Karlsruher Institut für Technologie

Estimating I/O Memory Bandwidth

Diplomarbeit
von

cand. inform. Jos Ewert

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Marius Hillenbrand

Bearbeitungszeit: 30. Juni 2016 – 28. Dezember 2016

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 28. Dezember 2016

Abstract

On multicore systems, the memory bandwidth is an important shared resource. However there is no direct way to measure the memory bandwidth. With this lack of data, the scheduler cannot make decisions that take this resource into account. This leads to an unfair distribution of the memory bandwidth and consequently suboptimal performance.

There is research into measuring application's direct memory bandwidth usage, however these systems cannot measure the memory bandwidth a process uses indirectly via the IO devices. Hence, the OS lacks a way to measure IO memory bandwidth to gain a complete view of a process's memory bandwidth consumption.

In this thesis I demonstrate how with extended accounting in the operating system and data from the hardware's performance counters the OS can estimate a process's IO memory bandwidth. I show that data from current state-of-the-art performance counters can provide precise data about the total IO memory bandwidth. Furthermore, I illustrate why the current accounting in the Linux kernel is insufficient to calculate the IO memory bandwidth for each process's devices.

To extend the OS to provide IO memory bandwidth accounting, I first identify the two groups of devices that the system should track, network and storage devices. I explain how they differ and how to add accounting to both groups. I present a system that combines this accounting with data from performance counters to estimate the IO memory bandwidth for each process.

I validate the system's accuracy and determine that in most cases there is no significant performance penalty. In the worst case it has a 16% performance penalty.

Contents

Chapter 1

Introduction

One of the core functionalities a operating system provides is to manage access to shared resources. The OS should fairly distribute these resources among the competing processes. To avoid one process from negatively influencing another, the OS needs data about what resources a process consumes. In current operating systems these resources generally are CPU time and memory usage. However, there are more resources that the processes share. The OS can generally manage and limit a processes IO capacity and bandwidth. All of these metrics are relatively easy to quantize and limit. For example CPU time can be limited by not scheduling a process, the memory usage by not allowing the process to allocate more memory.

Today's operating systems need to manage processors have many cores that allow multiple processes to run in parallel. While the cores operate independent from each other, they all share the same memory bus and thus its bandwidth. This means that there is a constant contention over the memory bus. Even though processes can run otherwise independently, they influence each other via the shared memory bus.

Consequently, the scheduler, responsible to distribute the CPU time, needs to be aware of the memory bandwidth. This is due to different processes negatively influencing their CPU time indirectly via the memory bandwidth, the processes can stall waiting for data and not effectively use their CPU time. To assure that each process has a fair share of the CPU time, the scheduler needs to also fairly distribute the memory bandwidth the system provides.

There are generally two ways a process can access the memory. Directly by reading its own data and indirectly by data that IO devices copy to the memory for a process. Most of the research so far concerns itself with the first case, memory that is directly used by the process. There is only very little research going into IO device's memory bandwidth needs.

In this thesis I show that IO devices have a significant impact on a process's

available memory bandwidth. Hence, to allow the scheduler to make more accurate decisions it needs data about the process's IO memory bandwidth. To provide the scheduler with data about this bandwidth, I propose a system that combines data from the operating system and the processor's performance counters.

This thesis is organized as follows: In Chapter 2 I give an overview of current research about memory bandwidth and IO bandwidth. These works provide a deeper look into issues that arise when the memory bandwidth is not properly shared between processes. I also give an introductions of the systems that I need to modify to measure the bandwidth. I explore and evaluate what data the hardware and software can provide in Chapter 3. In this chapter, I illustrate why a system needs data from both the hardware and the software. Furthermore I explore how the current accounting in Linux is too limited to provide per process IO memory accounting.

Based on the information from Chapter 3, I design and develop a prototype in Chapter 4 and Chapter 5. The prototype combines data from the processor's performance counters and the OS' accounting to provide per process IO memory accounting. In Chapter 6 I show that the system can provide accurate data and has in most cases no performance penalty, even in the worst case where there is a moderate performance penalty. Finally, in Chapter 7, I summarize my findings and present what I believe the next steps are to properly manage IO memory bandwidth.

Chapter 2

Background And Related Work

Memory accesses from processes happen outside of the OS control. Hence the OS cannot directly measure or distribute the memory bandwidth. In Section 2.1, I present research that demonstrates that this lack of accounting can lead to degraded performance. Moreover, research shows that scheduling policies that take the memory bandwidth into account can improve a system's performance. However most of the systems so far only take into account memory bandwidth that a process uses directly. There is research to extend the operating system's IO accounting capabilities which I need in this thesis.

In Section 2.2 I quickly describe the systems that can be used to estimate IO bandwidth and IO memory bandwidth. Finally, in Section 2.3 I introduce the network and storage stack that I use to get data about the device's activity.

2.1 Memory Bandwidth Management

Managing and abstracting hardware resources is one of the OS' main functions. The goal generally is to fairly provide and multiplex access to the resources provided by the hardware. Usually, the OS manages CPU time and memory as well as access to IO devices. The OS is able to provide fine grained policies over these resources as access requires the application to give up control to the OS. Memory access however usually happens on the hardware level when the OS is no longer in control over the flow of execution. The sheer number of memory accesses necessitates the now almost ubiquitous memory hierarchy with several levels of small caches and main memory that is managed by the hardware. Executing every memory access in software would be too slow.

To quantify the impact different processes running of different cores have on each other, Subramanian et al. propose a new model to estimate slowdown processes experience on a shared system [17]. They show that scheduling and par-

titioning the available memory and caches in a way that takes the slowdown into account can improve a system's performance. Another model to quantize memory bandwidth is proposed by Eklov et al. [6]. They focus specifically on memory bandwidth contention and also show that quantizing and managing memory bandwidth improves performance.

Moscibroda et al. show that without a way to properly manage the memory bandwidth on multi-core systems, processes can create a denial of service by denying other processes memory bandwidth [13]. These papers show the importance that proper accounting and scheduling for memory bandwidth have.

There are several papers that discuss scheduling processes while taking into account a process's memory bandwidth. Antonopoulos et al. describe several scheduling policies that take into account a process's memory bandwidth needs [3, 4]. They rely on data from performance counters they read during a process's execution to predict the process's future memory bandwidth needs. Xu et al. use a similar system and propose a more precise scheduling policy by also taking into account a process's exact behavior [18]. They show that with a bandwidth aware scheduler, the system's performance can increase significantly.

Using the performance counters to estimate a processes performance is rather coarse grained. There are more precise estimations that use CPU simulators, however they are slow. However modern processor's performance counters can provide additional data about events. With this extra data memory bandwidth measurement's can be precise and have a significantly lower performance penalty [11].

All the previous works show that scheduling according to a process's memory bandwidth needs is an important topic to improve a system's performance. However they all can only measure the bandwidth a process consumes directly. Memory bandwidth that a process uses indirectly via DMA transfers from devices cannot yet be accounted for. The first step to account for this is to determine a process's IO bandwidth needs.

There is only little research to accurately measure a process's usage of IO devices. Ranadive et al. evaluate the impact that Infiniband bandwidth contention has on processes that share a Infiniband connection [14–16]. Infiniband has drivers that are in userspace and bypass the kernel. They show that by observing the command buffers in userspace they can estimate the process's bandwidth needs. They develop a system that is bandwidth aware that can greatly improve a system's performance. They show a possible way to monitor a process's IO bandwidth needs and how to manage a shared resource. However, their work is only indirectly concerned with memory bandwidth by the way Infiniband transfers data directly into another process's address space.

The Linux IO stack has limitations about what data is available in what stage of the stack. This can lead to unfair IO scheduling. To remedy this lack of data, Yang et al. propose "Split-Level I/O Scheduling" [19]. They move data from

the upper layers of the stack down to the schedulers so they have more data to base scheduling decisions on. This paper shows how the Linux IO stack can be extended so that I can provide more data about process's IO needs.

While there is research to measure processes' memory bandwidth and schedule them while taking into account the bandwidth, none of the previous works consider IO memory bandwidth. Research into IO bandwidth so far was limited to special devices and are more concerned about sharing the device fairly and not the memory bandwidth. I combine both IO memory bandwidth estimation and the scheduler. I show that the systems to estimate IO bandwidth can provide valuable data to the process scheduler. By combining the data that these systems can provide, and data from the hardware I can estimate a process's IO memory bandwidth needs and add this data to the scheduler. I provide more detail about monitoring IO devices in the next section as it is an important part of this thesis.

2.2 IO Device Memory Access Monitoring

As my proposed system relies on IO bandwidth measurements I need a precise way to determine a device's IO bandwidth usage. In this section I describe a few methods to get information about I/O device's memory bandwidth usage. I first describe how to get information from drivers inside of the operating system and then those that are outside of the OS. As the software does not know what requests are serviced in the caches or main memory I need a way to estimate the memory bandwidth. I quickly go over how conceptually an Input–output memory management unit (IOMMU) can be used to estimate the bandwidth. Finally I show how performance counters can also provide data.

The classic way to control the interaction of applications with the hardware is to use a trap from user space or an interrupt from the hardware side to let the OS take control and copy the data from the device's buffers. Using this model it is relatively easy to add accounting during each of these interrupts and gauge the memory bandwidth. With the use of DMA, the OS only knows that an access to memory will happen in the future, but no longer has direct knowledge over how much data is transferred. However the drivers need to program the devices and know the amount of data that is transferred. This means that by adding accounting to a driver it can provide accurate data.

A system called memory introspection can be used for drivers that are not under the operating system's control and reside in userspace. This system maps parts of the (user space) command buffers used by the drivers to a another region where they can be periodically inspected [15]. This technique requires precise knowledge of the driver's buffer usage, layout, and algorithms. Memory introspection does not require the OS to take control over the device accesses and it can provide

precises data over what processes and devices communicated.

With these two methods I can measure a process's IO needs, however I do not know if a request is serviced in main memory or in the processor's cache. So I cannot estimate the actual memory bandwidth a process's IO devices need. To measure this I need more data that the hardware can provide.

Conceptually IOMMUs can be used to infer the memory bandwidth usage. The IOMMU handles address translations from IO devices, so for example a device can work on virtual addresses without having to know physical addresses. The IOMMU is situated between the IO devices and the main memory to handle every request to main memory from devices. The page tables used by the IOMMU can have "accessed" and "dirty" bits [2, 5, 7] that can be used to infer what memory page has been accessed. However devices available at the time of writing do not provide the necessary features. I explain this issue in Section 3.5.

Most devices and processors provide a multitude of performance counters. These can provide a detailed view of what operations were performed by the hardware or provide accounting for example over the memory usage. Counters on the CPU have been used in previous works to create scheduling policies that can take memory bandwidth into account [4, 11]. However these counters are used to estimate the process's memory bandwidth consumption and not the IO device's. Each of the devices provide different counters at different stages in their hardware, which means that mechanisms to collect data have to be tailored to each device. In Section 3.2.2 and 3.3 I show what exactly these performance counters can provide.

2.3 Network And Storage Stack

I rely on data about process's IO usage from the operating system. In this section I introduce the two IO stacks that I use in this thesis, the network and storage device stack.

The network stack in Linux provides applications with the socket interface. Applications can use this interface to send data over the network. The network stack roughly follows the layers of the OSI network model. The top layers in the applications interact with the sockets. The sockets pass the data down to the transport layer, through the network and link layers, until the driver writes the data to the device.

The storage stack in the Linux kernel is generally used by applications via files. The application reads or writes to a file. The kernel's file objects interacts with the filesystem. To improve the performance, the storage stack also utilizes a page cache to temporarily store data from the devices. Generally data is only written to the device when it is necessary, for example if the cache is full. To avoid

data loss Linux uses a process called `update` that periodically forces the cache to write data back to the device. When the data is written back to the device, it travels further down the stack until it reaches the device driver.

While both the network and storage stack have schedulers that manage when data is written to a device, it is important to distinguish these schedulers from the process scheduler. The IO schedulers are responsible from ordering and timing requests to devices. They generally have no concept about processes. In this thesis I concern myself about the process scheduler and how it can manage processes to share bandwidth.

Chapter 3

Analysis

In this thesis I explore how much IO memory bandwidth the different devices a process utilizes need. Therefore, I establish that IO has a significant impact the memory bandwidth in Section 3.1. Since there is no way to directly measure each process's IO memory bandwidth I propose an indirect approach by aggregating data from different sources to measure the consumed IO memory bandwidth.

All the experiments in this thesis are conducted on a Intel Xeon Haswell EP processor. Although the methods I present are processor family specific, they can be ported to other platforms that provide similar performance counters.

The system I propose needs data about the memory bandwidth, the different devices' IO bandwidth, and a process's usage of these devices. In this chapter, I describe what measurements hardware provides, potential ways to combine hardware and software, and finally what data the software already provides without further modification.

The hardware offers different performance counters to track all kinds of metrics. In Section 3.2 I give an overview of what data related to memory and IO bandwidth they can provide. In Section 3.2.1 and 3.2.2, I explore the potential and limits the performance counters on the Xeon processor have. I determine that the cache controller can provide the most valuable data and that the counters provide accurate data. Section 3.4 demonstrates that it is not possible to reliably identify individual devices from the performance counter data alone.

Since the performance counters alone do not provide sufficient data to determine the IO memory bandwidth, it is necessary to combine information from the software and the hardware. A possible, potentially precise and fast, combination is to estimate bandwidth using memory management units that can abstract the address space for IO devices, so called Input–output memory management units (IOMMU). However state-of-the-art hardware at the time of writing does not implement the necessary features. For completeness, I describe the idea and caveats of this method in Section 3.5.

		Bandwidth GiB/s	Bandwidth GiB/s
IO from programs		0	2.0
Stream:	Copy	20.6	18.1
	Scale	13.8	11.5
	Add	15.1	12.8
	Triad	15.1	12.8

Table 3.1: Memory bandwidth reported by the stream benchmark. The benchmark is run on one core and the IO consuming processes on others, making the memory bandwidth the shared resource. The bandwidth on the left column is measured without IO and on the right with IO from other processes.

Finally, in Section 3.6 I present the systems the Linux kernel has at its disposal for IO accounting and how they are insufficient to provide per process per device accounting. As the already present accounting in Linux is in either the high levels or the low levels of the IO stack, they cannot exchange data to provide per process accounting. To be able to connect the different layers of the IO stack, I explore how cgroups solve this problem. In addition, I illustrate how the systems from Split-level IO scheduling proposed by Yang et al. [19] can be used to connect the low and high layers of the storage stack. Finally, for completions sake, I quickly illustrate memory introspection described by Ranadive et al. [15] that can track drivers outside of the kernel.

3.1 IO Bandwidth Performance Impact

The first question I want to answer in this analysis is what impact IO devices have on the total memory bandwidth. To demonstrate the impact, I use the Stream benchmark [12]. Overall, the benchmark runs four tests.

1. “Copy” copies data from one array to another.
2. “Scale” is a scalar multiplication with a vector that saves the result in a new array.
3. “Add” is a sum of two vectors.
4. “Triad” is the vector sum of an array and an array multiplied by a number.

To provide a reference I run the Stream benchmark on an idle system first. In the second test I put the system under load with very simple benchmarks that write the same buffer to disk and network until they are killed. I run only the single threaded version of Stream in both tests and let the other benchmarks run on the

other cores. This way the different benchmarks only share the memory bus and the last level cache. The IO Bandwidth that is reported by the benchmarks only counts the data that is sent to the socket or file. It does not take into account any overhead from the file system or network headers.

Table 3.1 shows that the difference between both runs of Stream follows roughly the IO bandwidth created by the other benchmarks. In my test, the IO devices need at least 12% of the memory bandwidth that is available to stream. In the worst case IO devices need over 16% of the memory bandwidth. This shows that to accurately manage the memory bandwidth as a shared resource, the OS has to also know the IO memory bandwidth from each process.

3.2 Performance Counters

Performance counters keep track of many events on the CPU. In this section I give a small overview of what data related to IO memory bandwidth performance counters can provide. I show how a request from a IO device passes through the processor and where counters can gather data. Finally, I show the limitations the performance counters have.

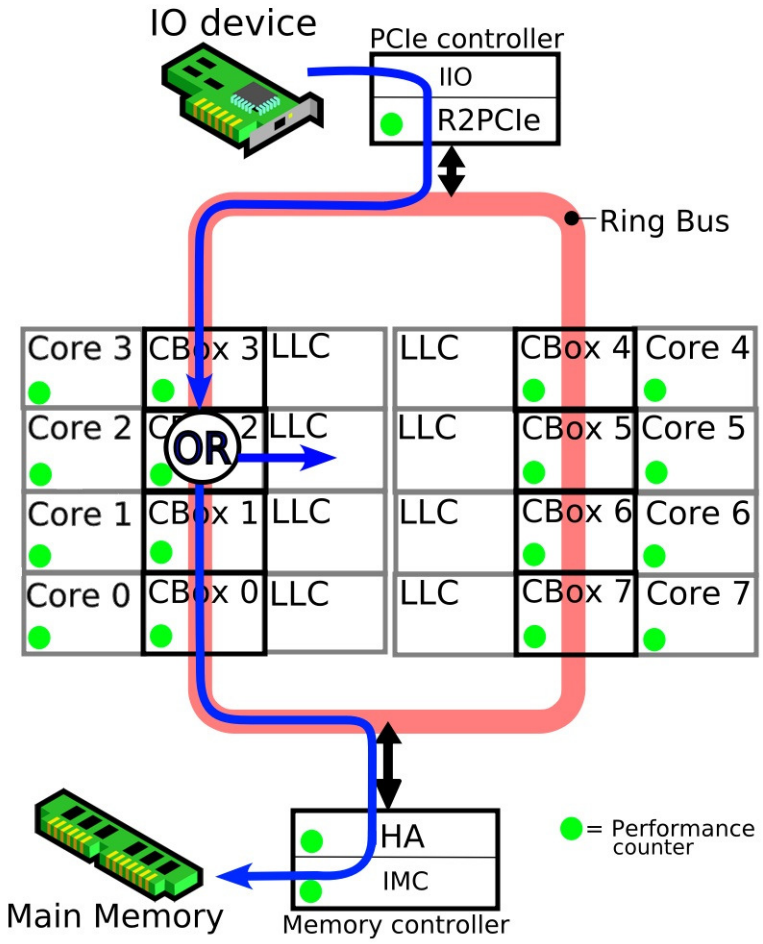
While there are many similarities between the counters on current processors, generally these counters and the events they can track are model specific. For example on Intel processors only the seven "architectural performance events" are trackable on any processor, any other event is processor family specific.

