UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

Analysis and mitigation of writeback cache lock-ups in Linux

Degree Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Alba Mendez Orero

In partial fulfillment
of the requirements for the degree in
TELEMATICS ENGINEERING

Advisor: Juan Jose Costa Prats
Barcelona, June 2020

Abstract

Linux caches disk I/O for performance: writes complete immediately from userspace perspective, and are committed to storage later. But in the presence of heavy writers, this buffering can easily hurt system responsiveness if not handled correctly.

This is well known and has received substantial work, but remains a problem today. In this thesis, we first research the relevant kernel subsystems & technologies, measure and understand what's happening, then design a PoC to mitigate the adverse effects.

We discover cache flushes to be the issue, because inodes are locked until written to disk. Today's large caches mean inodes from innocent processes can stay locked for long amounts of time. This often *replaces* the intended throttling.

We designed a daemon that detects heavy writers and lowers their I/O priority, allowing inodes from other processes to be written (and unlocked) first. It successfully reduces long pauses and improves responsiveness.

Resum

Linux cacheja les operacions I/O de disc: les escriptures acaben immediatament des de la perspective d'userspace, i més endavant s'escriuen realment al disc. Però en la presència de processos que escriuen sense límit, aquest buffering pot danyar fàcilment el temps de resposta del sistema si no es gestiona de forma justa.

Es tracta d'un problema conegut i que s'ha treballat bastant, però que segueix estant present a dia d'avui. En aquest projecte s'investiguen els subsistemes i tecnologies rellevants del kernel, es mesura i entén el que està passant, i llavors es dissenya un PoC per mitigar els efectes adversos.

Es descobreix que el problema sembla estar en els buidats de la cache, ja que els inodes és bloquegen fins que s'escriuen al disc. Avui dia amb caches tan grans, els inodes de processos innocents poden mantenir-se bloquejats durant un temps llarg mentre esperen ser escrits. Això fins i tot *reemplaça* el throttling normal.

S'ha dissenyat un daemon que detecta processos ofensius i els redueix la prioritat I/O, de forma que quan es buidi la cache, els inodes d'altres processos s'escriguin (i es desbloquegin) primer. Això aconsegueix reduir les llargues pauses i millora la fluidesa del sistema.

Resumen

Linux cachea las operaciones I/O de disco: las escrituras acaban inmediatamente desde la perspectiva de userspace, y más adelante se escriben realmente al disco. Pero en la presencia de procesos que escriben sin límite, este buffering puede dañar facilmente la responsividad del sistema si no se gestiona de forma justa.

Se trata de un problema conocido y que se ha trabajado bastante, pero que sigue presente a día de hoy. En este proyecto se investigan los subsistemas y tecnologías relevantes del kernel, se mide y racionaliza el comportamiento observado, y finalmente se diseña un PoC para mitigar los efectos adversos.

Se descubre que el problema parece estar en los vaciados de cache, ya que los inodes son bloqueados hasta que se escriben al disco. Hoy en día con cachés tan grandes, los inodes de procesos inocentes pueden mantenerse mucho tiempo bloqueados mientras esperan ser escritos. Esto aveces *reemplaza* el throttling normal.

Se ha diseñado un daemon que detecta procesos ofensivos y les reduce la prioridad I/O, de forma que cuando se vacie la caché, los inodes del resto de procesos se escriban (y desbloqueen) antes. Esto consigue reducir las largas pausas y mejora la fluidez del sistema.

Revision history and approval record

Revision	Date	Purpose
0	2020-05-29	Document creation
1	2020-06-18	Partial revision
2	2020-06-28	Final revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Alba Mendez Orero	me@alba.sh
Juan Jose Costa Prats	jcosta@ac.upc.edu

Written by:		Reviewed and approved by:	
Date	2020-06-10	Date	2020-06-28
Name	Alba Mendez Orero	Name	Juan Jose Costa Prats
Position	Project Author	Position	Project Supervisor

Contents

List of Figures

List of Listings

List of Tables

1 Introduction

As usual in most modern systems, the Linux kernel caches disk I/O to optimize throughput and latency. This is called the *page cache*, and is an almost essential part of any GPOS because of its performance benefits.

Disk writes are also 'absorbed' by the cache as well. They usually complete immediately from the process' point of view, and are committed to the disk at a later time. This part of the cache (which essentially acts like a buffer for writes) is substantially complex and is often called the *writeback cache*.

However, like any buffer, it can easily introduce unfairness and side-effects when there are heavy I/O tasks writing at a fast speed (such as downloads, backups, or uncompressions). These may eventually saturate the cache, forcing the kernel to throttle I/O, which often results in a lagging system. Introducing some level of fairness in the throttling would improve this.

This research project, which is carried out at the Department of Computer Architecture, aims to develop a Proof of Concept that will (partly) introduce this fairness:

- First, the existent (unfair) I/O throttling will be measured. Its impact over system performance and general responsivity will then be analyzed. The main goal is to understand the dynamics leading to unresponsivity as deeply as possible.
- Then, the Proof of Concept will be designed and developed to isolate the throttling, so that only the offending tasks are affected by it.
- As an optional goal, a proper patch to the Linux kernel can be developed in order to provide actual fairness.
- Measures will be taken again, and the PoC will be tested on production systems.
- The resulting improvement in system performance / responsivity will be analyzed and general conclusions will be drawn.

1.1 The problem

The idea for this independent project comes from several attempts of diagnosing system unresponsiveness from the author, often in production environments. Simply starting a system upgrade or a backup would render some machines almost *unusable*. This problem occurs frequently, as the only requisite is to have an application that writes lots of data to the disk.

The cache is always enabled by default, since it greatly improves performance:

- Tasks can continue their work without being having to wait for I/O writes, which means better latency and resource utilization, especially CPU efficiency.
- Prevents 'bumpiness' (greatly varying latencies) in the physical device from damaging the performance of tasks. This is especially relevant in "read-write loops" (i.e. a download from the network into disk) which greatly reduce their throughput if there are unstabilities in one of the ends [3].
- I/O is submitted to the disk in bulks, which allows the I/O scheduler (or elevator) to build efficient schedules for their transfer.

