VIO

Virtual IO through Memory Emulation

Malte Zahn

Assessors: Prof. Dr. Claudia Fohry, Prof. Dr. habil. Josef Börcsök

March 24, 2022

Contents

1 Introduction

When debugging or inspecting programs, the usual approach is to make use of breakpoints, a mechanism by which the execution of a program is stopped at predetermined points. This allows for inspection of the register state, of global and local variables, and of the so-called call-stack, which is the list of functions that called each other in order to reach the breakpoint.

Often, this is not enough to find the root cause of a bug or problem, and one may prefer to stop program execution not based on a certain instruction or line of code being reached, but as the result of a specific variable or memory address being read from or written to.

For this purpose, some processor architectures offer a mechanism referred to as *memory breakpoints* or *watch points*, which theoretically allows one to do exactly that. Unlike breakpoints that trigger when a certain instruction or line of code is reached, a memory breakpoint is triggered when a specific memory address is read from and/or written to.

Conventional *breakpoints* are usually implemented by replacing the instruction at the requested address with some kind of branch instruction that transfers execution to the breakpoint handler which in turn allows the user to inspect the system. When resuming execution, the replaced instruction is temporarily restored and executed, allowing the system to resume execution as if nothing had happened. This approach allows for an infinite number of breakpoints (bounded only by available system memory) to be placed in program text.

Unlike conventional breakpoints, memory breakpoints are a limited resource. For example, the `x86` architecture, while providing a mechanism for memory breakpoints, not only defines this functionality as an optional processor feature, but limits the number of simultaneous memory breakpoints to a maximum of 4[1]. The obvious approach of artificially increasing this number by rotating active memory breakpoints is unfeasible, as this would require perfect prediction of which memory breakpoints need to be active when executing a particular set of instructions. This cannot be achieved without a degree of complexity that would ultimately exceed that of disregarding memory breakpoints and executing one instruction at a time while monitoring for effects on memory. This is the solution used by debuggers such as

[1]While speculation, the author believes these to be the reasons why some debuggers do not bother including facilities for the creation of memory breakpoints.

`GDB`, though this is "[...] hundreds of times slower than normal execution."[2]

Given the obvious advantages of memory breakpoints, but considering the restrictions that come with using them, a different approach for achieving the same end goal is designed and considered in this thesis.

More specifically, we implement a system called *Virtual Input/Output*, which offers a pure software solution for the implementation of memory breakpoints. It doesn't require architectural support for memory breakpoints, but only depends on the presence of a memory management unit (see Section 2.2).

To facilitate our approach, access to the regions of memory containing variables one wishes to monitor is virtualized, such that any such access is handled through abnormal means discussed in Sections 3.2 and 3.3. This allows for introspection of arbitrary details regarding the access made, including the ability to monitor for reads and writes to individual variables. This way, VIO does not pose any performance penalties when not accessing a virtualized region of memory, and the number of such regions is not limited.

Besides the example detailed above, VIO also makes it possible to implement other functionality such as:

- Emulation of memory-mapped peripheral devices
- Software-implementation of non-volatile memory (*NVRAM*)
- Black-box analysis of interactions between functions and memory, aiding in reverse engineering
- Journaling of memory accesses made by programs, including all accesses made to memory

This document is structured as follows. First, necessary background knowledge is conveyed while stating requirements imposed upon computers hosting VIO in Section 2. Then, Section 3 describes how VIO is implemented and integrated into a kernel. Finally, Section 4 gives information on a reference implementation of VIO, before Sections 5 and 6 discuss use cases and caveats, respectively.

[2] `https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html#index-hardware-watchpoints`

2 Background

This section explains terminology and the essential features of current computers that are necessary for the implementation of VIO.

2.1 Computers and Memory

Any modern computer in existence today is derived from the model of a *Turing machine*[3] and its definition of memory.

Briefly stated, memory in a *modern* computer can be abstracted as an addressable array of locations that can be read from and written to in arbitrary order.

The original Turing machine specifies that memory be consecutive, and theoretically unlimited. It takes the form of a tape and includes a tape-head located somewhere on the tape with the ability to read, write, and seek left or right. Additionally, a Turing machine remembers its current program-state[4] which, if combined with the contents read from the tape, is translated into a new program-state, a tape-head movement that is one of "move left", "move right", or "keep position", as well as a new value that is written to the tape prior to the optional movement step.

A *modern* computer's memory differs from this, mainly in that further complexity is added for the sake of performance and ease in programming, though it has been proven that modern computers are just as *turing-complete*[5].

In the following, we list important aspects of a modern computer that are relevant to VIO:

- Memory is linearly indexed, and no read/write head is used for the purpose of performing an access. Instead, the term *linear memory access* is defined as a read- or write-access being performed by indexing into an array of sequential *physical memory addresses*.

  The array is referred to as *linear memory* or *physical memory*, and memory indices in general are called *addresses*.

  Each physical memory address is usually capable of holding its own value independently of any other.

[3] Named after its inventor, Alan Turing, born 1912

[4] In literature, a Turning machine's program-state is held by the *state-register*, but we will call it the PC-register.

[5] The ability to simulate a Turing machine.

As noted in Section 1, debugging a program that accesses variables stored in memory requires knowledge of which addresses correspond to specific variables, requiring a mechanism through which to obtain the address of a variable. Such a mechanism is not further elaborated here, as it is independent of VIO.

- In the context of a Turing machine, we define the register-state as the current position of the head, combined with the current program-state upon which future head movements and program-state transitions depend. Notably, the contents of memory are not considered part of the register-state.

- In a modern computer, the *register-state* is defined similarly and is held by a finite set of designated storage locations called *registers*. These registers can be categorized as a single *PC-register*[6] that holds the program-state, and an architecture-dependent set of *work-registers*. The combination of all register contents forms the register-state.

- Work-registers only serve the purpose of augmenting the program-state described by the primary PC-register. They serve no designated purpose within a VIO implementation, but carry architecture-specific relevance for VIO instruction emulation, as described in Section 3.3.

- The PC-register represents the program-state and translates to the address of the instruction that is– or will be executed next. In the context of a Turing machine, it can be thought of as holding an index into an array of all possible program-states.

- *Instructions* are addressable, meaning that it is possible to interpret the PC-register as an address. The contents of memory read at this address can then be decoded into said instruction. This process forms the basis of how a computer assigns semantic meaning to every possible value the PC-register might take.

- The exact behavior of individual instructions must be known and replicable. Replicating an instruction's behavior is called *instruction emulation*, which is done by combining a decoded instruction with read/write access to memory and work-registers, before using software to emulate

[6]Standing for "Program Counter"

the steps taken by the computer hardware upon encountering the instruction. VIO takes this a step further by defining special handling for memory accesses done during VIO instruction emulation; see Section 3.3.

2.2 Memory Management Unit

For a multitude of historical reasons, modern computers include a piece of hardware called a *memory management unit*, or short *MMU*. It is a peripheral memory controller with which the computer can interface in order to control and limit access to linear memory made by processes[7], specifically in the case where multiple processes are executed in parallel on a single computer.