With such a multitude in possible counters it is necessary to write specific drivers for each processor family. In the following sections, I present the performance counters that are on the my test system's processor. While the description is specific to this processor, other processors provide similar counters and this can help identifying relevant units and metrics to estimate bandwidth.

3.2.1 Relevant Functional Units

Intel Xeon Haswell EP processors have a multitude of functional units. In this section I identify the units that can potentially provide data to estimate IO memory bandwidth, i.e. the units that are on the path from a device to main memory and vice versa. A complete description of every performance counter can be found in the Software developers guide and the Uncore performance monitoring guide [8, 9].

Figure 3.1 shows the path a read from an IO device takes to get the necessary data. All the functional units on this path can potentially provide performance data to measure IO or memory bandwidth.



IIO	Integrated IO controller: Connects directly to PCIe devices and multiplexes/demultiplexes requests
R2PCIe	Ring to PCIe: Connects the ring bus with the IIO
CBox	Cache controller: Determines if a request can be handled in LLC and ensure coherency between sockets
LLC	A slice of the Last Level Cache, shared between all cores
HA	Home Agent: Responsible for the memory access protocol and orders memory access requests
IMC	Integrated Memory controller: Directly connects to the DRAM modules and handles the low level access to RAM

Figure 3.1: Memory read from an IO device on the Intel Xeon Haswell EP processor. Either the read is satisfied in LLC or in main memory. The green dot indicates if a functional unit has performance counters. Based on figure in [8, Fig. 1-3]

Going from devices to the memory, a request first passes through the PCIe controller. On the Xeon processor, the PCIe controller is split into two parts. The request passes through the Integrated IO controller (IIO) and Ring to PCIe Interface (R2PCIe). The IIO demultiplexes requests and passes them on to the R2PCIe. R2PCIe moves the request onto the ring bus. Using the ring bus, the request is served to the cache controller called "Caching Agent" (CBox) on the Xeon processor. On the Xeon processor each Cbox is responsible for a slice of last level cache (LLC) and a certain memory address range.

The cache controller determines if the request can be satisfied in cache or has to be moved to main memory. If the request cannot be satisfied using the caches, it is send via the ring bus to the memory controller. The memory controller on the Xeon processor has two parts; the Home Agent (HA) and Integrated Memory Controller (IMC). The home agent handles the protocol side of memory requests, such as keeping coherency on multi socket systems and maintaining the order of memory requests. It is in front of the IMC which finally does the actual access to DRAM.

On the Xeon processor, the relevant functional units thus are the IIO, R2PCIe, CBox, HA, and IMC. In Section 3.3, I evaluate the different events the functional units can track and how they can be used to track the memory bandwidth. Furthermore, I show the limits on what can be tracked in the next section.

3.2.2 Performance Counter Programming

In this section I show how the performance registers are used and the limitations the counters have.

On Intel processors, generally functional units with performance counters have four counter registers to track four different events. Each counter register is paired with a counter config register to select the event. Some functional units have a filter register to further filter the events. Additionally, each unit has a global register to control all performance counters, for example to start or reset the counters. Overflows are reported in a status register.

Counter registers provide the raw number of events that have been counted. This register is 64 bit long, however generally it overflows at 48 bits. On an overflow the "Box status" register sets the appropriate bit in the status register. Optionally the counter can send an interrupt. This is useful if a program wants to be notified after a certain amount of events have been counted. By initializing the counter to $2^{48} - \#events - 1$ the register will overflow after the set number of events and can then create an interrupt. This can for example be used to be notified after a certain number of CPU cycles.

The counter config register sets what event should be counted. These events can then be further refined with sub events. For example one event the cache

controller can count is every cache access, and filter to a subevent that only counts cache misses.

Depending on the functional unit and event, there is an additional filter that can be applied on the event. There is only one filter register and as such the filter is applied to every event that can use this filter. An example for a filter is the cache controller's Thread-ID filter that limit events to a certain Core-ID and Thread.

In summary, the performance counters are limited in the amount of different events that can be tracked. This is limited even more when I use an additional filter as it applies to every event that can use this event. Hence it is important to carefully select what events I need to track to estimate the memory bandwidth. Therefore, in the next section I describe the events each functional unit can provide and describe their uses to estimate IO memory bandwidth.

3.3 Selecting Performance Counters

In this section, I describe the events each of the functional outlined in Section 3.2.1 has. I limit this description to only events that can provide data about memory or IO bandwidth. Ideally, the performance counters can tell me how much data each device needs and how much data from IO devices needs to be processed in main memory. This makes it easy to calculate each devices memory bandwidth. However, as I show in Section 3.3.1, on the Xeon processor there is no way to directly measure each devices bandwidth. However, the process can estimate the total bandwidth. In the remaining subsections I explore which events can be used to estimate the total IO bandwidth and memory bandwidth.

3.3.1 IIO

The PCIe controller is connected to every IO device and is in an ideal position to provide data about every device's activity. The PCIe controller on the Xeon processor is split into multiple parts and it is the so called "IIO" that directly interfaces with IO devices. This makes it is the the ideal candidate to track requests from each device. However, it does not provide any performance counters. This means the closest to the devices we can extract data from is the second part of the PCIe controller, the R2PCIe. At this point however the IIO has already demultiplexed the requests, so the R2PCIe and functional units beyond it can only provide aggregate numbers of accesses.

3.3.2 R2PCIe

The second part of the PCIe controller on Xeon processors, called the R2PCIe, can potentially measure the total IO bandwidth. It sends and receives requests to and from the ring bus. The ring bus is divided into 2 directions, clockwise and counter clockwise. Each cycle the R2PCIe can send one 32 byte message in each direction. By counting the number of requests I can potentially calculate the bandwidth. However the R2PCIe has several limitations that prevent me from counting these events.

The event that counts the access to the ring bus is called `RING_BL_USED`. However this event has several caveats. First, it does not count when packets are being sent, and second it also counts packets that are only passing through the R2PCIe and are not destined to any IO device. As an example, this happens when a request passes from CBox 3 to CBox 4. This event is more focused on measuring how much load is on the ring bus, and less about how the PCIe controller uses the Bus. As such it does not provide data about the PCIe controllers bandwidth needs.

The R2PCIe provides more events that can potentially be used to estimate the bandwidth, but they are undocumented. Ingress events can be counted with `RxR_INSERTS`. However, while the documentation claims that it "tracks one of the three rings that are used by the R2PCIe agent" [8, Ch. 2.10.6], it does not provide information on how to select the ring that should be tracked. `TxR_NACK_CW` counts egress events, however the documentation has no detailed description of this event so it is unclear what exactly this event counts. The name of the event implies the use of the ring bus in the clockwise direction while the title is "Egress CCW NACK" [8, Ch. 2.10.6], which implies a use in the counter-clockwise direction. Consequently as neither event has a detailed description, I cannot use them in my system.

While the PCIe controller is a potential candidate to gather data about IO device's activity, on Haswell EP Xeon processors neither of the two parts can provide good data. The next unit on a requests path to the main memory is the cache controller. As I show in the next section it provides all the data that my system requires.

3.3.3 CBox

Each request from an IO device to the main memory necessarily needs to pass through the cache controller. This includes uncacheable requests. The so called CBox, the Xeon processor's cache controller, has versatile performance counters because the Cache controller is involved in every kind of memory transaction. Because of the detailed performance counters, the cache controller provides all the necessary data to estimate both IO and IO memory bandwidth.

For the purpose of estimating IO and memory bandwidth, the most useful events are TOR_INSERTS, that counts operations on the cache, and LLC_VICTIMS, that tracks evictions from the cache [8, Ch 2.3.6].

TOR_INSERTS counts each entry inserted into the "Table Of Requests" (TOR) that tracks every pending cache controller transaction. Each request can be filtered by its opcode using a special filter register. The opcodes can filter what kind of transaction has been inserted into the queue and more importantly they can also filter by the request's origin. With this data I can measure the total PCIe bandwidth. Additionally, an single socket systems a read cache miss in the last level cache necessarily needs to be satisfied by the main memory. Consequently read misses represent parts of the memory read bandwidth.

LLC_VICTIMS counts the number of cache lines that have been evicted from the cache. The cache coherency protocol on Xeon processors only writes to main memory if a modified cache line is evicted. Thus, this event provides numbers about how much data the cache writes to main memory.

In the rest of this section I detail both events and how they can be used to estimate IO device's memory bandwidth.

TOR_INSERTS

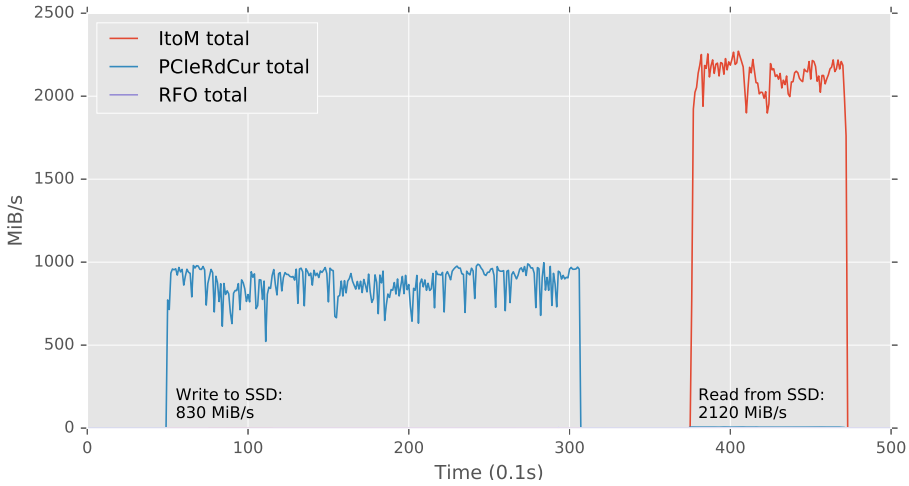
The cache controller uses a queue to process memory requests called the "Table of Requests" (TOR). The number request to the queue are counted by the TOR_INSERTSevent and it can discern what kind of requests have been inserted into its queue. This event has subevents that can filter the type of request with the filter register. Moreover, according to the technical documentation [8, Ch. 2.3.5], with the correct thread-ID all the counted requests are limited to PCIe. With the correct filter, I can accurately estimate the total bandwidth from IO devices.

As each cache controller only has one filter register, every event that uses the filter on the controller tracks the same request type. There are many different request types and a limited amount of controllers. Too much sampling leads to a high overhead as the counters need to be constantly changed. This means that I need to identify the most relevant types of requests to track IO bandwidth.

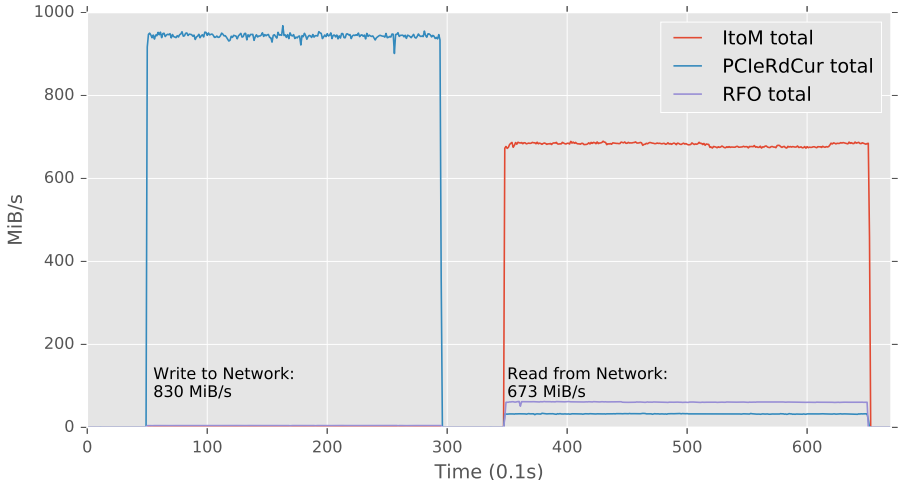
According to the documentation [8, Table 2-20], the relevant request opcodes are :

- RFO: Read for Ownership. This opcode is used when a device or CPU wants to write a partial cache line. The device first needs to read the cache line, modify it and then write the modified line back. It sets every other copy of this cache line to invalid so they are updated on the next read.

- PCIWiL: An uncacheable write from PCIe of a partial cache line. This is only used for memory mapped IO, which is generally used to read or write



(a) Storage benchmark. From 5s to 32s the benchmark writes to disk, from 37s to 47s the benchmark reads from disk. RFO is too small to show on the scale



(b) Network benchmark. From 5s to 30s the benchmark writes to the network, from 35s to 65s the benchmark reads from the network.

Figure 3.2: Events counted for TOR_INSERTS filters. Network and Storage benchmarks are displayed separately. The IO bandwidth reported by the `PCIRdCur`, read from memory, and `ItoM`, write to memory, opcodes follow the reported bandwidth. Partial writes are counted by RFO and are comparatively rare events.

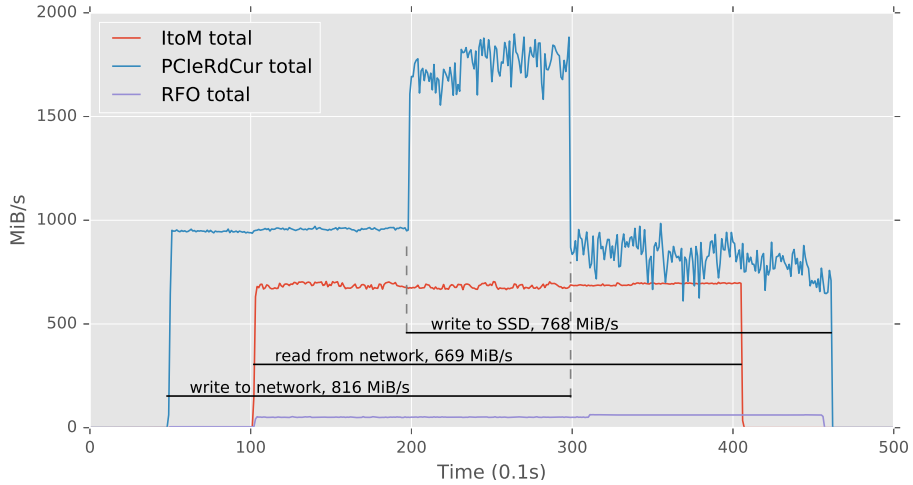


Figure 3.3: Events counted for TOR_INSERTS filters. ItoM tracks writes, PCIeRdCur tracks reads from memory. RFO tracks partial writes. Network and storage benchmarks run simultaneously. The bandwidth reported by the counters follows the total bandwidth.

from a device's registers. Drivers use these registers for example to notify a device about new data.

- `PCIWiLF`: Like `PCIWiL` but for full writes.
- `PCIItoM`: A cacheable write from PCIe. This is used for regular DMA traffic. As this request handles full cache lines I can calculate the total amount data that IO devices write by multiplying it by the size of a cacheline.
- `PCIRdCur`: A read from PCIe. The cache line is not changed. Like `PCIItoM` this operates on full cache lines and thus gives the amount of data IO devices read.
- `ItoM`: Similar to `PCIItoM`.
- `PCINSRd`: PCIe Non-Snoop Read. As there is no snoop, there is no cache coherency. This is only used for addresses that are not allowed to be cached, like memory mapped IO.
- `PCINSWrF`: PCIe Non-Snoop Write. Like `PCINSRd` but for full cache line writes.

To measure how TOR_INSERTS behaves for writes using different opcodes, I run very simple benchmarks on an otherwise idle system that write the same buffer to disk or the network until they have written 20 GiB.

For the writes on the disk the benchmark calls sync after after every write to avoid the block cache stopping writes to disk. For reads on the disk, the benchmark clears the block cache to ensure that reads are satisfied by the device. The program then reads to the same buffer until the entire file is read. Each read is as long as the buffer.

For reads of the network device I use the nc tool. Network devices write to the memory whenever a packet arrives. This is independent of the application that reads the socket. As such the application's read behavior does not influence the network device's memory bandwidth consumption. So any application that can open a port works to measure incoming network traffic.

The read and write benchmarks use the kernel space Linux drivers. The benchmark reports the the average bandwidth after it has finished.

Figure 3.2 shows the results from the network and storage benchmarks. The benchmarks all use a 1MiB buffer. Only RFO,PCIRdCur, and ItoM are plotted as the other opcodes only produce 0. The results are:

- PCIItoM, PCIWiL, PCIWiLF, PCINSRd, and PCINSWrF are always 0. This might be a bug on the performance counter.

- RFO is used significantly less than PCIRdCur and ItoM. Since the buffers that the drivers use are aligned to cache lines, I expect very few partial cache line writes.

- PCIRdCur and ItoM follow closely the bandwidth that the benchmarks report. Each event equals to a 64 bit cache line, so it is easy to calculate the bandwidth. The counters report a slightly higher bandwidth than what the benchmark reports. This is expected as the benchmarks cannot account for any network or file system overhead.

To further analyze the counters behavior, I let the benchmarks run simultaneously. Figure 3.3 shows the behavior of RFO, PCIRdCur, and ItoM. From the data I conclude that PCIRdCur and ItoM are independent from each other as I expect. ItoM also behaves as expected and correctly reports the cumulative bandwidth of both write benchmarks. Again RFO events only make a small portion of the total counters.