However, it is also unfair because it treats all writes equally. So, when the cache gets full, all applications that perform I/O are allegedly slowed down (I/O throttling), affecting the whole system.

To avoid this problem, an application that wants to write lots of data may explicitely enable `O_DIRECT`, a POSIX flag which causes these writes to bypass the writeback cache, preventing the cache from filling (and therefore, avoiding I/O throttling). However this is far from a solution, because:

- The user must modify the application source code in order to enable it (users don't normally know about this source code).
- Applications don't usually know whether they're going to cause the cache to fill; it's hard to predict if the bottleneck will be at the disk writing speed. It's the kernel that knows.
- Using an option that does an unwanted thing (it disables the cache, reducing performance) just to prevent a system-dependent side effect (I/O throttling) doesn't feel right at all.

Linux also allows disabling the writeback cache on an entire filesystem, by remounting it in sync mode. However this is also not ideal; it degrades system performance as indicated above.

Linux also allows to adjust the threshold at which the cache is considered full (i.e. the queue size). Threshold, bandwidth and I/O scheduling can also be tweaked per block device. However, neither of this prevents the throttling from triggering; at most, it can isolate it to applications using that block device, but this doesn't usually help.

1.2 Previous efforts

The writeback cache (and its throttling) has been a well known source of unresponsiveness. In 2010, modifications were made to the Linux kernel where (among other things) I/O is now rate-limited early to avoid reaching the actual limit where full throttling begins. From patch `143dfe86` [3]:

> [. . .] long block time in balance_dirty_pages() hurts desktop responsiveness [. . .]
>
> Users will notice that the applications will get throttled once crossing the global (background + dirty)/2=15% threshold, and then balanced around 17.5%. Before patch, the behavior is to just throttle it at 20% dirtyable memory in 1-dd case.
>
> Users will notice a more responsive system during heavy writeback.
> "killall dd" will take effect instantly.

Some other changes have also been made in respect to throttling since then; however throttling still has a substantially noticeable effect on responsiveness on the mainline kernel at the time of this writing.

Some possible reasons this hasn't been properly addressed are:

- The writeback cache is an extremely complex component. It doesn't cleanly belong in a single subsystem, instead being managed jointly by the *memory subsystem* and the *filesystem layer* at the same time, and also interfaces with the *block I/O layer* to commit changes to the disk.
- Its behaviour isn't easy to test, inspect or reason about due to their macroscopic nature and the variety of situations one can find.
- It's a critical part of the system in terms of performance.
- There may be hidden dynamics we're unaware of at the time of starting this project.

It's important to note that throttling appears to have been partially task-fair at earlier versions of the Linux kernel; however that doesn't seem to be the case today.

On the other side, subtle control features have been made available (memory cgroups, BIO annotation support), but aren't —to the author's knowledge— being actively used to address I/O throttling fairness. This project attempts to build on these features. However throttling is a complex process, so we first need to understand & measure it in detail.

1.3 Project structure & requirements

Since we're dealing with a complex component of the kernel, substantial research will be needed at the early stages of this project. Understanding what leads to the throttling is critical for success.

Due to the nature of Linux there is plenty of documentation, discussion and commit history publicly available on the matter; however the scope is limited and will likely not allow for a 100% correct, in-depth, source-code level analysis of the problem. Likely, we are not aiming for a definitive fix, or even a proof-of-concept for one, but at least a solution that can successfully mitigate the problem partially.

Requirements

- Tooling to measure and analyze when and how I/O throttling occurs, and what tasks are affected.
- The measuring process should be as less invasive as possible, so it can be used on a production system and measurement itself doesn't alter the results.
- The developed Proof of Concept must be able to isolate the I/O throttling to a small part of the system, ideally at process- or task- level.
- The Proof of Concept should be developed preferably in the form of a *userspace daemon*.
- Optionally, a proper kernel patch may be developed.

General specifications (both PoC and measurement tooling)

- The kernel must be compiled with accounting and tracing support.
- In order to implement fairness, we first need to track the origin of block writes. If these writes come from a filesystem (which will usually be the case), then explicit support is required from the FS in order to track the origin. According to the kernel's documentation, only ext2, ext4 and btrfs have this support implemented. **Implementing tracking support on filesystems is out of scope**. Thus, this isn't guaranteed to work on other filesystems.

PoC operation specifications

- Maximum time from (a) start of offending writes to (b) throttling isolation: 5 s
- The kernel must be compiled with cgroup support.
- If multiple processes are writing to the same inode, and at least one of them is performing offending writes, throttling isolation is not guaranteed.
- Custom kernel patches might be required.

Project: Analysis	WP ref: WP1	
Major constituent: measurement & analysis	Sheet 1 of 1	
Short description: Develop necessary tools to non-invasibly measure and analyze the dynamics of the I/O throttling and how it affects other processes.	Planned start date: 2020-03-01 Planned end date: 2020-03-28	
Internal task T1: real-time monitor Internal task T2: ftrace analysis Internal task T3: dummy loads / processes Internal task T4: perform basic tests Internal task T5: perform cgroup tests	Deliverables: None	Dates: None

Project: Implementation	WP ref: WP2	
Major constituent: design & development	Sheet 1 of 1	
Short description: Understand throttling dynamics. Design & develop Proof of Concept to (partly) isolate throttling	Planned start date: 2020-03-29 Planned end date: 2020-05-02	
Internal task T1: understand throttling dynamics Internal task T2: design general operation, validate it Internal task T3: PoC development Internal task T4: perform basic tests	Deliverables: Critical review	Dates: None

Project: Discussion	WP ref: WP3	
Major constituent:	Sheet 1 of 1	
Short description: Test the PoC in production systems, measure improvement, optionally develop proper kernel patch	Planned start date: 2020-05-03 Planned end date: 2020-06-08	
Internal task T1: production test, measures Internal task T2: improvement anaysis Internal task T3: [Optional] kernel patch development	Deliverables: Final memory, source code	Dates: None