This is done by assigning each process its own self-contained instance of linear memory, then called an *address space*. The exact specifics of how address spaces are controlled is beyond the scope of this thesis. However, what is important here are some logical consequences that stem from the introduction of a MMU:

- Any MMU offers the ability to mark specific ranges[8] of addresses as $\overline{\texttt{PRESENT}}$, which stands for *not present*. Any access performed to such a range will differ from a normal memory access, in that the computer will branch to a designated program location (by means of loading a special value into the PC-register) called the *page fault handler*.

- The *page fault handler*, or short *PF-handler*, is usually a part of the kernel[9], and contains a set of instructions defining the consequences of such an access. Traditionally, a PF-handler serves a number of purposes[10], though this document will not go into further detail about such uses.

  The exact details of how a MMU is programmed are specific to that MMU, but every MMU must provide the means of specifying a function $P_{MMU}(a)$ to determine if a given address a is present:

$$P_{MMU}(a) \in \{\texttt{PRESENT}, \overline{\texttt{PRESENT}}\}$$

[7] An actively executing instance of a program
[8] Term defined in Section 2.4
[9] See Section 2.3
[10] e.g. *demand paging*, *copy-on-write*, and the termination of a miss-behaving process

- The PF-handler also has access to the computer's register-state prior to it being branched to. The value of the PC-register at that time is referred to as the *Fault-PC*, and the instruction it indexes is referred to as the *faulting instruction*, while the register-state at the time is called the *fault-state*.

Unless otherwise stated, all mentions of the equivalent terms address and *pointer* throughout the rest of this document refer to virtual addresses, meaning locations relative to the specific address space of the currently running program (unless otherwise stated), as opposed to linear memory.

2.3 Kernel

A *kernel* takes the form of a set of instructions that are loaded into every program's address space (as described in Section 2.2).

As such, a kernel forms a supervisor or controller of programs running on a system. It is responsible for distributing resources, and controlling when and how processes are executed, among other things. The exact details of most of these tasks are not of relevance to VIO.

The PF-handler is assumed to be part of the kernel, and must be designed or modified to include semantic support for VIO. Semantic requirements and modifications are described in Section 3.1.

Additionally, an interface for programs to control VIO must be provided by the kernel. Other parts of the kernel may also need to be adjusted or redesigned to support VIO. This is further discussed in Section 5.

2.4 Other Terms

The following terms are used throughout this document:

- *NULL*: A null-value, meaning a placeholder value that does not equal any other valid value except for itself. By convention, it can only be assigned to pointers.

- *Address ranges*: An address range $R = [R.\texttt{minaddr}, R.\texttt{maxaddr}]$ is a pair of addresses, one of which represents a lower bound ($R.\texttt{minaddr}$), while the other represents an upper bound ($R.\texttt{maxaddr}$). By convention, these bounds are always inclusive.

Given an address a and an address range R, we define:

$$a \in R \Leftrightarrow a \geq R.\texttt{minaddr} \wedge a \leq R.\texttt{maxaddr}$$

Given two address ranges A and B, we define:

$$\begin{aligned} A \cap B &= [max(A.\texttt{minaddr}, B.\texttt{minaddr}), min(A.\texttt{maxaddr}, B.\texttt{maxaddr})] \\ A = \emptyset &\Leftrightarrow A.\texttt{minaddr} > A.\texttt{maxaddr} \\ A \subseteq B &\Leftrightarrow (A.\texttt{minaddr} \geq B.\texttt{minaddr} \wedge A.\texttt{maxaddr} \leq B.\texttt{maxaddr}) \\ &\Leftrightarrow (A = (A \cap B)) \end{aligned}$$

- *Memory mapping descriptor*: A memory mapping descriptor describes the semantic behavior of memory accesses performed to a specific address range. This includes the intended and permitted use by programs, taking the form of mapping-specific semantic rules enforced by the PF-handler and MMU[11]. We define it as a tuple $(R, P, ...)$, where R is the address range it maps (describes), and ... represents additional, memory mapping descriptor-specific data (relating to the aforementioned rules).

  For an address $a \in R$, $P(a)$ equals the value of $P_{MMU}(a)$, as described in Section 2.2. The value of $P_{MMU}(a)$ must not necessarily be identical for every $a \in R$. Additionally, an MMU defines a so-called `PAGESIZE` constant (which is usually at least 4096) that states the size and alignment of chunks of memory for which P_{MMU} must be the same. As a consequence, `PAGESIZE` is also the size of the smallest address range which any kind of memory mapping descriptor can span. We define:

$$\begin{aligned} R &\neq \emptyset \\ (R.\texttt{minaddr} \ \textbf{mod} \ \texttt{PAGESIZE}) &= 0 \\ (R.\texttt{maxaddr} \ \textbf{mod} \ \texttt{PAGESIZE}) &= \texttt{PAGESIZE} - 1 \\ \forall a, b : \left(\lfloor \frac{a}{\texttt{PAGESIZE}} \rfloor = \lfloor \frac{b}{\texttt{PAGESIZE}} \rfloor \right) &\Rightarrow (P_{MMU}(a) = P_{MMU}(b)) \end{aligned}$$

[11] A violation of these rules, such as writing to a read-only address or accessing an inaccessible address (s.a. P_{MMU}), traditionally results in the program being terminated.

Furthermore, the value of $P(a)$ may change over the course of the memory mapping descriptor's lifetime. Any such changes are usually opaque to the program, a portion of whose memory the memory mapping descriptor is used to describe, but can be used to facilitate advanced memory access techniques, such as demand paging or copy-on-write. Such techniques are not further elaborated here, but VIO poses no hindrance to their implementation.

Traditionally, memory mapping descriptors can be categorized into two classes:

- *File mapping descriptor*: Generally used for memory regions occupied by a program's instructions.
- *Anonymous mapping descriptor*: Generally used for dynamically allocated memory, as might be returned by the C function `malloc(3)`.

In Section 3 we will introduce a new class for the purposes of VIO.

- *Page*: A page G is an address range whose size equals `PAGESIZE`. Furthermore, it's base address (G.`minaddr`) is divisible by `PAGESIZE`. In turn, every address $a \in G$ yields the same value for $P_{MMU}(a)$.

- *PFA*: In order to aid a kernel's handling of page faults, some computer architectures include means for the PF-handler to retrieve the precise address a to which a memory access[12] was performed that violated the MMU's constraints imposed on that address. This includes the case where $P_{MMU}(a) = \overline{\texttt{PRESENT}}$, though other functionality might exist which facilitates the same result (for example: writing to an address the MMU has been told to only permit read-accesses for). Such functionality is neither required for–, nor detrimental to the implementation of VIO.