So far I have only analyzed how the benchmarks behave on an idle system. To check if there is any influence on the counters if the system is under load, I run a benchmark that fills the cache to 15MiB of the 20MiB and constantly operates on the data. I have not included a figure as from the point of view of

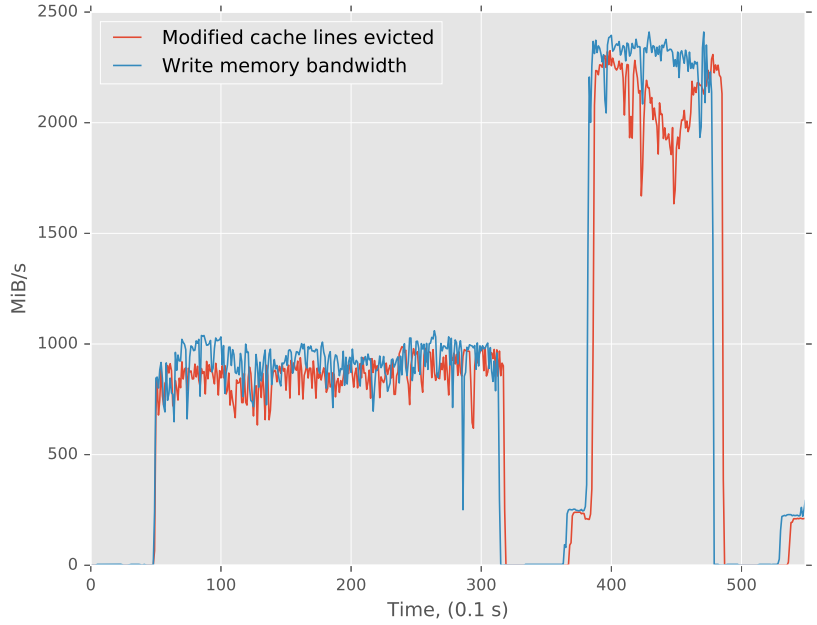


Figure 3.4: Comparison of the number of evicted modified cache lines with the write memory bandwidth reported by the memory controller. The bandwidth reported by the cache controller follows the bandwidth reported by the memory controller. The drift in the data is due to how the data was collected.

`RFO`, `PCIRdCur`, and `ItoM` nothing changes, the numbers figure are exactly the same. While there is a significantly higher amount of memory bandwidth, the counters that are responsible to measure the PCIe traffic are not influenced by it.

This shows that the `TOR_INSERTS` event counting `PCIRdCur` and `ItoM` requests can deliver accurate data about PCIe IO bandwidth. Both events behave as expected, they accurately count the events that the documentation specifies. Their results are not falsified when the system is under load. The `RFO` event sees significantly less use than the other two. Because `RFO` handles partial writes to cache lines, it is not evident how I can precisely calculate any kind of data from the `RFO` event. As `PCIRdCur` and `ItoM`'s data are behaving as expected and report the correct bandwidth, I believe that on my system I do not need to track `RFO`.

LLC_VICTIMS

To calculate the write bandwidth to main memory I use the`LLC_VICTIMS` event. It counts all the cache lines that are removed from the cache. This happens when there is a collision of two cache lines and one is replaced with the new one. Ac-

cording to the cache coherency protocol on the Xeon processor, only cache lines in the modified (M) state are written back to main memory. By setting `LLC_VICTIMS` to only evicted modified cache lines, I can easily calculate the amount of data sent to main memory by multiplying the counted number of events with the size of a cache line, 64 byte.

To verify that `LLC_VICTIMS` behaves correctly, I use the same benchmark than before for `TOR_INSERTS`. I compare the data from the event to data from the memory controller. In Section 3.3.5 I describe how to read the data from the memory controller. The benchmark is run on an idle system. I run the benchmark once for each event. As the benchmark does not always behave exactly the same there is some drift between the measurements. Figure 3.4 illustrates that the amount of events follows almost exactly the total amount of bytes written to main memory.

This shows that I can accurately measure the write memory bandwidth on the cache controller. It is preferable to the cache controller over the memory controller as the cache controller only counts the write bandwidth caused by evicted cache lines. The memory controller can potentially also count writes that bypass the cache. Unlike `TOR_INSERTS`, `LLC_VICTIMS` can not discern where a request originated from. As such while `LLC_VICTIMS` provides an accurate measurement of memory write bandwidth, I cannot use the `LLC_VICTIMS` event alone to estimate the write memory bandwidth of IO devices. However by combining the data from the IO bandwidth and the memory write bandwidth I can infer the write IO memory bandwidth.

Summary

The performance counters in the cache controller provide valuable data. With the correct events and filters the cache controller gives almost direct and accurate data about the total IO bandwidth and the total IO read memory bandwidth. It also provides data about the memory write bandwidth caused by cache misses which can be used to estimate the write IO memory bandwidth.

3.3.4 Home Agent

The memory controller on the processor handles every request to main memory. On the Xeon processor it is split into two parts, the Home Agent and Integrated memory controller. The Home Agent handles the protocol side of memory requests and orders them. It is mostly concerned about ensuring coherency and order on multi socket systems. Because of its role, the Home Agent's performance counters are mainly concerned with the QuickPath Interconnect (QPI) protocol that is used to link several sockets together.

It does have two events that count access to main memory. The `IMC_READS` and `IMC_WRITES` events can count read and write events to the integrated memory controller. As these events are first processed by the CBoxes, it is no longer possible to determine the origin of these requests.

3.3.5 Integrated Memory Controller

The second part of the memory controller is the so called Integrated Memory Controller (IMC). It handles the low level access to memory banks. The events that it can count are all concerned with the details of the low level access to memory.

The IMC allows to count reads and writes directly to memory. The event `CAS_COUNT` tracks the number of read or write commands to DRAM. This provides accurate memory bandwidth numbers. The amount of data can be calculated by multiplying the number of commands by 64 bytes. As the IMC is behind the home agent, it too cannot discern where an request originated from.

This illustrates that while the memory controller has accurate information about the memory bandwidth, It lacks a way to discern what caused the access to memory. This makes the cache controller the preferred functional unit to measure memory bandwidth as it has more data about a requests origin.

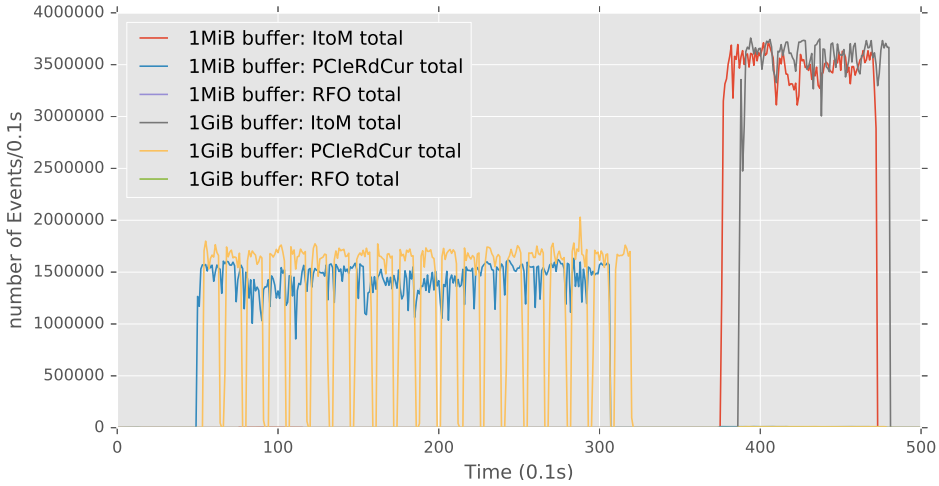
3.4 Identifying IO Device Behavior With Performance Counters

Ideally I want data about the bandwidth every device needs. This allows the system to easily measure the bandwidth of each device by using the hardware and then take data from the software about each process. That way the system can calculate the IO memory bandwidth each process needs. As I describe in Section 3.3.1, the only part of the processor that can still identify different devices does not provide performance counters. In this section I show that this is the only part of the processor that can reliably discern different devices, and that I cannot reliably identify devices by observing the performance counters.

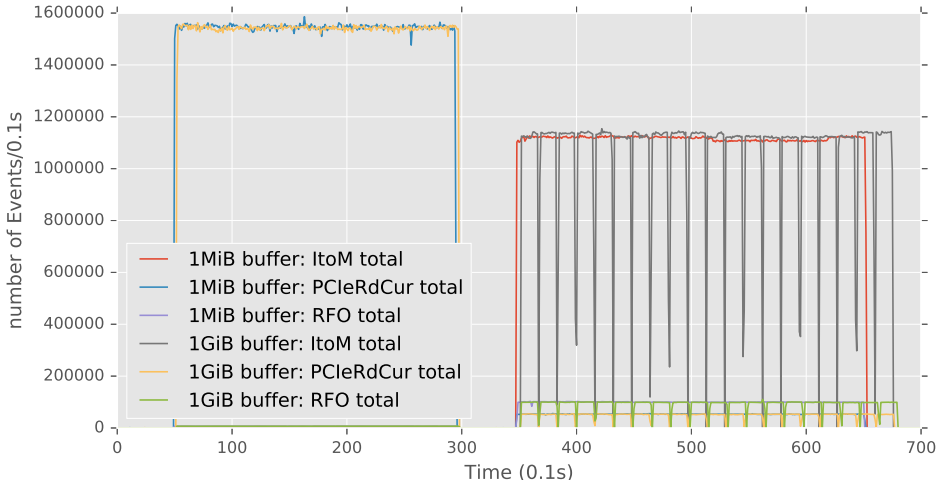
To determine if it is possible to discern different devices by observing their behavior with performance counters, I run several simple benchmarks while I record the counters. These are the same benchmarks I use in Section 3.3.3.

These benchmarks can use different sizes of buffers to read or write. In Figure 3.5 I compare the difference between a benchmark that writes to disk and one that writes to the network. Each graph shows the data from the performance counters with two benchmarks that use different buffer sizes.

The figure shows that the bandwidth on the SSD has more fluctuations than



(a) Storage device benchmark. The benchmarks are run with different buffer sizes. With 1GiB buffers, the write benchmark has a pattern that follows the time a buffer is sent to the OS.



(b) Network device benchmark. The benchmarks use different buffer sizes. With 1GiB buffers, read benchmark has a pattern that follows the time when the buffer is sent to the OS on the sending machine.

Figure 3.5: The difference between network and storage device behavior. The difference in behavior is mostly due to how the software uses the device rather than the device itself.

the network. For 1 GiB buffers, there is an obvious pattern where the bandwidth drops when the buffer is sent to the OS. For the SSD this happens on writes, for the network it happens on reads.

Both of these behaviors are due to the circumstances the benchmark was used in. For the network the lack of fluctuations in the bandwidth is due to the network being perfectly idle when the benchmark was run. If the network has some congestion, the behavior changes drastically. The performance counters also show that for 1GiB buffers, the data stream sent by the other system exhibits a clear pattern that follows the time the buffers are sent to the OS. However my test system can handle large buffers without causing any pauses and so it does not have the same pattern. For the storage device I force a sync after every write, the behavior again changes when the page cache is used. In both cases the data that the performance counters observe changes drastically when the software uses the devices differently

The two devices exhibit a very similar behavior in what events they trigger. For sending data and receiving data they both use `PCIRdCur` and `ItoM`. The only difference in what events they trigger is that receiving data on the network devices causes a higher amount of `RFO` events than the storage device. Moreover Figure 3.3 illustrates that `RFO` events continue even after both network benchmarks are finished. I conclude that I cannot use differences in events triggered to reliably to identify the device either.

From my measurements I determine that the most significant change in the number of counted events is due to the software that uses the device and the circumstances the device is used it. The numbers depend less on the device itself. So I cannot reliably use information from performance counters to discern the behavior of different devices. In the following section I present a different way that could be used to identify different devices, but current hardware does not support yet.

3.5 IOMMU

In this section I present a potential way a Input–output memory management unit (IOMMU) can be used to get detailed data about different device's and processes' memory usage. However, even though the different standards theoretically allow this system, at the time of writing, state-of-the-art hardware does not support the needed functionality.

The IOMMO for devices is similar to the memory management unit (MMU) for processes. Both units create a virtual address space that can isolate different users of the memory from each other. The IOMMU manages the memory access from devices and isolates them from the rest of the system. A device can no longer

directly use physical memory addresses, it has to use a virtual address space. The IOMMU is programmed with page tables similar to those used in a MMU. With the page tables, the IOMMU can potentially help to identify different device's and processes' memory bandwidth needs.

The Intel [7], AMD [2], and ARM [5] IOMMU specifications all support "accessed" and "dirty" flags similar to those in the page table. Like with the MMU and its page tables, these two flags are set if a page has been accessed or changed. These flags are often used on the MMU or by the OS to manage the limited space in the translation lookaside buffer. By regularly clearing these two flags, the OS can check which pages have been accessed and altered in a certain period of time. That way the OS can calculate the bandwidth each device needs.

At least the Intel standard allows multiple IOMMU page tables. This can be used to give different processes access to the device, without the processes being able to use the device to overwrite memory outside of their own address space. This is also useful to create an address space where video memory and main memory are represented as one continuous block. With this feature it is potentially possible to get information about each process's use of a device.

The issue with using the IOMMU is that this feature is optional. Setting the flags is up to the device. In my research I could not find any devices that support setting these flags.

For ARM only devices that implement the V2 SMMU specification can have set these flags. I could only identify the Mali 500 MMU that implements this specification. Looking at the chip's specifications, I conclude that it does not implement setting these flags, since it does not have the required registers to program the IOMMU to set the flags.

For Intel chips, in theory the HD Graphics 5000 GPU supports the the optional "Process Address Space ID" (PASID) feature that can set the accessed/dirty flags. This is to create a unified address space for main and video memory. Researching further however, the Intel IOMMU driver in Linux [1, drivers/iommu/intel-iommu.c] states that this feature actually does not work on Intel processors released at the time of writing even though they report that they can. Moreover, while the original specification from 2014 [7] states that the extended capabilities register reports PASID support on bit 28, the revised specification from 2016 [10] has changed this bit to state it is "reserved (defunct)". It is now on bit 40.

This illustrates that IOMMUs can theoretically provide the necessary information to the Operating System about what process accessed what device. However processors and IO devices available at the time of writing either do not support or have hardware bugs that prevent these features from being used. As such I need to rely on the software to provide the needed data. In the following section I detail what IO accounting is already present in Linux.

3.6 IO Accounting In Linux

I determine that by using only the hardware there is no reliable way to get per device data, let alone per process data. I need to rely on the software to provide the needed data. In the first section I describe what the Linux kernel provides to allow IO accounting per device and per process. This support is very rudimentary and I describe methods from other research that can be used to extend it in the other sections.

3.6.1 IO Accounting Structures

To track how much data each process sends to and receives from devices, I need to rely on the software to provide accounting. In this section I present the accounting that the Linux kernel already provides and how it is lacking important data.

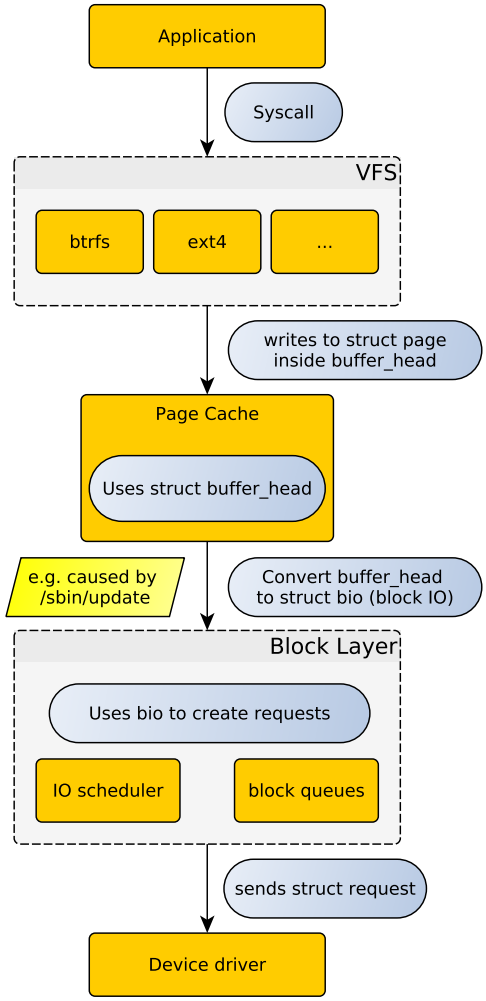
The Linux kernel tracks processes and threads that can be scheduled to run on the processor in data structures called `task_struct` [1, linux/sched.h]. The `task_struct` tracks, amongst others, the process ID, open files, or the last time the process was scheduled. To identify the process that made the syscall that the kernel is processing, it provides a special `task_struct` called `current`. For IO accounting, the `task_struct` includes a `task_io_accounting` struct [1, linux/task_io_accounting.h]. Additionally the kernel has functions to update the accounting data [1, linux/task_io_accounting_ops.h].

The mainline Linux Kernel when compiled with `TASK_XACCT` counts every byte a process reads or writes by hooking inot the unix read and write syscalls. The `TASK_IO_ACCOUNTING` option enables more accounting that tracks the number of bytes a process reads and writes to storage devices.