Table 1: Work packages for this project

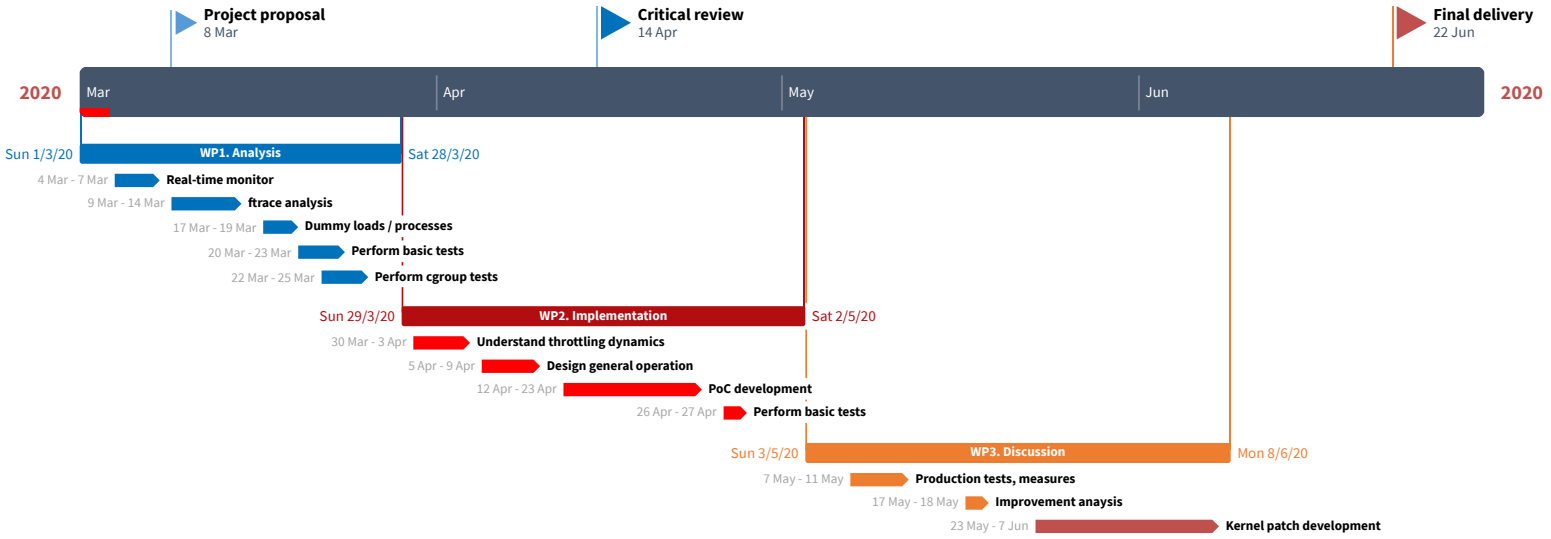


Figure 1: Gantt diagram showing initial work distribution

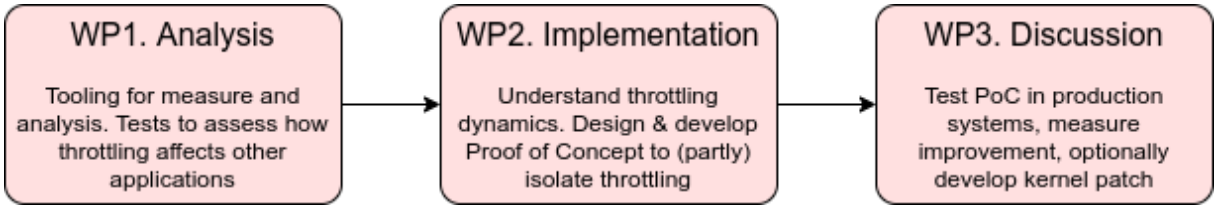


Figure 2: Work breakdown structure of this project

Initial work plan Figure 2 shows the general structure of the work breakdown, table 1 shows the work packages and figure 1 shows the Gantt diagram. Critical review was initially planned to happen shortly after starting WP2.

Deviations The main source of deviation was the unexpected cause for pauses: we didn't expect them to be caused by cache flushes. This required researching new approaches and studying the block layer in more detail. In the end, we were still able to get both a functional mitigation, and a working Proof-of-Concept. Some of the tasks from WP2 and were conducted in parallel with WP3, and most of WP3 couldn't be carried out. In addition, WP1 ended up getting more weight than WP2. We believe incidentals of this magnitude are natural in any research project, though.

Another unexpected deviation not reflected in the Gantt was that we ended up spending more time building appropriate tooling than using that tooling. On the other hand, that also gave us a deeper understanding of many kernel technologies beyond the immediate subject of the project.

This project was also substantially impacted by COVID-19, originating changes in both the work distribution (about a week of delay at the time of the critical review) and the deadline schedule from ETSETB.

2 State of the art

This chapter presents a conceptual overview of the current state of the relevant kernel subsystems and APIs we'll be working with during this project. Linux is a huge piece of software and its source code can be hard to navigate, so having a conceptual understanding of the relevant components & the roles they play out is crucial.

The central part to analyze is the **memory (VM) subsystem**, which is directly responsible of the writeback cache, but as we'll see later, the **block I/O subsystem** and **filesystem (VFS) layer** are of special importance as well. Those are presented within sections 2.1 & 2.2.

During analysis, we'll probably need to get insight on what is happening in the kernel. For complexity and practicity, this should be preferably done in a *non-invasive way* that avoids modifying the kernel or altering the results of the experiments themselves. Section 2.3 presents an overview of the currently available mechanisms for tracing & debugging the Linux kernel.

When developing a possible solution —be it in form of kernel patches, loadable modules or userspace daemons— we'll likely rely on **accounting mechanisms** to track resource usage of (groups of) tasks and provide fairness. **Resource control** mechanisms will also be needed to enforce limits on (groups of) tasks, in the case of a userspace daemon. Section 2.4 details relevant accounting & control mechanisms readily available in Linux.

Finally, section 2.5 details other technologies used in this thesis.

2.1 Linux and the I/O stack

This project is centered around the Linux kernel, and we'll be working on the **mainline version** at this time.

