Faster, smolBSD! Boot! Boot!

Pierre Pronchery

IT-Security Consultant, Defora Networks GmbH
<pierre@defora.net>
Developer & Board of Directors, The NetBSD Foundation
<khorben@NetBSD.org>
Developer, The FreeBSD Project
<khorben@FreeBSD.org>

***Abstract*—Virtualisation technology has gone a long way. It began as early as the 1960s with IBM CP/CMS, and was even available as Open Source back then. On modern hardware though, the technology exploded in the mid-2000s, and has become a new normal in the form of containers since 2014. BSD systems have both pioneered and lagged behind on different aspects, between innovation and catch up with the latest industry standards. The gang of NetBSD, EdgeBSD, and smolBSD projects is no stranger to this: coming from the desert, it is now proudly lining up for drag races to the boot finish line, still plundering ideas and patches all over to solve the case.**

Keywords—EdgeBSD, Firecracker, FreeBSD, KVM, MicroVM, NetBSD, NVMM, OpenBSD, paravirtualization, PVH, QEMU, smolBSD, virtualization, VirtIO, VMware, Xen

I. New Use Cases for a Paradigm Shift

The popularity of virtualization and paradigm shift towards container-based applications and workloads can be explained in different ways. Before even considering security aspects, one could argue that the diversity and growing complexity of Linux-based environments, coupled with its explosion in popularity, has led a new generation of engineers onto a platform without the opportunity (or desire) to take advantage of one of its most prominent cultural strengths: packaging.

Regardless, it became possible to ship an entire environment around an application, instead of just the application itself. This, together with the availability (or force feeding) of ever more capable hardware [1], has allowed a generation to effectively distribute entire installations of an Operating System as application images. In turn, this impacts bare metal systems in multiple ways, down to their conception. Suddenly, the performance of the boot sequence became important enough of an issue to drive the traditional System V init system out of every major Linux distribution [2].

When it comes to BSD-based systems though (or really anything not using the Linux ABI) in their original shape and form, they cannot simply be started as container processes within a Linux environment. However, their kernel can be started instead, leveraging the kernel as a translation layer between the corresponding BSD ABI and that of the host. Suddenly, the performance of the entire boot process, *including that of the kernel* is effectively key to the overall performance of BSD-based containers. In some workloads, short-lived containers may be started and stopped continuously and repeatedly, further increasing the importance of shorter boot times; this paper will delve into how this has been benchmarked and improved for the NetBSD Operating System.

In practice, virtualization has relied on the simulation in software of hardware components typically found in physical hardware. However, this introduced unnecessary complexity, limitations, and negative impacts on performance, especially before the introduction of hardware-assisted virtualization. A first solution was introduced in the form of paravirtualization [3], like used by the Xen hypervisor [4]. With this communication mechanism, seamless integration can be ensured between the host and its guests, without any such artificial constraints. A number of innovations have been developed as a result, such as PVH guests [5].

Going one step further, the architecture of the virtualized environment itself could be stripped down to a minimal implementation. As recently as November 2018, Amazon introduced its new Firecracker virtualization technology [6]. This Virtual Machine Monitor (VMM) enables the deployment of lightweight Virtual Machines (called "microVMs"), which in turn allows interacting with virtualized environments almost like regular system processes - which is particularly relevant for the workloads mentioned above. The reference implementation chosen in our Open Source environment is that provided by the MicroVM machine type from the QEMU project [7]; this new architecture brings its own set of requirements and improvements, which are listed in this paper as well.

References

[1] Moore's law on Wikipedia, https://en.wikipedia.org/wiki/Moore%27s_law
[2] init on Wikipedia, https://en.wikipedia.org/wiki/Init
[3] Paravirtualization on Wikipedia, https://en.wikipedia.org/wiki/Virtualization#Paravirtualization
[4] Xen on Wikipedia, https://en.wikipedia.org/wiki/Xen
[5] Guest Types from Xen, https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview#Guest_Types
[6] Firecracker, https://firecracker-microvm.github.io
[7] QEMU, https://www.qemu.org

II. Improving Boot Time

First, raw boot time of the kernel becomes crucially important, since it turns into a recurring operation. This first section details how this was achieved on NetBSD, from benchmarking to actual improvements.

A. Measuring performance with tslog(4)

Following the introduction of Firecracker, Colin Percival (FreeBSD developer cperciva@) worked on porting FreeBSD to this environment [1], with significant assistance from other FreeBSD developers. Thanks to this collective effort [2], a number of improvements could be tested on the NetBSD kernel as well.

