eXecute-In-Place (XIP) Support for NetBSD

Masao Uebayashi
Tombi Inc.
`uebayasi@tombi.co.jp`

April 5, 2010

Abstract

XIP is a technique to execute programs directly from NOR FlashROMs without user programs copied into RAM as *page cache*. Its basic functionality is realized with small changes in *NetBSD* because *page cache* management is already concentrated in single place called *UBC*, where all file data access (read() / write(), and mmap()) go through. This paper explains the design and implementation of the changes made to the *NetBSD* kernel to support *XIP*.

1 Background

1.1 What is *XIP*?

In Unix-like systems, programs are usually stored in filesystems mounted on block devices, most typically hard disk drives. OS copies programs to execute, because executable region has to be memory mapped. *XIP* is a technique to execute programs directly from NOR FlashROMs without programs copied into RAM as *page cache*.

XIP is very commonly seen in embedded OSes (e.g. RTOS), whose memory model is much simpler than Unix. Unix system run-time image is made of kernel and user spaces. Kernel is always resident in RAM during its execution. Kernel XIP is easily achievable because it's already almost XIP by nature; it has a single copy of image. Kernel XIP is widely used in Linux to speed up boot time by omitting initial copy of kernel image[3].

Userspace XIP is more difficult, because user program execution is more complex activity built on top of a number of levels of abstraction to meet various requirments of users and user programs (applications); various devices and filesystems, dynamic execution

This paper explains only userspace XIP support.

1.2 Who needs *XIP*?

The purpose of XIP is all about reducing memory usage in user program execution. We want to reduce memory concumption mainly in two reasons:

- Internal simplicity

  Dynamic memory allocation is a complex activity. By executing user programs directly from ROM, the system can omit loading a mapped file from backing store devices. It also saves dynamic memory (page cache) allocation.

- Power consumption

  RAM consumes more power than ROM. Products can run longer if you can execute programs running directly on ROM. This matters for embedded productions that are driven by batteries.

On the other hand, XIP comes with many restrictions described later sections. It's not an option to solve problems magically. It's an option for developers who understand its implications and limitations and best used only when the product is carefully designed with other technologies like NAND FlashROMs as data storage.

1.3 Why *XIP* looks odd?

As already mentioned, programs are usually copied from a backing store (a block device) to RAM. Why? CPU can execute only instructions that are directly accessible from CPU, which means, the device's data is mapped in CPU's physical address space. The main memory (RAM) serves that.

Traditionally Unix has had an assumption that user programs are placed in a filesystem, which in turn is placed in a backing store (block device). Unix has ignored situations where programs are executed directly from such devices, because it's expected to break almost every assumption made in VM and fintroduces lots of oddities.

In *NetBSD*, however, it has turned out to be not that difficult to achieve *XIP* because most part of problems are already solved by *UBC*[2]; it concentrates access (both via mmap() and read()/write()) to pages in single place, *page cache*. The details are explained later.

2 Goals

Simplicity

Both of the design and implementation have to be simple enough to be understood by all users and developers. We don't try to solve everything in one place; rather one thing to solve in one place. Users who want our XIP to solve a simple problem can easily solve it. Users who face a complex problem (programs, data, hardware limitations, many other requirements, ...) have to solve their problem by combining multiple techniques, and designing their products by themselves. We assume that users are not stupid.

Correctness

NetBSD is known as its well thought design. The changes made by XIP must be proved correct. Ideally all the combinations of the possible operations are identified and tested. In reality, it was quite difficult because the UVM[1], especially the fault handler, was doing too many things in scattered places. The code has to be clarified before the XIP changes are applied, to prove the correctness of the relevant change in the code, instead of adding complicate code fragments and claim that ”it just works”.

Code cleanliness

Changes made by this work to the *NetBSD* kernel touch VM code, which is one of the most critical code path and almost everything in *NetBSD* relies on. The changes have to be not only correct, but also harmless for non-*XIP* case performance, and clearly separated from non-*XIP* code.

Changes made to support *XIP* can't be centralized in one place but aare sprinkled in the source tree by nature. Such a functionality is difficult to maintain if its overall design is not clearly documented. Design documentation is desirable.

Memory efficiency

Save as much memory as possible. We save not only extra copy of executable files to RAM, but also page state object (`struct vm_page`) which is usually allocated one for each physical memory page. All the `struct vm_page` objects occupy about 1.8% memory of the size of the real physical memory pages on NetBSD/sgimips 5.0. [1]

Compatibility

While *XIP* is used in somewhat limited contexts, we don't want to require any unnecessary tasks. Our implementation is independent of filesystem types. Userland programs don't need any modification; both statically linked and dynamically linked programs work as usual. Access to file data within executable files is consistent. Although dynamically linkd programs don't really make sense, because they're copied into RAM.

Our *XIP* implementation resulted in introducing a few new concepts in the *NetBSD* kernel. We had to teach the new knowledges to the VM. We tried best to avoid any API backward-incompatible changes to any VM primitives. We did some cleanups to some complex VM code fragments (especially the fault handler) to isolate conditions and clarify responsibilities, but still, no backward compatible changes are made.

Performance

Read performance of ROM devices depends on device's access speed, which is usually very slow compared to RAM. CPU's cache (both for instruction and data) is mandatory to make *XIP* usable in practice. Although it might be possible to invent some page cache mechanism dedicated for *XIP*, we don't consider this for simplicity in this paper. It would be useful if filesystem is optimized for *XIP*. At this point we use FFSv1 as a proof-of-concept target. Filesystem optimization is discussed in later sections.

3 Use cases

3.1 Possible users of *XIP*

XIP makes sense best where available RAM size is extremely limited. Most typical cases are embedded devices like cellphones. Smaller RAM size gives us not only the reduction of cost but also power consumption at run-time. Advanced users who want to run *NetBSD* with very low power consumption, and who can customize userland are all possible targets of *XIP*.

3.2 Requirements

NOR FlashROM

This is the most important, but the only hardware requirement. *XIP* doesn't make sense at all without this. Once you have write a filesystem image into a ROM, you don't need write operation at run-time.

[1]On *NetBSD* 5.0 sgimips, the size of `struct vm_page` is 72. Given that page size is 4K, (1 * 1024 * 1024) / (4 * 1024) = 256 `struct vm_page` objects are allocated for 1MB ROM and total size is 72 * 256 = 18.4KB.

Kernel customization

While *NetBSD* is moving toward modular and binary distribution, *XIP* is not provided as a part of binary distribution. It's is also very unlikely that *XIP* support is enabled in the official binary images, because *XIP* is used only in certain contexts. It adds extra code, though it's very small, in some very critical code paths like fault handler. Enabling *XIP* is not a good option for the single official binary release.

