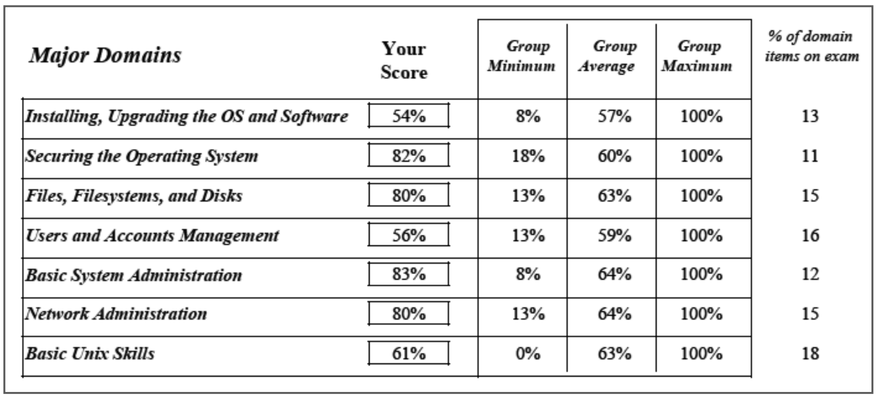
The JTA outlined the f ollowing knowledge domains and their representative percentages on the exam:

Domain	Weighting
• Installing & Upgrading the OS and Software	13%
• Securing the Operating System	11%
• Files, Filesystems, and Disks	15%
• Users and Accounts Management	16%
• Basic System Administration	12%
• Network Administration	15%
• Basic Unix Skills	17%

The JTA also determined the nature of the questions that were eventually incorporated into the BSDA exam . Th e exam format and question pool were finalized in 2007 and the exam was launched at SCALE in 2008.

BSDA Exam Report

Every candidate who takes the BS DA receives a score report that contains the results of their score and how their scores co mpare with other examinees. Figure 1 shows a representative example of a score report. The score report identifies all the Knowledge Do mains on the exam and the candidate’s score for each one. The score report also contains representative numbers for the m inimum and the maxim um scores from other candidates. Note t hat the scores for the group m aximum for all domains is 100% indicating that some othe r candidates have managed to achieve a score of 100%. Likewise, the minimum scores show that the scores by other candidate have fallen as low as 0% (for ‘Basic Unix Skills’) to 18% (for ‘Securing the Operating System’).



Major Domains	Your Score	*Group Minimum*	*Group Average*	*Group Maximum*	*% of domain items on exam*
Installing, Upgrading the OS and Software	54%	8%	57%	100%	13
Securing the Operating System	82%	18%	60%	100%	11
Files, Filesystems, and Disks	80%	13%	63%	100%	15
Users and Accounts Management	56%	13%	59%	100%	16
Basic System Administration	83%	8%	64%	100%	12
Network Administration	80%	13%	64%	100%	15
Basic Unix Skills	61%	0%	63%	100%	18

Figure 1 - BSDA Score Report

BSDA Exam Analysis

Figure 2 shows the Pass/Fail Ratio for the exam over the period 2009 – 2013. It is noteworthy that the f ail rate is fairly high. This indicates that the exam questions are neither too easy, which would result in a much lower fail rate, nor too hard or not understandable which would result in a much higher failure rate. It also indicates that candidates need proper preparation for the exam. Anecdotal evidence fro m both candidates and proctors indicate that candidates usually indicate that the were surprised by the quality of the exam.

As noted ab ove, the exam has a number of questions from each knowledge Domain.

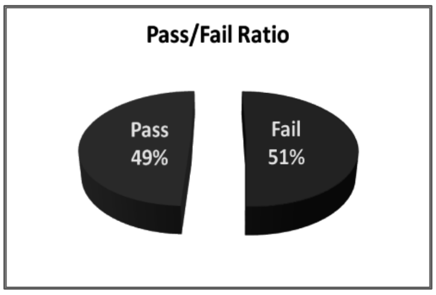


Figure 2 - BSDA Pass / Fail Ratio

Each Knowledge Domain contributes an important percentage to the overall score. In addition, questions in each Kn owledge Domain are spread across all four BS D operating systems supported on the e xam – DragonFly BSD, FreeBSD, NetBSD, and OpenBSD. It is not possi ble to pass the exa m if you are not basically familiar with all four BSD systems.

