Project proposal - Hardware virtualization for Haiku's QEMU port

Introduction

I'm Daniel Martín, a last year student in Computer Science and Mathematics at Complutense University of Madrid (Spain). I discovered Haiku around 2013 and played with Alpha 4 back then. I kept following Haiku through the years: In 2016 I made sure GnuSocialShell (my GNU Social client) worked in Haiku. That year I also participated in Google Code-In completing some Haiku tasks (pretty simple tasks though). As Beta 1 came out in 2018 I got some of those commemorative installation DVDs that were sold at the forum. That year I also put Haiku specific instructions for building wget2 at the project readme file (they're still there). Yet, I haven't done any contribution to Haiku project itself. Since I'm interested in operating systems, drivers and low-level stuff I thought GSoC would be a great way to get into the project and learn more about those topics.

Regarding programming experience I have done mainly C programming on my own. On university courses we have used Java, C++, Assembly, Haskell and Prolog. I have some contributions to the popular GNU Wget (actually, to the "new" wget2) and a silly PS/2 mouse driver I did for the operating system xv6 in 2021. I took an elective course in 2022 called "Linux and Android Internals" which essentially was about writing kernel modules and drivers for the Linux kernel.

Project title: Hardware virtualization for Haiku's QEMU port

Project goals

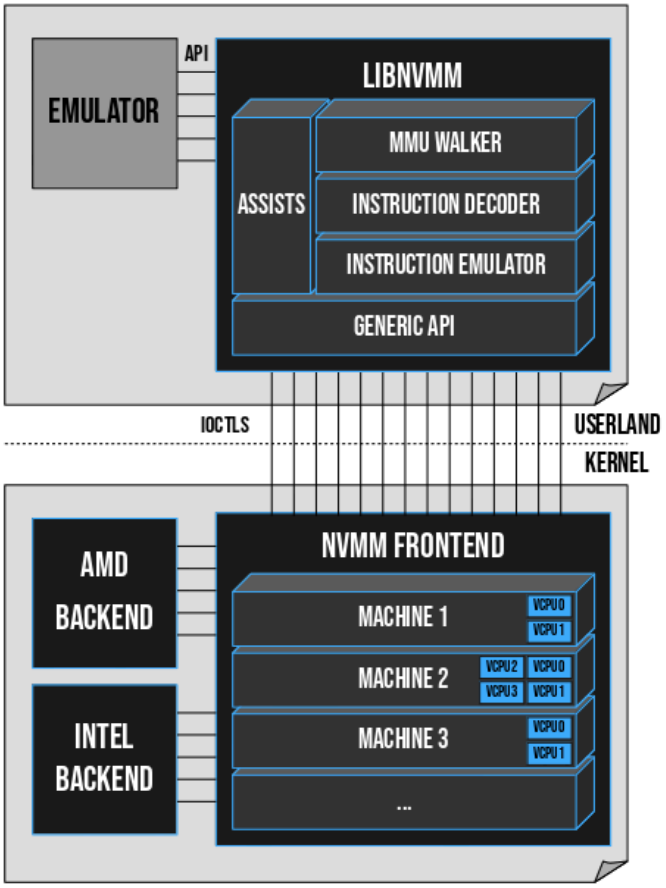
- NVMM driver ported to Haiku (VMX backend only)
- QEMU capable of accelerating virtual machines through NVMM

Technical overview

https://blog.netbsd.org/tnf/entry/from_zero_to_nvmm

NVMM is made up of 3 parts:

- NVMM backend (VMX or SVM): Handles all the low-level stuff. It's the part that effectively talks to the hardware.
- NVMM frontend: Provides a machine independent interface to handle virtualization, through exposing a device driver (UNIX style) that can be talked to mainly by using ioctl().
- libnvmm: Provides a friendly interface for userspace applications. Documentation at: https://man.dragonflybsd.org/?command=libnvmm&section=3 (this is actually a pretty good documentation not only of the API, but of the whole NVMM code).



Currently there are two main, divergent, implementations of NVMM: One at NetBSD and another one at DragonFlyBSD. The DragonFlyBSD version it's more OS independent (the majority of OS-dependent logic is confined to a nvmm_osname.c file) and it's supposed to have received functional improvements during the porting process from NetBSD. Thus, this is the implementation I would be porting.