  When the computer branches into the PF-handler, it also passes along the value of a, which we call the *primary faulting address*, or short PFA. For example, the `X86` architecture allows the PF-handler to determine the PFA by reading from the `%cr2` register. By collecting the set of address ranges accessed by the faulting instruction, it is possible

[12] The implied width of this memory access is 1 address unit.

to implement VIO without the use of the PFA, but for the sake of simplicity, this thesis assumes that the PFA is directly accessible to the PF-handler.

- *Function pointer*: A pointer that may be invoked by writing it to the PC Register, thus facilitating a branch to the pointed-to set of instructions.

  Like with normal functions, semantics allow a function pointer to accept arguments, as well as have a meaningful return value. The meaning of arguments and return values is context-sensitive and described as appropriate. A function or function pointer that does not yield a meaningful return value is said to return a value of type `void`.

  We use `C`-notation to encode functions and function pointers. For example, `long (*funptr)(int arg)` describes a function pointer named `funptr`, whose semantics state it returns a value of type `long`, as well as take a value of type `int` via an argument named `arg`.

- *V-Table*: A V-Table, short for *virtual table*, is an array of function pointers. By convention, elements of this array are given names, rather than indices.

- *Address unit*: The delta between two addresses is referred to as an integral value expressed in address units. For example, the size `N` of an address range `R` can be expressed in address units, by use of the expression `N = (R.maxaddr - R.minaddr) + 1`. Conventionally, computer architectures assign the width of a *byte* as the address unit.

- *Memory access*: A memory access describes the act (or intent/attempt) of accessing memory at a specific address range `R`. For the sake of simplicity, a memory access is often said to refer to a singular address a. In this case, $a =$ `R.minaddr`, alongside a (sometimes implied) word width that is equal to the size `S` (in address units) of the range `R`. The type of the word being accessed is written as `Word<S>`.

  For example, the expression `x = *(Word<4> *)p` performs a read-only memory access from an address range $R = [\mathtt{p}, \mathtt{p} + 3]$.

  The precise set of possible values for `N` is equal to the set of memory access sizes N_{valid} which can be expressed using any instruction accepted by the computer architecture. This is an architecture-specific set of

positive numbers and usually consists of only a few, small power-of-2 integers. For example, most 32-bit computers define $N_{valid} = \{1, 2, 4\}$, while most 64-bit computers define $N_{valid} = \{1, 2, 4, 8\}$. The greatest element of this set is called $N_{max} = \max(N_{valid})$.

- *Recursive page fault*: A page fault is said to be recursive if it is triggered from inside of the PF-handler, or some other function invoked by it, implying that the kernel was still in the process of handling a preceding page fault.

  Care must be taken to ensure that no page fault can ever re-trigger itself or cause infinite recursion in some other manner. Otherwise, the most likely outcome will be what is referred to as a *stack overflow*.

  VIO both uses and relies on recursive page faults to function properly, but also takes precaution to ensure that infinite recursion is impossible without making alterations or extensions to descriptions given by this thesis.

- *Memory access primitive*: A semantic description of what a memory access tried to achieve. Examples for memory access primitives include:

  - `Word<N> read(Address a)`
    Read `N` bytes starting from address `a`.
  - `void write(Address a, Word<N> v)`
    Write `v` to `N` consecutive bytes starting with address `a`.

  Other examples might include atomic operations, but for the sake of simplicity, this thesis forgoes their discussion.

- *VIO memory*: an address range R is said to represent VIO memory if every address unit it contains is associated with some VIO mapping descriptor (as described in Section 3):

$$\forall a \in R : \texttt{getVioMappingAt}(a) \neq \texttt{NULL}$$

  Note that not all addresses must necessarily refer to the same VIO mapping descriptor for the range to be considered VIO memory.

  Note also that instructions used to implement program code detailed in– or referenced by the following sections, are never allowed to reside in

VIO memory. The same also goes for pre-existing parts of the kernel's PF-handler, as well as its dependencies. Careless failure to adhere to these restrictions might result in a stack overflow or system crash. In general, it is never a good idea to use VIO memory to describe program instructions. Doing so is possible, but the primary intended use of VIO is to virtualize data, rather than instructions accessing said data.

3 VIO

To achieve its goal of virtualizing memory accesses, VIO extends the functionality of a kernel's PF-handler, as well as a kernel's capabilities of defining the semantic meaning of address ranges, as done by means of creating memory mapping descriptors using facilities such as the **NIX*[13] system call[14] `mmap(2)`.

For this purpose, a new class of memory mapping descriptor is defined, called a *VIO mapping descriptor*. A VIO mapping descriptor is defined as a tuple (R, P, V), where R and P serve the same purpose as they do for every other class of memory mapping descriptors (see Section 2.4). V represents a V-Table that is further detailed in Section 3.2. The creation of VIO mapping descriptors in general is detailed in Section 3.5.

VIO mapping descriptors always define $P(a) = \overline{\texttt{PRESENT}}$, thus guaranteeing that every memory access to an address $a \in R$ causes the computer to branch to the PF-handler (see Figure 1 on page 14).

Another data structure called the *VIO mappings table* exists for every process, as shown in Figure 2 on page 22. The VIO mappings table of the current process is controlled by the kernel and is accessible to the PF-handler. It allows for translation of an address a into the VIO mapping descriptor whose address range R contains a. This functionality is used to implement `getVioMappingAt()`.

Once a memory access into VIO memory has passed control to the PF-handler, it is up to said handler to determine if a VIO access was the cause before acting accordingly. The specifics of actions taken by the PF-handler to determine cause, as well as the consequences thereof, are described in Section 3.1.

[13] Collective term for operating systems such as Unix, Posix, Linux, BSD, KOS, ...

[14] The common term for mechanisms by which programs interface with a kernel.

To simplify VIO as described by this thesis, the following assumptions are made about the computer architecture. These assumptions are not necessary, but they help to simplify the presentation in this thesis.

- N_{max} (as defined in Section 2.4) is smaller than– or equal to `PAGESIZE`. From this, the greatest number of distinct pages (and thus potential VIO mapping descriptors) with which a singular memory access may overlap, becomes limited to 1 or 2 (2 only when its range's lowest address unit is part of a different memory mapping descriptor than its greatest address unit).

  In practice, this assumption **can** always be made, and the author is unaware of any (non-theoretical) computer architecture to which it would not apply.

- No instruction will ever attempt to perform a so-called unaligned memory access. An *unaligned memory access* is defined as a memory access spanning N consecutive address units starting at address a, such that $(a \texttt{ mod } N) \neq 0$. Consequently, an *aligned memory access* (the only possible kind as per our assumption) guaranties that $(a \texttt{ mod } N) = 0$. Combined with the previous assumption, this guaranties that any memory access always references exactly 1 page, meaning that the greatest number of distinct VIO mapping descriptors with which a singular memory access may overlap also equals 1.

  In practice, this assumption **cannot** be made, as even computer architectures which don't natively support unaligned memory accesses can be (and are[15]) made to support such behavior via special handling integrated into kernels.

  Note that the author's reference implementation, as described in Section 4, does *not* make this assumption.