Flow of an I/O operation Before beginning work, it was important to have an overview of the whole I/O stack in Linux, even if simplified. When an I/O operation is issued from userspace upon a **mounted filesystem**, the following happens:

1. **VFS:** The VFS (short for 'Virtual File System') layer handles the operation and calls the appropriate handler on the corresponding filesystem.
2. **Writeback cache:** The pages holding data to be written are marked as *dirty*. The **memory subsystem** (also called VM or MM) keeps track of dirty pages. The VFS operation then usually completes immediately, and at a later time, kernel workers enqueue (some of) the dirty pages to be actually written to the underlying block device. The pages are now in *writeback* state.
3. **BIO:** At this point, the operation is called a BIO —short for 'Block I/O'— and it's handled by the **block layer**. It's placed on a per-device queue[1] (or set of queues), and

[1]Some special block devices (like loop devices, or the device mapper) don't use a queue or I/O scheduler.

the **I/O scheduler** (or elevator) selects operations from that queue and issues them to the hardware (disk drive). There are many elevators on Linux, such as `bfq` (which provides fair scheduling, explained in section 2.4) or `noop` which is simple FCFS.

4. **Disk drive:** When the BIO is selected from the queue, the block driver issues it to the drive. After this, the operation is complete. However, the disk drive itself is often capable of caching the received operations. Since this caching layer is in hardware it's usually transparent to the kernel, except for the need to issue cache flushes when requested. Utilities like `hdparm` may be used to enable or disable the drive's cache.

If the I/O operation is directly upon an open block device, it goes directly to step 3 and isn't of interest. Also of note is that the VFS layer does have some internal caches for inodes and dentrys, but this doesn't seem to be relevant either.

Step 2 (the writeback cache) is what we'll work with, and its behaviour and interface was further researched and is explained in section 2.2.

Figure 3 shows a representation of the flow. Please note how the page cache (i.e. the VM layer) isn't *after* the VFS layer, but *next to it*. As will be seen later, these layers interact in both directions.

Skipping the cache Some operations may skip the writeback cache. In this case step 2 is not performed (thus, the memory subsystem is not involved); the I/O operation is immediately scheduled and completes when issued to the disk. There are many ways for this to happen:

- From userspace, if the file was open with `O_DIRECT`. This is a POSIX flag that does precisely that; it instructs operations on the file to skip caching.

  A related option is `O_SYNC`, which instructs data to be written synchronously to disk. They have different meanings, but `O_SYNC` involves among other things skipping the cache.

- By mounting the filesystem in `sync` mode (versus `async` mode, the default). This causes all operations on that filesystem to skip caching. A command like the following may be uesd to disable the writeback cache, on the fly, in the root filesystem:

  ```
  mount -o remount,sync /
  ```

(This list isn't exhaustive.)

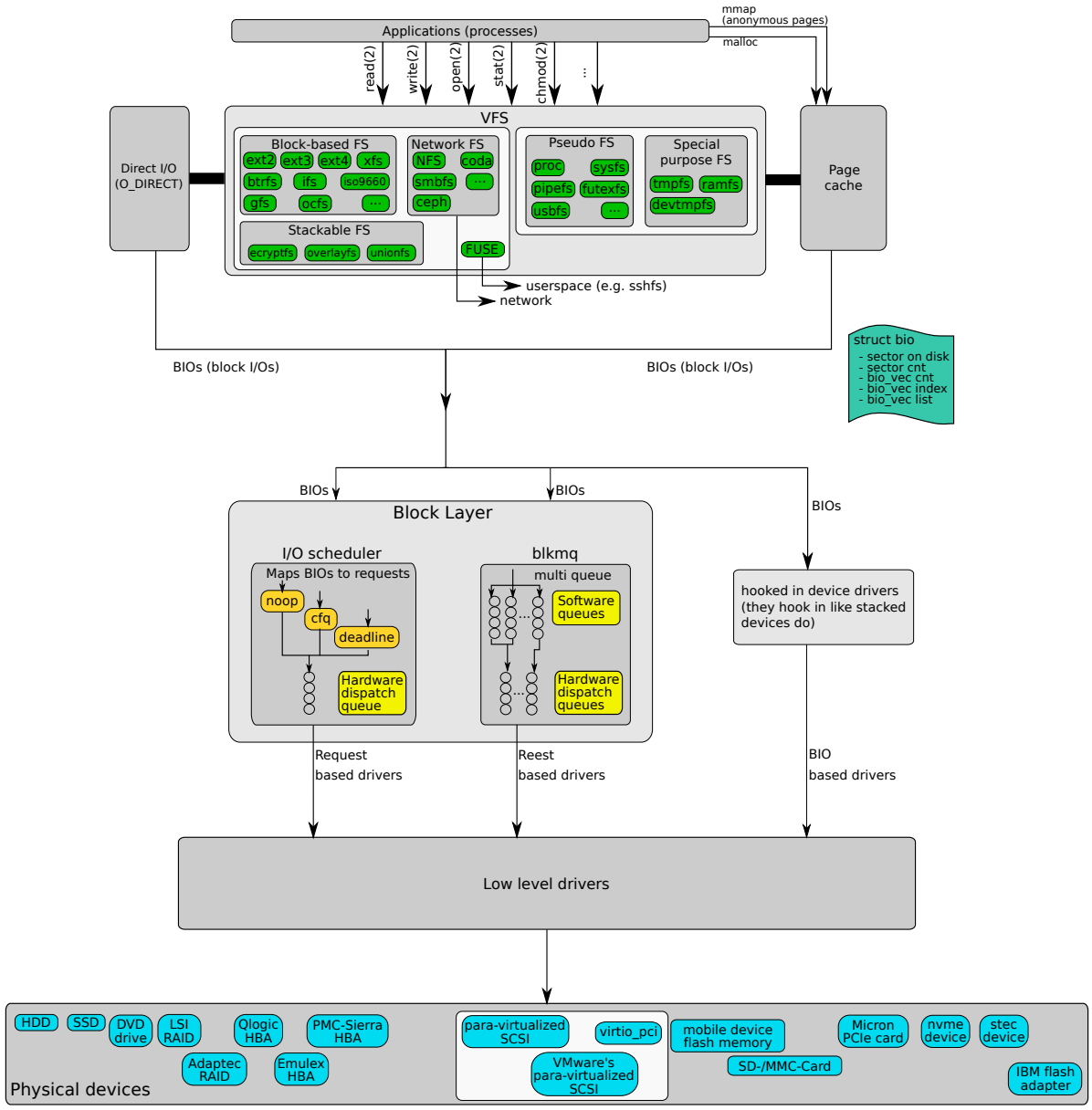


Figure 3: Simplified view of the Linux I/O stack, adapted from [1]

2.2 The writeback cache

Introduction The writeback cache is a subset of the functionality provided by the VM's **page cache**. The page cache is an essential component in any general purpose OS, whose main intent is to reduce I/O operations on the underlying block device and improve perfomance.