Userland customization

Although our XIP implementation can execute any userland programs, users are recommended to customize and build their own userland.

When a write occurs into XIP'ed process address space, the VM handles it as copy-on-write; allocate a page on memory and copy the original data (on NOR FlashROM) to a RAM page. This means that users have to carefully avoid copy, otherwise everything is copied into RAM, and XIP doesn't make sense at all. Typically users are recommended to build a single statically linked program in userland. Such a program has to be configured and built by themselves.

3.3 Restrictions

Can't run from NAND FlashROMs

NAND FlashROM is getting cheaper and more widely used. Unfortunately NAND FlashROM can't be used for *XIP* because its block (data) is not directly memory-mappable; its data are accessed by exchanging a sequence of commands and a buffer. As a result, NAND FlashROM is used in a very different way from NOR FlashROM; NOR is for mainly executables like bootloaders or *XIP*-ready programs (either kernel or userland) and data files. Other data, imagine MP3 files in mobile audio players, would fit in NAND FlashROM. It's very common for embedded systems to implement both of NOR and NAND FlashROMs.

Can't compress filesystem images

It's been a common technique to compress the system's image to reduce the size of the image stored in the ROM. Typically such a compressed image is loaded and inflated from ROM to RAM by the bootloader. Obviously such a compressed ROM image can't be executed directly. Image compression is a very opposite approach of *XIP* where the size of RAM is considered more important than ROM. Users decide which

3.4 Development

Basically our *XIP* implementation will become part of the standard NetBSD source distribution. NetBSD's development environment is very consistent and self-contained; users can build NetBSD image (compiler, kernel, userland, filesystem image) on almost every POSIX-like operationg systems.

Our *XIP* implementation needs no special change except users have to build their own userland programs as a crunched binary, whose build procedure is a little special and different than the standard build procedure provided by NetBSD, build.sh. *XIP* makes sense for statically linked programs, which don't need to rewrite

3.5 Deployment

The current implementation doesn't support write. Users have to unmount the XIP'ed filesystem to update the filesystem image. Update could be done via write(2) to the flash(4) block device driver in NetBSD, or using firmware. Users have to carefully design the system and its deployment plan.

3.6 Operation

All users have to do is to mount the filesystem as read-only and xip options explicitly specified. Root partition (/) can be mounted as XIP too.

4 Overview

To realize *XIP* the following parts in *NetBSD* are involved:

4.1 Host (development environment)

As already explained, XIP will become part of the standard NetBSD source distribution. NetBSD distribution is truely self-contained; it contains compilers to build NetBSD, and both userland and kernel source. The whole NetBSD image can be built on most POSIX like operating systems like Linux and Mac OS X.

The current XIP implementation is transparent to filesystems, which means, users can use any filesystem for XIP image in theory. But in practice, NetBSD"s filesystem image creation command (makefs(8)) only supports ffs(4) and cd9660(4). Users are strongly recommended to use ffs(4) for XIP.

To avoid unnecessary copy from ROM to RAM, executables in XIP'ed filesystem image should have read-only text segment, most typically statically linked executables. In highly customized environments, NetBSD offers crunchgen(8) to build a bundled static binary as

Linux's well-known BusyBox does. The development interface of crunchgen(8) is a little inconvenient; developers have to get used to it.

Altenative to statically linked or crunched binaries is position independent, dynamically linked executable with read-only text segments. Unfortunately these formats are not common in NetBSD yet. When these will become available, these should be just part of NetBSD's toolchain, and users just have to build their binary with enabling some configuration option.

In summary, while NetBSD lacks many features now, the development environment is self-contained, and will be so even after new features are added.

4.2 Hardware

Basically the only hardware requirement to function XIP is memory-mappable ROM, that is NOR FlashROM. In reality, you should carefully design your product considering XIP's characteristics and your use cases. First of all, XIP's performance heavily relies on the read access speed of NOR FlashROM. Secondly, CPU's cache (both instruction and data) matters. Ideally user programs' text segments are fully resident in instruction cache. Actual cache and memory behavior depends on user programs' behaviors.

In extreme situations, you want to save active PTE used in MMU for text segment. Mapping long, read-only text segment using super page is easy in theory. But unfortunately NetBSD's VM doesn't support super pages yet.

4.3 Target (userland)

Our XIP needs help of userland programs in that they have to be built in an XIP-friendly form. That is, unnecessary write to text and data segments are best avoided, otherwise those pages are copied into RAM as copy-on-write.

Users are encouraged to use "crunched binary". Crunched binary is a statically linked executable file with bundled library functions. Usually statically linked executable files have copies of library functions. By crunching those statically linked executable files into a single file, those duplicate library functions are shared.

While our XIP implementation is transparent to userland executables, users have to understand the internal behavior to benefit from XIP, otherwise pages are implicitly copied into RAM and just keep running normally without any error.

Text segment is modified at run-time mainly for two cases; code relocation and debugging. Code relocation occurs in dynamically linked libraries whose actual addresses are determined at run-time. The dynamic (runtime) linker collects necessary dynamic libraries, maps them into the process's address space, and fixes the addresses embedded in the text segment of the main program and libraries. The actually modified code and data depend on the format of executable files. Some architectures may support read-only text relocation, where code is carefully generated to be position-independent and addresses are resoved by referring to the intermediate relocation table in .data section. Thus text segment is not modified at runtime. This topic is beyond this paper's scope. Those who want to build a complicate set of userland programs may consider this approach.

Text segment may be modified for debugging purposes. Internally this is seen to VM as a write operation to a private map. This just works in XIP environment.

The organization of user programs should be carefully designed, especially placement of data. The basic rule is to concentrate read-only data and don't modify them. UVM doesn't allocate memory for .data and .bss sections until pages are really modified and write access is trapped by protection fault. Read-only data marked as "const" in C code result in read-only segment (".rodata" in NetBSD) in the executable file. Other data will be modified at runtime sooner or later. There is no point to record these data as is in NOR FlashROM, because those data are destined to be copied to RAM. Users may consider to compress these data in filesystem and read them via I/O. Some advanced filesystems like AXFS[3] automatically compress data segments and store them transparently. Even with such an assist from filesystems and other infrastructures, users have to carefully design their programs.

4.4 Target (kernel)

Block device drivers

XIP-capable ROM devices behave in two ways. Firstly it's a block device to provide usual block I/O interface for kernel to access filesystem metadata. Secondly it's directly mapped to user process spaces.

Usually such a memory mapping operation is done via character device's `mmap()` interface. In *XIP*, the device only has to pass the physical address to filesystem subsystem at mount time. `mmap()` is called by kernel's exec handler, and resolved in the succeeding fault handler.

Filesystem

XIP makes sense only for directly memory mappable ROM devices. These will be recognized as a block device, written a filesystem image, and mounted onto a mount point.