First, while the tslog(4) framework itself has not been imported into NetBSD's source tree at this time, it has been successfully adapted for the system [3] by Émile Heitor (NetBSD developer iMil@). It has been used extensively when implementing or testing most of the improvements documented in this paper. Once the corresponding patch applied, the components relevant for kernel-level performance measurements are as described here.

a) Configuration option for the kernel

The kernel option `TSLOG` has to be enabled within the desired kernel configuration file, e.g., `sys/arch/amd64/conf/GENERIC`:

```
option TSLOG
```

b) Kernel framework

The tslog(4) framework is available after including the header `<sys/tslog.h>` from kernel code, which exposes the `tslog()` function itself:

```
void tslog(const lwp_t *, int, const char *,
const char *);
```

A number of helper macros are available as well:

```
#define TSENTER() TSRAW(curlwp, TS_ENTER,
__func__, NULL)
#define TSENTER2(x) TSRAW(curlwp, TS_ENTER,
__func__, x)
#define TSEXIT() TSRAW(curlwp, TS_EXIT,
__func__, NULL)
#define TSEXIT2(x) TSRAW(curlwp, TS_EXIT,
__func__, x)
#define TSTHREAD(td, x) TSRAW(td, TS_THREAD,
x, NULL)
#define TSEVENT(x) TSRAW(curlwp, TS_EVENT, x,
NULL)
#define TSEVENT2(x, y) TSRAW(curlwp,
TS_EVENT, x, y)
#define TSLINE() TSEVENT2(__FILE__,
__XSTRING(__LINE__))
#define TSWAIT(x) TSEVENT2("WAIT", x);
#define TSUNWAIT(x) TSEVENT2("UNWAIT", x);
#define TSHOLD(x) TSEVENT2("HOLD", x);
#define TSRELEASE(x) TSEVENT2("RELEASE", x);
#define TSFORK(p, pp) TSRAW_USER(p, pp, NULL,
NULL)
#define TSEXEC(p, name) TSRAW_USER(p, (pid_t)
(-1), name, NULL)
#define TSNAMEI(p, name) TSRAW_USER(p,
(pid_t)(-1), NULL, name)
#define TSPROCEXIT(p) TSRAW_USER(p, (pid_t)
(-1), NULL, NULL)
```

c) Adding probes

Without diving into the implementation details of tslog(4), in the patch attached to the e-mail, iMil@ registered probes into the system as follows:

- In `sys/kern/init_main.c`, function `main()`, `TSENTER()` right at the beginning of this function, and `TSEXIT()` just before handing control over to the init(8) process (PID 1)
- In `sys/kern/init_main.c`, function `start_init()`, `TSENTER()` right at the beginning of this function, and `TSEXIT()` if init(8) exits;
- In `sys/kern/kern_exec.c`, function `execve1()`, `TSEXEC()` just before `execve_runproc()`
- In `sys/kern/kern_exit.c`, `TSPROCEXIT()` in `exit1()`
- In `sys/kern/kern_fork.c`, `TSFORK()` in `fork1()`
- In `sys/kern/kern_kthread.c`, `TSTHREAD()` in `kthread_create()`
- In `sys/kern/subr_autoconf.c`, `TSENTER2()` and `TSEXIT2()` in `config_attach_internal()`

With these probes in place, it was possible to monitor the exact timing of kernel threads, device driver attachments, as well as user processes.

d) Obtaining results with sysctl(8)

Then, the log buffers can be obtained from a user program through the `debug.tslog` and `debug.tslog_user` sysctl(7), as follows:

```
# sysctl -n debug.tslog
```

```
0x0 163043419271429 ENTER main
0x2 163043589716213 THREAD idle/0
0x3 163043589798639 THREAD softnet/0
0x4 163043589846413 THREAD softbio/0
0x5 163043589919985 THREAD softclk/0
0x6 163043589964721 THREAD softser/0
0x7 163043590024553 THREAD xcall/0
[...]
# sysctl -n debug.tslog_user
1 0 172737354585999 0 "/sbin/init" ""
31 123 172740234772401 172740249387583 "" ""
63 123 172740234547299 172740235482361 "" ""
95 127 172740333737791 172740335851907 "" ""
123 1 172740202524183 172741236028583 "/
rescue/sh" ""
124 123 172740227635497 172740230554267 "" ""
125 123 172740228373173 172740234310955 "" ""
126 125 172740231595483 172740233954105 "" ""
127 190 172740331192515 172740343274743 "" ""
159 226 172740373769929 172740378274071 "/
rescue/mkdir" ""
[...]
```

e) Visualizing performance

Finally, the system performance can be compared visually through the generation of flamegraphs. Here also, a fork [4] of Colin Percival's freebsd-boot-profiling repository [5] was tweaked by iMil@ when working on the NetBSD targets; the helper script `tslog.sh` obtains the log buffer, while the corresponding flamegraph is generated with the help of Perl scripts in the same repository:

```
$ git clone https://github.com/NetBSDfr/
freebsd-boot-profiling.git
$ cd freebsd-boot-profiling && ./mkflame.sh
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//
EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/
svg11.dtd">
[...]
```