[15] For example, `arm` does not natively support unaligned memory accesses, but the linux kernel handles attempts to perform one via emulation in `arch/arm/mm/alignment.c`.

The following diagram shows an example of the control flow and actions taken in response to an instruction `mov4 [A], B` performing a read of 4 address units from an address `a = A` into register `B`. Following a fault, the PF-handler determines the cause being VIO-related, before acting accordingly.

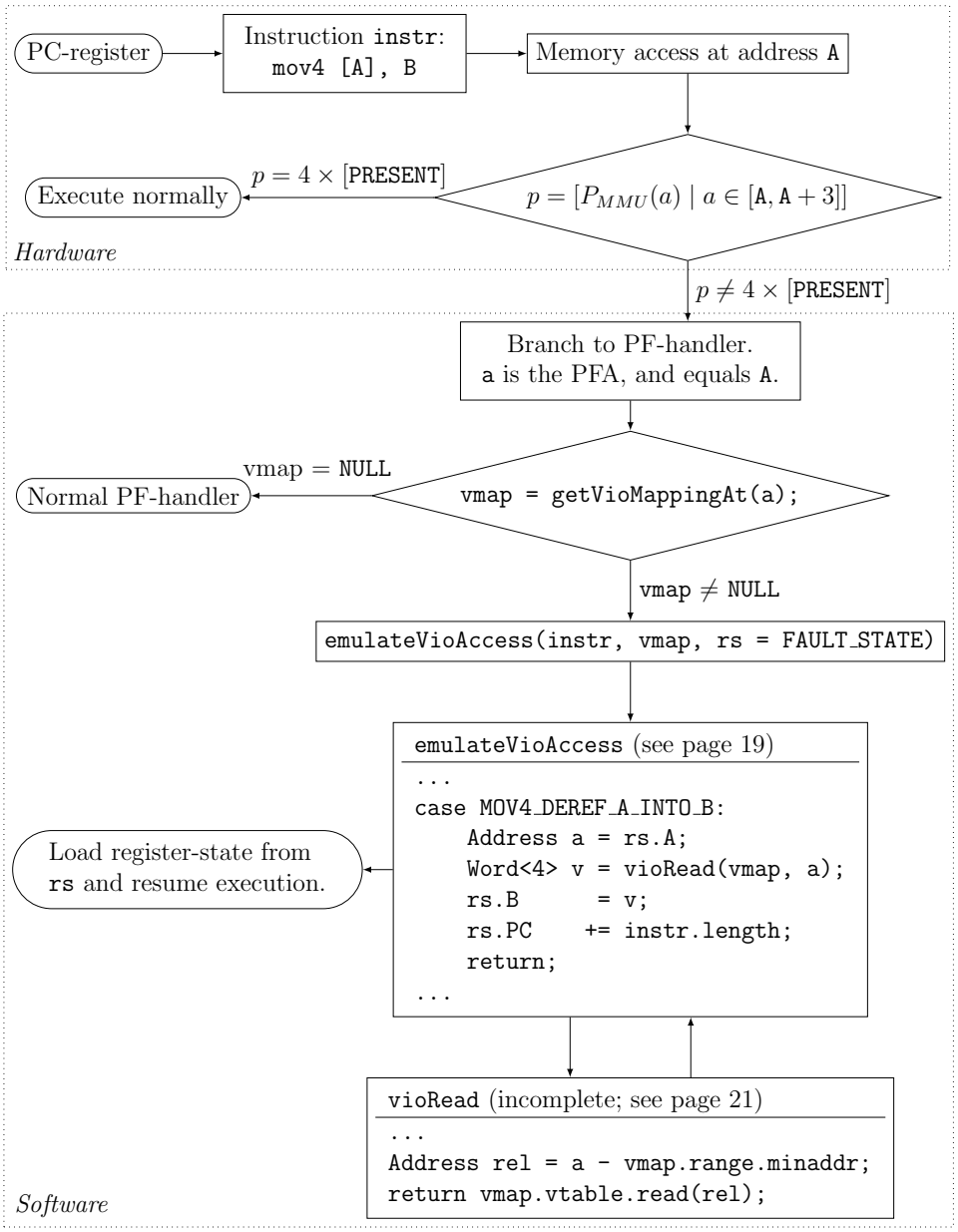

Figure 1: VIO Control Flow

3.1 PF-handler

Many reasons can exist for the PF-handler being entered. For the purposes of VIO, the only one of relevance is that of a VIO mapping descriptor existing at the PFA. When this is the case, the page fault is said to be a *VIO fault*.

Because a VIO mapping descriptor (R, P, V) implies $P_{MMU}(a) = \overline{\texttt{PRESENT}} \mid \forall a \in R$, and the PF-handler can query for the presence of VIO mapping descriptors, a page fault with $a =$ PFA becomes a VIO fault exactly in those cases when `getVioMappingAt`$(a) \neq$ `NULL`.

In order to detect and handle the case of a VIO fault, modifications are made to a kernel's existing PF-handler. These modifications first test if a VIO fault happened, in which case a call to `emulateVioAccess()` is made in order to handle it. Any other reason for the PF-handler being entered will be implicitly handled by falling through to the kernel's existing PF-handler (see Figure 1 on page 14).

Some computer architectures may not provide any means for the PF-handler to (easily) determine the PFA. In this case, the faulting instruction must be analyzed in order to test if it *might* perform a memory access to VIO memory. For the sake of simplicity, this case will not be further discussed.

When the PFA *can* be determined by the PF-handler (such as by reading from `%cr2` under `x86`), modifications made to the PF-handler look as follows:

```
PF_HANDLER() {
    Address               faddr = PFA; /* "Primary Fault Address" */
    VioMappingDescriptor vmap  = getVioMappingAt(faddr);
    if (vmap != NULL) {
        Address       fpc   = FAULT_PC;    /* "Fault-PC" */
        RegisterState rs    = FAULT_STATE; /* "Fault-state" */
        Instruction   instr = readInstruction(fpc);
        emulateVioAccess(instr, vmap, rs);
        return;
    }

    ...  /* Normal, kernel-specific PF-handler */
}
```

The act of retrieving the VIO mapping descriptor overlapping with an address is facilitated by use of the `getVioMappingAt()` function, which takes

said address as argument. Its implementation makes use of the current process's VIO mappings table, and its return value is the overlapping VIO mapping descriptor. When no such descriptor exists, NULL is returned instead. Details regarding VIO mapping descriptors are discussed in Section 3.2.

The `readInstruction()` function loads and decodes the instruction pointed to by the fault-PC. Such functionality has already been described in Section 2.2.

The `emulateVioAccess()` function represents the heart-piece of VIO's "Memory Emulation", and takes information about the instruction that caused the page fault (here: `instr`), the VIO mapping descriptor associated with the PFA (here: `vmap`), as well as access to the fault-state (`rs`). In the form as it is passed to `emulateVioAccess()`, `rs` can be thought of as a copy of the entire register-state as it was at the time of the page fault happening (the point where hardware transitioned to software in Figure 1 on page 14). Once the PF-handler returns, its contents are written back into the cpu's actual register-state, causing execution to resume at whatever context it describes at that point (including any modifications made).