Essentially, the page cache puts free memory pages to good use by mapping them to storage. "Free" in this context approximately means "pages not reserved by the kernel, network buffers or user processes". (This hasn't always been that way, though; some time ago, swappable process memory was considered free as well. [4])

In that regard, **writeback caching** allows writes to be cached as well, and performed at a later time. This is useful for two important reasons:

- Tasks can continue their work without being having to wait for I/O writes, which means better latency and resource utilization, especially CPU efficiency.
- Prevents 'bumpiness' (greatly varying latencies) in the physical device from damaging the performance of tasks. This is especially relevant in "read-write loops" (i.e. a download from the network into disk) which greatly reduce their throughput if there are unstabilities in one of the ends [3].
- I/O is submitted to the disk in bulks, which allows the I/O scheduler (or elevator) to build efficient schedules for their transfer.

Conceptually, it acts like a large buffer for writes.

General operation Writeback caching works by tracking pages that become **dirty**, i.e. modified with respect to what's stored in the block device. Pages can be modified by regular operations on open files, or through other means such as mapped file memory.

Kernel worker tasks wake up periodically and transition some of the pages into **writeback** state, which means they're being written to the block device (resulting in one or more in-progress BIOs). This is done according to some criteria, such as how long the page has been dirty, or whether the current amount of dirty pages surpasses a configured **background dirty threshold**.

Once the page has been written (the BIO has finished), it's marked as clean and is now eligible to be removed from the cache.

Integration with VFS It's a complicated component, because even though part of the memory management layer (VM), it has to coordinate with the VFS layer to maintain file-to-storage mappings and flush changes to disk when either of the parts request it (i.e. `sync` has been called, or the system is running low on memory).

And this integration between both components is often tricky, because the VFS layer always works in terms of *inodes*, whereas the VM layer always works in terms of pages [5]. Many pages

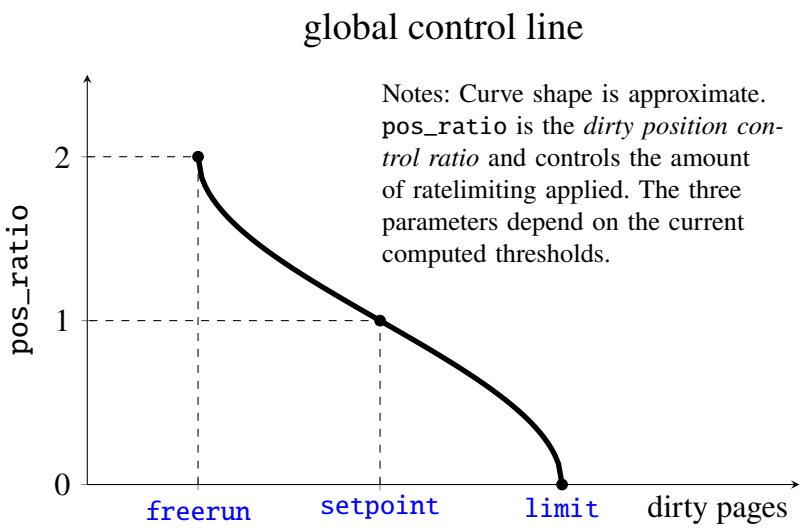


Figure 4: Representation of the global throttling curve, according to [2]

can belong to the same inode.

Throttling Like every buffer with finite capacity, the writeback cache needs a mechanism to throttle tasks that dirty pages to prevent it from filling indefinitely.

The main user-facing parameter to control this throttling is the **dirty threshold**, which defines the maximum percentage[2] of the *free memory* (see above) in the system that can ever hold dirty pages[3]. Together with the **background dirty threshold** mentioned before, these define most of the throttling characteristics.

Previously, throttling seemed to be simple: when the dirty threshold is surpassed, new writes are no longer cached and effectively become synchronous (i.e. complete after processed by the block device). This was hurting interactivity and performance, and was later replaced by a **soft-throttling** scheme based on scheduler rate-limiting [3].

Instead, processes now begin to be throttled *before* the dirty threshold is reached (at around the midpoint between it and the background dirty threshold, called the **setpoint**), and the pauses are designed to increase as the dirty limit is approached. The details are a bit more complicated: the curve that governs the throttling is represented in figure 4, the curve parameters are marked in blue [2].

Furthermore, the kernel supports block device-specific throttling, which is intended to apply backpressure from the disk queues into the cache & applications. It's enabled through

[2]Modern kernels allow defining an absolute amount of memory instead of a percentage of a dynamic value [6]. This is useful in systems with lots of RAM, as only *integer* percentages are accepted.

[3]In most contexts, especially throttling, *dirty pages* includes pages in writeback state as well.

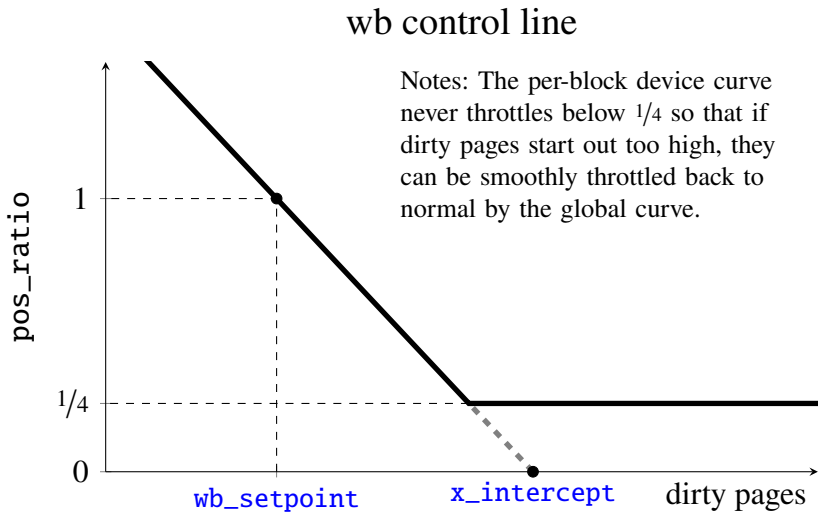


Figure 5: Representation of the per-block device throttling curve, according to [2]

`CONFIG_BLK_WBT` and implemented through a separate curve, represented in figure 5. The source code appears to indicate that the minimum of both curves is in effect.