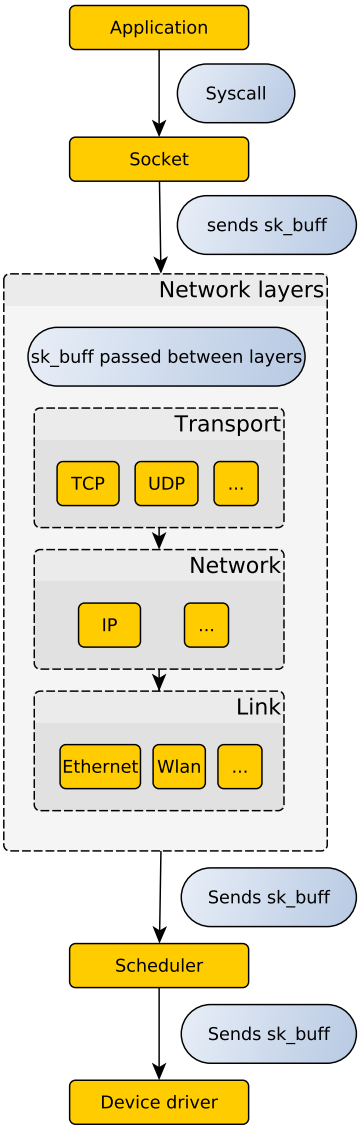
The`TASK_XACCT` option adds accounting to the various read and write syscalls. In Figure 3.6a and 3.6b the accounting happens between the application and the VFS or socket layer. To identify the process, the function this option provides uses the `current` process. As this option's accounting is on the syscall level, it cannot identify if the call operates on a storage or network device. To do so it has to reimplement most of the logic that are in lower layers of the IO stack.

To track reads and writes to storage devices, the `TASK_IO_ACCOUNTING` option adds accounting that hooks into lower level storage device functions. In Figure 3.6a this happens between the VFS layer and the Page cache. Filesystems either call the accounting functions directly or use abstractions from provided by the VFS layer. This call is still at a high enough level that the `current` task is the process that caused the read or write.

At the levels of the IO stack these two accounting mechanisms operate it is not possible to directly find what device services the request. The device that services the request is at the lower levels of the IO stack. On the other hand the layers that



(a) Simplified part of the storage device stack in Linux. Different data structures are used in each step. Different programs can cause data from the page cache to be written to disk



(b) The network stack in Linux. The same data structure is used throughout all the layers.

Figure 3.6: Comparison of data structures in the storage and network stack

these accounting functions hook into can reliably identify the process that called the original read or write with the `current` process.

Lower layers of the stack can identify what device request is sent to, but because of buffers and different schedulers, they can no longer reliably identify the process that caused the original request.

For example with block IO, it is likely that a request is first stored in the page cache. The block is only written to the device after a process like `update` forces a write to disk. This also sets the `current` task to the `update` process, which is not the process that actually caused the write. This leads to inaccurate accounting information.

The built in IO accounting structures are not accurate enough to track per process, per device accounting. However they establish a framework I expand to provide this information. They illustrate where the kernel can identify processes. In the next two sections I describe the Linux kernel's facilities to track network and storage device's activity.

3.6.2 Accounting For Network Devices

To calculate the IO memory bandwidth each process needs, I also need to measure the total amount of data every device has processed. To further understand where this accounting is best implemented in the stack and how to handle per device accounting I describe what the Linux kernel already implements for device accounting. In this section I start with network devices and in the next I describe storage devices.

The linux kernel's representation of a network device is the `net_device` [1, linux/netdevice.h]. It has `net_device_stats` to track the amount of packets, bytes, errors, etc. the device has had. For example`ifconfig` displays some of this data. Layers higher up in the stack than the device drivers are unable to count all the data. For example transmission errors or invalid packets are not always reported up the network stack. Because every device is different, it is up to each network device driver to update these stats.

This illustrates that the only stage where the kernel can track all the data about a network device is in the device drivers. It also lays the foundation for further accounting in network devices. To further investigate per device accounting I describe storage device accounting Linux provides in the next section.

3.6.3 Accounting For Storage Devices

Apart from accounting for network devices, the Linux kernel also provides accounting for storage devices. The statistics can be read in `/proc/diskstats`.

The data saves for example the amount of requests and sectors the device has processed since the system booted up. This data is saved per device and per partition.

To perform accounting on storage devices, the Linux kernel uses the `requests` that the device driver receives. Because storage devices usually use a standard interface like SCSI or NVMe, there are far less drivers for storage devices than network devices. Generally, these drivers update the device's accounting data when a request is finished. Some devices however finish a request in several iterations, which means that the layers above the device driver cannot accurately provide accounting as they do not receive partially finished requests. Each driver needs to use the accounting functions to update the accounting data either by calling them directly, or more commonly via a helper function that the block layer provides.

Like with the network stack, this shows that the accounting for devices can only be accurately done in the device drivers.

While both layers track somewhat similar data, they represent it differently. The network stack reports bytes and packets, while storage stack reports sectors and requests.

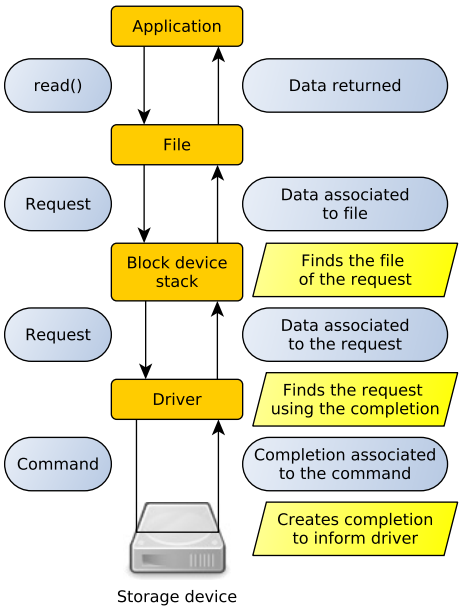
3.6.4 Connecting Upper Layers With Lower Layers

As I show in Section 3.6.1, the only the higher layers of the IO stack can reliably identify a process, while Section 3.6.2 and 3.6.3 illustrate how only the low level device drivers can accurately track a device's activity. For a system to provide per process per device accounting, data from the higher levels of the stack needs to be connected with data from the lower layers. In this section, I explore how the network stack and storage stack differ when trying to connect the high and low layers of the stack.

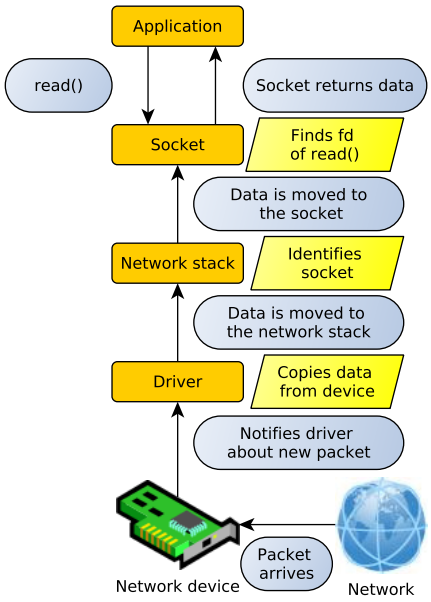
There are two models for reads that devices follow. Either they behave like a storage device or network device. Figure 3.7 shows the difference between the two groups of devices. Storage devices first need a request from an application before they start any kind of transfer. Network devices however can receive a new packet at any moment without any involvement from an application. The different data flows mean that connecting the different layers cannot be handled the same way in each IO stack.

In this section I focus mostly on how reads are handled, as they have the most significant difference. For a write it is easy to connect the upper with the lower layer. For both stacks the request is moved from the application down to the device. This means that this request can be used to transport additional information that can be used for accounting.

This is more complicated for reads, especially for network devices. In the following sections I first describe how the storage device allow for an easy transfer



(a) Reads on storage devices cause the device to write data to memory



(b) Reads on network devices only read data from the socket. The device can write data to memory at any moment when a packet arrives

Figure 3.7: Different access models for reads. Some devices only write to memory after a request like storage devices, some devices can write to memory at any time like network devices.

of data through all layers and then how for network devices this becomes a much more involved process.

Reads On Storage Devices

Figure 3.7a shows how a read request is processed by the storage device stack. The application uses the system call and the request is moved down the stack to the device. After the device has copied the data it creates a so called completion to notify the driver. The completion has the necessary data that the driver can associate the data to a request. With this information the data is moved back up the stack.

This means that storage devices can at every layer of the stack identify what request data is associated with. While the current implementation in Linux does not move data down the stack to identify the calling process, the data flow makes identifying the reading process easy. The data flow allows that all data copied to main memory from a device can be uniquely attributed to a read call and as such be accounted to the correct process.

Reads On Network Devices

Network devices behave differently on reads than storage devices. Figure 3.7b gives an overview of how network devices handle reads. The main difference is that a packet can arrive at any point in time without a process requesting any data first. The device then notifies the driver about a new packet. The driver copies the data from the device to a buffer and hands it up to the network stack. At this point the device has already consumed the memory bandwidth, but the network stack does not know yet what process will eventually read the data. Consequently, the network stack cannot do any per process accounting at the time the IO request happens.

The network stack uses the various network layers to identify the socket the packet is destined for. Generally only the highest layer, the transport layer, can identify the socket. The application has no involvement in this process, it only picks up data that is already stored in the socket. Reading data from the socket and the device writing data to memory are generally independent of each other. This is in contrast to the storage stack where each read of data leads to the device writing data to memory.

Because the device writing data to memory and the process reading data are independent, the network stack can not identify the reading process when it delivers data to a socket. However a system that provides accounting for each device a process uses needs this information. So instead of directly identifying a process in the network stack, I explore how a process can be tied to a socket using the data

structures that are present in the Linux kernel. I describe how this method, while simple, cannot solve all the problems that can arise in the next section.

Identifying Socket Owners

The transport layer can potentially use information about the process that reads the socket, the owner, to update a process's accounting on network reads. Identifying the owning process of a socket however is complicated. While the the socket has a field to identify the owning process of the socket, this is not always set. This is due to the `fork` system call migrating file descriptors from one process to the next, which means that multiple processes can potentially read the socket. In other words there can be multiple owners. In the mainline Linux kernel, sockets only use the owner field when a process wants to receive the `SIGURG` signal. In that case the process needs to set itself as the owner with `fcntl`. This means that generally I cannot expect that the socket's owner is set. In the following I discuss several ways to identify the owning process.

A straight forward way to update the owner of a socket is to set it when the file descriptor is created. This happens in the `socket` and `accept` system calls. As on the system call layer the kernel can reliably identify calling processes, it can set the owner of the socket to the process that caused the system call. However because of `fork` there can be multiple processes that utilize a socket. A common use-case are processes that listen to new connections and call `accept`, then `fork` to handle new connections in separate processes. The parent process then closes the new socket and listens to incoming connections again. While from the perspective of the processes it is easy to identify what process has opened which socket, the socket layer is oblivious to the change. The socket layer does not keep track of what process uses what socket, since this information is saved in the process's data structures. In this case the owner of the socket is set to the parent process and is not changed even after the child process takes over the socket. This is problematic as this wrongly attributes reads to a process when it creates a child process to handle requests. For example this creates the wrong data for SSH, where a new session is a new process that is run with the user's privileges.

The OS can alternatively update the socket owner on every `read` or `write`. This solves the issue when a process `accept`s and then `close`s the socket. However there is a race condition when a packet arrives before the new process has called a `read` or `write`. This is unlikely to cause a significant error in accounting as I can expect the new process to `read` the incoming data in a timely manner and thus update the owner. It however does not solve the issue when several processes share a socket. In that case there is another race condition, a process can update the owner while there is data in flight from another process. This leads to incorrect accounting data.

Both proposed solutions have shortcomings and cannot correctly identify the process that reads a socket in every case. The only way to correctly account for all network devices a process uses is to use data from the driver and move it up the stack. When a process reads the socket the information from the lower layers can then be used to correctly update the process's accounting data.

As a socket can have multiple interfaces, each byte in the socket needs to be tied to a device. Each byte of data needs to be tagged and able to be tracked back to the interface that has received the packet since a process is free to read as much or little data it wants from a socket.

This highlights the issues that the data flow of the network stack causes. Because a packet can arrive at any moment, the network stack does not know what process will read the data. While easy solutions to define a process as a socket's owner can solve some of the problems, the only solution to avoid all these problems needs to be much more involved.

3.6.5 Summary

After exploring the systems that are in Linux I conclude that the accounting in Linux does not satisfy the needs of my system. However it shows how such a system can be implemented.

The two kernel options that add per process accounting show that accounting for processes can only be reliably done in the upper layers of the IO stack. This system works for a global counter that tracks all data a process has written or read. However for a system to provide per process per device accounting, it needs data from the devices. This data can only accurately be provided by the low level device drivers. This means that both the upper and lower layers need to be connected. However due to differences in the network stack and storage stack, both need a different approach to implement this.

3.7 Possible Modifications

The Linux kernel has only limited capabilities to provide IO accounting. In this section I analyze several systems and how they can be used to extend the capabilities the kernel has to provide accounting.

3.7.1 Cgroups

To have per process IO accounting the Linux kernel needs to provide a way to identify the process in different layers of the IO stack. Linux already has a powerful system that passes through the different layers of the IO stack, so called

control groups (cgroups)[1] [1, Documentation/cgroups]. In this section I illustrate what cgroups already provide and how they lay a foundation to move additional data through the IO stack.

The Linux cgroups are a way to group processes and enforce certain restrictions. Cgroups can handle different tasks depending on what subsystems are loaded, for example isolating processes with the use of namespaces, or restricting memory and CPU usage. To estimate IO bandwidth cgroups provide the block IO (`blkio`) and the network subsystem.

Cgroups For Storage Devices

For storage devices, the `blkio` controller subsystem can restrict the IO bandwidth a cgroup can consume on a certain device. That means that the block IO cgroup needs to keep track of how much data has been sent over time. The cgroup subsystem uses hooks in the IO scheduler to measure the group's data usage. It checks how many requests, bandwidth, or bytes a cgroup has used and if it is still within the limits. If a request is over the limit, the scheduler does not pass the request to the device immediately and wait until there are enough resources available to schedule the request.

The scheduler finds the cgroup of a block io request by checking the block IO (`bio`) for this request. Each block io has a cgroup subsystem state that keeps track of the cgroup the block IO originated from.

Setting the cgroup however is a complex task as a request can take many different paths through the block layer. Depending on the device, some requests are immediately written to the device or they are first cached and at some point written back to the device. The Linux kernel handles devices that bypass the cache by setting the cgroup to that of the `current` process. This does not work for requests that are written to the block cache first as the cache uses a different data structure.

When a block is saved in the page cache it is no longer considered a block IO, but is a memory page. To allow the cgroup data to persist when the data is transferred from a block IO to a memory page and vice versa the `blkio` subsystem implicitly loads the memory subsystem. This way cgroups can be associated with certain memory pages. Every page in the page cache has memory cgroup information. When a block IO request is written to disk, the Linux kernel needs to copy this information from the page in the cache to the block IO that is transmitted to the device driver.

The `blkio` cgroups implementation shows how a system to track per process block io requests can be built. It shows how information from the upper layers of

[1] For an overview about cgroups: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html

the stack can be moved down to the scheduler level. It is necessary to attach data to each request and migrate this data when the block is written to disk.

Cgroups For Network Devices

While the cgroup subsystem for storage devices provides an extensive and complete way to move data through different layers of the storage stack, the network subsystem is more limited.

The networking cgroup subsystems can only set priorities. These are the same that can be set with socket options. The network subsystem only tags for the traffic controller (TC) in Linux. The TC is part of the network scheduler in Linux, it determines what packets are sent or dropped and in what order they are sent. To identify what policies should be used on a buffer, it can use the class identifier that is set per cgroup.

However the `sk_buff` only saves parts of the class identifier, a 16 bit `tc_index`. Depending on the scheduler, this index can be used differently. Also depending on the scheduler is the traffic controller protocol that is used to process the buffer. To get per cgroup data from the traffic controller there have to be extensive modifications to the code to allow other layers of the network stack to access the internal data of the different classifiers.

Therefore the network cgroups subsystem is too limited to provide the data to identify processes to different layers.

Cgroups For IO Bandwidth

For IO bandwidth accounting it is desirable to have a new cgroup subsystem. That way administrators can easily set a policy and add more processes to the cgroup. Cgroups are a powerful mechanism that can implement bandwidth measurements and tracking limits. Both the storage and network cgroups can limit bandwidth already, however they only work with their specific scheduler, not the process scheduler. Neither of the existing cgroups can provide the data that the scheduler needs. They however lay out the building blocks on how a new IO memory bandwidth accounting cgroup can be implemented.

For storage devices the system is almost implemented, all the needed systems to copy the accounting information as the data travels through the stack are already in place.

For networking the implementation is more complicated for the same reason I present in Section 3.6.4, it is hard to correctly attribute incoming data to a process. Processes inherit the parent's cgroup, but they are not locked to that cgroup and are free to change it. While an administrator can set a policy that no process can

change cgroup, there are programs, for example ssh, where changing the child process's cgroup is desirable.

The existing cgroups show how a new accounting and limit system can be implemented but run into the same issues that accounting for each process has.

3.7.2 Split Level Scheduling

The block layer in the Linux kernel has many branches a request can take[2]. This makes accounting on the block layer especially complicated. Yang et al. describe a way to give the block IO scheduler more information about who or what cause IO operation [19]. They add a new data structure they call "cause" to track different events on a block IO's path.

When a process creates a block IO in the VFS layer, it is tagged with a cause that saves the process. Similar to block IO cgroups, at each step, for example when the block is moved to the cache, the causes are copied to the new data structure. Every time the causes are copied or the block IO is touched, new causes are added to track how the block was processed. This way the IO scheduler can make a more informed decision on how to order requests.

As they track what process caused the initial write, this system can be used to precisely track what process caused which device to start DMA operations.

3.7.3 Memory Introspection

So far I have only discussed drivers that the Linux kernel has direct access to. However, for performance reasons, some devices can use drivers that are in user-space. The kernel is only involved in setting up the process's and device's shared memory region and programming the device. Any transfers from the device to process are directly written to the processes' address space. The kernel cannot see how much data the process sends or receives.