Specifics regarding the implementation of `emulateVioAccess()` are described in Section 3.3.

3.2 Mappings

The central idea behind VIO is the virtualization of memory accesses. This is achieved by breaking down any possible memory access into a set of memory access primitives, such as `read` or `write`, that are combined within the *VIO operator table*. The VIO operator table takes the form of a V-Table.

Every VIO mapping descriptor (R, P, V) includes an instance V of a VIO operator table, thus making it possible to define custom behavior for memory access primitives on a per-VIO mapping descriptor basis.

The primitives stored inside a VIO operator table are conditionally invoked when dispatching memory accesses made during emulation of the faulting instruction following a VIO fault. The conditions for invocation to take place, as well as the process of emulation in general, is described in Section 3.3.

The set of memory access primitives defined by the VIO operator table must be capable of describing any type of memory access that can also be expressed using a computer architecture's native instructions. Undefined function pointers can sometimes be substituted, but for the sake of simplicity,

we assume that any function pointer used from a VIO operator table will have been defined during the VIO mapping descriptor's creation.

In the following, the layout and semantic contents of a typical VIO operator table (as also presented in Figure 2 on page 22) will be described while elaborating on the semantic behavior of memory access primitives encountered on typical computer systems.

Ignoring the possibility of atomic operations, we define simple `read` and `write` primitives, as also referenced in Section 3.3 and the rest of this thesis:

```
Word<N> (*read)(Address addr)
void    (*write)(Address addr, Word<N> value)
```

Unlike usual, during a VIO access the function pointers stored in VIO operator tables may point into the address space of a process different from the one performing the access. This makes it possible for one program to specify the VIO operator table used to describe memory accesses performed by another program. The creation and use of VIO mapping descriptors inside of programs is discussed in Section 3.5.

Note that it is usually not possible for a kernel to directly call a function pointer that has been defined outside of the kernel itself, without also risking the possibility of the entire system misbehaving or security to be compromised. For the sake of simplicity, we ignore such restrictions, but an implementation must take this into account and devise means to circumvent this problem. Such means are not further elaborated here, but the reference implementation provided in Section 4 solves this problem by using a request & response client/server model.

3.3 Emulation

As described in Section 3.1, any instruction that performs a memory access into VIO memory will cause the computer to unconditionally branch into the kernel's PF-handler. Next, the PF-handler confirms the cause of the page fault to be a VIO fault, before invoking the function `emulateVioAccess()`. As previously stated, the following information is passed during this call:

- The faulting instruction `instr`, as described in Section 2.2.
- The VIO mapping descriptor `vmap` $= (R, P, V)$ associated with the primary fault address, as described in Section 3.2. In the context of

`emulateVioAccess()`, `vmap` is also referred to as the *fault mapping descriptor*. Pseudo-code references the elements of `vmap` as `vmap.range` $= R$ and `vmap.vtable` $= V$.

- A copy of the fault-state (called `rs`), whose contents will be loaded by the PF-handler once `emulateVioAccess()` returns.
- Furthermore, `emulateVioAccess()` retains the ability of performing normal memory accesses to the linear memory of the current process, and will do so to facilitate recursive page faults and do non-vio memory accesses. When and how this is done is discussed later in this section.

The implementation of the `emulateVioAccess()` function takes a closer look at the faulting instruction (`instr`) in order to recreate its semantics in a manner that matches the instruction's natural behavior. This process is referred to as *VIO instruction emulation*. The term *natural behavior* here refers to the instruction's expected behavior, had it not attempted to access VIO memory.

During the act of VIO instruction emulation, any memory access which **might**[16] overlap with `vmap.range` is not performed directly, but is instead done by dispatching the relevant memory access primitives within the VIO operator table (`vmap.vtable`). The selection between invocation of operators from the VIO operator table, and performing a direct memory access, is encapsulated by VIO wrapper functions such as `vioRead()`. VIO wrapper functions are elaborated upon later in this section.

The act of VIO instruction emulation is highly dependent upon the set of instructions accepted by the computer architecture (including their composition and semantics). Without exception, **all** instructions must be emulated, even those that might not appear to include any means of accessing memory, and in turn VIO memory.

This is because any instruction is implicitly capable of accessing memory due to the fact that prior to being executed, the computer must first load said instruction from memory, similar to how we do the same with our `readInstruction()` function.

As an example, an implementation of `emulateVioAccess()` with support for an instruction we call `mov4 [A], B` is provided below. For this purpose,

[16] Instructions that access multiple memory locations might reference both VIO memory and non-VIO memory.

said instruction converts the value of work register `A` into an address. Next, this address is used to load `4` consecutive address units from memory, before storing the loaded value in work register `B`.

```
void emulateVioAccess(Instruction instr,
                      VioMappingDescriptor vmap,
                      RegisterState rs) {
    switch (instr.identifier) {
    ...
    case MOV4_DEREF_A_INTO_B:
        rs.B = Word<4>(vioRead(vmap, Address(rs.A)));
        rs.PC += instr.length; /* Adjust Program Counter */
        break;
    ...
    default:
        /* Can't get here because there is no
         * instruction that doesn't get emulated */
        UNREACHABLE;
    }
}
```

Here, the `vioRead()` function is used to differentiate between accesses to memory part of–, and not part of `vmap`. Because it acts as a wrapper for the memory access primitive `vmap.vtable.read`, it is classified as a VIO wrapper function. VIO wrapper functions are further discussed below.

Note that even though, as specified in Section 2.1, and repeated by the `default` case in the above code block, *all* instructions must be known and emulated by `emulateVioAccess()`, in practice this is likely impossible to achieve for a multitude of reasons. These reasons include the introduction of new instructions in computer architecture revisions released after a specific VIO implementation was written, or instructions left undocumented by computer architecture vendors.

As such, any practical implementation of VIO can only ever give a *best-effort* guaranty in terms of instructions that are supported for accessing VIO memory. Generally, this is sufficient because any reasonable use-case of VIO implies some amount of knowledge as to which instructions might reasonably be encountered during execution. There should be no situation where this set isn't a subset of those instructions that have been publicly disclosed for a

specific computer architecture, and as such can be expected to be supported by a sufficiently recent VIO implementation.

To work around the issue of unknown instructions, an implementation might branch to the kernel's *illegal instruction handler* when the default-case would be reached. The specifies of what an illegal instruction handler is, or what to do when no such handler is defined, are not further discussed here.

Memory accesses for at least those address ranges which **may** overlap with `vmap.range` must be passed to a designated *VIO wrapper function* (such as `vioRead()`). A memory access which **never** accesses VIO memory can be performed as usual.