This means that the capability of *XIP* is known at mount time. If an *XIP* capable block device is mounted, kernel marks the mount point as *XIP*-capable. If a vnode is created on that mount point, kernel marks the vnode as *XIP*-capable. This reduces run-time checks.

Except these condition checks, filesystem is independent of *XIP* in the current implementation. When VM handles a fault to map a device page, VM asks filesystem the address of a given page-sized segment of a file. It's specific to filesystems, but nothing special is needed for *XIP*, because metadata is accessed via block device interface, which is provided by all *XIP* capable block device drivers.

NetBSD has two interfaces to access filesystems; buffer cache and page cache. The former is used mainly for metadata (file attribute) and raw data retrieval without opening a file via the standard file I/O interface for users. Buffer cache relies on the block I/O interface. This means that XIP capable NOR FlashROM drivers have to provide not only mmap() but also the block I/O interface so that its data can be accessed as a filesystem metadata by the kernel.

Kernel execution handler

Userland program execution is a special kind of memory mapped file access tightly coupled with execution context (process). Like usual file mmap(), program execution is done in two stages, preparation (mapping) and actual access resolution (fault handling) as explained in the following sections. When a process requests the kernel to execute a program, the kernel looks up the given path, locates the file, and read its program header metadata. In NetBSD, this part of access is done via buffer cache interface. Next kernel prepares a process context including per-process address space data structures. The kernel asks VM to map the needed program sections into the created address space following the read program header metadta.

Userland program's address space entries are always associated with vnodes; after the executed process's address space is created, the content of the executed file is accessed via VM's vnode interface (vnode pager). Once program execution is bootstrapped with help from the kernel, and its execution is started, remaining behavior is transparent to the kernel and VM. Userland programs may map a new executable code to extend itself (dynamic link).

When the mapped address space is executed in userspace, the machine's *MMU* causes a memory access fault. VM is responsible to load the actual page and register the H/W mapping information so that the faulting process will see the data when it's resumed its execution.

VM

VM plays a major role in the XIP functionality. Especially XIP is about device pages (like NOR FlashROM). The author had to teach a new knowledge of device pages to the NetBSD's VM subsystem called UVM. It was a fundamental design change (despite the small amount of diff) made against one of the most complex subsystems in NetBSD. The detail of UVM is described below.

5 The UVM

5.1 History

UVM[1] was developed by Charles D. Cranor and Gurudatta M. Parulkar. It was merged into NetBSD 1.4 in 1999. The initial motivations why it was developed to replace BSD VM were the following four reasons:

- complex data structures
- poor performance
- no virtual memory based data movement mechanisms
- poor documentation

Later another big change called UBC[2] was added into NetBSD 1.6 in 2002. UBC unified buffer cache and page cache so that it could provide a unified access to page caches both for I/O methods (read() / write()) and memory mapped file (mmap()).

UVM and UBC are often told together as one of the beggest and greatest features of NetBSD, and have proved its usefullness. Unfortunately, these are actually very complex systems, and understood only be a limited number of people even in NetBSD developer's community. Such a situation has promoted the illusion of the greatness of UVM and UBC unnecessarily.[2]

5.2 Key features

One of the most complex data structures seen in BSD VM was object chains. BSD VM tracked copy-on-write data by creating a new overwriting object and linking it to the lower overwritten object. Thus if an object repeats copy and overwrite, the chain extends. Managing such a chain is a very complex operation. UVM addressed this by introducing overlay layer (amap) and flattening the chain; copy the overlay layer data rather than chain it, while managing those objects cleanly and avoid unnecessary copies. This also improved performance for copy-on-write operations.

Another big change introduced by UVM was virtual memory based data movement (zero-copy), which is considered to be faster for transferring large chunks of data

[2]The author believes that great codes that are not understood are not so great.

than copying them. According to the paper, UVM introduced three mechanisms to achieve zero-copy; page loaning, page passing, and map entry passing. Unfortunately, it seems that there is only one feature, page loaning, that is implemented and working. Its behavior is difficult to understand. It's known that page loaning are not good for multi-processing environments.

UBC totally changed the file I/O and page cache behavior. While the core UVM interface was not affected very much, the dirty work to manage page cache consistency is now responsible to pagers. Especially vnode pager's page cache management code (genfs) is known as one of the most complicate codes in NetBSD. XIP benefits UBC's work of unified page cache; the substantial change needed for XIP is only to represent device pages as page caches. In that regard XIP is considered to be an extention of UBC.

5.3 Virtual address space map (`struct vm_map`) manager

UVM keeps virtual address space information as `struct vm_map`, which represents a single contiguous virtual address space. This doesn't mean all the address space it has are really used. `struct vm_map` is responsible to all the operations associated with the virtual address space. Note that address space and actual memory on RAM are distinguished concepts. `struct vm_map` is only about the former.

Actual space allocated in a given `struct vm_map` has a set of `struct vm_map_entry`, which represents a single contiguous virtual address space with a coherent attribute (protection, etc.). A variety of operations are made against these data structures, like allocation, deletion, copy, share, changing attributes, etc. The map manager concentrates on the efficiency and correctness about given constraints. *UVM* has to keep information as simple as possible even after time goes.

It doesn't, however, really need to focus on performance; history has shown that map entry management is so complex that enhancing it causes additional complexity. Especially kernel has a single address space and reuse its memory heavily. NetBSD is moving toward implementing another layer (called kmem(9)) of space management in between kernel memory allocator API and UVM map manager. kmem(9) caches address spaces and memories given by UVM map manager. It utilzes given resources as far as possible, and ask more resource only when it's really needed. kmem(9) is modeled after Solaris.

XIP has nothing to do with this layer, because map manager interacts actual pages only via the pager interface. XIP'ed pages are transparent to the pager interface.

5.4 UVM Object and pager

UVM's basic functionality is to help access to a variety of things laid in various devices, using page-sized memory chanks as a cache. UVM provides an uniform method to access those devices. UVM represents these as an object; a set of methods and data (backing store devices and caches) with linear address. This is called UVM object (`struct uvm_object`). Such an object-oriented abstraction is known as a well-established methodology to design a complicate system like VMs. UVM's core part doesn't need to know what kind of backing store a given UVM object belongs to, and can focus on provided page cache.

The method part to manage consistency between page cache and backing store is called pager. Pager is responsible to manage its linear address, where page caches are usually mapped. In the UBC's world, all access to UVM object is via as memory mapped; even file I/O access is converted to memory mapped access by remapping user's I/O buffer onto kernel address space temporarily. Thus pager is always entered from fault handler caused by memory access fault.