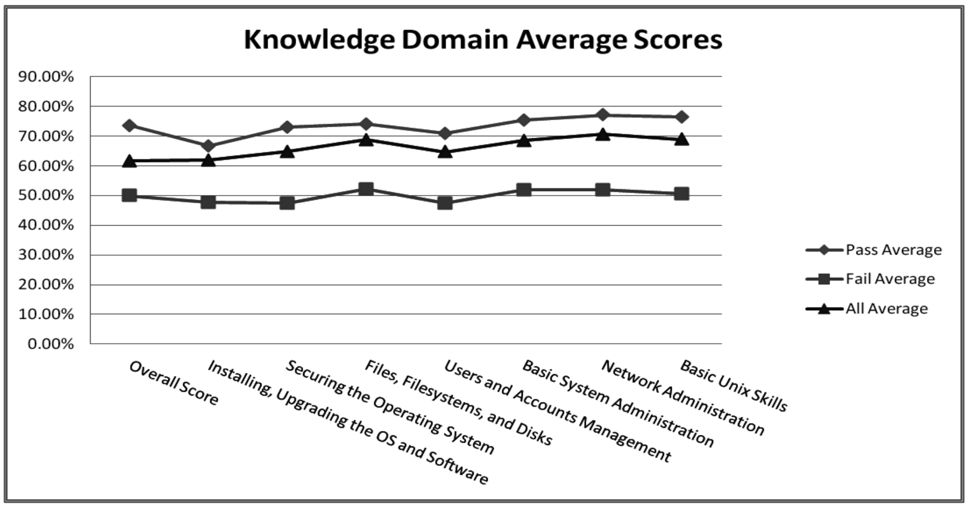


Figure 3 - BSDA Knowledge Domain Average Scores for all Domains

Figure 3 shows the percentage of correct responses for all Knowledge Domains for all candidates, candidates who passed, and candidates who failed. The figure shows that the those who pass the exam score approximately 20% higher on every Knowledge Domain than those who fail. This indicates that success on the exam is the result of preparation, or a combination of experience and preparation across the board.

Figure 3 also shows which domains are consistently the highest scores. "Network Administration", "Basic System Administration", and "Basic Unix Skills" are the highest scoring domains overall. This is a reflection of the high value that BSD administrators and users place on these topics.

The lowest domain score, "Installing and Upgrading the OS and Software", may be a reflection of the fact that you generally only install a system once and perform upgrades infrequently thereafter. Improvement in these skills, along with "Users and Accounts Management" will likely result in improved passing scores overall.

Also of interest is the change in performance for each Knowledge Domain over time. For the years 2009 through 2013, the averages for each domain have been calculated as shown below in Figure 4.

Note that while scores for 'Basic System Administration' have decreased slightly, the scores for 'Basic Unix Skills' have increased. Also note that the scores for 'Securing the Operating System' have jumped within the last year paralleling a wider industry trend regarding increased scrutiny of information security best practices.

However, if we adjust the score axis and zoom out (see insert), we can see that the distribution of scores *as a whole* remains fairly constant. Collected BSD knowledge in all Knowledge Domains by all candidates has remained fairly stable and is not on the decline.