A way to still do accounting in this case is to periodically monitor the shared memory regions between the device and process. Memory introspection as described by Ranadive et al. [15] monitors the user-space command buffers and can estimate the process's bandwidth needs. The kernel knows what process has initialized what memory region and can add accounting data about activity that otherwise bypasses the kernel.

This system needs detailed information about how the driver and device interact to be able to gain the necessary information.

[2] A much more complete graph showing the Linux storage stack: https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram

A system like this can be used to extend the accounting for the kernel to also track user-space drivers, though it needs to be implemented per device.

Chapter 4

Design

In this chapter I describe the design of the system to estimate IO memory bandwidth. I illustrate the system's general design in Section 4.1. The system takes detailed but aggregate memory bandwidth usage data from the hardware, and combines it with per process accounting data from the operating system. I describe the hardware side in Section 4.2. In Section 4.3, I describe the goals of the Software. I describe the two classes of devices, network and storage devices, that I track and how the systems handles their differing behavior. With the information from the hardware and software I describe how to calculate the IO memory bandwidth of each device or process in Section 4.4. To allow these calculations, the system needs different data structures that I detail in Section 4.5. Finally, in Section 4.6, I show how the system is integrated into the Linux kernel.

4.1 Overview

Current operating systems do not have data about a process's memory bandwidth consumption. Like any other shared resource in the system, the memory bandwidth should be distributed fairly among the processes. There are systems to estimate a process's memory bandwidth, however they do not provide data about the memory bandwidth IO devices consume while they process requests issued by the application. As I demonstrate in Section 3.1, memory bandwidth that processes indirectly use via IO devices has a significant impact on other processes. In my case over 10% of the available memory bandwidth is used by IO. Without accounting for the IO memory bandwidth, the operating system does not have a complete picture of a process's memory usage. Consequently, systems that only measure the memory bandwidth that a process consumes directly cannot fairly distribute the shared memory bandwidth. The goal of this thesis is to develop a system to provide data for the operating system about applications' IO memory

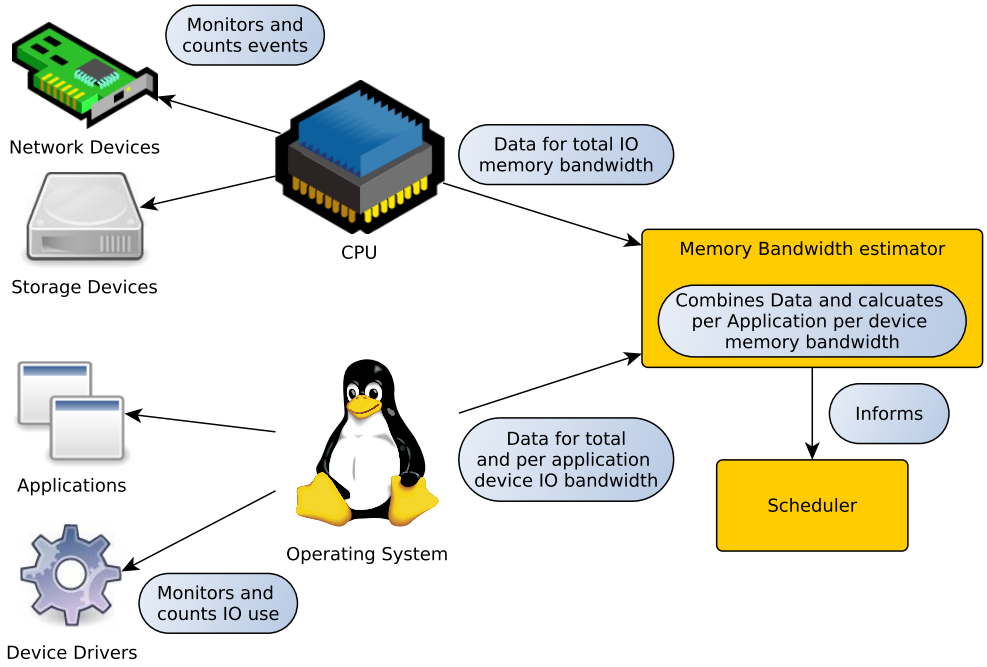


Figure 4.1: Overview of the IO memory bandwidth estimation system's design

needs so that the scheduler can make more informed scheduling decisions and allow a fair distribution of the memory bandwidth.

The Operating system generally has no knowledge of the memory bandwidth that is consumed on the system. The OS needs data from the hardware about the memory bandwidth. However, the performance counters can only provide aggregate numbers. In Section 3.4 I show that after analyzing the data from the hardware, I can not reliably discern different device's activity. On the other hand the operating has detailed information about process's and devices' activity. I propose to combine the information both the performance counters and the OS can provide to estimate the IO memory bandwidth.

Figure 4.1 depicts the system I propose. The hardware provides accurate total numbers about the devices activity. The Operating system monitors the device drivers and applications to provide data about how the processes use the devices. By combining the data from the CPU's performance counters and the OS, the memory bandwidth estimator can calculate each process's IO memory bandwidth consumption. This data can then be used by the scheduler.

With this system the scheduler has the necessary data to also include memory bandwidth a process consumes indirectly via IO devices. In the remaining sections I detail each subsystem's role and design.

4.2 Hardware

The system I propose need data about the memory bandwidth, the activity of processes, and the activity of devices. I have determined that the most reliable way to get data about the memory bandwidth is to read the CPU's performance counters. I limit myself to single socket systems in my design, but multi socket support can be implemented if the counters can discern what socket requests originate from. In this section I describe what data the hardware needs to provide for the system.

At the bare minimum the hardware needs to provide data about the total IO bandwidth the devices produce and the total memory bandwidth. Consequently the performance counters need be able to filter the data to only include requests from IO devices. If they cannot, it is almost impossible for the software to accurately calculate the bandwidth for each process.

The functional unit best suited to give detailed information about memory bandwidth is the cache controller. This is because the cache controller needs to process each request to determine if it can be satisfied by the cache. This way the cache controller has information about the origin and the type of request that it processes. The memory controller interacts only with the cache controller and thus has less information about the requests. This leaves the cache controller in a an ideal position to provide quite a variety of data concerning memory bandwidth.

On the Intel processor in the test system, the cache controller can count the amount of cache hits, and more importantly, to estimate memory bandwidth, the amount of cache misses. While read misses on some systems can be satisfied by copying data from other caches, on a single socket system a miss in last level cache must be handled in memory. On writes the Intel architecture only accesses the main memory when the processor writes a modified cache line back to main memory. As I demonstrate in Section 3.3.3 the Intel processor provides events to track both metrics.

In an ideal system, the hardware also provides a means to discern how much IO bandwidth each device consumes. With per device IO bandwidth information it is easier to calculate per device memory bandwidth usage. This can only be counted by the PCIe controller as it is the only part of the processor that is directly connected to every device. However as I show in Section 3.3.1 the PCIe controller in my system cannot provide this data. This means that the software needs to provide the per device bandwidth data.

Finally it is important to consider what information the data provides. Depending on the load that is on the caches, an IO device can cause a a much higher memory bandwidth consumption than on an idle system. The question is who do I attribute this increase of bandwidth to? Is it the device's fault that the application using the cache has more misses or is it the applications fault because it uses the cache so heavily? To answer this question I decide to only account for data

that the CPU can directly link to the device. When a device writes to the cache and has a miss that leads to a memory access, I count it against the device. If an application has a miss because the device has overwritten a cache line, I do not count it against the device as the hardware can only provide data about the first case. Anything else is speculation. Consequently I only count cache misses that the device directly causes.

In the system I propose, the hardware provides accurate data to calculate the memory bandwidth. However, the hardware cannot reliably provide data about each device's or process's activity. This data needs to come from the software which I discuss in the next section.

4.3 Software

In the previous section I show that while the hardware has detailed information about the total IO bandwidth and total memory bandwidth, it cannot provide information about a device's or a process's activity. In this section I describe what data the software needs to provide to the memory bandwidth estimator. The software needs to collect data for each device and each process. To gather this information I need to answer several questions:

- What devices should be tracked?
- What classes of devices are there and what are their differences related to getting accounting data?

After answering these two questions I can determine how to proceed with the system's design and answer the following questions:

- Where in the software stack can the OS gather bandwidth data?
- How can the OS attribute the data sent or received to a certain process?

In the following sections I aim to answer these questions.

4.3.1 Devices To Track

The first question to answer is of course what devices should be tracked? As I want to allow the scheduler to fairly distribute the memory bandwidth, I can safely ignore any devices that only need very little IO and thus memory bandwidth as they have a negligible impact on the systems performance. Devices like keyboards, or sound cards barely produce any data.

The devices I believe are the most important to observe are storage devices and network devices. Storage devices, especially modern SSDs that are directly attached to PCIe, can produce vast amounts of data in a short time. They can reach up to several GiB/s of bandwidth. Modern Network hardware provides high bandwidth like to 10 Gb Ethernet or 112 Gb Infiniband. This data needs to be transported from main memory to the device and vice versa, taking much of the PCIe and memory bandwidth that is available. As I show in Section 3.1, these devices can consume a considerable amount of the available memory bandwidth from applications.

For this thesis I limit myself to devices that have drivers in the kernel. I track the bandwidth of a SSD and a 10 Gb Ethernet controller. The next question that has to be answered is how these devices differ and how this impacts how they can be tracked.

4.3.2 Classes Of Devices

In the previous section I determine to track network devices and storage devices. In this section I propose that these devices belong to two classes of devices that fundamentally need different methods to track their per process activity.

I analyze their behavior and how it dictates how the OS can track their activity in Section 3.6.4. I identify two classes of devices: Devices that only produce data after a request, and those that can produce data at any point. Figure 3.7 depicts how both classes behave.

The first category includes devices like hard drives or SSDs. Only after a request from the an application does the device write it into a buffer. This makes it easy for the OS to control how much data the device produces and how much bandwidth it has available. Moreover it facilitates tracking what process receives what data as there is always a request that needs to travel from the process to the driver before the data is transported back up to the process. As I limit myself to storage devices in this thesis, I refer to this class of devices as storage devices. Figure 3.7a shows the typical way an application reads from such a device.

The second category is devices like Ethernet controllers or Inifiniband network cards. Figure 3.7b shows how these devices behave on a read. These devices can receive new packets of data at any point from the network. This data needs to be copied to memory first and processed before the OS can do any kind of accounting on it. This means that there is no straight forward way to limit the bandwidth on these devices and it becomes harder to determine what process the data is destined for. I refer to this category of devices as network devices.

In Section 3.6.4 I show how the different access models change how the OS needs to handle these devices. For storage devices it is relatively easy to track what process caused a read, while for network devices it is a complicated task.

In short, there are two different classes of devices that behave differently and thus need to be handled differently to get accurate data. In the next section I describe where in the IO stack the OS can gather the necessary data.

4.3.3 Tracking Device Activity

I want to track network and storage devices. The next question is where in the IO stack can the Operating system find the most accurate data? In this section I describe how the Linux network stack provides a robust foundation to implement such a system.

Linux does not have a standard way to collect bandwidth data. However, in Linux it is easy to see how much data has been sent over a network interface, for example by calling `ifconfig`. The network stack shows a possible way to implement such a system.

In Section 3.6.2 I show how each network device keeps accounting information saved in its own data structure. Device drivers then implement a way to update the accounting. With each call to the driver, the upper layers provide a way for the driver to identify what device the data is destined to. This is also true the other way round, the driver knows what device data arrives on. That way the driver can always identify the device to update the accounting data. A driver is free to choose how or when it wants to update the accounting numbers.

The next question obviously is why Linux updates the accounting at the lowest possible layer?

The first reason is that every device is different and only the driver knows what exactly has to be done to account for all the activity on the device. The second reason is that only the lowest level layer of the network stack knows about all the activity on the link. This layer is the only that knows if there are to be retransmissions or if there are packets that are dropped because of full buffers or faulty checksums. These packets are obviously not transported to higher layers.

These reasons also hold for storage devices as I describe in Section 3.6.3. Different protocols handle requests differently. Some finish requests with several iterations, some always return complete requests. Higher layers no longer know how many sectors were transferred to the OS. For example if only one byte of a sector contains data that the application needs, the entire block needs to be read first.

The network stack shows that the best way to gather accurate data for IO devices is in the driver. This is also true for storage devices. However on these layers of the IO stack it is no longer possible to do accounting for processes, only for devices. In the next section I describe what additional data is necessary to monitor processes.

4.3.4 Process IO Accounting

Usually data that is transferred needs additional metadata. For example network headers or file system metadata. Accordingly to accurately measure the bandwidth the system needs to take this additional data into account. Which means that the system needs data from within the IO stack where it ideally can still identify the process the data is destined to or originates from, and where it can determine how much data is sent to the hardware. However, as I describe in Section 3.6, 3.6.2, and 3.6.3, these two goals oppose each other. Identifying processes is reliable on the system call layer. Exact numbers on how much data needs to be transferred to a device are only available on the driver level. In this section I describe how the different layers of the IO stack need to be connected to solve this conflict.

Traversing The Storage Stack

For the storage device stack, due to its complexity and different paths the data can take, moving accounting data through the different layers is a complex task. As I show in Figure 3.6a different data structures to move the data are used on different layers of the storage stack. The most drastic change is when the data is written to the page cache. At that point the data is no longer handled as a storage device IO but as a memory page. When the data is written back to the storage device, the data from the memory page is transferred into a storage device IO request and then transferred to the device. This illustrates how data needs to be copied from the various data structures to the next.

The block IO cgroup implementation shows a possible way to keep track of what data originated from where. I describe this cgroup in more detail in Section 3.7.1. When a block is moved from the block cache to the disk, the cgroup information needs to be copied or inferred from the memory page. The device driver can then use the information from the memory page in the storage device IO data structure to update the group's accounting.

One implementation to track a process's block IO activity that follows the block IO cgroup implementation is the system by Yang et al. proposed in the split level IO scheduling paper [19]. Instead of just keeping one state however, they chain so called "causes" each time the data is handled by a different subsystem. That way the IO scheduler can track what process caused what operations on a request for a storage device. This system can be used to track what process has sent what data.

While this system has only implemented the ability to track writes, the authors claim that adding support for reads is easy. As I explore in Section 3.7a, block devices only send data after it has been requested from the software. To track reads, the request needs to be tracked from the syscall level to the driver where

the driver can then update the accounting data. So it closely resembles the way writes are tracked.

I propose to follow the block IO cgroup implementation to move accounting data through the storage stack. The system from the split level IO scheduling paper implements my proposal and can easily be extended to support both reads and writes. With this system in the storage stack, the OS can connect the upper and lower layers of the stack with the lower layers and track the IO bandwidth each process creates on each device.

Traversing The Network Stack

The network stack is easier to modify than the storage stack as the request's path follows the network layers and the only branches are different protocols on each layer. As I show in Figure 3.6b, every network request is transported with one `sk_buff` from the transport layer down to the driver and vice versa.

Going from top to bottom, each `sk_buff` includes the original socket that the request came from. As each socket also includes a way to set the owning process, information about the socket can be transported to the driver so the driver can update the process's accounting data.

Unlike storage devices, network devices can receive data at any time, the driver has no way to identify what process eventually receives the data. Since each `sk_buff` tracks the size of the data and headers, it is easy to also add a field to track the driver's data overhead. The transport layer identifies the socket that the data is transported to, at this point the transport layer can update the accounting data of the process.

As discussed previously in Section 3.7b it is hard for the socket layer to know what process will read the socket. So I cannot always accurately identify what process should have the accounting data updated. I assume that the most common case are applications that directly use a socket and applications that fork and then close the new socket in the parent process. In this case, as I mention in Section 3.7b, the most accurate system is to update the socket's owner on each read or write. However for the sake of simplicity I decide against this and only update the socket's owner on creation. This means that my system cannot accurately account for applications like SSH. It is however sufficient to show that the system can produce viable data.

This shows that while the network stack is easy to traverse, It cannot easily attribute data to a process. Because of this difficulty I chose to have a less accurate per process accounting for the sake of simplicity.

4.4 Combining Hardware And Software

In the previous sections I discuss what data I gather from the hardware and what data the software provides. On their own, neither source of data can provide the information about what process's device usage caused how much memory bandwidth. In this Section I describe how both datasets can be combined to provide this data.

The hardware does not allow me to track each device's memory bandwidth or bandwidth in general. However I can calculate their bandwidth.

First I discuss how to calculate the read memory bandwidth. On a single socket system read misses in LLC cause a read from main memory. So I can calculate the read IO memory bandwidth from the number of the read cache misses caused by IO devices.

For writes I verify in Section 3.3.3 that the amount of evictions of cache lines in the modified state follows the write bandwidth closely as I expect from the cache coherency protocol. The hardware, however, cannot attribute write-backs to an origin. To solve this problem, I assume that the chance of a cache miss causing a write back to memory is the same for every miss. I can assume this as memory addresses are evenly distributed across the last level cache. The Intel processor uses a proprietary hash function that evenly distributes the physical addresses in the cache [8, Ch 2.3]. This means that two devices producing the same amount of misses also produce the same amount of write-backs. With this data I have an accurate way to calculate the total memory bandwidth from IO devices. I calculate the share IO devices have on the total cache misses and multiply this with the amount of modified cache lines the cache controller writes back to memory.

With these two components, the formula to calculate the total memory bandwidth is:

$$M = (R_{IOmiss} + (\frac{W_{IOmiss} + R_{IOmiss}}{W_{miss} + R_{miss}} \bullet WB_{Mstate}))/time \tag{4.1}$$

with M=total IO memory bandwidth, R_{IOmiss}= read misses from IO, R_{miss}= all read misses, W_{IOmiss}= write misses from IO, W_{miss} = all write misses, and WB_{Mstate}= Write back of modified cache lines.

The next step is to split that data according to the ratio each device or process uses of the total.