Unfairness However, the throttling seems to be applied the same way no matter the task. There is a manual exception for tasks that have the `PF_LOCAL_THROTTLE`, which get throttled in less cases, but little more. This is expected, given that the memory subsystem has reduced capability of tracking the ownership of each page (but see section 2.4).

The block device *does* seem to be taken into account, as we saw, but this doesn't help on most Linux installs which have all the system in one disk.

Taking that into account, it's reasonable to expect the same throttling applied to innocent, non-cache-starving processes than is being applied to the starving ones. Also, workers don't seem to apply any fairness when selecting pages for writeback.

Existing parameters Writeback cache parameters can be adjusted on the fly through the sysctl interface [7]. These are the relevant ones:

`vm.dirty_background_ratio` The threshold, as a percentage of free memory, at which dirty pages start to be transitioned to *writeback* state (default: 10 %).

`vm.dirty_background_bytes` Like `vm.dirty_background_ratio`, but specified as an absolute quantity in bytes (default: none).

`vm.dirty_ratio` Target maximum dirty + writeback pages, as a percentage of free memory (default: 20 %).

`vm.dirty_bytes` Like `vm.dirty_ratio`, but specified as an absolute quantity in bytes (default: none).

`vm.dirty_writeback_centisecs` Interval, in centisecs, at which kernel workers wake up to write dirty pages (default: 5 s).

`vm.dirty_expire_centisecs` Time, in centisecs, after which dirty pages are always written to disk regardless of threshold (default: 30 s).

Some parameters may be adjusted per block device as well, but aren't detailed here for brevity and because of its limited use for this project.

2.3 Kernel tracing tools

When performing the experiments, we'll likely need to debug the kernel in some way to better understand what's happening. Methods like traditional debugging (kgdb) are out of question because we're trying to analyze behaviour over time, and the problem is hard to reproduce precisely. In addition to that, we'd like debugging to be as unobtrusive as possible and avoid modifying the experiment itself (which is tricky, because we're debugging the VM & FS).

Thus, *tracing* seems to be what we need. Available kernel tracing mechanisms were researched; this section presents the most relevant ones.

The kernel tracer (ftrace) Linux has a generic tracing system, which consumes **events** from multiple sources and aggregates them into a ringbuffer. Events are encoded in a binary format and include a timestamp (many clock sources are available), PID, CPU number and event-specific fields.

The tracer has a flexible filter system. Specific events may be enabled or disabled in many ways: either manually, for some time, or in response to another event being fired. This allows to pin down selected events, which is relevant in situations like this one, where interesting data may be buried in a frequently logged event (I/O). There's also some support for aggregating data into histograms, for instance.

Userspace can safely consume the ringbuffer and store it somewhere via the `splice` syscall, which is handy (as the process itself may get throttled). There's also a text-based ringbuffer consumer, but that doesn't support `splice`.

Among the event sources supported are **tracepoints** (explained below), an included **function tracer**, and other profilers. Userspace can also log custom data into the ringbuffer, in the form of `print` events.

We won't go into the details of the user interface, for brevity and because this will be handled by a user-space tool (presented later).

Tracepoints Tracepoints are a debugging mechanism. They are statically inserted in kernel code through a macro, at points of interest. Then, through the API, the user can supply a callback that will be injected at the tracepoint they wish. The only cost of tracepoints, if enabled, is a branch condition check when not in use [8].

They can also serve as event sources. When inserting a tracepoint, the developer can specify additional *glue code* specifying the event name, fields and their representation. The tracepoint may then be enabled in the tracer and will generate an event every time it's hit [9].

Tracepoints are organized in *subsystems*. In our case, we're mostly interested in the `writeback` subsystem, and specifically in the following events, which were identified to be useful in getting a general overview of the state of the cache:

- `writeback:global_dirty_state`: This tracepoint fires conditionally at the end of the `domain_dirty_limits` function, which calculates the thresholds mentioned earlier from the configured percentages of free memory. The event reports:

  `nr_dirty` Current number of pages in dirty state

  `nr_writeback` Current number of pages in writeback state

  `background_thresh` Calculated background dirty threshold, in pages

  `dirty_thresh` Calculated dirty threshold, in pages

  `dirty_limit` Not completely sure what this is, seems to be the absolute limit after which all operations still become synchronous. It's higher than `dirty_thresh`.

  `nr_dirtied` Accumulated counter of dirtied pages

  `nr_written` Accumulated counter of pages written to disk

- `writeback:balance_dirty_pages`: This tracepoint is placed in multiple points of the `balance_dirty_pages` function, which holds the main / root logic responsible for throttling. It's periodically called for processes that dirty pages, and regulates the amount of throttling (i.e. ratelimit) applied to them.

  The event reports too many fields to list them here, but amongst them there's the amount of dirty pages, the parameters (and input) to the throttling curves, and the resulting ratelimit for the task.

Other relevant events include the `syscall` subsystem, which has two events for every possible syscall (enter and exit) and allows similar functionality than `strace`.