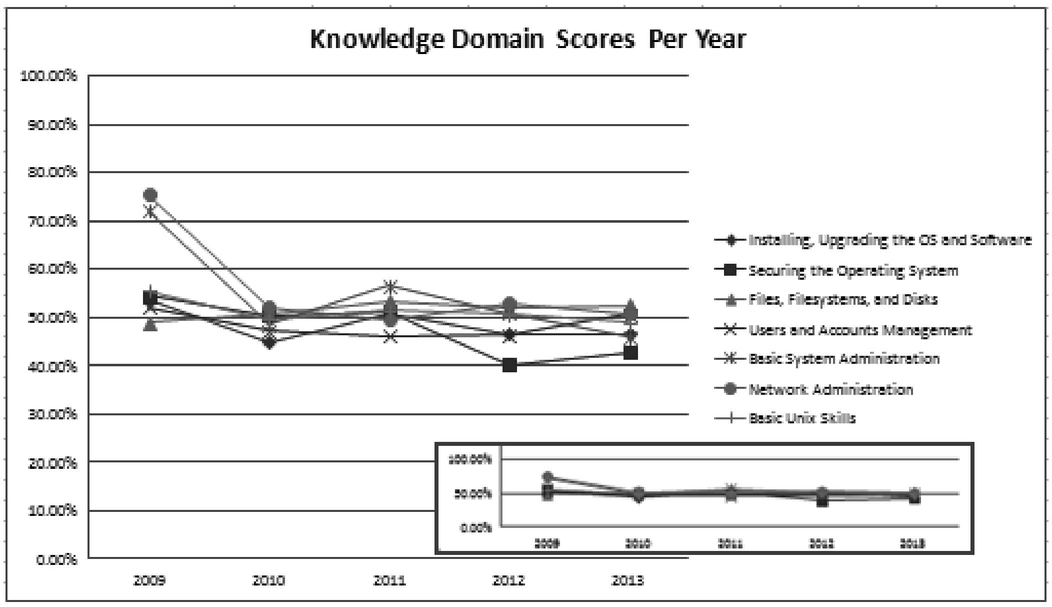


Figure 4 - BSDA Section Scores 2010 - 2013

BSDA Score Trending

Figures 5 and 6 show the percentage of passing and failing scores per year, respectively. The figures show a decline in passing scores and corresponding rise in failing scores, during the 2009-2012 period. Reasons for the decline are not clear, though as noted above, proper preparation for the exam is essential. Passing scores rose during the 2013 year by approximately 12%.

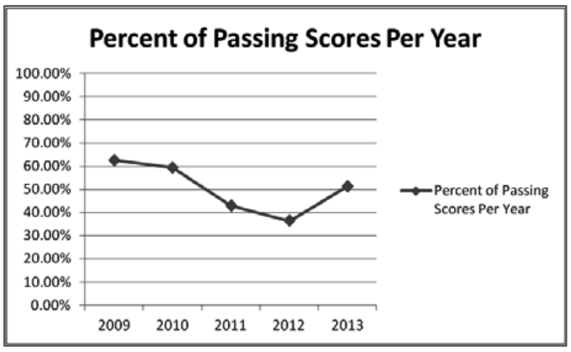


Figure 5 - Percentage of BSDA Passing Scores Per Year

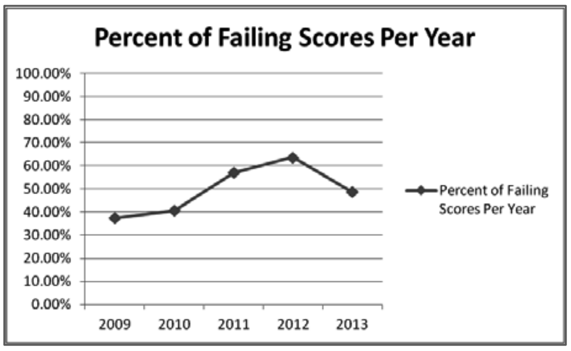


Figure 6 - Percentage of BSDA Failing Scores Per Year

BSDA Exam Security

One of the concerns of any certification group is the security of the exam questions, and more importantly, the answers. There are several strategies that certification exams use for protecting their exam security. In fact, a good certification program will use multiple overlapping strategies to protect its integrity.

One of the best strategies is to have a proctor administer every exam. The role of the proctor is to ensure that the candidate is who they say they are, to securely handle the test materials, and to ensure there is no communication between exam candidates. This improves reliability of the exam results, because the only source of exam knowledge is from the exam candidate, and no one else.

Another excellent strategy is to have a very large pool of questions about the exam. The larger the pool, the less likely it is that someone can subvert the entire certification by publishing questions. By having a large enough random pool of questions, the ability of an evil candidate to publish meaningful information about the exam (either questions or questions and answers) declines as in the ratio

$$\frac{\text{number of questions on exam}}{\text{number of questions in pool}}$$

where the number of questions in pool is any number greater than 10 times the number of questions on the exam.