As the OS tracks the total data that each device sends or receives, the system can calculate the portion each device has of the total. I assume that the distribution of misses and thus write-backs is proportional to the amount of data a devices produces. In other words, a device's share of the IO bandwidth is the same as its share of the IO memory bandwidth. The same is true for each process. As the OS can calculate the amount of data each process needs from each device, it can

again calculate the share the process consumes of the total bandwidth and use that share to calculate the IO memory bandwidth.

The formula for the memory bandwidth of a device or process is:

$$m = \frac{d}{D} \bullet M \tag{4.2}$$

where m=IO memory bandwidth of a process or device, M=total IO memory bandwidth, d=IO data of a process or device, and D=total IO data.

As the OS can track reads and writes separately the system can also split the data up into reads and writes if necessary.

With this system the OS can calculate accurate data about the bandwidth from each device or process. To allow the system to calculate the bandwidth it needs data structures to save and provide the needed data. In the next section I explain the necessary data structures.

4.5 Data Structures

In the previous section, I demonstrate how the OS can calculate the IO memory bandwidth. In this section I discuss the data structures the system needs for each process or that are globally tracked.

4.5.1 Data Structures For Processes

I want to track the bandwidth each process uses, as such I need to track data for each process. What I need to answer now is how fine grained I want my design to be and how this influences the data structures that allow this.

My goals are :

- To calculate the bandwidth each process used since the last time it was checked.
- To track received and transmitted data separately.
- To have a fine grained overview of what device produced how much bandwidth for each process.
- To support an arbitrary number of devices.

This means that:

- For each process I measure the length of time that passed and the amount of data that was produced since the last update.

- I need to save both incoming and outgoing numbers of bytes separately.
- Each process needs to save how much it utilized each device.
- The data structures can adapt to different numbers of devices.

In the following I show what data structures each process needs to achieve these goals.

As the system concerns itself with bandwidths, it obviously needs to measure the time between each interval a calculation occurred. So the system needs to save the last time the bandwidth was calculated for each process. While any kind of measure of time can be used, like CPU cycles, I decide to use the wall clock. I choose this measurement of time as it is the easiest to understand and debug.

I need the amount of data the process has produced since the last check. So the OS has to save the state every time the system updates the bandwidth to calculate the difference on the next check. The OS also needs to save the state of the performance counters for each process each time the bandwidth is calculated. This data is necessary to calculate the total IO memory bandwidth. As the processor constantly counts the number of events, the OS can always get the current total from the performance counters.

Because I want to track both incoming and outgoing bandwidth, the OS has to track these numbers separately. Every data structure utilizes a separate variable for each direction.

The OS does not know in advance how many devices each process uses. Consequently the data structures have to be able to hold information about an arbitrary amount of different devices. As a process generally does not access all the devices on the system, I choose a data structure that each device the process has accessed can add itself to. Every time a devices handles a request on behalf of the process it can update the data. That way the system does not waste memory to track devices that are never used. This also speeds up calculations as the system does not calculate the bandwidth of every possible device, only those that are in use. I choose to use a hash map that holds the per device state.

Finally each process needs to save the global state of all the devices. This is the state of the performance counters and the global state of the device's IO bandwidth as tracked by the OS. Since this state incorporates all the devices, it can be saved in two variables.

The data structures I describe in this section fulfill the goals I have set for the per process data tracking. However, this still leaves the problem that each process needs to access counters to update the per process data structures. I discuss this in the next section.

4.5.2 Global Data Structures

The kernel needs to track the total amount of IO bandwidth to calculate the bandwidth each process consumes. The processes save the amount of data they produce to calculate their activity during a certain time. All I need from the globally accessible counters is that they separate each device and they separate outgoing and incoming data.

Like for processes every data structure saves incoming and outgoing data separately.

To allow each process to update their private counters, The system has to save the amount of data each device has produced since the system started. For the same reasons as for processes, this data structure should only track devices that add themselves to it. Again I choose a hash map to hold this data.

The globally accessible data structures are simpler than those of the processes as they do not need to track any time frames, they are updated whenever a device sends data.

With these data structures the OS can provide the data that is necessary to update each process's internal data structures.

In summary the system saves data for each process that allows it to calculate the difference of data that is processed in a certain time. It also saves each device's total amount of data so it can calculate the difference in total IO data. Finally I need to decide when the system should update the data and when to calculate the bandwidth. I discuss this in the next Section.

4.6 Calculating The Bandwidth

The final step in the system is to start the calculating the bandwidth at the appropriate time. What is an appropriate time? To answer this question I set the following goal: To avoid wasting CPU time, only calculate the bandwidth when there is new data. In other words I do not want to calculate data for processes that have no IO and processes that were idle and could not produce any IO. In this section I discuss how my system fulfills this goal.

There are processes that do not do any kind of IO operations. This means that there is never any new data and as such they should not be updated. Because I use data structures that only store data of devices that the process actually used, this issue is already solved. The system skips empty data structures.

Processes that are idle do not produce IO operations and thus have no new data. I do not want to waste CPU cycles so I do not want to calculate the bandwidth of processes that are idle. This of course makes a system that periodically checks every process not viable. A process can only use a device if the process has been

scheduled to run. While this does not necessarily mean that the process has had any IO, I believe this is a viable compromise between complexity and efficiency.

Consequently my system updates a process's IO memory bandwidth each time the process is removed from the ready queue. One of the reasons why a process is taken off the ready queue is when it is blocked waiting for an IO operation. This means that only processes that run are updated. This also has the side effect that the busiest processes have the most precise data as they are updated most frequently. As it likely that the busiest processes interfere with each other the most, it is desirable to have precise data for these processes.

With this I have a complete system. It uses the data from the hardware to measure the total IO memory bandwidth. The system combines this data with the per process and device data the OS counts. Finally it only calculate the bandwidth whenever a process is descheduled to avoid wasting CPU cycles. This concludes the design, in the next chapter I discuss the implementation of the system.

Chapter 5

Implementation

This chapter describes how I realize my design to demonstrate it's feasibility and analyze the impact it has on a real operating system. In the first section I describe the hardware and operating system I use to implement the prototype. In the following Section 5.2, I describe how I program the hardware counters and realize the data structures as proposed in my design. In Section 5.3 I outline the why I only implement the complete accounting on the network stack and limit myself to only per device accounting on the storage stack. As my design asks for driver modifications, I describe how I modify the TCP, Ethernet, and NVMe drivers in Section 5.4. In Section 5.5, I show where in the Linux fair scheduler I implement updating a process's accounting data. In the last section, I detail the limitations of the prototype and how they can be overcome.

5.1 Experimental Setup

The system I use to implement the prototype is a single socket Intel Xeon processor with 32 GiB of ram. The system is equipped with a 10-Gigabit Ethernet controller to validate my system on Network devices. Additionally the system features a NVMe SSD to test storage devices. The operating system is Ubuntu linux 16.04 with the 4.4.21 Linux kernel. In rest of this section I describe each part in detail and how they influence the prototype.

The system uses an Intel Xeon Haswell EP CPU. I choose this processor because of the extensive performance monitoring support it provides and the detailed documentation that is available. I describe the performance counters that the processor provides to measure memory and IO bandwidth in Section 3.2 and 3.3.

The network device is an is a Intel 10-Gigabit Ethernet controller. The controller uses the Intel *ixgbe* driver in Linux. In order to have per process accounting for the network device, I add additional accounting to this driver.

The storage device I use is a NVMe PCIe SSD. This card uses the Linux NVMe driver which I also modify do provide additional accounting.

The Operating system is Ubuntu Linux 16.04. The kernel version is the Ubuntu version of Linux 4.4.21, the Ubuntu package version is 4.4.0-45.66. While it has some Ubuntu specific modifications, like ZFS for Linux, these modifications are not in parts of the code that concern this thesis.

5.2 IO Accounting Structures

I describe in Section 4.5 what data structures my system needs. In the first section I show how the system programs the performance counters and it updates each process's state with the values from the counters. In the second section I describe how the system saves both the total and per device data it need for each process.

5.2.1 Performance Counters

My system relies on data from the performance counters. First I need to identify what performance counters I want to use, then how to sample them and finally how to give their data to the Kernel. Ii this section I describe how to use the cache controller's counters and how the system samples the events by distributing the the different events over all the available cache controllers.

As I describe in Section 4.2, the system needs counters that can provide data about the total IO memory bandwidth. The processor I use does not provide a direct way to measure this, so I describe in Section 4.4 how the system can calculate the bandwidth. According to the design, I need the total amount of read and write misses, the amount of read and write misses caused by the IO devices, and finally the amount of modified cache lines that the CPU writes back to memory.

As I demonstrate in Section 3.3.3, the cache controller can provide all these metrics.

The prototype tracks the total amount of misses using the `TOR_INSERTS` event. I show in Section 3.3.3 that this event can filter the counter by the request's origin, but only has one filter register. This limits the system to one filter per cache controller. I decide to only use two different filters, to limit the errors from sampling. The prototype uses the `PCIRdCur` opcode filter to accurately track reads from the cache to a device. For writes from a device to the cache I use the `ItoM` opcode filter because `PCIItoM` always produces 0. Like the documentation suggests, the prototype sets the Thread-ID filter to 0x3E [8, Ch. 2.3.5]. While the documentation does not state what this Thread-ID is, setting it changes the amount of events counted. To measure the total amount of cache misses, the prototype uses `TOR_INSERTS` without the filter to count all misses.

To count the total amount of cache lines that the CPU writes back to main memory, the system uses the cache controller's `LLC_VICTIMS` event and sets it to only count lines that are in a modified state.

The processor only provides four counters per cache controller and only one filter register. Consequently the prototype needs to sample the data. I do not want to constantly change the counters programming so they can run without the OS interfering. In order to sample, the prototype uses every second cache controller to either count read or write misses from PCI. In the remaining three counters the system counts the total read and write misses, and cache line write backs.

The kernel sets up the processor's performance counters during the kernel's initialization, after the scheduler has been initialized, shortly before the kernel starts loading modules.

The processor can be programmed to throw an interrupt when any counter overflows so that they can be reset by the software. To correctly handle overflows the OS needs to save additional data so it can recreate the previous state. However, I assume that overflows are rarely an issue as the counters can hold up to $2^{48} - 1$ events. For example with around 10 million events per second, it takes a bit less than a year until an overflow occurs. To avoid the overhead of the additional data structures and systems to deal with the very unlikely case of a overflow, the prototype explicitly does not implement handling overflows. The processor however does allow the OS to catch and process overflows, so this can be implemented in case overflows become an issue.

Because the prototype ignores overflows and does not sample by time, the kernel does not need any kind of global state for the performance counters. It can always read the performance counters when needed.

To track the total amount of memory bandwidth used by the devices, the prototype provides helper functions that read the counters and then calculate the amount of bytes the devices consume of the memory bandwidth. These functions implement the two terms of Formula 4.1 for the total memory bandwidth without the time component.

As I discuss in Section 4.5, the processes need to save the state of the performance counters so the OS can accurately calculate the difference. To realize this functionality, I modify the already present `task_io_accounting` structure in Linux to hold two integers to save the total amount of reads and writes to main memory.

With these modifications and new functions I implement the performance counters according to my design. In the next section, I describe how I implement the data structures that provide the data from the software's accounting.

5.2.2 Per Device Accounting

As I demonstrate in Section 4.4, my design requires the total sum of bytes sent and received from each device. The system also needs the amount of data consumed by each device for each process. And finally the system needs to save the time the process was accessed the last time.

To calculate the time that has passed, the system uses a variable in each process's `task_io_accounting` structure to keep track of the last time it was checked.

The OS does not know in advance how many devices are used or are in the system. The scheduler should also be able to quickly look up a specific device's bandwidth. Consequently the system uses a hashmap as it fulfills both requirements. In this hashmap the prototype saves for each device the amount of data sent and received so far. To index the hashmap, the system uses the device's pointer address in the device tree. As the system needs a global counter and per process counters, the prototype uses a per process and a globally accessible hashmap to track each device's current activity.

The data that is saved in the global and per process hashmaps is very similar. As they both save similar data, the prototype uses the same data structure for both. The data structure in the hashmap tracks the data a device has sent and received globally or for the process. The data structure is initialized with 0 and then whenever the device has activity, the driver adds the amount of data to the data structure.

Processes need to track the state the devices were in on the last check. This means that they need an additional hashmap to save each device's last state. For ease of use and debugging the prototype also uses a hashmap to save the last bandwidth. This is not necessary as the bandwidth can be calculated at any time from the data that is already present.

To write and add new data into the global hashmap, drivers call a helper function `membw_io_accounting_add`. The driver sends a pointer to the device it is handling and the bytes that have been read or written. The function uses the address of the device in the device tree as the key of the hashmap. If the device is not found in the hashmap, the function adds the device and initializes the numbers to what the driver sends.

A driver can update a process's hashmap by calling `task_membw_io_accounting_add`. This function is almost identical to the one of the global hashmap, except that it takes a pointer to the process's `task_io_accounting` structure that holds the process's hashmaps.

The per process hashmaps need to be cleaned up when the process exits. This happens in the `release_task` [1, kernel/exit.c] function.

With the hashmaps to track the data and the functions to update the data in

place the OS can access all the data that the system needs to calculate the IO memory bandwidth. In the following sections I describe how I modify the drivers so that they can update these data structures.

5.3 Connecting Stack Layers

With the data structures in place, I need to modify the drivers to update them. The drivers need information about the process that has caused the IO. As I describe in Section 4.3.4 there are differences in how information can flow through the network or storage stack. In this section I describe what I implement of the design.

The prototype can only track per process data in the network stack. In the block IO stack the prototype only tracks the total amount of data the device produces. This is because of the complex nature of the block device stack and the many paths a request can go. The patch from the split level IO paper is incompatible with the prototype's kernel and only tracks writes. Porting the patch to a newer kernel version and expanding it to also track reads is outside of the scope of this thesis.

The prototype does however implement per process accounting in the network stack. The network stack only uses only one data structure that is passed through the entire stack. The prototype uses the `sk_buff` data structure to move additional data that is used for accounting from the driver up to the transport layer and vice versa.

The prototype uses an new variable in `sk_buff` so that drivers can inform the other layers about the overhead of the control messages the devices needs.

To identify a socket's owner, the prototype uses modified `socket` and `accept` syscalls to save the `current` process's pid in the socket. The code uses the same data structure that is used when fcntl calls F_SETOWN. This means that the prototype can only accurately track processes that do not pass their sockets to other processes.

Since the split level IO system is not compatible with the prototype, the prototype cannot use the additional accounting. However, the network stack links the data that the higher layers provide with the data from the lower layers. With this I can verify the prototype's correctness.

5.4 Driver Modifications

After adding the necessary structures to save the data in each process and linking the different layers of the IO stack together, each driver needs to be modified to update the amount of data it consumes.

In this section, I detail the modifications needed to track the global and per process amount of data each device consumes.

5.4.1 Network

The bandwidth estimation needs data for each device's total activity and each process's activity. I show how this can be done with the example of the the network stack. The prototype uses modified TCPv4 and ixgbe drivers that update the global and per process accounting.

The network stack needs to be modified in the transport protocol layer and the link layer driver. This is due to the fact for outgoing packets, the size of the headers is only known when they are transmitted on the link layer and for incoming packets, the network stack can only identify the the destination socket in the transport layer.

To move data through the network stack, the Linux kernel uses the `sk_buff` structure. It already has a field for packet's data, and header lengths. The packet's data length includes the transport layer header. The prototype provides an additional field that saves the length of the device's command descriptors.

For incoming traffic, the ixgbe Ethernet driver adds the size of the command descriptors that it sends to the device to start DMA transfers to the `sk_buff`.

I only modify the TCPv4 handler to count incoming traffic. The transport layer identifies the socket the data is destined for and then copies the data to the socket. The `sk_buff` saves the device the packet has arrived on and the header sizes of the lower layers. The prototype uses the socket's owner to identify the process using the socket. With that information the driver calls `task_membw_io_accounting_add` on the process's IO accounting structure with the data from the `sk_buff`. The driver also calls `membw_io_accounting_add` on the device saved in the `sk_buff` and the data length information to update the global counter.

To track outgoing traffic, the prototype uses a modified version of the ixgbe Ethernet driver. The network stack saves the socket a request originated from in the `sk_buff`. Like the TCP driver, the Ethernet driver uses the data from the `sk_buff`. All it needs to add is the device and length of the descriptors. Analogous to the TCP handler, using the socket and data length, the driver updates the per process and global counters for outgoing traffic.

With this, the prototype provides per process and global tracking of the Ethernet controllers incoming and outgoing traffic.

5.4.2 Block Devices

For block devices, the prototype only implements tracking the total amount of data the device produces. The accounting is implemented the low level PCIe interface

of the Linux NVMe driver.

As I discuss in Section 5.3, the split level IO system does not work for the prototype's kernel version. This means the NVMe driver cannot reliably determine the originating process. Once there is a way to connect the higher levels of the storage stack with the driver, it can use the same method as I describe here to update a process's accounting.

The device driver determines the size of each request. The size of the request is always aligned with the sector size of the device. This means that the NVMe driver always counts entire sectors, even if higher layers only read part of the block. In other words, the driver counts the amount of data that the device needs to transfer, no matter what the original request was.

The device driver can identify if a request is a read or a write and add the length of the request to the global counter.

By hooking into the low level PCIe driver of NVMe, the prototype provides the device's total data consumed.

5.5 Scheduler Integration

With a system that can track the necessary data, the performance counter numbers, the global device data counters, and the per process counters, the scheduler can calculate the processes memory bandwidth. As I discuss in Section 4.6 the scheduler updates a process's consumed bandwidth every time the process is dequeued.

As per the design, the prototype measures the time difference with the wall clock. The "fair" Linux scheduler keeps track of the wall clock for each queue in a `clock` field. This `clock` is updated periodically from data of the queue's CPU clock. This clock has a nanosecond resolution and a "bounded drift between cpus" [1, kernel/sched/clock.c]. This drift might introduce some errors when a process is scheduled on another CPU, but the error only persist until the process is scheduled on the same CPU again.