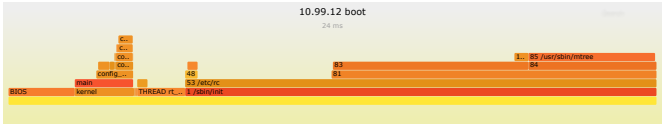

Fig. 1. NetBSD 10.99.12 boot performance measured by iMil

Thanks to these measurements, two areas in particular were identified with potential for improvements to the boot time:

1. Removing the calibration loop for the processor.
2. Eliminating the calls to `DELAY()`.

Both items are presented below.

B. Obtaining the CPU frequency

On Intel-compatible processors, the Time Stamp Counter (TSC) [6] counts the number of CPU cycles since its reset. It is used for time keeping, but requires calibration to provide accurate results. In NetBSD on that platform, this is performed in `sys/arch/x86/x86/cpu.c`, in the function `cpu_get_tsc_freq()`, where a loop calls `delay_func(100)` 1000 times, always delaying the boot time by at least 10 milliseconds:

```
/*
 * Now do the calibration.
 */
freq = 0;
for (int i = 0; i < 1000; i++) {
        const int s = splhigh();
        t0 = cpu_counter();
        delay_func(100);
        t1 = cpu_counter();
        splx(s);
        freq += t1 - t0 - overhead;
}
freq = freq * 10;

aprint_debug_dev(ci->ci_dev, "TSC freq "
    "from delay %" PRIu64 " Hz\n", freq);
```

In virtualized environments, the TSC calibration is already known to the host Operating System; it makes sense to obtain it from the hypervisor directly. VMware provides a `CPUID` leaf to that effect, and offered to standardize it [7] across the industry. The implementation uses one of the Model-Specific Registers (MSR) reserved for software use to that effect (`0x40000010`).

In NetBSD, the support for this instruction has been introduced by iMil@ on May 2nd 2025, in time for inclusion into the 11.0 release.

C. Eliminating calls to DELAY()

Just like in the example above, a number of differents parts of the kernel use the `DELAY()` mechanism to intentionally slow down the boot process. While it is a necessary, documented property of some hardware components to ensure their proper initialization, in some cases this behaviour could be avoided either by improving the initialization process, or by leveraging virtualization as a safer context. This is thanks to the added control over the hardware simulated or provided to the guest system; it notably affects serial ports.

a) Serial ports

Serial ports are typically emulated when exposed to guests. There, the operating system should be able to skip delays waiting for output upon initialization. Another 10 milliseconds were gained this way, thanks to the following patch by iMil: a new system property, `skip_attach_delay` (boolean), was added to the system and used to teach serial drivers to conditionally avoid waiting for pending output. The corresponding patch affects two

places; first, `sys/arch/x86/x86/x86_autoconf.c`, in the function `device_register()`, setting the property's value:

```
+    if (device_is_a(dev, "com") && vm_guest
> VM_GUEST_NO)
+
prop_dictionary_set_bool(device_properties(de
v),
+            "skip_attach_delay", true);
```

And then, `sys/dev/ic/com.c`, in the function `com_attach_subr()`, avoiding the delay if appropriate:

```
+    bool skip_attach_delay = false;
[...]
+    prop_dictionary_get_bool(dict,
"skip_attach_delay", &skip_attach_delay);
[...]
+    /* No need for a delay on virtual
machines. */
+    if (!skip_attach_delay)
+        delay(10000); /* wait for output to
finish */
+
     /* Make sure the console is always
"hardwired". */
-    delay(10000);     /* wait for output to
finish */
```

In NetBSD, this change was introduced on February 12th 2025, in time for inclusion into the 11.0 release.

References

[1] FreeBSD/Firecracker FreeBSD Status Report 2022Q3, https://www.freebsd.org/status/report-2022-07-2022-09/firecracker/

[2] C. Percival, Profiling the FreeBSD kernel boot, https://www.daemonology.net/papers/bootprofiling.pdf

[3] É. Heitor, Benchmarking NetBSD/amd64 with tslog(4), https://mail-index.netbsd.org/tech-kern/2024/10/24/msg029802.html

[4] NetBSDfr/freebsd-boot-profiling repository on GitHub at branch "netbsd", https://github.com/NetBSDfr/freebsd-boot-profiling/tree/netbsd

[5] cperciva/freebsd-boot-profiling repository on GitHub, https://github.com/cperciva/freebsd-boot-profiling

[6] Time Stamp Counter on Wikipedia, https://en.wikipedia.org/wiki/Time_Stamp_Counter

[7] CPUID usage for interaction between Hypervisors and Linux on Linux Weekly News from October 1st 2008, https://lwn.net/Articles/301888/

III. Improvements for the MicroVM Architecture

In the MicroVM machine type as implemented by the QEMU project, the system is stripped down to a minimalistic architecture. While multi-processing remains possible, there are no PCI or ACPI buses anymore. Instead, virtual devices perform their operations through Memory-Mapped I/O [1] (MMIO), with the enumeration and configuration information obtained in command-line parameters provided by the hypervisor.