There are a few "special" pagers; UBC pager, device pager, and Xen's priviledge page pager. These have a special fault handler so that it can override the default handler's responsibility. UVM objects of these pagers don't have actual page caches. XIP's behavior can be seen similar to device pager, because XIP also deals with device's pages. Unfortunately the current device pager can't handle copy-on-write or other features to execute programs.

As already explained, pagers are basically entered only from the fault handler. The basic pager operations are "get pages" and "put pages". When fault handler wants to get a page cache in a given UVM object, it asks the object's pager to "get" the page. The pager looks up a page cache already loaded in memory. If it doesn't exist, the pager executes I/O operations to retrieve the relevant data from backing store.

Vnode pager is responsible to manage vnodes (files). Vnode management is a super complex task because filesystem is complex. Filesystem provides a standard set of operations to manage data stored in storage devices. UBC vnode pager maps page size granularity page caches and blocks in filesystems. This mapping is one of the reasons of the complexity. The another reason is that vnode behaves asynchronously for performance reasons.

Write operation ("put pages") collects pages to be written back to backing store. Vnode pager has to carefully manage pages. Pages may be memory mapped to user's address space. Pages may be shared among multiple address spaces too. Clean pages should not be written back.

5.5 Overlay (*AMAP*) manager

The overlay manager in UVM is composed of two main data structures; amap (`struct vm_amap`) and anon (`struct vm_anon`). amap is a mapping with a linear address like UVM object. amap is allocated for each address space to keep track of overwritten part. amap has slots to store its own privately modified pages. These private overwritten pages are anons. anons fill amap's slots. These are special forms to avoid object chain for privately modified, copy-on-write data. They're shared and reference counted where possible.

anon has no actual backing store, because it's content is private; the modified data is visible only to the existing process. anon may be swapped out instead, when available memory is getting smaller. Swap is optionally configured by user to temporarily put these "anonymous" memories in running system.

5.6 Physical page and segment management

The primitive data structure to represent freely available "physical" memory page is `struct vm_page`. When a kernel boots, its image is copied by boot loader into the main RAM. Kernel calculates available memories as segments asking firmware or lookup hardware configuration. After kernel allocates some persistent data structures (like page tables), it re-calculates available memories. After things are bootstrapped, kernel registers register those freely available memory pages as segments to the VM.

VM allocates VM pages for each freely available memory pages. VM page is a metadata of freely memory page. It keeps state of the associated page-sized memory. VM page has mainly two distinct states; paging and H/W mapping. Paging is activity to read or write backing store data from / to the memory page. These pages are called page cache. VM keeps such a state in VM page because I/O is considerably slow operation compared to memory operations.

H/W mapping state information is about consistency of memory page and relevant CPU cache. When a memory page is mapped with CPU cache disabled, VM doesn't need to keep track of the memory page, because non-cached access is volatile, stateless. Cache means a copy of data put in the original location. It's often software's responsibility to "manage" these copies and consistency. This is also important where a single page is mapped in multiple address spaces too. UVM and its ancenstors have used the structure called `struct vm_page_md` to store "machine-dependent" part of VM page. Since H/W mapping is done in machine-dependent code called pmap (explained later), we can assume H/W mapping related state is kept in `struct vm_page_md` for now.

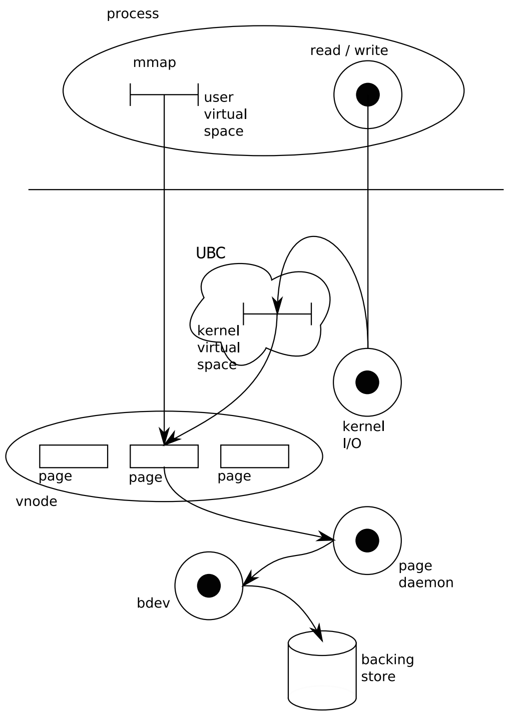

Figure 1: UBC and page cache

UVM records a contiguous physical RAM segment as VM segment, `struct vm_physseg`. VM segment has an array of pointers to its VM pages. VM segment's role has been limited, because most information of memory pages are in VM page. VM segment, however, will be more important because it's the best data structure to store physical segments of device pages.

The author has extended the use of VM segment to store device pages too with newly added members meant for managed device pages. Note that we don't allocate VM pages for managed device pages, because we don't need most part of VM page which stores paging activity state. Instead we decided to allocate only H/W mapping related information. The detail will be explained in later sections.

5.7 *pmap*

pmap is the abstraction layer of hardware's physical memory management unit (*MMU*) handling. *pmap* data structures are devided into two parts; per-address-space structures (`struct pmap`), and per-page structures (`struct vm_page_md`).

Per-address-space data is mainly about page tables and page table entries. pmap stores all address spaces' H/W mapping data in main RAM so that it can lookup the relevant page table entry and reload it into MMU, when MMU has lost the page table entry. Kernel mappings that are not never paged out ("wired") are called unmanaged pages.

Page fault for unmanaged pages never enters UVM's fault handler, because unmanaged pages are wired, and don't have any backing store.

Per-page data structure is used to keep track of physical-to-virtual mapping (PV map for short) entries for managed pages. This is because a) we want to lookup H/W mapping state using virtual address as a key, and b) pmap has to notify H/W mappings sharing a single page. There are many cases where a page is shared (commonly mapped) among multiple virtual addresses. If a shared page is changed somehow, pmap has to invalidate all the H/W mappings referring to the changed page, otherwise those virtual addresses will see the stale cache data.

5.8 Fault handler

Kernel and processes running in Unix and other modern operating systems have virtual address space. Its behavior can be roughly described in two stages; preparation and resolution. Virtual address space users prepare a "map" by associating their address space to a memory or some object that can be abstracted by a linear address like devices. Next, VM resolves actual accesses to memory or devices by catching MMU's page access fault. The latter is called page fault handling. The code doing paging fault handling is called fault handler.

Unmanaged page is easy to handle; what needs to be done is only about filling MMU's H/W mapping entries. OS doesn't need to take anything into account other than H/W mapping entries, because unmanaged pages have no cache, no backing store, no duplication of data, and no inconsistency.