To calculate the amount of data a process has consumed since the last check, the scheduler uses the data from the last check and the current measurements. It uses the global and per process data to calculate the bandwidth according to Formula 4.2. The scheduler calculates the share the process's data consumption has on the total data consumption since the last check and multiplies that share with the total IO memory bandwidth.

Finally the scheduler saves each device's IO memory bandwidth in the process's data structure.

With the scheduler updating the processes, the prototype has all the parts that the design calls for. In the last section I detail the limitations the prototype has.

5.6 Limitations

In this section I illustrate the prototype's limitations and how it can be extended.

I do not port the split IO scheduling system to my kernel version. Hence the prototype cannot identify a block IO's original process. So the prototype only provides the global accounting numbers for storage devices. However I can verify the principle to estimate IO memory bandwidth using the network stack.

I only save a socket's owner when the socket is created. The implementation only works correctly for processes' that do not fork. Processes that call accept and then fork to a new process that use the new socket have all the data attributed to the process that called accept, which is the parent process. However, as I demonstrate in Section 3.6.4 solving this problem is a complex task but does not invalidate the basic principle. The prototype can easily be extended with a system that more accurately tracks what process read which data from a socket.

The prototype is limited to NVMe storage devices. For network devices, it can only track TCPv4 and network adapters that use the *ixgbe* driver. However as my prototype provides functions that are general enough to support any device, additional drivers can easily implement the needed accounting.

Despite these limitations, the implementation covers sufficient system usage to show the general principle of the system and to evaluate it.

Chapter 6

Evaluation

In this chapter I validate my design by evaluating the prototype's accuracy and performance. I already demonstrate the accuracy of the counters in Section 3.3, so in this chapter I limit myself to evaluating the accuracy of the system itself and the performance impact the modifications have on the OS. I describe the experimental setup in Section 6.1 and establish the prototype's correctness in Section 6.2. In Section 6.3 and 6.4 I demonstrate that while the measured dequeuing latency in the scheduler increases significantly, this does not have an impact on most benchmarks' performance. In the worst case there is a 16% performance loss.

6.1 Experimental Setup

The system I use to evaluate the prototype is a single socket Intel Xeon processor with 32 GiB of RAM. The system has a 10-Gigabit Ethernet controller to evaluate network devices and an NVMe SSD to test storage devices. The operating system is Ubuntu Linux 16.04 with the 4.4.21 Ubuntu Linux kernel.

The system uses the Intel Xeon E5-2618L v3 CPU. This is a 8 core CPU with 20 MB of last level cache. It uses DDR4 memory and has four memory channels. For the memory the system uses four 8GiB DDR4 1866 DIMMs, one for each memory channel to maximize the memory bandwidth.

The network device is an is a Intel 10-Gigabit X540-AT2 Ethernet controller. The controller uses the Intel *ixgbe* driver in Linux. As I only concern myself with kernel space drivers, I use the controller through the Linux socket interface to ensure all processing happens in the kernel.

As a reference I use an unaltered Ubuntu Linux 16.04 kernel of Linux 4.4.21, Ubuntu packet version 4.4.0-45.66. To measure the change in performance, I compile the kernel once with only the added per process accounting, and once with both the changes to the scheduler and accounting.

	Reported (MiB/s)	Measured (MiB/s)
Read	870	879
Write	678	675

Table 6.1: Average IO bandwidth reported by the benchmark and measured by the system. This demonstrates how the IO bandwidth is close to the expected values

	Read (MiB/s)	Write (MiB/s)
IO bandwidth	879	0.78
IO memory bandwidth	4.38	0.27

(a) Write to network

	Read (MiB/s)	Write (MiB/s)
IO bandwidth	3.82	675
IO memory bandwidth	0.18	20.3

(b) Read from network

Table 6.2: Idle system: Average estimated IO memory bandwidth compared with measured IO bandwidth. The IO memory bandwidth never exceeds the IO bandwidth and is thus in the expected range.

6.2 Memory Bandwidth Estimation

To demonstrate the correctness of my system's data, I periodically read the estimated bandwidth the prototype reports. Since there is no system that allows to directly measure the IO memory bandwidth, there is no reference measurements that I can compare the prototype's results against. Consequently, I only demonstrate that the accounting the prototype uses is correct and that the estimated memory bandwidth is plausible. To demonstrate the prototype's accuracy I validate the Performance counter's accuracy in Section 3.3. With the information from both evaluations I conclude that the prototype's reported estimation provides an accurate view on a process's IO memory bandwidth.

To demonstrate that the accounting and the time measurements are correct, I add an additional calculation to the memory bandwidth estimator to measure the IO bandwidth of a process's device. The prototype reports data that in is incoming from the network as a write as the device writes to the memory and outgoing data as a read because the device reads data from memory. Table 6.1 shows the IO bandwidth for a read and a write benchmark. Both use the reported bandwidth from the benchmark as a reference. In both cases the reported and the measured bandwidth are close together, illustrating that the accounting in the system works as expected.

	Read (MiB/s)	Write (MiB/s)
IO bandwidth	876	51
IO memory bandwidth	526	0.8

(a) Write to network

	Read (MiB/s)	Write (MiB/s)
IO bandwidth	4.98	677
IO memory bandwidth	0.13	69.6

(b) Read from network

Table 6.3: Loaded system: Average estimated IO memory bandwidth compared with measured IO bandwidth. Compared to the idle system the IO memory bandwidth increases but stays below the IO bandwidth, demonstrating that the system behaves as expected.

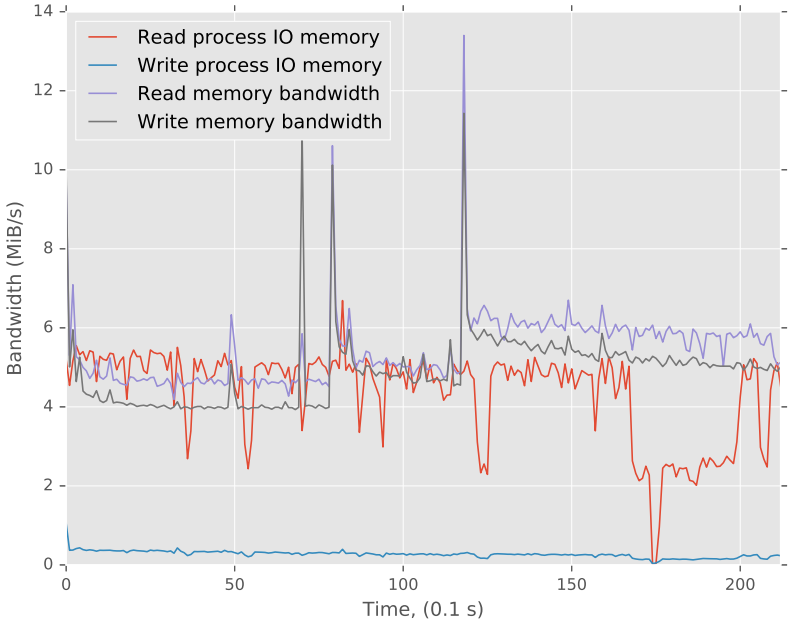


Figure 6.1: Total memory bandwidth compared to process IO memory bandwidth. The reported IO memory bandwidth is close to the total memory bandwidth. There is a slight difference in the data as it had to be gathered in separate runs. This demonstrates that the system provides plausible data.

I expect the IO memory bandwidth to be lower or close to the measured IO bandwidth. In Table 6.2 I note the average IO bandwidth and IO memory bandwidth for a read and a write benchmark on an idle system. In both cases the IO memory bandwidth is far lower than the IO bandwidth. This behaves as expected. With the system idle and the payloads of the benchmarks being smaller than the cache, most of the data can be provided by the caches without the main memory. Consequently like I expect the reported IO memory bandwidth is fairly low.

I repeat the same experiment on a system that is under load by a benchmark that uses 15 of the 20 MiB of cache. I report the results in Table 6.3. Because of the higher load that is on the cache, I expect a higher number of cache misses caused by IO devices. Hence the IO memory bandwidth should be higher than on an idle system but remain below the total IO bandwidth. The data reflects this, compared to the benchmarks on the idle system, the memory IO memory bandwidth increases, but stays below the total IO bandwidth. Again the system reports data that behaves as expected.

Finally I compare the IO memory bandwidth to the total memory bandwidth. The IO memory bandwidth should be close or below the total memory bandwidth. I need to run the benchmark twice as the system to read out the memory bandwidth overwrites the performance counters of the memory bandwidth estimation. The benchmark runs on an idle system so that most of the memory bandwidth is caused by the device. I plot the total memory bandwidth and the IO memory bandwidth in Figure 6.1. As the figure shows, the IO memory bandwidth is close or lower than the total memory bandwidth. There is a slight difference in the first 7 seconds, I attribute this to slight differences between the two runs. This demonstrates that the estimated values are within the estimated range.

In all of these tests the estimated IO bandwidth and IO memory bandwidth are within the expected range. The IO bandwidth is close to the reported bandwidth and the IO memory bandwidth never exceeds the IO bandwidth. The IO memory bandwidth also increases on a busy system as expected by the higher potential amount of cache misses. Finally the IO memory bandwidth does not exceed the memory bandwidth, showing that in every case the data that the IO memory bandwidth estimation delivers is plausible. This combined with the knowledge that the performance counters produce accurate numbers, I believe that the data from the system provides an accurate estimation of the real IO memory bandwidth.

A scheduler can use this data as additional information to help scheduling decisions. While this data can improved a system's performance, it is not useful if estimating the bandwidth slows down the system. In the remaining two sections I demonstrate that while reading the performance counters is relatively slow, it does not have a significant impact on the system's performance.

Function	Average Time (μs)
Update process or device accounting	0.12
Get total device IO as counted by OS	0.25
Get read IO memory bandwidth from performance counters	2.25
Get write IO memory bandwidth from performance counters	12.33

Table 6.4: Helper function latencies reported by ftrace. The first function is used by the drivers on every IO operation, the remaining three are used by the scheduler on every dequeue.

6.3 Function Latencies

To evaluate the prototype's performance penalty I first measure the function latencies. While I demonstrate in this section that the latency for reading performance counters is relatively high, I show in the next section that the impact on the whole system is in many cases negligible.

To get an idea about the impact of the added functions on performance, I use the built in Linux tracer ftrace [1, Documentation/trace/ftrace.txt]. I measure my helper functions' and the modified functions' latencies.

Table 6.4 shows the average latency of each helper function. While the drivers use use two different functions to update the global and a process's accounting on every IO operation, these functions are almost exactly the same. Hence I group them together as one in the table. The latency is $0.12\mu s$. Looking up the correct slot in the hashmap to update and adding the new values is a very fast operation.

The other functions in Table 6.4 are used by the scheduler to calculate the bandwidth of each process. The scheduler uses a helper function to get the total amount of data sent and received by all the devices. This function looks up every device in the global hashmap and sums up their values. With a latency of $0.25\mu s$ it is also a fast operation.

Both of these functions should not have much of an impact on the system's performance.

The two functions that need the performance counter's data, however have a comparatively high latency with $2.25\mu s$ and $12.33\mu s$ respectively. Calculating the read IO memory bandwidth requires the function to read one performance counter on every second core to determine the read misses. To calculate the write IO memory bandwidth, the function needs to read three performance counters from every processor. So the write bandwidth function needs to read six times more performance counters than the read bandwidth function. Both functions need to perform a few calculations to produce the data, but they should not contribute much to the overall latency. With the latency of `membw_get_wr` being around six times that of `membw_get_rd`, I believe that the largest portion of the latency comes from read-

ing the performance counters. From these measurements, I assume that reading each performance counter takes around 0.5μs .

To check the impact these new functions have on the modified drivers and scheduler, I record the latencies as they are reported by ftrace. Figure 6.2 shows the results.

The Ethernet driver updates the process and global accounting in it's transmit function `ixgbe_xmit_frame`. The data in Figure 6.2b shows is no noticeable difference between the different kernel versions. With or without accounting the times are about equal. This means that additional accounting should not impact the system's performance.

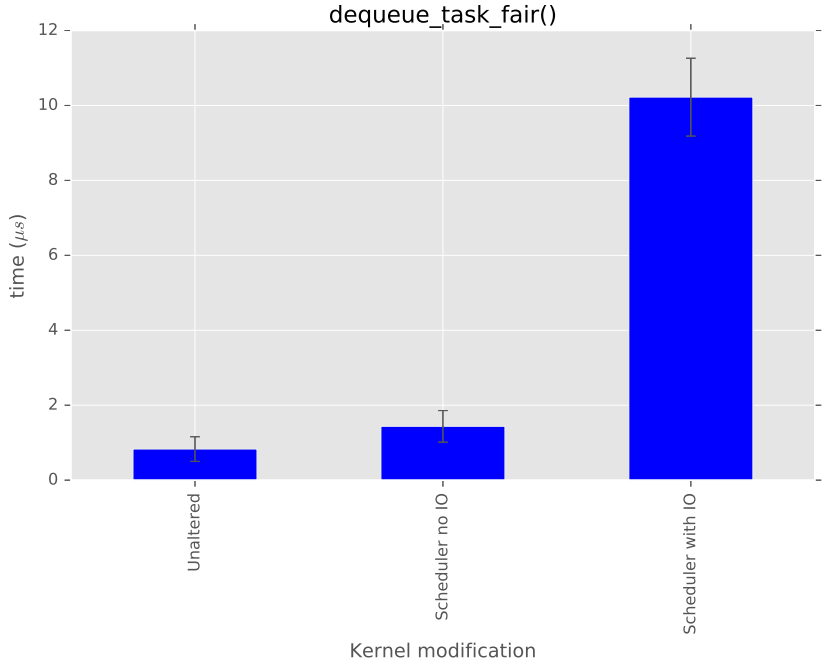
The scheduler uses needs to get the total IO accounting from the OS and get the bandwidth numbers from the performance counters to calculate the IO memory bandwidth a process consumes. If there is no device in the hashmap, the process has not used any IO devices that are tracked. In this case most of the code is skipped. These processes are grouped under "Scheduler no IO" in Figure 6.2a. The average latency in the unmodified kernel is 0.83μs and in the modified kernel 1.43 μs . There is a 0.6μs difference. While this is a significant change in latency, processes generally run for several ms before they are descheduled again. In that context, 0.6μs are a negligible quantity. So the modifications should not have much impact on the performance of processes that do not use IO.

However for processes with IO, due to the high latency reading out performance counters has, the latency to deschedule a process becomes much higher. It goes from 0.83μs up to 10.22μs . I am uncertain why ftrace measures a longer latency for reading the necessary data than for the descheduling itself. However this shows that there is a significant difference in latency, almost 10μs. In the context of scheduled processes, if a process runs for $1ms$, the change in latency makes up 1% of the total time.

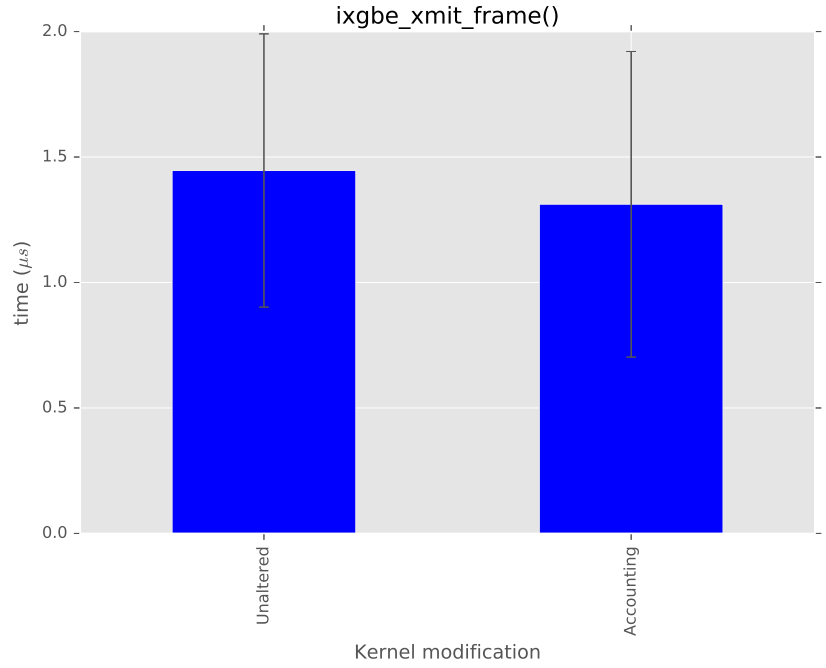
These results show that looking up or adding to the OS's data structures are a fast operation. Neither of the functions that operate on the data structures have a high latency. They have no noticeable latency impact on the functions that use them. Operations on the performance counters however have a significant latency, and as the change in descheduling latency shows, have a significant impact on functions that call them. I demonstrate in the next section that while there is a significant change in latency, this does not always impact the system's performance negatively.

6.4 Benchmarks

In the previous section I show that in some cases, the prototype can increase a functions latency by more than an order of magnitude. To evaluate the impact



(a) Scheduler. Compares the latency of the same function to dequeue a process. Once on an unaltered kernel and twice on a kernel that implements updating process's IO memory bandwidth. In one case the process produces no IO and in the other it does.



(b) 10Gbit Ethernet driver. Compares the latency of the Ethernet driver's transmit function in an unaltered kernel and in a kernel with additional accounting in the transmit function.

Figure 6.2: Latencies of modified functions.