A. Generic PVH boot

One major obstacle to the efficient startup of Virtual Machine environments has to do with the hardware initialization sequence and early boot process. Assuming Intel-compatible platforms, on bare metal systems with a BIOS, kernel code was entered by a bootloader running in "real mode", with significant restrictions [2] on the initial execution environment:

- Less than 1 MB of RAM is available.
- There is no hardware-based memory protection, nor virtual memory.
- The default CPU operand length is only 16 bits.
- The memory addressing modes provided are restrictive.
- Accessing more than 64 kB requires using segment registers.

Even on modern systems supporting the Unified Extensible Firmware Interface (UEFI) [3], Operating System kernels need a bootloader [4] linked into a PE executable, placed in a FAT32 partition, itself with a minimal size of 32 MB. The real kernel is typically chainloaded from there, from another partition in the native format for the system.

None of these requirements or behaviour make sense in virtualized environments. Worse even, they really slow down the execution: a virtual BIOS or firmware has to be initialized and provide access to the simulated hardware platform, in a bug-for-bug compatible manner with the actual, physical hardware.

Instead, platforms like Xen introduced the "PV boot mechanism", where the kernel is loaded and launched by the hypervisor directly [5]. In practice, a specific ELF note section has to be added to the kernel, containing the virtual memory address of the kernel entry point. The hypervisor will then jump into the kernel entry point with paging already enabled (possibly in 64-bit mode directly). Support for this boot mode had been introduced by Manuel Bouyer (NetBSD developer bouyer@) to the NetBSD kernel on May 2nd 2020, but it only worked when the kernel was started by the Xen hypervisor specifically. Thanks to another patch by iMil@ on December 2nd 2024, it became possible to start the NetBSD kernel in direct boot mode, according to the Generic PVH (GENPVH) specification.

Starting the NetBSD kernel with QEMU, in direct mode

with the MicroVM architecture and with NetBSD's MICROVM kernel, can then be performed as follows: (here on a macOS/amd64 host)

```
$ qemu-system-x86_64 -m 512 -accel hvf \
  -display none -serial stdio -cpu host \
  -action reboot=shutdown \
  -kernel netbsd-MICROVM \
  -append "console=com rw -q" \
  -drive file=rescue-
amd64.img,format=raw,if=virtio
[   1.0000000] WARNING: system needs entropy
for security; see entropy(7)
[   1.0000000] [ Kernel symbol table missing!
]
[   1.0000000] Copyright (c) 1996, 1997,
1998, 1999, 2000, 2001, 2002, 2003,
[   1.0000000]     2004, 2005, 2006, 2007,
2008, 2009, 2010, 2011, 2012, 2013,
[   1.0000000]     2014, 2015, 2016, 2017,
2018, 2019, 2020, 2021, 2022, 2023,
[   1.0000000]     2024, 2025, 2026
[   1.0000000]     The NetBSD Foundation,
Inc.  All rights reserved.
[   1.0000000] Copyright (c) 1982, 1986,
1989, 1991, 1993
[   1.0000000]     The Regents of the
University of California.  All rights
reserved.

[   1.0000000] NetBSD 11.99.5 (MICROVM) #22:
Fri Feb 13 20:08:10 CET 2026
[   1.0000000]
khorben@reapr.station.defora:/home/khorben/
Projects/EdgeBSD/src-current/obj-amd64/sys/
arch/amd64/compile/MICROVM
[   1.0000000] total memory = 511 MB
[   1.0000000] avail memory = 427 MB
[   1.0000000] Found mainbus0 (root)
 (QBOOT    000000000000)
[   1.0000000] Found cpu0 at mainbus0: Boot
Processor
[   1.0000000] Found ioapic0 at mainbus0
[   1.0000000] Found isa0 at mainbus0
[   1.0000000] Found com0 at isa0
[   1.0000000] Found pv0 at mainbus0
[   1.0000000] Found virtio0 at pv0
[   1.0000000] Found ld0 at virtio0
[   1.0000000] Found virtio1 at pv0
[   1.0100030] WARNING: no TOD clock present
[   1.0100030] WARNING: using filesystem time
[   1.0100030] WARNING: CHECK AND RESET THE
DATE!
[   1.0100030] kernel boot time: 253ms
Created tmpfs /dev (1835008 byte, 3552
inodes)
#
```

Support for direct boot mode is expected to be part of the 10.2 and 11.0 releases of NetBSD.