The source code has the following tree:

```
nvmm/
├── Makefile
├── nvmm.c
├── nvmm_dragonfly.c
├── nvmm.h
├── nvmm_internal.h
├── nvmm_ioctl.h
├── nvmm_netbsd.c
├── nvmm_os.h
└── x86
    ├── nvmm_x86.c
    ├── nvmm_x86.h
    ├── nvmm_x86_svm.c
    ├── nvmm_x86_svmfunc.S
    ├── nvmm_x86_vmx.c
    └── nvmm_x86_vmxfunc.S
```

Out of this we would take the frontend (nvmm.c, nvmm.h, nvmm_internal.h, nvmm_ioctl.h) and the backend (probably only the VMX backend, since I don't own any AMD machine), plus we would have to add a Haiku region at nvmm_os.h plus writing a new nvmm_haiku.c file to contain all the Haiku-specific code.

NVMM frontend

Since nvmm it's exposed through a device driver on /dev (accessed through open(), closed() and ioctl()) Haiku's legacy driver API should do the job.

Essentially nvmm.c, nvmm.h, nvmm_internal.h and nvmm_ioctl.h should be ported with as little changes as possible, as OS-dependent logic goes into nvmm_os.h and nvmm_haiku.c. Despite being more OS-independent the DragonFlyBSD version of NVMM still has some code that is OS-dependent (but common for both NetBSD and DragonFlyBSD). As an example, while working on the code contribution I found that several low-level constants regarding CR0 register bits needed to be redefined as they're named differently in Haiku than in UNIX-like OSes. This shows that some changes will need to be done to the supposedly OS-independent parts of NVMM, although we're not talking about big changes.

nvmm_os.h contains a variety of things: synchronization mechanisms (locks, mutexes), memory allocation (at kernel space), standard output and atomic arithmetic operations (increment, decrement, …), several CPU operations (get CPU number, GDT table, ...), cpusets (CPUSet can probably be used) and preemption (enabling and disabling preemption). All of these seem to be available in one way or another in Haiku, but some (preemption and memory management mainly) will need some work before they can provide the support needed for the port (this work may or may not involve changes to Haiku's kernel itself).

As shown in the rest of this document the main task for porting NVMM's driver is putting Haiku-specific code at the nvmm_os.h header and some implementations at a new nvmm_haiku.c file. Taking a look at nvmm_netbsd.c and nvmm_dragonflybsd.c we can see that the first functions there are os_vmspace_create() and os_vmspace_destroy(), which create and destroy a virtual memory address space (which nvmm uses to create the physical address space of the guest). Regarding working with virtual memory Haiku offers areas, which unluckily don't allow to create a whole address space (they're limited by the amount of RAM available) since their main objective is to lock memory at RAM. However, the virtual memory subsystem already has the structures needed (VMAddressSpace) for this. NetBSD does this through their UVM virtual memory system.

The functions to create and handle virtual memory objects follow. A virtual memory object it's just a contiguous region of virtual memory that it's actually backed by some system facility. In the case of NetBSD these are uvm_objects, which essentially is a struct that has a list of pages + operations to manage those pages. This is the way in which nvmm maps memory to the guest. The file also contains several functions for allocating kernel memory, by pages. While Haiku has a nice virtual memory API (vm/vm.h) and the structures that are needed are already there (VMAddressSpace, VMArea, …) it might be necessary to do some changes at

the kernel level, as the way in which NVMM handles memory appears to have some subtleties (I think this is mainly because it relies on the virtual memory system of *BSD).

My plan is to port both frontend and backend in parallel, similar to what I did on the code contribution: Copy all the nvmm code we aid to port to Haiku, comment all of it, and then slowly uncomment as we port it: The idea is that by doing this we'll get a working driver from the beginning (even if at the beginning it's only capable to do VMX hardware detection) that we'll keep improving: Week by week the functionality of the driver will improve as we get close to a complete port. This should allow us to do some testing in the middle too, avoiding finding "nested" bugs at a later time, when they will be harder to fix.

NVMM backend (VMX)

Much of what's being said in the frontend section applies to the backend too, as the OS-dependent logic at nvmm_os.h is used by the backends as well.

To successfully port NVMM some knowledge of VMX will be of use. I have found a blog series of articles that provides a friendly introduction to VMX (there is a continuation series too). The VMX page at OSDev will be of help too, as well as the official Intel Manual.

NVMM backend (SVM)

After porting the VMX backend, porting the SVM backend should be a bit easier, as I would already have the experience of the VMX port (also many of the dependencies of the SVM port will be available already, as they would have been included as part of the VMX port).