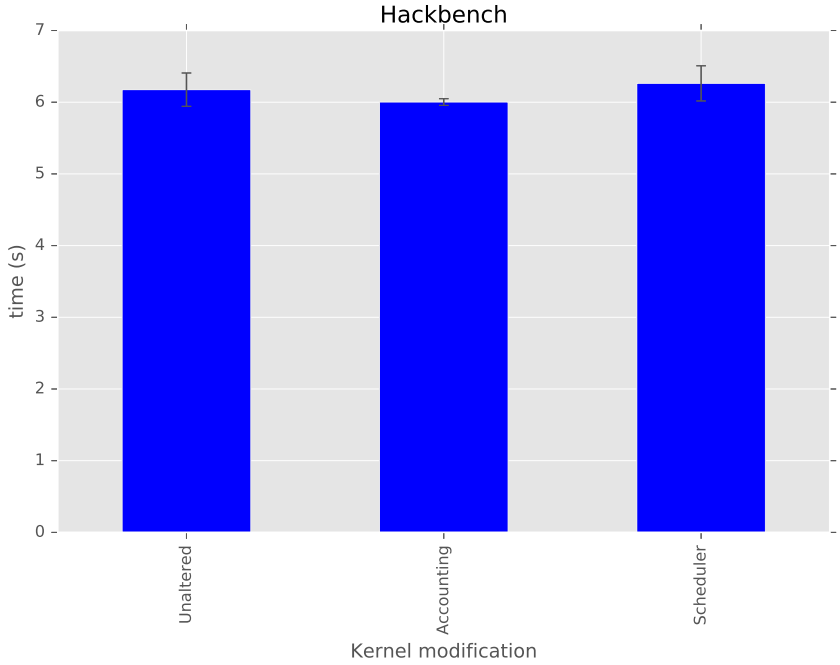


Figure 6.3: Hackbench results. Stresses the scheduler. Compares an unaltered kernel, one with the additional per process IO accounting, and one with the scheduler changes.

on the system's performance I use Several benchmark to stress the scheduler, and benchmarks to stress the CPU and network. In the first section I use synthetic benchmarks that stress the scheduler with a high amount of context switches and in the second I use application benchmarks that stress the network and CPU. I demonstrate that in many cases the higher latency in the dequeuing function has no impact, only in the worst case it has a 16% performance impact.

6.4.1 Synthetic Benchmarks

In this section I evaluate the impact the higher latency that I measure in Section 6.3 has on the process switching performance. I demonstrate that for processes without IO there is no significant difference, and for processes that would trigger the slow path, that the context switch latency increases by around twice the added dequeuing latency.

To stress the scheduler I use hackbench, a scheduler benchmark[1]. It spawns different processes that use sockets to send data to each other. I set it to send 10000

[1]The manpage of hackbench:http://man.cx/hackbench(8)

messages. As this benchmark spawns many processes that need to be scheduled to communicate, an efficiency loss in the scheduler manifests in longer runtime. The results are in 6.3. There is no significant difference in the runtime between the different kernel versions. As hackbench does not use an IO device, it skips most of the new descheduling code. I also use the lat_ctx benchmark that measures the context switch latency. It behaves like hackbench and shows no difference between versions. This shows that the measured difference in the descheduling latency has no impact on processes that do not use IO devices that are tracked.

In the previous test the scheduler skips most of the code as neither benchmark uses IO devices. There is no synthetic benchmark to measure the context switch latency that also uses an IO device. To simulate the higher dequeuing latency processes with IO have, I modify the Linux kernel to always have a 10 μs delay in the descheduling function as I measure in the previous section. I use the lat_ ctx benchmark again to measure the latency. In this case the measured latency increases from around 9 μs to 31 μs. So I can expect a higher process switch latency for processes that use IO. In the following I demonstrate that this only has an impact on performance in a worst case scenario where the system is under full load and many short requests are serviced.

6.4.2 Application Benchmarks

After evaluating the impact the higher latency in the dequeue function has on the process switching performance with synthetic benchmarks, I use application benchmarks to evaluate the impact the prototype's modifications have on the system. I demonstrate that for most scenarios there is no statistically significant change in performance and that in the worst case there is a 16% performance penalty.

To see how the network driver's performance changes with the added accounting and what the impact of the higher dequeue latency is, I use Iperf [2]. It is a network benchmark to measure the network bandwidth. Hence it both stresses the network stack and scheduler as the benchmark needs to wait for the data to be sent. I use it in the default settings, using large packets and TCP. I record the CPU usage with the `time` utility as the benchmark does not use all of the processor's resources to saturate the network. Any added overhead in the driver should show up as a higher CPU usage. Figure 6.4 shows the results. There is no significant change in the average CPU usage. There's a slightly higher variance in the recorded CPU usage, but that is likely due to the coarse grained data that that `time` provides. The driver calls the comparatively fast functions to update the total and the process's data usage. So I do not expect any significant overhead from

[2]The Iperf benchmark: https://iperf.fr/

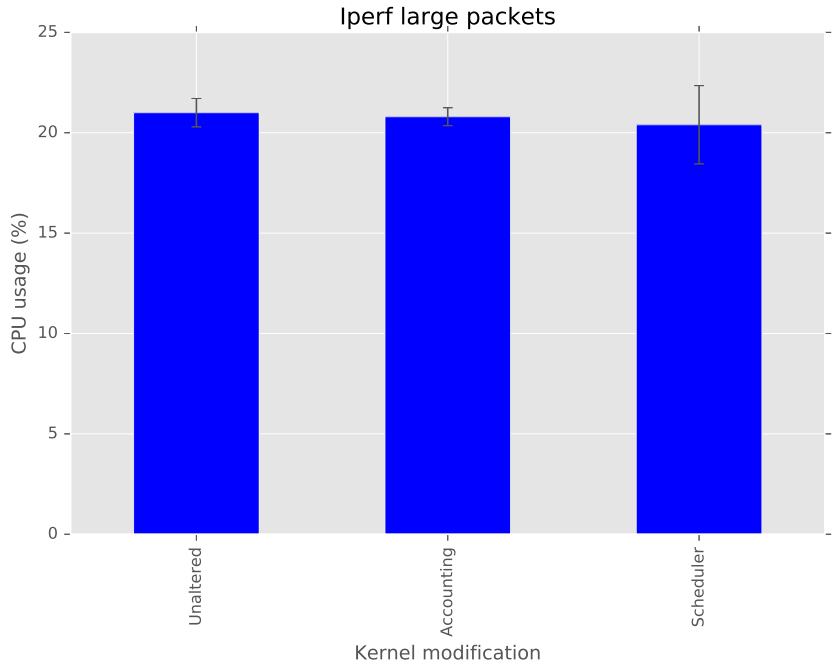


Figure 6.4: Iperf, with large packets. As the benchmark saturates the network this reports the CPU usage on an unaltered kernel, one with additional per process accounting, and one with the accounting and scheduler modifications.

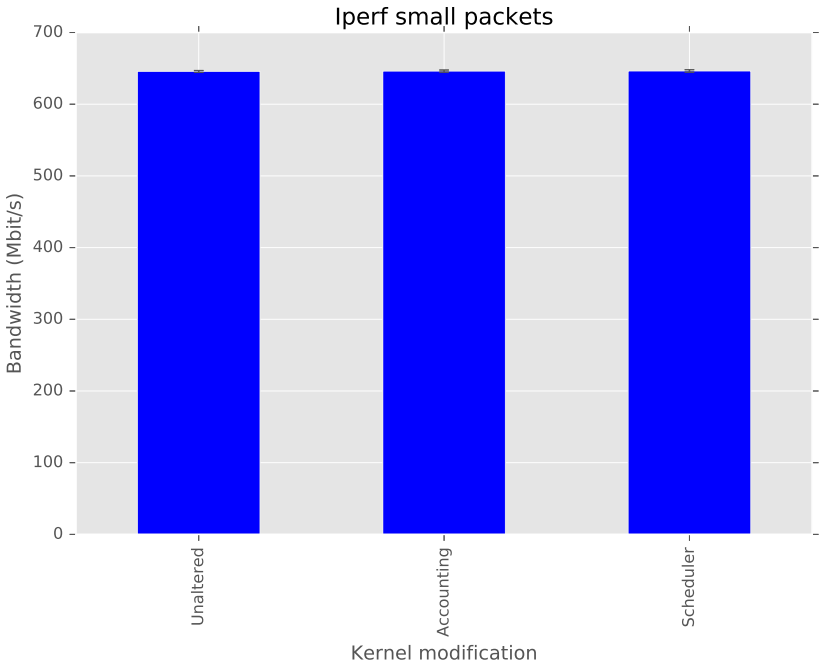


Figure 6.5: Iperf, with small packets. As the CPU is constantly at 100% utilization, this reports the Mbit/s bandwidth the benchmark achieves. The results are reported for a kernel without modifications, with only the accounting in the driver, and with the changes to the scheduler and accounting.

that. On the other hand, as this benchmark does use a IO device that is tracked, it also uses the slow path through the scheduler. I expect some sort of change in the CPU usage, but the data shows that on an otherwise idle system the higher latency of the scheduler has no significant impact on processes that send large buffers to an IO device.

To further explore how the network stack and the scheduler behave, I use Iperf with small, 100 byte, packets. This cannot saturate the network and the CPU always runs at 100%. Hence I record the bandwidth Iperf reports. As Iperf sends as many small packets as the CPU allows, any decrease in efficiency of the network stack and scheduler show as a reduced bandwidth. Figure 6.5 shows the results. There is no significant drop in performance. This shows that neither the changes to the network stack nor the scheduler have a significant impact on the benchmark's performance. So even with processes that are blocked often for IO, the change in scheduler latency has no impact. In both cases I assume that the network stack can compensate for the higher process switching latency by buffering more data in the socket.

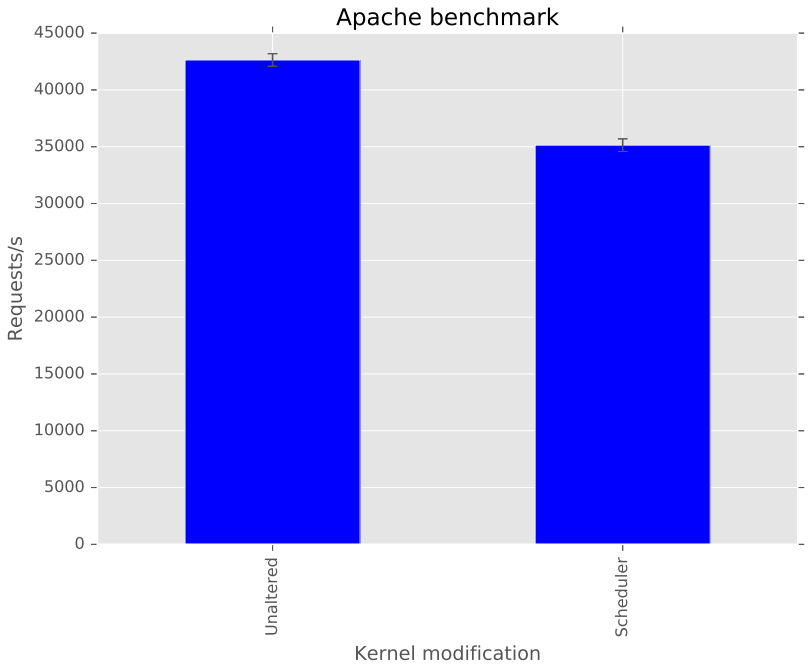


Figure 6.6: Apache benchmark. Reports the requests per second to a Apache 2.4 server on the test system. The server is run on a system with an unaltered kernel and on a system with the kernel implementing the scheduler and accounting modifications.

So far all the benchmarks were run on idle systems. To evaluate the performance impact the prototype has on a system under load, I use the Apache benchmark[3]. This is a HTTP benchmark that tries to send as many requests per second as possible. I use Apache 2.4 on the test system with the default configuration and website. The benchmark stresses both the network and the scheduler as Apache spawns many threads to process the requests. This represents the worst case scenario where a process produces a high amount of small requests that are each handled by one thread and thus completely saturates the CPU. The network stack cannot compensate for the slower process switching as each request is delivered to another process. Consequently any delay in process switching also delays the request's answer. Figure 6.6 shows the requests per second that the benchmark reports for an unaltered kernel and the kernel with scheduler modifications. There is around a 16% drop in performance. This demonstrates that for workloads that spawn many processes and threads that each process a request, the performance is worse with a higher dequeuing latency.

These results show that the modifications have in many cases no significant impact on performance. The added latency from updating the accounting data has no impact on the networks performance. While there is a significant change in the latency to deschedule a process, this added time often does not impact the system's performance. Only for the worst case scenario there is a 16% drop in performance because of the scheduler latency.

With these results, I am confident that a system like this can be implemented in an OS and provide additional data to the scheduler without having a crippling performance penalty.

[3]https://httpd.apache.org/docs/2.4/programs/ab.html

Chapter 7

Conclusion

Memory bandwidth is an important shared resource that the operating system needs to fairly distribute among processes. In this thesis I demonstrate that schedulers that are memory bandwidth aware cannot rely only on data about the memory bandwidth a process uses directly. The scheduler also needs to take the memory bandwidth into account that the processes consume indirectly via IO devices. This illustrates that the IO memory bandwidth is an important resource that the scheduler needs to be aware of when making decisions. To furnish the scheduler with this data I propose a system that provides detailed information about how much IO memory bandwidth each device used by process consume.

I establish that to estimate the IO memory bandwidth, the OS requires data from both the hardware and the software. I show how the hardware, while it produces accurate aggregate bandwidth numbers, cannot provide the operating system with per process or per device data. I demonstrate how the OS can, through its drivers, determine the activity of each device and potentially provide per process device accounting. Consequently, the OS needs to be modified to track the hardware's counters and provide additional accounting per process and per device.

I demonstrate how the network and the storage stack, while being vastly different, can provide the necessary data, but lack the ability to link this data to a process. Hence, in the mainline Linux kernel, they cannot provide per process accounting and need to be modified to provide this data. I demonstrate that the storage stack, due to how the devices function, can potentially always identify the calling process. By following the implementation of storage cgroups the higher layers of the storage stack can be linked to the lower layers. For the network, stack I determine that the layers can be linked together relatively easy with the shared data structure that is handed through the stack. However, for network devices, determining the process that caused a read is hard as the access to memory happens before the process can be identified. I propose several solutions to this problem.

Based on the information about the performance counters and the OS, I design

a system to provide the scheduler information about each process's IO memory bandwidth. By taking advantage of the extensive data performance counters provide and the accounting data from the OS, the system calculates each process's device's IO memory bandwidth. With this system the scheduler has additional information it can use to base scheduling decisions on.

My prototype's evaluation illustrates that this design is a viable approach. The system provides accurate data and in most cases does not have a performance penalty. Even in the worst case, the performance penalty is moderate.

As such I believe that the system I present in this thesis can be adopted into schedulers that use memory bandwidth as a metric to base decisions on. With this data the scheduler has additional data about the processes' memory bandwidth consumption. Combined with methods that measure the direct memory bandwidth, the scheduler has a complete picture of each process's memory bandwidth usage. With this additional information, the scheduler can make decisions based on complete and accurate data.

For future works, as most of the performance penalty is due to the performance counter's latency, I believe that systems that reduce this latency, like periodically caching the current state, can improve the system's performance. Moreover there is still the complex issue of correctly accounting for each process's network reads. It is worth evaluating if a system where every byte is tracked is necessary or if a simpler system that assume one owner per socket is sufficient.

Bibliography

[1] Linux 4.4.21. `https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.4.21.tar.xz`.

[2] Advanced Micro Devices Inc. AMD I/O Virtualization Technology (IOMMU) Specification. (February), 2015.

[3] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling algorithms with bus bandwidth considerations for SMPs. *Proceedings of the International Conference on Parallel Processing*, 2003-Janua:547–554, 2003.

[4] C D Antonopoulos, D S Nikolopoulos, and T S Papatheodorou. Realistic workload scheduling policies for taming the memory bandwidth bottleneck of SMPs. *High Performance Computing - Hipc 2004*, 3296:286–296, 2004.

[5] ARM Limited. ARM System Memory Management Unit Architecture Specification. (15 July), 2015.

[6] David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. Bandwidth Bandit: Quantitative characterization of memory contention. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013*, 2013.

[7] Intel Corporation. Intel ® Virtualization Technology for Directed I / O. (October), 2014.

[8] Intel Corporation. Intel ® Xeon ® Processor E5 v3 and E7 v3 Product Families Uncore Performance Monitoring Reference Manual. (February), 2014.

[9] Intel Corporation. [3]Intel ® 64 and IA-32 Architectures Software Developer ' s Manual. *System*, (April), 2016.

[10] Intel Corporation. Intel ® Virtualization Technology for Directed I / O. (June), 2016.

[11] Florian Larysch. Fine-Grained Estimation of Memory Bandwidth Utilization. (September 2015), 2016.

[12] John D McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia. http://www.cs.virginia.edu/stream/.

[13] Thomas Moscibroda, Onur Mutlu, Thomas Moscibroda, and Onur Mutlu. Memory Performance Attacks : Denial of Memory Service in Multi-Core Systems Memory Performance Attacks : Denial of Memory Service in Multi-Core Systems. pages 257–274, 2007.

[14] Adit Ranadive. VIRTUALIZED RESOURCE MANAGEMENT IN HIGH PERFORMANCE FABRIC CLUSTERS VIRTUALIZED RESOURCE MANAGEMENT IN HIGH. (December), 2015.

[15] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. IBMon : Monitoring VMM-Bypass Capable InfiniBand Devices using Memory Introspection. *Performance Computing*, (March):25–32, 2009.

[16] Adit Ranadive, Ada Gavrilovska, and Karsten Schwan. FaReS: Fair resource scheduling for VMM-bypass infiniband devices. *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 418–427, 2010.

[17] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model. *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pages 62–75, 2015. http://dl.acm.org/citation.cfm?id=2830772.2830803.

[18] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, page 237, 2010. http://dl.acm.org/citation.cfm?id=1854273.1854306.

[19] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-kiswany, Rini T Kaushik, Andrea C Arpaci-dusseau, and Remzi H Arpaci-dusseau. Split-Level I / O Scheduling. 2015.