B. The pv bus

The pv(4) bus was implemented by iMil@ (again), largely inspired by OpenBSD's own pvbus(4) [6]. It does not correspond or match any physical bus or device, but lets the VirtIO drivers [7] attach there in the absence of the PCI or ACPI buses.

In the MICROVM kernel configuration file shared by the i386 and amd64 ports, `sys/arch/x86/conf/MICROVM.common`, the pv bus is used as follows:

```
# Virtual bus for non-PCI devices
pv* at pvbus?

## Virtio devices
# Use MMIO by default
virtio* at pv?

include "dev/virtio/virtio.config"
no viomb* at virtio?
no vioscsi* at virtio?
no viogpu* at virtio?  # unsupported
```

In turn, the `dev/virtio/virtio.config` file lists the device types supported:

```
viomb*    at virtio?  # Memory balloon device
ld*       at virtio?  # Disk device
viocon*   at virtio?  # Serial device
viogpu*   at virtio?  # GPU device
vioif*    at virtio?  # Network device
viornd*   at virtio?  # Entropy device
vioscsi*  at virtio?  # SCSI device
vio9p*    at virtio?  # 9P device
```

In NetBSD, this new bus was introduced on January 2nd 2025, well in time for inclusion into the 11.0 release.

C. pvclock(4) driver

The pvclock(4) driver supports the paravirtualized clock available on KVM and other hypervisors. It uses a shared page between the guest and the hypervisor to synchronize the TSC clock in an efficient way.

This driver was initially written for OpenBSD [8] by Reyk Floeter (OpenBSD developer reyk@). It implements support for the second version of this protocol, thus accessing a stable and reliable source for the Time Stamp Counter described above. Although currently being tested in smolBSD, the driver has not been imported into the NetBSD kernel yet.

D. virtio_mmio(4) from the command line

In the absence of the PCI and ACPI buses, there is no mechanism available to the kernel to enumerate the location of the registers and interrupts for the devices available. Instead, with the MicroVM architecture, this information is provided to the kernel by the hypervisor, via command-line parameters to the kernel. This can be seen in the kernel's system message buffer, as printed by dmesg(8):

```
# dmesg
[...]
[     1.000000] virtio0 at pv0
[     1.000000] virtio0: kernel parameters:
console=com rw -x
virtio_mmio.device=512@0xfeb00e00:12
```

```
[     1.000000] virtio0: viommio:
512@0xfeb00e00:12
[     1.000000] virtio0: VirtIO-MMIO-v1
[     1.000000] virtio0: block device (id 2,
rev. 0x00)
[...]
```

The virtio_mmio(4) bus driver was extended to look in the command line for the string "virtio_mmio.device=". It then parses the value provided, expecting:

```
value:        size [multiplier] "@"
base_address ":" interrupt [":" id]
size:         UINT64_T
base_address: UINT64_T
interrupt:    UINT64_T
id:           UINT64_T
```

With this information, the corresponding VirtIO drivers can be accessed and used normally, through direct memory access, without the overhead of a physical bus. This feature was introduced in the FreeBSD kernel by cperciva@, and imported into the NetBSD kernel by iMil@.

In NetBSD, this driver was introduced on January 15th 2025, well in time for the NetBSD 11.0 release.

E. viocon(4) as kernel console

When debugging or running virtualized environments, the console is the most critical communication channel available to the user. Serial console emulation is a common occurrence, but it is subject to a number of limitations. On top of its disadvantages as an emulated hardware device, it is fundamentally slow: often set to a default speed of 115200 bauds, this is far from the capacities of modern buses.

Thankfully, modern hypervisors provide access to a console device with the VirtIO protocol. Unfortunately, in NetBSD, this is not supported in an official release nor imported into the development branch as of the creation of this document. However, a patch from Taylor Campbell [9] (NetBSD developer riastradh@) has been imported in smolBSD, where it is in regular use for testing.

F. initrd support for amd64

Booting the kernel in direct mode was demonstrated above, accessing the root filesystem through the ld(4) driver, itself attaching as a virtio(4) device. However, there is a way to access the root filesystem in memory directly, without any intermediate translation through a device driver: using a RAM disk [10] (aka. initrd). This is particularly useful and efficient for short-lived, small, volatile payloads, with no driver backend for saving information back into the hypervisor's system.

On NetBSD, RAM disks are usually supported and used in a few different contexts: starting the installer (with `ramdisk.fs`), detecting and mounting an encrypted partition as the root filesystem (with `ramdisk-cgdroot.fs`), or setting up a ZFS pool containing the root filesystem (with `ramdisk-zfsroot.fs`). These files are provided in the corresponding releases where supported at e.g., `amd64/installation/ramdisk` for the amd64 port. However, support for RAM disks when booting in direct kernel mode (as per the Generic PVH specification above) was unfortunately not working until the corresponding patch was merged on December 5th 2025.