Kprobes A more recent technology, Kprobes (originally part of Dprobes) allows injecting code on arbitrary positions in memory, barring a few exceptions. They are essentially dynamic tracepoints. How this is accomplished depends on the architecture, but it generally involves patching the instruction at the supplied address, replacing it with a jump to user code, and then

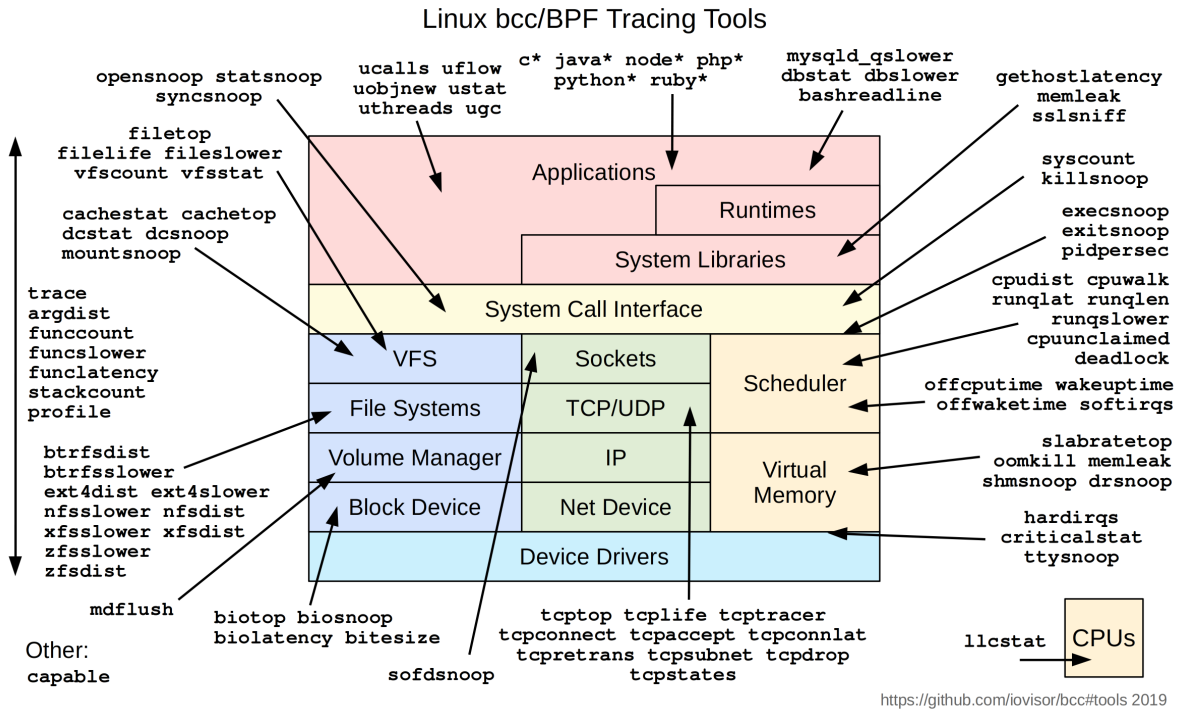


Figure 6: Overview of current kernel BCC/BPF tracing tools

executing that missing instruction at the end (before resuming normal execution).

Kprobes are useful as they allow patching any function or spot on the fly. There's also a second type of Kprobes, called Kretprobes, that fire when an arbitrary function returns.

Like tracepoints, Kprobes has a kernel API. However, it can be used together with the tracer as event sources, which is convenient [10]. They can also invoke BPF programs. Both of these solutions are entirely user-space.

To register Kprobes as event sources, the user writes entries to:

```
/sys/kernel/tracing/dynamic_events
```

These entries specify what info should be obtained and how (i.e. registers, variables, arguments, dereferencing), the symbol+offset to insert the Kprobe at, and an event name. This then appears among the other events and may be enabled or disabled as usual.

BPF Short for Berkeley Packet Filter, it's a VM (Virtual Machine) that lives in the kernel, along with its bytecode format. BPF allows user-space to upload custom, architecture-independent programs that are interpreted in kernel-space at certain contexts.

It is primarily designed to be safe and compact: the kernel verifies the bytecode before running it to make sure there's no infinite loops, etc.

BPF was originally created to filter network packets when capturing traffic, and the first version has been renamed to *classical BPF* (or simply, cBPF). The new version, *extended BPF* (eBPF), performs better and allows the programs to communicate among themselves and with user-space, through shared data structures [11].

Since then, BPF has found more and more uses. A very powerful one is **attaching BPF programs to Kprobes and tracepoints**. This (especially with BCC, see below) provides a powerful way to inspect kernel structures at an arbitrary point of execution that is easy, safe, portable and unobtrusive, entirely from user-space (i.e. instead of modifying the kernel or writing a kernel module).

Programs are verified and can only read memory, not modify it, so it's guaranteed that they won't damage the system. BPF Kprobes are thus safe to load even in production systems. A number of diagnostic tools and possibilities have been made possible, see figure 6.

User-space solutions Until now we've specifically covered available kernel technologies. User-space tools have been created to make the use of some of them easier, and it's worth looking into them due to the limited scope of a degree thesis.

- **trace-cmd**: This is the official user-space tool for the kernel tracer. A bit unmantained, but seems to be feature complete and easy to use. It is able to configure the tracer and log events to a single `trace.dat` file safely using `splice`, and handles some other technicalities.