Every VIO wrapper function serves the purpose of selecting between invocation of function pointers from the VIO operator table associated with the fault mapping descriptor (`vmap.vtable`), or performing a normal memory access. They do so based on the passed address wa to which an access should be emulated (in the above example's call to `vioRead()`, $wa =$ `Address(rs.A)`). Because we assume that unaligned memory accesses are impossible, a memory access partially overlapping with the start or end of `vmap.range` also becomes impossible, and it suffices to check the association with the base address of the accessed address range ($R.$`minaddr` $= wa$), rather than the association of every address unit inside of the range $R = [wa, wa + N - 1]$).

1. $wa \in$ `vmap.range`: The accessed address is described by the fault mapping descriptor. In this case, the access is handled solely by means of dispatching through function pointers from `vmap.vtable`.

2. $wa \notin$ `vmap.range`: The accessed address doesn't overlap with the fault mapping descriptor. In this case, the memory access can be facilitated entirely by means of performing a normal memory access, which in turn is allowed to recurse when wa overlaps with a VIO mapping descriptor other than `vmap`.

   In turn, because wa is known not to overlap with `vmap.range` in this case, the PFA_{inner} of a recursive page fault will be relative to wa (for simplicity, our assumption of only aligned memory accesses being allowed allows us to assume that $\text{PFA}_{inner} = wa$), and thus won't overlap with `vmap.range`, either. Because we assume that any memory access always references a singular page, the case where that page belongs to another VIO mapping

descriptor will see the recursive invocation of the PF-handler make another call to `emulateVioAccess()` using `getVioMappingAt`(PFA_{inner}).

`emulateVioAccess()` then emulates the instruction used by the original fault's VIO wrapper function to do what it believed to be a normal memory access. However, during this second call of the VIO wrapper function, case *1* applies (because $wa \in$ `getVioMappingAt`(PFA_{inner})`.range`), thereby allowing the memory access to be handled by functions from `getVioMappingAt(wa).vtable` and preventing any further recursion.

Thus, infinite recursion could only become possible when memory accesses made by functions pointed-to by the VIO operator table `vmap.vtable` erroneously perform a memory access to their own VIO mapping, or some other that does similarly and ends up forming a circular dependency loop. Such use of VIO is not intended and can easily be prevented by never (intentionally) accessing VIO memory from inside functions pointed-to by VIO operator tables. Should this restriction not be adhered to, the error lies in the program(s) that provided the functions that ended up forming the loop.

As an example of VIO wrapper functions, the aforementioned `vioRead()` function is implemented as follows. Note that without our assumption of no unaligned memory accesses, VIO wrapper functions would become significantly more complex due to a third case where the accessed address range only partially overlaps with `vmap.range`.

```
Word<N> vioRead(VioMapping vmap, Address wa) {
    if (wa >= vmap.range.minaddr && wa <= vmap.range.maxaddr) {
        /* Case #1 */
        return vmap.vtable.read(wa - vmap.range.minaddr);
    } else {
        /* Case #2 */
        return NORMAL_READ(wa);
    }
}
```

As shown in the above pseudo-code, the `Address` argument taken by function pointers from the VIO Operator Table represent the relative offset of the lowest address unit that is being accessed, based off of the beginning of the associated VIO mapping descriptor. As such, it also represents the

effective address within virtualized memory, thus making it possible for VIO mapping descriptors to be placed anywhere in memory without the implementation of memory access primitives such as `vmap.vtable.read` needing to concern themselves with the associated mapping descriptor's base address (`vmap.range.minaddr`).

3.4 Data Structures

The following figure displays how VIO data structures relate to each other and to existing kernel components. The *VIO-specific* components displayed by the figure are explained throughout the preceding sub-sections. Terminology for existing components (not part of the *VIO-specific* block) follows conventional *NIX-style names that are not further elaborated here.

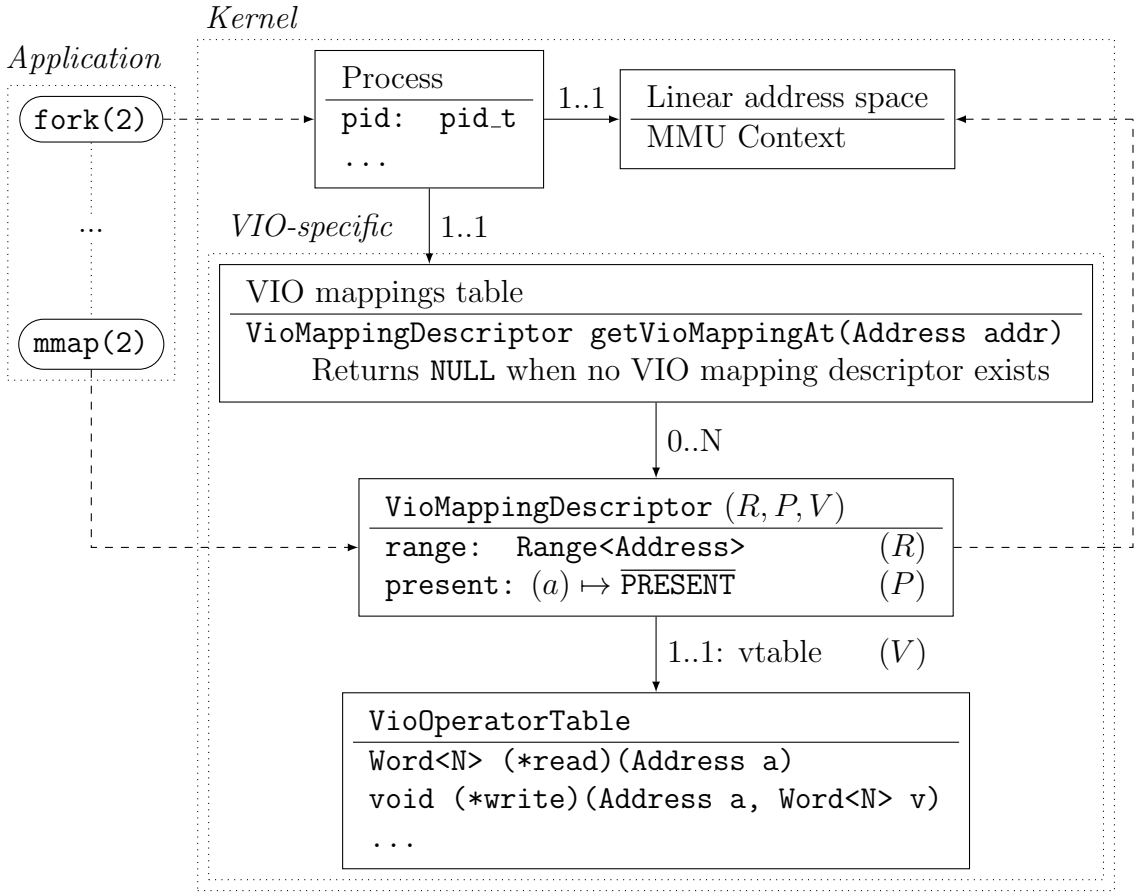

Figure 2: VIO Data Structures

3.5 Usage

As stated in Section 3, a kernel with support for VIO must provide some means for programs to construct VIO mapping descriptors, and in turn VIO memory. The following is pseudo code for a program that does this:

```
Word<4> read4(Address addr) {
    printf("In read4: 0x%x\n", addr);
    return 42;
}
VioOperatorTable operators = {
    .read = { [N: 4] = &read4 }
};

void main() {
    int fd = createVioMappingObject(&operators);
    Address mem = mmap(NULL, 0x10000, PROT_READ, MAP_PRIVATE, fd, 0);
    printf("Before read\n");

    /* This memory access invokes ‘read4(0x350)’ above. /
    Word<4> val = *(Word<4> *)(mem + 0x350);

    printf("After read: %d\n", val);
}
```