Managed page's fault handling needs more work; VM is responsible to prepare the faulting page as a page cache before returning back to the original execution context which is accessing the virtual address space. If the page cache is found in memory, VM just re-enters the H/W mapping entry to MMU. If the page cache is not in memory, VM invokes the pager to retrieve the relevant data from the backing store using I/O subsystem. This means that fault handling is potentially a very slow operation. A variety of techniques are introduced to reduce the chance to cause I/O, which makes the fault handling code more complicate.

UVM's fault handler is in a little odd shape because the existence of special handlers where pages are not usual page cache of backing store objects. Other than those special handlers, UVM's fault handler is roughtly divided into two parts, lower (UVM object) and upper (overlay). At the entry the fault handler checks if the faulting address is covered by the overlay (amap). If yes, the handling is passed over to the upper handler. Otherwise the lower handler.

Lower fault handler is responsible to retrieve the faulting page cahe from the backing store using I/O. Organization of page caches and backing store is specific to UVM object (pager) types. Vnode pager maps UVM object's linear address to actual block address of storage devices by querying filesystem's code (`VOP_BMAP()`). Another UVM object pager is "aobj". aobj uses swap as its backing store. This means that its contents' lifetime is only during the system's uptime. aobj is useful for kernel data structures that can grow, but doesn't need to be always resident in memory.

When the faulting address is overlaid the upper handler resolves the fault. Overlay data structures are designed to be shared among multiple address spaces (VM map entries) so that UVM doesn't need to manage page chains for privately modified data. UVM copies a privately modified page on-demand. This is possible because MMU can catch protection fault; VM registers writeable pages as read-only H/W mappings so that write access to those pages cause write protection faults. The fault handler knows the original protection (writeable) and the faulting (access) protection. The fault handler belatedly copies the faulting page into a new page, updates the H/W mapping to point to the new physical page address, then back to the trapped context.

The on-demand page copy to the overlay is called "promotion". Promotion can happen both from the lower UVM object layer and the upper overlay layer. When a promotion from the lower layer occurs, the fault handler has to prepare two data structures, amap, and its slot, VM anon.

6 Design

6.1 Device page handling in the fault handler

The biggest question to realize *XIP* in *UVM* is how to represent those memory-mappable device addresses in the fault handler. *UVM* has basically 2 special fault handling cases (*UBC*, character device) and the generic handler. Two possible approaches were considred. One is to teach copy-on-write handling to the character device handler. The other is to teach device pages to the generic handler.

After much consideration[3], we decided to go to the latter, because copy-on-write handling implemented in the generic handler is very complex. Reimplementing it in the device handler is not good too in that code is duplicated.

It turned out that teaching device pages to the generic handler, along with vnode pager, was not that difficult.

[3]The author experienced tough time to made this decision (changed directions LOTS of times), because it was early development stage, and he didn't understand the code very well.

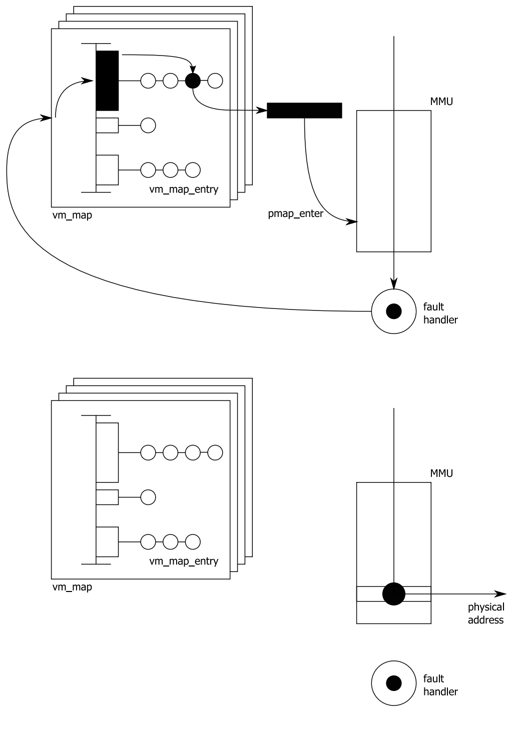

Figure 2: Page fault handler

This was done by abstracting the `struct vm_page *` object. `struct vm_page *` object is a struct, whose object is allocated at system boot time, one for each page cache. It's the metadata of page cache; most importantly it keeps the state of the matching page cache's paging state. In fault handing's context, the only relevant information in `struct vm_page *` is physical address of the page cache.

6.2 Device page representation in *UVM*

To design the representation of device pages in *UVM*, we identified the characteristic of device page and compared it to page cache.

	page cache	device page
mapping	physical memory	physical device
paging	yes	no
attribute	per-page	homogeneous
metadata	`struct vm_page`	`struct vm_physseg`

Those are substantially different in that device page are persistent (never involved in paging) and homogeneous. Considering these, we concluded that device page doesn't need `struct vm_page` metadata like page cache. This works because

- The fault handler allocates an array of `struct vm_page *` on the stack. It's filled by the vnode pager, and later passed to pmap(9) to register H/W mappings. That is, the `struct vm_page *` object is almost opaque to the fault handler.

- The `struct vm_page *` is dereferenced in only limited number of places for related to paging activities and some exceptions like wired and/or loaning.

```
/*
 * encode a device's physical address
 * into struct vm_page *
 */
#define PHYS_TO_VM_PAGE_DEVICE() ...

/*
 * decode a device's physical address
 * from struct vm_page *
 */
#define VM_PAGE_TO_PHYS() ...
```

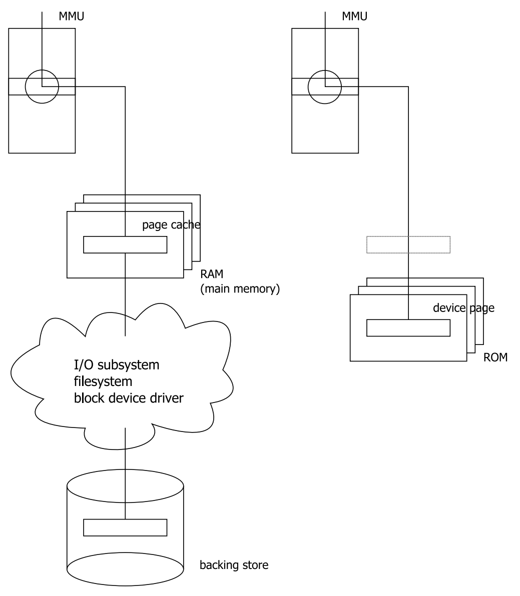

Figure 3: Page cache vs. device page

In the current implementation, the `struct vm_page *` for device page is a pointer with a magic value is encoded. This value is encoded by the vnode pager when the physical address of the page in the block device is known by calling `PHYS_TO_VM_PAGE_DEVICE()`.

```
/* register device memory for general use */
bus_space_physseg_t
bus_space_physload(
    bus_space_tag_t space,
    bus_addr_t addr, bus_size_t size,
    int prot, int flags);
void
bus_space_physunload(
    bus_space_physseg_t seg);

/* register managed device pages */
bus_space_physseg_t
bus_space_physload_device(
    bus_space_tag_t space,
    bus_addr_t addr, bus_size_t size,
    int prot, int flags);
void
bus_space_physunload_device(
    bus_space_physseg_t seg);
```

6.3 Device physical segment

Managed device page

Before device pages are introduced, UVM and PMAP had no knowledge about mmap'ed character devices. UVM only managed physical segments of the system's memory (RAM). When PMAP is given a physical address to map, it looks up the managed physical memory segments. If the physical address is included one of those segments, PMAP considers the physical address as "managed". Otherwise, PMAP assumes the address as "unmanaged" and maps it as uncacheable page. This is not acceptable behavior for XIP for performance. XIP pages should be cacheable like other executables, or performance would be miserable.