Collusion, defined as the practice of a maintaining a secret understanding between two or more persons to gain something unlawfully or unfairly, can be a problem for certification exams. “Brain Dump” sites that exist to give an unfair advantage to those willing to pay have been a problem for many certification programs. However, collusive behavior can be detected, and more importantly can, in some instances, be traced back to the originator.

Consider Figure 7, a chart of the scores for a single Knowledge Domain (Basic Unix Skills) of those who passed the exam. The chart reveals a very wide variety of scores over a broad period of time. The scores reveal that the distribution of scores is fairly stable – many score well, but many also score less well. This chart is typical of a non-collusive environment.

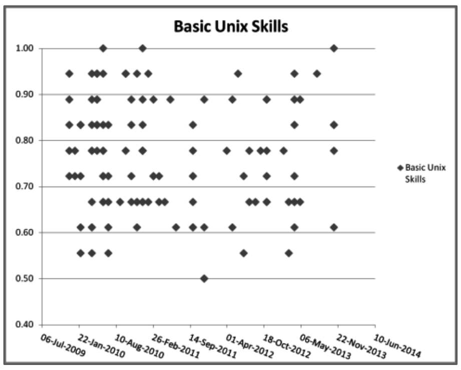


Figure 7 - Typical Scores of Passing Candidates in A Domain

Now consider Figure 8, which is a chart that shows collusive behavior. Notice how the distribution of scores begins to homogenize toward a single score. The period of time where the collusive behavior began is very likely somewhere within the red bounding box. There are still some people who scored less well, but the number of those declines very quickly over time as more and more candidates opt for the "brain dump" or "cheat sheet" sites to get the answers. A psychometrician who is concerned about collusion will quickly focus on the exams within the bounding box and re-examine those scores for other clues of cheating.

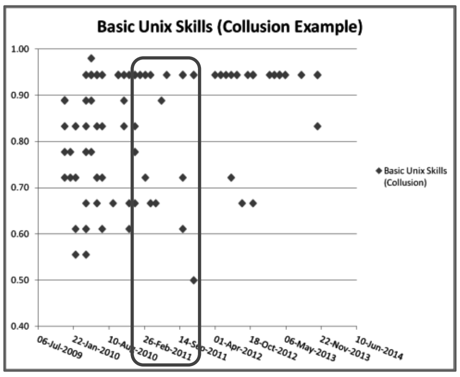


Figure 8 - Example of the Effects of Collusion

Fortunately to date the BSD Certification Group has not been targeted for serious collusion. Psychometrics provides a valuable toolkit to detect collusive behavior and this is yet another reason why all BSD Certification Group exams are based on psychometrics.

BSD Professional Exam

The BSD Associate (BSDA) exam is only the first exam produced by the BSDCG. The next exam in development is the BSD Professional (BSDP) exam, scheduled for release in 2014. The published Knowledge Domains for the BSDP are[1] :

Domain	Weighting
• Installation and Setup	5%
• Security	10%
• Files and Filesystems	8%
• Users	6%
• General Administration	8%
• Common Services	7%
• General Networking	10%
• Backup and Restore	16%
• Virtualization	8%
• Logging and Monitoring	16%
• High Availability / High Performance	7%

The BSDP exam will incorporate both a written exam and a lab exam. In this way, both cognitive knowledge and practical skills will be tested.

We will track results for the hands-on practical skills separately from the written exam scores. It will be interesting to revisit these statistics in the next five years to see if hands-on practical skills improve exam scores.

Future Directions

The world of certification exams is crowded with many competing exams – some good, and some not so good. In the BSD world, the BSD Certification Group has put together these exams to provide the best possible vehicle to assess knowledge and skill in using and

1 http://www.bsdcertification.org/certification/certification/bsd-professional

administering BSD operating systems. Additional exams are in the planning stages.

Another development is the establishment of partnerships for the adaptation and delivery of the BSDA exam and future exams. Through a recent partnership with BSD Research, the BSDA exam will soon be available in Japanese and delivered locally. This will allow future examination of the effects of translation on exam results.