Running the above program's `main()` function produces the following output:

```
Before read
In read4: 0x350
After read: 42
```

The above program defines a function `read4()` to facilitate a read from VIO memory. This operator is linked for `N=4` read-operations in the definition of `operators`, which is then passed to `createVioMappingObject()` to create a kernel object that is used with `mmap(2)` in order to turn it into a VIO mapping descriptor. Afterwards, VIO memory is read at offset `0x350`, causing the `read4()` function to be invoked with that same offset. Its return value `42` is then printed.

Explanation of previously unmentioned functions used in the above code:

- `int createVioMappingObject(VioOperatorTable *ops);`
  This function represents a new system call introduced for VIO that creates a so-called *file descriptor* which can be passed to `mmap(2)` in order to map it into memory. In the reference implementation (see Section 4), this function takes additional arguments and is called `vio_create(3)`.

- `void *mmap(void *hint, size_t size, int prot,`
  `int flags, int fd, off_t offset);`
  This function is a well-established *NIX-style system call that is used to map the object referenced by `fd` into memory, by asking the kernel to construct a new memory mapping descriptor. The function's return value is that memory mapping descriptor's R.`minaddr`. `mmap(2)` includes many features not relevant to us, but what is relevant is that `size` is the size of the constructed memory mapping descriptor, and (relevant only for its use in Section 3.6) when `flags` contains `MAP_FIXED`, R.`minaddr` = `hint`, with preexisting and overlapping memory mapping descriptors getting truncated or overwritten.

  Note that `void *` is the `C`-name for what we refer to as an `Address`. More details regarding `mmap(2)` can be found in its man-page[17].

3.6 Watchpoints

As stated in Section 1, VIO makes it possible to implement watchpoints that do not impose any performance penalties for memory accesses performed to non-VIO memory. The following pseudo code defines a function `createWatchpoint()` that can be used to create VIO-based watchpoints, as well as demonstrates how that function may be used:

```
/* Overlay pages of the given address range [addr, addr+size) with
 * a read-/write-through memory mapping descriptor, but any access
 * to the given range itself first calls 'handler' */
void createWatchpoint(Address addr, size_t size, void (*handler)()) {
    Address aligned_addr   = FLOOR_ALIGN(addr, PAGESIZE);
    Address aligned_end    = CEIL_ALIGN(addr + size, PAGESIZE);
    size_t  aligned_size   = aligned_end - aligned_addr;
```

[17] `https://www.man7.org/linux/man-pages/man2/mmap.2.html`

```
    Address watch_mindelta = (addr           ) - aligned_addr;
    Address watch_maxdelta = (addr + size - 1) - aligned_addr;
    Address copy = duplicateMemoryMappings(aligned_addr, aligned_size);
    Word<N> readWrapper(Address offset) {
        if ((offset + N) > watch_mindelta && offset <= watch_maxdelta)
            handler(); /* Read from monitored area */
        return *(Word<N> *)(copy + offset);
    }
    void writeWrapper(Address offset, Word<N> value) {
        if ((offset + N) > watch_mindelta && offset <= watch_maxdelta)
            handler(); /* Write to monitored area */
        *(Word<N> *)(copy + offset) = value;
    }
    VioOperatorTable operators = {
        .read  = { [N: *] = &readWrapper },
        .write = { [N: *] = &writeWrapper }
    };
    int fd = createVioMappingObject(&operators);
    mmap(aligned_addr, aligned_size, PROT_READ | PROT_WRITE,
         MAP_FIXED | MAP_PRIVATE, fd, 0);
}

/* Custom watchpoint handler */
void myWatchpointHandler() {
    printf("WATCHPOINT!\n");
}

void main() {
    int array[1024];
    createWatchpoint(&array[94], sizeof(int), &myWatchpointHandler);
    for (int i = 0; i < 1024; ++i) {
        printf("Writing to array[%d]\n", i);
        array[i] = i;
    }
}
```

Running the above program's `main()` function produces the following output:

```
Writing to array[0]
```

```
Writing to array[1]
...
Writing to array[93]
Writing to array[94]
WATCHPOINT!
Writing to array[95]
Writing to array[96]
...
Writing to array[1022]
Writing to array[1023]
```

In the above code, `createWatchpoint()` calculates the page-aligned boundaries of the address range it is given by `main()`. It then duplicates whatever memory mapping descriptors are already present within that range by use of `duplicateMemoryMappings()`, and stores the base-address of this duplicate in a variable called `copy`. Then, that same address range is replaced with a VIO mapping descriptor whose operator table is defined to forward all accesses to `copy`, only that any access made which overlaps with the originally given address range calls `handler()` prior to being forwarded.

Explanation of previously unmentioned functions used in the above code:

- `FLOOR_ALIGN` and `CEIL_ALIGN` are defined as:

$$\texttt{FLOOR\_ALIGN}(x, a) = \lfloor x/a \rfloor \cdot a$$
$$\texttt{CEIL\_ALIGN}(x, a) = \lceil x/a \rceil \cdot a$$

- `void *duplicateMemoryMappings(void *addr, size_t size);`
  This function asks the kernel to duplicate all (sub-ranges of) memory mapping descriptors that overlap with the given address range $R = [\texttt{addr}, \texttt{addr}+\texttt{size}-1]$, which is assumed to be page-aligned. It's return value is the base address of a new set of memory mapping descriptors that reference the same physical memory or VIO operator tables, as are already referenced by memory in R. Following its return, any access to an address $\texttt{addr} + a \mid a \in [0, \texttt{size})$ behaves the same as one to address $\texttt{return} + a$. It is implied that $\texttt{addr} \neq \texttt{return}$.