To address this, UVM should provide information for PMAP to judge a given physical page is managed or not. We added physical address segment data for those device pages too. Now when PMAP is given a

Device physical segment registration

Device drivers have to register their memory-mappable region to the VM, so that VM recognizes such a region and manages it. We'll introduce new functions as part of device drivers API.

`bus_space_physload()` and `bus_space_physload_device()` register a specified device space as part of VM aware managed space. `bus_space_physload()` is for managed device spaces used as page cache, and `bus_space_physload_device()` is for managed device spaces used as device page.

Copy-on-write

UVM is so smart that it delays to allocate data segment's page cache when it's firstly written. This means that the device page mapped into a process seeing its data segment is replaced with a writable page cache (anona), while the virtual address stays the same address. With cache enabled, UVM is responsible to make the process see the newly replaced data, by invalidating the cache content assosiated for the mapped virtual address and page.

This is usually done in PMAP layer for the faulting process whose address is being updated. The problem is when the upper layer to which a page is promoted is shared among other processes. UVM has to tell PMAP layer that the physical page replacement affects other processes too. In order for PMAP to invalidate other processes' cache, PMAP layer has to track what process's address is mapped to what physical address. This is called as PV mapping.

Each PMAP implementation is responsible to PV management. In reality, those implementations are classified into two categories:

- Have a global hash (x86).
- Associate those PV entries to the relevant `struct vm_page` objects (arm, mips).

The former type works as is because it doesn't make any difference about added PV entries. The latter needs remedy because we don't allocate `struct vm_page` objects for device pages. Here we have two choices:

- Give up shared amap.
- Maintain PV entries for device pages separately.

Giving up shared amap means that we have to copy amap everytime a process is forked. This overhead could be considerably big in usual Unix use-cases where processes are very often forked. It's also possible that some user want to run a highly simplified userland, where only a few processes run and they don't fork and copying amap for every process is expected to have little impact about memory usage.

To maintain PV entries for device pages, we need some additional code and data in UVM. We implemented a very simple, global hash lookup table. If PMAP is given a physical address, and it's known to belong to a device page, PMAP looks up the physical address in the hash and finds the PV entry header, walks the list and finds the matching PV entry.

The decision to make this simple was made because such a operation is considered a rare operation. When an

```
        struct vm_page *pg;
        ...
+   if (!uvm_pageisdevice_p(pg)) {
        pg->flags &= ~PG_BUSY;
+   }
        ...
```

XIP program is exec'ed, its sections are mmap'ed. When a page in the data section is first written, the access is trapped by the MMU and then the fault handler allocates a page cache, copies the data, replaces, and promotes it. Once a page is promoted to anon, it's dealt with as a page cache, and no more XIP specific handling needs to be taken account into.

The another point is that device pages that are supposed to be promoted are all in data sections. Those pages are very likely to be placed continuously in the filesystem image. Small hash size should not be problematic.

7 Implementation

7.1 Physical Page and Segment Manager

The first thing is to define device page. We took an approach which affects the least impact to the existing code, while achieving the goal to make `struct vm_page *` opaque. The new definition of `struct vm_page *` is that:

- `struct vm_page *` points to either page cache or device page.
- Which type `struct vm_page *` points to is queried by a new function, `uvm_pageisdevice_p()`.
- `struct vm_page *` can be dereferenced only if it's page cache.

Thus we can leave almost all the code as is. Code paths that have to deal with device page are very limited; mainly the fault handler, and some pager codes. Most changes, whose details are explained in the following sections, are to skip page cache handling if a given page is device page. Typical code fragment looks like:

7.2 Fault Handler

The fault handler's responsibility with respect to device page is almost transparent; it traps a page fault, looks up the faulting page's metadata, asks a pager to load the page, then enter a H/W mapping. However, the fault handler has become very complex to support UVM's enhancements.

```
uvm_fault() {
        check_and_prepare();
        if (upper) {
                handle_upper();
        } else {
                handle_lower();
        }
}

handle_lower() {
        if (need_io)
                do_io();
        if (need_promote)
                do_promote();
        enter_hw_mapping();
}
```

The heart of this paticular task is to realize the responsibility of the fault handler, identify the relevant code, judge if it's related to device page, then insert conditions in places.

Basic code flow of the UVM fault handler looks like XXX. Device page is handled in lower fault code, because it belongs to vnode, which is one class of an object pager. The changes made in the lower fault code path are to skip either page cache specific behavior, or some special features like page wiring and page loaning. We gave up wiring and loaning support because they rely on the metadata (`struct vm_page *`) to keep the those special states.

7.3 Vnode Pager

In XIP, a device page in an executable file are mapped in a file, which is represented as an object with a range. If a user accesses the file, either mmap or read/write, a page fault is triggered, and the fault handler traps it. Next the vnode pager is asked by the fault handler to do two things in order:

- If a set of pages is resident, return status
- If a set of pages is not resident, address the file blocks matching the requested pages, do I/O, then return status

It's obvious that this scheme is to manage the complexity of filesystems where:

- Files are put in a backing store. To read / write a file from / to there is very expensive task and slow. Which leads to introducing page cache. While page cache improves performance, it also brings more complexity.

- Mapping of files are complex, because filesystems manage directories, long names, etc.

However, careful investigation revealed that we can effectively omit most of these difficulties for XIP'ed device files, by using a dedicated vnode pager for XIP. Its behavior looks like this:

- Map the requested region into page addresses, then return them

Pretty much simple, because device pages always exist where CPU can map and address. No paging and I/O are involved. This is enough for XIP whose filesystem is read-only.

Note, however, how page addresses and file blocks are used. In the usual vnode pager, those mapping is used to I/O; pager queries to a filesystem the actual block address of a given page cache. In VM, a file is represented as a linear object. In filesystems, the real blocks of the file are likely to be scattered in the backing store.