This patch effectively implements support for kernel modules, provided by the hypervisor as per the `struct hvm_start_info` in file `sys/external/mit/xen-include-public/dist/xen/include/public/arch-x86/hvm/start_info.h`:

```
struct hvm_start_info {
// Contains the magic value 0x336ec578
// ("xEn3" with the 0x80 bit of the "E" set)
uint32_t magic;

// Version of this structure.
uint32_t version;

// SIF_xxx flags.
uint32_t flags;

// Number of modules passed to the kernel.
uint32_t nr_modules;

// Physical address of an array of
// hvm_modlist_entry.
uint64_t modlist_paddr;

// Physical address of the command line.
uint64_t cmdline_paddr;
// Physical address of the RSDP ACPI data
// structure.
uint64_t rsdp_paddr;

// All following fields only present in
// version 1 and newer
// Physical address of an array of
// hvm_memmap_table_entry.
uint64_t memmap_paddr;

// Number of entries in the memmap table.
// Value will be zero if there is no memory
// map being provided.
uint32_t memmap_entries;

// Must be zero.
uint32_t reserved;
};
```

During the boot process, early kernel code in `locore.S` (written in 32-bit assembly) goes through the array of `hvm_modlist_entry` located in memory at struct member `modlist_paddr`. With this patch, it makes a copy of every module passed to the kernel at a new location in memory, safer for access later. The function

`x86_add_xen_modules()` was then added to the file `sys/arch/x86/x86/x86_machdep.c`, called by `init_x86_clusters()` when booting in GENPVH mode.

Unlike kernel modules loaded by the regular boot loader, there is no classification of the modules as `BI_MODULE_ELF`, `BI_MODULE_IMAGE`, `BI_MODULE_RND`, or `BI_MODULE_FS` as per file `sys/arch/x86/include/bootinfo.h`. Instead, the module type is inferred through the presence of magic values:

- `\177ELF` at offset 0 for ELF kernel modules.
- `\211PNG\r\n\032` or `\377\330\377` at offset 0 for PNG or JPEG images, respectively. (e.g., for splash screen images)
- Fallback to filesystem images in every other case.

In the case of the MICROVM kernel configuration, this also required enabling the `MODULAR`, `MEMORY_DISK_HOOKS`, and `MEMORY_DISK_DYNAMIC` options into the kernel configuration file at `sys/arch/amd64/conf/MICROVM`. Unfortunately, while in practice NetBSD usually supports loading RAM disks embedded into ELF kernel modules, this is not implemented by this patch; filesystem images need to be provided directly instead.

Regardless, support for this feature has been retrofitted to be included into the NetBSD 11.0 release.

References

[1] Memory-Mapped I/O on Wikipedia, https://en.wikipedia.org/wiki/MMIO

[2] Real Mode on OSDev.org's wiki, https://wiki.osdev.org/Real_Mode

[3] UEFI vs. legacy BIOS on OSDev.org's wiki, https://wiki.osdev.org/UEFI#UEFI_vs._legacy_BIOS

[4] System Initialization (x86) on OSDev.org's wiki, https://wiki.osdev.org/System_Initialization_(x86)

[5] PVH Specification from the Xen project, https://xenbits.xen.org/docs/4.6-testing/misc/pvh.html

[6] pvbus – paravirtual device tree root from OpenBSD, https://man.openbsd.org/pvbus.4

[7] VirtIO on libvirt.org, https://wiki.libvirt.org/Virtio.html

[8] pvclock – paravirtual clock driver from OpenBSD, https://man.openbsd.org/pvclock.4

[9] Commit 0b64bdd3b6633e26361541c1c9f9a42148169a77 of the NetBSDfr/NetBSD-src repository on GitHub at branch "netbsd-smol-11", https://github.com/NetBSDfr/NetBSD-src/commit/0b64bdd3b6633e26361541c1c9f9a42148169a77

[10] RAM drive on Wikipedia, https://en.wikipedia.org/wiki/RAM_drive

IV. Tying It All Together

smolBSD is a meta-operating system [1] built on top of NetBSD. It uses the MICROVM kernel and provides a framework to build and run minimalistic, bootable filesystem images. With all of the improvements described above, on our reference hardware, it can take less than 10 milliseconds from the moment QEMU hands over control to the kernel, to the first instruction executed in userland.