4 Reference Implementation

Beyond the thesis itself, the author has created a functional reference implementation of VIO. It is part of their kernel named *KOS*[18], which can be found under `https://github.com/GrieferAtWork/KOSmk4`[19]. Files and lines relevant to VIO are listed in the following:

`PF_HANDLER()`	`src/kernel/core/arch/i386/fault/handle_pagefault.c:797`
`emulateVioAccess()`	`src/libviocore/arch/i386/viocore.c:103` `include/libemu86/emulate.c.inl:3229`
VIO operator table	`include/libvio/vio.h:112`
VIO creation (user)	`src/libvio/vio.c:484`
VIO creation (kernel)	`src/kernel/core/memory/uvio.c:988`
Usage example	`src/apps/playground/main.c:665`

5 Caveats

Implementing and using VIO comes with some caveats, which we list and discuss in the following:

- While memory accesses to non-VIO memory are completely unaffected, any access to VIO memory is many times slower than expected. This is because any such access requires the faulting instruction to be decoded and emulated in software (see Section 3.3).
- VIO breaks the natural expectation made by programs, of any memory access completing in finite time. For example, because almost no restrictions are put on what the user-provided functions from a VIO operator tables are allowed to do, actions taken by them might include sleeping for an indeterminate amount of time.
- Adding support for VIO to widely-used kernels such as NT (Windows) or Linux would be highly difficult. This is because of the aforementioned fact that a VIO access might sleep for an indeterminate amount of time, or perform arbitrary other operations during its execution, none of which may result in a system crash or deadlock. As a result,

[18] A home-made, monolithic, but still modular kernel for *i386* and *x86_64*.
[19] Files/lines are relative to git commit `6ec1573d091da60db0ea7bb4607cf1733c93abc5`.

this means that whenever a kernel with VIO support makes an access to potential VIO memory (for which any region of memory it is given by a programs qualifies), it can **never** hold any kind of lock while doing so. Otherwise, should a function from a VIO operator table simply wait indefinitely, the entire system might deadlock due to lock-starvation.

A similar problem arises when a program is terminated while emulating a VIO access. Doing so must be made possible to ensure that programs can always be force-closed. As a result, any access made by the kernel to memory with the potential of being VIO memory, must provide some means of being aborted for the purpose of handling a termination request[20].

Kernels such as those mentioned above already contain an existing mechanism called *swap memory* that imposes restrictions similar to those required by VIO. Swap memory is used to simulate additional memory by reusing memory that's already in use. This is done by writing the contents of memory to-be reused to a mass-storage medium (such as a hard-disk), and using the PF-handler to read back the contents when they are needed again. As such, accessing memory that has been turned into swap memory implies the potential of having to wait for a hard-disk to start spinning, but this is still less restrictive than what is required for VIO, since reading from a hard-disk is guarantied to complete in finite time (assuming an intact hard-disk), and thereby isn't something that necessarily needs to be interruptible.

6 Going Further

Access to memory is one of the fundamental features shared by every computer in existence. VIO expands upon this idea by making it possible to break pre-existing semantics and notions of how memory *should* work, such that a simple read from memory might result in an entirely distinct sub-program being executed to compute and eventually return the result of the preced-

[20]The author's kernel (see Section 4) is written in *C++* and uses exceptions to accomplish this, but this would not work for the Linux kernel, which is written in *C* and isn't designed with exceptions in mind. Furthermore, source code for the Linux kernel assumes that accessing memory can only ever fail due to "faulty" memory (`EFAULT`), but not due to a need to be interrupted (`EINTR`).

ing read. The same goes for writes and other memory access primitives; see Section 3.2.

As such, aside from the example use case of efficiently simulating watchpoints, there are many more use cases where VIO can be applied:

- A very simple example is a memory address that can be read to yield the current time in an arbitrary format. The same can be done to construct an address to yield a unique random number each time it is read.
- The emulation of so-called *MMIO*, which stands for *Memory-mapped I/O*. This is a mechanism commonly used by hardware peripherals to expose control interfaces to the computer, and in turn to running programs. This is also what led to the name VIO (which stands for *Virtual I/O*).
- VIO can be used to keep a transactional journal of all memory accesses performed by a program over the course of its execution, making it a tool for doing black-box analysis of interactions between functions and memory, or even to roll-back memory modifications made past a certain point in time.
- So-called *NVRAM*, standing for *non-volatile RAM*, can be implemented in software using VIO. NVRAM acts like normal memory, but any modification made to its contents is retained even after a power failure.

  Emulating read-operations by reading from an arbitrary persistent storage medium, and doing the equivalent for write-operations, VIO can define a region of memory that does the same without the need of dedicated hardware. Furthermore, said persistent storage medium need not necessarily be local, but could even reside in a remote location reachable over a network.

7 Related Work

While the general idea behind VIO – as presented by this thesis – is new, a somewhat similar technique is already being used in practice.

A mechanism present in kernels such as the Linux kernel allows for unaligned memory access on architectures that do not natively allow this [2]. To

achieve this goal, these kernels define an exception handler[21] that is triggered when an instruction attempts to perform an unaligned memory access.

Inside of this handler, the last-executed instruction is retrieved and decoded, before then being emulated by means of splitting the memory access it attempted into multiple smaller accesses, all of which can now be performed whilst properly aligned. In the case of a read, the results of this are then put back together and stored at the location or locations (normally, a singular work-register) indicated by the last-executed instruction. The inverse is done if the instruction attempted a write.

As detailed by this thesis, VIO does something similar. We hook the PF-handler and use it to decode and emulate the last-executed instruction, only that in our case we are forced to emulate **all** instructions, rather than only those that might be able to perform an unaligned memory access in a manner that triggers the associated exception handler. Furthermore, our emulation does not only perform regular memory accesses, but may also call functions from VIO operator tables in order to dispatch the access.

8 Conclusions

In this thesis we have described the semantics of VIO and how it can be implemented in modern computer systems.

By defining a new class of memory mapping descriptor which we call VIO mapping descriptor, it becomes possible to force execution to branch to the PF-handler following an access to VIO memory. This is because VIO memory is always mapped as non-present as far as the MMU is concerned.

After expanding the PF-handler with the ability to detect VIO accesses, as well as decode and emulate the faulting instruction, we are able to virtualize the semantics of what an access to VIO memory does, means, returns, and/or should accomplish.

Because any region of memory[22] can be replaced by VIO memory, this also means that any memory access can now become– or be replaced by a sub-program, without any modifications needing to be made to the program that performs the access.

We also presented a couple of use cases for VIO functionality, such as pseudo-code for implementing VIO-based watchpoints, as well as others like

[21] The PF-handler used by VIO can also be classified as an exception handler.

[22] Memory used by the VIO implementation itself must not reside in VIO memory.

software-based NVRAM.

More uses for VIO likely exist, and we encourage further research to look into these.

References

[1] QW (pseudonym) – Sep 15, 2010: *Re: Kernel requests via page faults*[23] `http://forum.osdev.org/viewtopic.php?f=15&t=22521` [21.01.2022] – Discussion on using page faults (and the PF-handler) to perform system calls.

[2] LINUS TORVALDS – 1995: *Linux Kernel* – Emulation of unaligned memory access on ARM (`arch/arm/mm/alignment.c`) [21.01.2022] – Hooking a CPU exception vector to emulate the unaligned memory access of the faulting instruction.

[3] INTEL® – Dec 2021: *64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide* – Chapter **4**: Paging `https://cdrdv2.intel.com/v1/dl/getContent/671447` [21.01.2022] – Detailed instructions on how to program the MMU of an `x86` computer.

[23] For archival purposes, a snapshot of the page can be found on web.archive.org.

9 Index