For XIP, such page/block mapping are used to map as a H/W mapping. Which means that the XIP vnode pager has to pass these addresses back to the fault handler, which in turn passes the addresses to PMAP, which handles the actual H/W mapping operation. And the only available way for the XIP vnode pager to return a set of addresses is to encode the address into the argument array of `struct vm_page *`.

Another thing to consider here is handling of unallocated blocks. For the usual vnode case, the given pages are zero-filled. For XIP, it's pointless to allocate zero'ed pages for each unallocated blocks because of memory consumption. We made these blocks to be redirected to a single dedicated zero'ed page. All unallocated blocks in all processes are mapped to this page. The XIP vnode pager encodes the physical address of this zero'ed page in the array as well as other blocks.

7.4 Kernel memory manager

As explained just above, we need a dedicated single zero'ed page. We allocate a page from pool's backend allocator ("pool page"). Pool page is good in that it returns a page-sized, wired memory, and on platforms with direct mapping, it's used (no KVA waste). Such a dedicated, zero'ed page may be useful in other parts (like /dev/zero driver). We plan to merge these in the future.

7.5 Pmap

It was considred to give PMAP a hint that a given physical address is device page. We didn't do this, because we didn't want to revise PMAP API only for XIP. We instead decided to impose PMAPs that support XIP to learn a little knowledge about device page. The needed changes to handle device pages are simple; only a few conditionals and look ups.

Another change here is PV maintainance. To support UVM's delayed overlay copy feature, we have to track PVs for all device pages that can be promoted (data segment). PV management is implementation dependent. Some has a global hash, others use `struct vm_page`. For the latter case, we have to maintain PVs of device pages somewhere. We chose to implement a generic global device page PV manager using a very simple hash code.

8 Other changes

8.1 Filesystem

Our XIP implementation is independent of filesystem types. However, due to the design of NetBSD filesystem, we have to change filesystem mount code path so that XIP is enabled when mounted block device is capable of XIP, and XIP mount option is specified. When filesystem mount code is passed XIP option, it queries the block device's physical address. If the block device supports XIP, it returns the base physical address back to the mount code, and it's recorded in per-mount data structure.

8.2 Block Devices

We developed a simple NOR FlashROM driver, because NetBSD has never had any MI flash driver. This is a block device which supports the usual strategy interface. This is needed because NetBSD accesses file's metadata via the buffer cache interface, which is different than page cache which is integrated with VM.

XIP capable block devices have to provide the physical address of the device. This is queried and told to the filesystem mount layer when the block device is mounted. This is only a cache; to avoid the physical address each time when device pages are handled in the vnode pager.

9 Measurement

(Apologies.)

10 Consideration

10.1 Memory consumption

(To be done.)

10.2 Other implementations

(To be done.)

11 Issues

11.1 Fault handler

We did a major clean-up of the UVM fault handler before applying the XIP change, because it had a very long, complex function, `uvm_fault()`[4], which had too many things to consider. We had to prove the changes made for XIP don't affect other parts badly, and also are placed in the right place. So we decided to split the big function into smaller pieces, where context is narrower and responsibility is clearer. After the clean-up, the XIP changes look reasonable; they're only about skipping paging handling, and ignoring some special cases like loaning or wiring. Even after the clean-up, the fault handler is still complex and needs more improvements as follows:

Special fault handlers

As explained previously, UVM has 4 fault handlers: the generic handler, and 3 special handlers (UBC handler, character device handler, and Xen's priviledge fault handler). Those special handlers are responsible to do everything which is done in the generic handler. The problem is that the responsibility is ambiguous. There are many code fragments duplicate among these handlers. We should establish a well-defined responsibility of the fault handler, and the code performing it should be concentrated in a single place.

Super pages

Now *UVM* assumes that all pages are of the same size (`PAGE_SIZE`). Most processors support multiple page sizes to reduce the amount of H/W mappings, and the frequency of page fault. In *XIP*, user program's text segment is truely read-only, thus suitable to be mapped by large pages. Large pages are useful for other memory-mappable devices like framebuffers, whose attribute is homogeneous.

The author plans to change the fault handler to deal with not only `struct vm_page *` but also `struct vm_physseg *`; meaning that managed page belongs to either page cache (the former) or device page. Device page fault actually needs "offset" too. When a fault is triggered against the middle of a device page, device pager returns the faulting device page as a segment (`struct vm_physseg *`) and its offset back to the fault handler, which in turn passes the pair to pmap. Device pages are don't involve paging activity, of course.

Supporting super pages for device pages is not difficult. Things become a little more complex when large pages are used for page cache. You have to maintain freelists of multiple sizes, split and/or merge pages, and teach all kernel subsystems that assume the size of page is fixed. This will result in involving huge amount of changes. Supporting large pages only for device pages would be a good step to start with for the moment.

[4]Actually `uvm_fault()` is an alias macro of `uvm_fault_internal()` which has the real code.

11.2 Reliability and predictability

Related to the previous topicss, one of the biggest problems of UVM is lack of resource management. UVM checks resource limits at higher level like when mmap() is called. However, after it once allows usrs to their resouces, UVM doesn't check resouces available for the system to function. Most typically example is that UVM always register H/W mapping entries for neighbor fault without caring the available resouce at the moment. Note that H/W mapping entry registration potentially allocates memory to keep on-memory copy of the mapping information. The resulting system's behavior has less reliable and predictability, which is a critical problem for serious embedded systems.

11.3 Loaning

UVM has a kind of zero-copy send, called loaning. When user's data is copied to kernel, kernel subsystem checks the size. If the size is big enough to compensate the cost of preparing zero-copy rather than copying the buffer (which needs no preparation). In *UVM*, the buffer is remapped into kernel's address space, then the real pages are marked as read-only, then they're passed to kernel. The user is not allowed to have to wait for the I/O to be done. If the user mistakenly writes while the I/O is on-going, *UVM* catches a fault to resolve the situation; allocate a new page, copy the data, then install the newly allocated page to the user's address space. Note that the user has a newly allocated page than the original one which was loaned to the kernel.

In the *XIP* context, loaning of device pages could be possible in that device pages are never paged out. The problem is that the loaning implementation in *UVM* needs the page metadata (`struct vm_page *`) to track the loaning count (because one page can be loaned multiple times). To address this problem, the loaning code has to revised to not use the `struct vm_page *` object. That should makes sense considering that loaning is not a persistent state; its lifetime is limited to the activation of I/O. Loaning is also a rare operation; only big data pages are loaned.

11.4 Wiring

NetBSD wires (pin-down) pages in some unclear situations. Those uses should be investigated one day, but unfortunately left in a pretty bad shape now. Wiring has two meanings in *UVM*. One is to prevent a page cache to paged out (pin-down). This is a straight-forward idea considering the cost of doing I/O to retrieve a page from a backing store is not acceptable in some situation. Page wiring is used.