A. smolBSD as minimal BSD system

When combined together as a virtualized environment, the MICROVM kernel from smolBSD and the filesystems created implement exactly what was described at the beginning of this article: a lightweight, efficient environment for the execution of container-like workloads. Getting started with smolBSD can be as simple as described here:

```
$ git clone https://github.com/NetBSDfr/
smolBSD
$ cd smolBSD
$ cat ~/.ssh/id_ed25519.pub >> service/sshd/
etc/mykey.pub
$ bmake SERVICE=sshd build
➡️ starting the builder microvm
➡️ fetching sets
➡️ creating root filesystem (512M)
✅ image ready: sshd-amd64.img
➡️ killing the builder microvm

$ ./startnb.sh -f etc/sshd.conf
[   1.0092096] kernel boot time: 14ms
Starting sshd.
Server listening on :: port 22.
Server listening on 0.0.0.0 port 22.

$ ssh -p 2022 ssh@localhost
```

In this example, a copy of smolBSD is obtained from its GitHub repository [2]. The system created is the equivalent of an SSH server running as a process, giving remote access to a shell on that virtualized system. After providing a public key to smolBSD, a system image is built using the "sshd" service definition, incorporating the public key to the build. Once the image ready, QEMU is started through the `startnb.sh` script, and an instance of the OpenSSH server is available within milliseconds.

The following table lists some of the services currently available in smolBSD:

Service	Description
bozohttpd	Starts a web server
clawd	Runs OpenClaw inside a MicroVM
games	Provides the programs available in NetBSD's games set
lhv-tools	Runs a collection of online tools within a PHP-based environment
nitro	Test environment for the nitro init system
rescue	Provides an environment with the rescue set installed
runbsd	Test environment for the runit init system
sshd	Starts an OpenSSH instance
systembsd	Test environment for the dinit init system
tslog	Benchmarking environment for tslog(4)
usershell	Provides a minimal environment with an OpenSSH client

Table 1. List of services available in smolBSD

B. Disk images as RAM disks

During the development of smolBSD, it became clear that it would make sense to support loading the root filesystem either as a virtual drive or directly as a RAM disk, at the flip of a switch. However, this respectively requires the presence or the absence of a partition table inside the corresponding image file. As a consequence, different files must be used, depending on the boot mode; this makes this solution rather impractical.

In order to help with this issue, a new patch was proposed [3], adding support for parsing partition tables from RAM disks. It chooses and uses the root filesystem desired when detecting a GUID Partition Table (GPT) [4]. To do so, it looks for the magic value `EFI PART` at offset 512; this is known as the LBA 1 block, located right after the boot sector. This check is performed just before the generic fallback as direct filesystem image. The partition effectively chosen for booting is the first one found to be marked with the `BOOTME` GPT attribute.

The patch only implements this behaviour when booting in Generic PVH mode, as documented above. However, it would make sense to extend this support in every situation where NetBSD supports loading RAM disks: this would also reduce the need to provide different image files as part of official releases, according to the boot mode. As suggested in the resulting e-mail thread by Valery Ushakov (NetBSD developer uwe@), the patch should be extended to offer this possibility in more contexts before possibly being integrated into NetBSD.

In the meantime, this change has already been included into smolBSD's own kernel. Adding support for alternative selection methods for the root partition has also been discussed within the project, like looking for the `BOOTONCE` GPT attribute, or looking for a specific partition by `NAME=`.

C. ZFS as root filesystem

As an example of quick prototyping made possible thanks to all of the improvements introduced here, it was possible to design and implement an automated installation process for NetBSD, with the root filesystem on a ZFS pool. Until it can fully work correctly when loaded as a RAM disk image and booting in direct kernel mode (GENPVH), it is implemented as a bootable CD-ROM image instead. It was confirmed to work on bare metal on an Intel Xeon E3-1275v5 system rented from a server auction at Hetzner [5].