However, since I don't have any AMD machines with SVM it doesn't feel reasonable to include it on the proposal. That said, I might be able to get an AMD machine, or find someone in the forums interested on help testing the SVM flavour. So, if time and circumstances allow this will get done, but it's not a priority. I prefer to prioritize getting the whole thing (NVMM + QEMU) done rather than having both VMX and SVM support, but no functional QEMU to make use of it.

libnvmm

```
libnvmm/
├── libnvmm.3
├── libnvmm.c
├── libnvmm_x86.c
├── Makefile
└── nvmm.h
```

libnvmm consists of three code files: libnvmm.c, nvmm.h and libnvmm_x86.c:

- nvmm.h: Exposes the API of libnvmm
- libnvmm.c: Contains the implementation of such API, implemented through ioctl() calls to NVMM's device driver file at /dev.

- libnvmm_x86.c: As the image of the second page summarizes libnvmm also handles MMU, instruction decode and emulation. The code for that it's here.

The MMU walker, the decoder and the instruction emulator aren't needed for a working NVMM virtualization driver. They are just extra additions to the lib for making it easier to develop applications over NVMM. That said, QEMU depends on this additions so we'll need them. Porting the lib should actually be the easiest part of all the project, since the libnvmm.c part it's simple and few code and libnvmm_x86.c, despite having more than 3000 lines of code, mainly involves reading and writing to "tables" (matrixes, arrays, …) as part of the disassembly + getting ready for emulation.

The library comes with a small test set that we may use for error detection before getting to port QEMU. In general we'll try to avoid nested bugs as much as possible by doing early testing (when possible).

How to make QEMU work with NVMM?

A patch (https://www.netbsd.org/~maxv/nvmm/patch-nvmm-support.diff) was submitted by the author of NVMM to QEMU back in 2019. As of now NVMM support is included on the latest official QEMU code: https://github.com/qemu/qemu/tree/master/target/i386/nvmm

DragonFlyBSD applies the following performance patch to the QEMU code: https://github.com/DragonFlyBSD/DPorts/blob/master/emulators/qemu/dragonfly/patch-target_i386_nvmm_nvmm-all.c

We should see whether this patch is useful or not to improve performance even further, once QEMU+NVMM is working on Haiku. Since QEMU v8.1 is available at HaikuPorts, which already contains NVMM support (at its source code) this will be the version we'll be trying to get to work.

Programming languages involved for the project

We'll surely need C (it's the language NVMM is written in), we will probably need some C++ (for all the virtual memory subsystem stuff and for any other kernel related work) and we might need x86 assembly (libnvmm extra features make heavy use of it), but I don't think this will be the case.

Timeline

This is a big project and the estimated time to complete it is 350 hours.

Community bonding period (May 1 - May 26)

- [I ONLY WILL BE ABLE TO DO MINIMAL TO NO WORK THE FIRST 2 WEEKS]
- Read documentation regarding VMX to help understand what exactly NVMM's VMX backend expect from certain kernel technologies it depends on.

- Get acquainted with the virtual memory subsystem, the scheduler and any other kernel parts that are needed to provide the kernel support needed for the port.
- Plan kernel changes that are needed (if any).

First month of coding (May 27 - June 31)

- [I ONLY WILL BE ABLE TO DO MINIMAL TO NO WORK BETWEEN JUNE 19 - JUNE 26]
- Implement any kernel technologies that are necessary for the port (if any).
- Begin porting both NVMM frontend and backend (probably in parallel).

Second month of coding (July 1 - July 31)

- Finish porting NVMM frontend and backend
- Port libnvmm
- Port libmvmm's test suite
- Run the test suite, identify any potential bugs/problems and fix them.

Third month of coding (August 1 - August 26)

- Make QEMU (HaikuPorts version) work with the brand-new NVMM driver
- Test QEMU+NVMM with a variety of guest operating systems. Use this table, which summarizes the status of many guests, to make sure QEMU+NVMM support is comparable to that in DragonFlyBSD.
- Fix any potential bugs that become known at this point and issues with the guests (if any).

After Google Summer of Code

Port NVMM's SVM backend too.

Expectations from Mentors

I expect mentors to be a reliable source of information regarding Haiku. Being able to talk with someone that knows the codebase makes finding some things easier and avoids losing significant amounts of time on dead-ends. Since mentors are more experienced than I am, and (according to the FAQ) it's desirable that I share small chunks of code on a daily basis, they can point out mistakes or suboptimal decisions that will let me improve both my coding skills and the output of the project.