The another is to forcibly mark a H/W mapping to be permanent in an MMU unit. This is a rather odd operation. Unix's virtual memory management has had an assumption that the number of the H/W mapping entries registered in an MMU is limited. The kernel instead keeps mapping information, which is needed to rebuild a H/W mapping entry, in local memory. In other words, H/W mappings in MMUs are assumed to be never persistent. The H/W mapping wiring violates this basic assumption.

XIP and device pages in general are wired as never retired into a backing store. However, because of the confusions of ”wiring” handling described abve, *XIP* avoids wiring where possible by telling the caller that the wiring request was failed.

11.5 Filesystem optimization

(To be done.)

11.6 Memory disk support

NetBSD has a pseudo block device called md(4), which emulates backing store using kernel memory. The content of md(4) is initialized either statically (mdsetimage(8)) or dynamically (typically by boot loader, or kernel's early boot code) before it's being mounted. When a program in a filesystem mounted on an md(4) block device is executed, the kernel allocates page caches then fills them by reading the backing store, that is the md(4)'s memory. This means the kernel has two copies of the program in memory. If XIP is applicable to this situation, we could omit page caches and save memory usage.

The problem is that the current implementation of XIP assumes device pages, which are never part of kernel memory. They're exclusive. The current md(4) implementation uses kernel memory as backing store, which contradicts the assumption made by XIP. A possible solution to this is to exclude md(4)'s backing store memory region from kernel's memory. Thus kernel is not aware of the md(4) memory region, and it recognizes the region as ”unmanaged”. Later if the region is registered as device physical segment, kernel will recognize the region as ”managed device pages”.

The solution describe above needs total rewrite of the md(4) driver and other related tools, which is beyond this paper. Thus this is reserved as a future work.

11.7 Dynamically linked programs

Dynamically linked programs are executed with help of the dynamic linker which reads the program header, loads (mmap()'s) the given program's sections, then reads the symbol table to resolve unresolved symbols; which means the dynamically linked programs' image copied onto memory are potentially modified by the dynamic linker before the real program execution starts.

However, it's possible to avoid modifications of read-only text segments by making all the code referencing external symbols lookup the indirect symbol lookup table. This means that the code (text) segment is not modified, but only the reloc is. Such a program can be generated with compilers and linkers. Dynamic linkers also need changes to handle those relocs. These are specific to architectures and beyond this paper. We're planning to convert all architectures to support such a feature in the future.

11.8 Kernel XIP

Kernel is already almost *XIP*, except that the details are very machine-dependant, and kernel can't do *copy-on-write*. Users need to map the kernel text ROM range at the right address. This has to be done in either boot loaders or machine-dependent early initialization code. In either cases, this is beyond the scope of this paper which deals with machine-independent parts.

11.9 Mount option handling

As already explained, our *XIP* implementation is neutral to filesystem. However, we lack a consistent way to pass mount options from mount commands to filesystems. We have to change the mount option handling code in every filesystem to be used for *XIP*. There is an on-going work to centralize such code in one place.

11.10 Zero'ed page handling

XIP is not the only users who need a zero'ed page in the kernel. `zero(4)` is a pseudo device which is shown as `/dev/zero` and read as 0s. It fills a given user's buffer with 0s. This can be efficiently processed if the kernel has a page-sized region filled with 0s.

Another use is, like *XIP* addresses, filesystem in general encounters unallocated blocks, which should be seen to users as 0s. *NetBSD* fills pages caches of unallocated blocks with 0s by calling `memset()` using the temporarily allocated kernel address space. (This is one of the reasons the kernel has to allocate a kernel virtual space (`pager_map()`) while handling I/O, even for direct I/O.) This should be rewritten to use the pre-allocated zero'ed page to avoid access to user's buffer via remapped kernel address space.

As of writing this, *NetBSD* has no consistent way to manage such a pre-allocated zero'ed page. We're planing to address this soon after the *XIP* is merged into the trunk.

11.11 Development environment

While our *XIP* implementation needs nothing special to develop a *XIP* capable system, the interface of creating a crunched program is a little inconvenient to use. It was developped mainly for creating installation media, like floppies, where available size of images is very limited. The infrastructure to build crunch binaries is a mixture of BSD make and AWK scripts, which is totally different than the one to build usual, non-crunched programs. The helper scripts parse a user's configuration file[5] and generate glue makefiles on-the-fly, which make programs read and decide how to build a special program ready to be linked against a final crunched program. The build procedure is very unclear. It also means that users have to maintain build procedures of crunched programs in scattered places.

Crunch binary is actually something like a collection of static libraries (archives); programs are compiled as a statically linkable object, with unnecessary symbols are hidden or renamed so that they won't conflict with other programs which will be crunched together into a single crunched program.

This situation could be simplified by moving the knowledge of the build procedure into a single place, that is, the system wide make file templates (`/usr/share/mk`). The problem is where to install intermedia object files; they are neither usual programs nor static libraries. If we once establish a consensus how to handle these files, the crunch build procedure would be done in almost the same way as the default build procedure for a collection of individual programs and libraries.

12 Conclusion

We have successfully implemented *XIP* into the *UVM* without losing any existing functionality nor any major code impact. *UVM*'s 2 layered representation avoids unnecessary data copy across proccess fork. *UBC* addresses data consistency of `mmap()` and I/O of device pages. While designing all the issues we've realized many existing design issues in *UVM* and found some interesting ideas which will benefit contexts beyond *XIP*.

[5]The configuration is called a "list" file, meaning we don't know how to call it other than the file name itself.

13 Acknowledgement

Matt Thomas `<matt@3am-software.com>` kindly suggested this topic, gave me the overall development direction, and pointed out many failures of my design. Chuck Silvers `<chuq@chuq.com>` taught me *UVM* / *UBC* behavioural details and the expected problems of this work. Noriyuki Soda `<soda@sra.co.jp>`, Antti Katee `<pooka@cs.hut.fi>`, and Masanari Tsubai `<tsubai@segv.jp>` gave me many useful comments in discussions at AsiaBSDCon 2009 and/or Japan NetBSD Users Group Annual Meeting. I'd also thank everyone who encouraged me on ICB.

References

[1] Charles D. Cranor and Gurudatta M. Parulkar. The uvm virtual memory system. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 9–9, Berkeley, CA, USA, 1999. USENIX Association.

[2] Chuck Silvers. Ubc: An efficient unified i/o and memory caching subsystem for netbsd. In *USENIX Annual Technical Conference, FREENIX Track*, pages 285–290, 2000.

[3] Sören Wellhöfer. Application execute-in-place (xip) with linux and axfs. sep 2009.