A virtualized, local test could be performed as follows:

```
$ qemu-system-x86_64 -m 1024 -accel hvf \
  -display none -serial stdio -cpu host \
  -action reboot=shutdown \
  -cdrom zfsinstall-com.iso \
  -drive file=ld0.img,format=raw,id=hd0
Welcome to the NetBSD 11.0_RC1 boot-only
========================================
 install CD
===========

This media contains only the installation
program.  Binary sets to complete the
installation must be downloaded separately.
The installer can download them
if this machine has a working internet
connection.

If you encounter a problem while booting,
report a bug at
https://www.NetBSD.org/.

     1. Install NetBSD (root on ZFS)
     2. Drop to boot prompt

Choose an option; RETURN for default; SPACE
to stop countdown.
Option 1 will be chosen in 0 seconds.
27141392+811136+1286016
[1369043+1654200+1239919]=0x20117f8
Loading /stand/amd64/11.0/modules/solaris/
solaris.kmod
Loading /stand/amd64/11.0/modules/zfs/
zfs.kmod
Loading /zfsinstall.kmod
Loading /stand/amd64/11.0/modules/cd9660/
cd9660.kmod
[...]
zfsinstall: configuring the network
dhcpcd-10.2.3 starting
[...]
zfsinstall: detected 1 disks: wd0
zfsinstall: downloading set: base
zfsinstall: downloading set: comp
zfsinstall: downloading set: etc
zfsinstall: downloading set: games
zfsinstall: downloading set: gpufw
```

```
zfsinstall: downloading set: man
zfsinstall: downloading set: misc
zfsinstall: downloading set: modules
zfsinstall: downloading set: rescue
zfsinstall: downloading set: text
zfsinstall: downloading kernel: GENERIC
zfsinstall: downloading ramdisk: zfsroot
zfsinstall: partitioning disk wd0
/dev/rwd0: Partition 1 added: c12a7328-
f81f-11d2-ba4b-00a0c93ec93b 2048 1048576
/dev/rwd0: Partition 2 added: 49f48d5a-
b10e-11dc-b99b-0019d1879648 1050624 1048576
/dev/rwd0: Partition 3 added:
516e7cba-6ecf-11d6-8ff8-00022d09712b 2099200
1900544
/dev/rwd0: Partition 2 marked as bootable
zfsinstall: formatting partition: dk0 (EFI)
/dev/rdk0: 1046496 sectors in 130812 FAT32
clusters (4096 bytes/cluster)
zfsinstall: formatting partition: dk1 (FFSv2)
zfsinstall: setting up partition: dk0 (EFI)
zfsinstall: setting up partition: dk1 (FFSv2)
zfsinstall: installing kernel: GENERIC
zfsinstall: installing ramdisk: zfsroot
zfsinstall: installing bootloader
zfsinstall: installing modules
zfsinstall: loading module: solaris
modload: solaris: File exists
zfsinstall: loading module: zfs
modload: zfs: File exists
zfsinstall: creating ZFS pool: rpool
zfsinstall: creating ZFS volume: rpool/ROOT
zfsinstall: mounting ZFS volume: rpool/ROOT
on /altroot
zfsinstall: installing set: base
zfsinstall: installing set: comp
zfsinstall: installing set: etc
zfsinstall: installing set: games
zfsinstall: installing set: gpufw
zfsinstall: installing set: man
zfsinstall: installing set: misc
zfsinstall: installing set: modules
zfsinstall: installing set: rescue
zfsinstall: installing set: text
zfsinstall: exporting ZFS pool: rpool
zfsinstall: importing ZFS pool: rpool
zfsinstall: mounting ZFS volume: rpool/ROOT
on /altroot
zfsinstall: restarting init
init.root: / -> /altroot
Sun Feb 15 22:00:58 UTC 2026
[...]
Sun Feb 15 22:01:04 UTC 2026

NetBSD/amd64 (Amnesiac) (constty)

login:
```

This process is of particular interest because it is currently missing from the official NetBSD installer, sysinst. While it has been documented in NetBSD's wiki [6], this procedure is difficult to get right. The implementation demonstrated here is still experimental and a work in progress, but has been published nonetheless as the khorben/edgebsd-11/zfsinstall branch [7] of NetBSDfr's NetBSD-src mirror on GitHub.

References

[1] smolBSD - build your own minimal BSD system, https://smolbsd.org/#about

[2] smolBSD on GitHub, https://github.com/NetBSDfr/smolBSD

[3] P. Pronchery, Support for disk images when loading ramdisks, https://mail-index.netbsd.org/tech-kern/2026/02/09/msg030807.html

[4] GUID Partition Table on OSDev.org's wiki, https://wiki.osdev.org/GPT

[5] Server auction at Hetzner, https://www.hetzner.com/sb/

[6] Root On ZFS on NetBSD's wiki, https://wiki.netbsd.org/root_on_zfs/

[7] NetBSDfr/NetBSD-src repository on GitHub at branch "khorben/edgebsd-11/zfsinstall", https://github.com/NetBSDfr/NetBSD-src/tree/khorben/edgebsd-11/zfsinstall

V. Conclusion

In this paper, different innovations improving performance when running NetBSD-based systems in virtualized environments were detailed. Their list is not exhaustive, but they should constitute enough evidence for the potential of the NetBSD kernel and smolBSD project as a viable solution for fast-pace workloads as BSD containers. When reaching sub-10 millisecond boot time for the kernel on our reference hardware, it becomes possible to offer a fully isolated solution for the routine execution of programs in hostile environments.

This effort is expected to continue in the future, keeping the NetBSD platform attractive not only for esoteric or obsolete hardware, but also in the race for elegant yet optimal solutions for modern workloads.

To quote the President of the NetBSD Foundation as the ending credits:

> FASTER!!1
>
> -- William J. Coldwell (NetBSD Developer billc@)