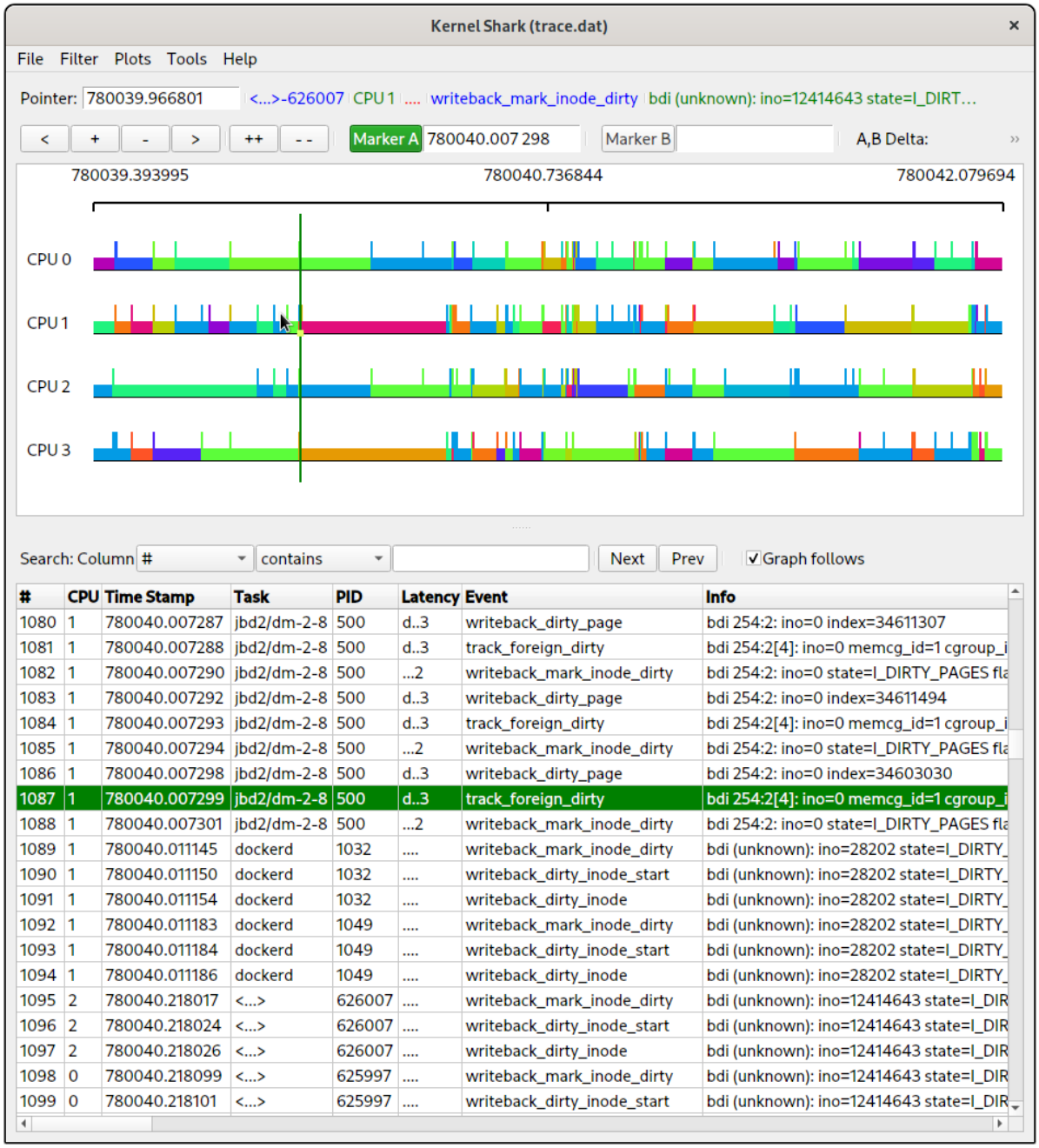
  It also contains libraries for parsing the event binary format, and a Python interface which can be very handy. Also included is **kernelshark** (figure 7), a graphical viewer for event capture files produced by `trace-cmd`. The format is designed for efficiency, and kernelshark can quickly open logs with millions of events.

- **blktrace**: Tool that listens for events logged from the block layer, and relayed through debugfs in a similar way to the tracer. Contains very detailed info about block I/O including scheduler operation, driver operation, queues, latencies. . .

- **perf**: Kernel profiler. Not investigated as it appears of little relevance to the project.

And some tools are more generic, allowing the user to script their behaviour:

- **BCC**: Set of tools that make BPF development significantly easier. These include:
  - Toolchain that allows BPF programs to be compiled from C code, linked against kernel headers, validated, etc.
  - Generic library that can invoke the toolchain, load the (resulting) BPF programs into the kernel, and manipulate the shared structures from user-space. The library has a Python interface as well.



#	CPU	Time Stamp	Task	PID	Latency	Event	Info
1080	1	780040.007287	jbd2/dm-2-8	500	d..3	writeback_dirty_page	bdi 254:2: ino=0 index=34611307
1081	1	780040.007288	jbd2/dm-2-8	500	d..3	track_foreign_dirty	bdi 254:2[4]: ino=0 memcg_id=1 cgroup_i
1082	1	780040.007290	jbd2/dm-2-8	500	...2	writeback_mark_inode_dirty	bdi 254:2: ino=0 state=I_DIRTY_PAGES fla
1083	1	780040.007292	jbd2/dm-2-8	500	d..3	writeback_dirty_page	bdi 254:2: ino=0 index=34611494
1084	1	780040.007293	jbd2/dm-2-8	500	d..3	track_foreign_dirty	bdi 254:2[4]: ino=0 memcg_id=1 cgroup_i
1085	1	780040.007294	jbd2/dm-2-8	500	...2	writeback_mark_inode_dirty	bdi 254:2: ino=0 state=I_DIRTY_PAGES fla
1086	1	780040.007298	jbd2/dm-2-8	500	d..3	writeback_dirty_page	bdi 254:2: ino=0 index=34603030
1087	1	780040.007299	jbd2/dm-2-8	500	d..3	track_foreign_dirty	bdi 254:2[4]: ino=0 memcg_id=1 cgroup_i
1088	1	780040.007301	jbd2/dm-2-8	500	...2	writeback_mark_inode_dirty	bdi 254:2: ino=0 state=I_DIRTY_PAGES fla
1089	1	780040.011145	dockerd	1032	....	writeback_mark_inode_dirty	bdi (unknown): ino=28202 state=I_DIRTY_
1090	1	780040.011150	dockerd	1032	....	writeback_dirty_inode_start	bdi (unknown): ino=28202 state=I_DIRTY_
1091	1	780040.011154	dockerd	1032	....	writeback_dirty_inode	bdi (unknown): ino=28202 state=I_DIRTY_
1092	1	780040.011183	dockerd	1049	....	writeback_mark_inode_dirty	bdi (unknown): ino=28202 state=I_DIRTY_
1093	1	780040.011184	dockerd	1049	....	writeback_dirty_inode_start	bdi (unknown): ino=28202 state=I_DIRTY_
1094	1	780040.011186	dockerd	1049	....	writeback_dirty_inode	bdi (unknown): ino=28202 state=I_DIRTY_
1095	2	780040.218017	<...>	626007	....	writeback_mark_inode_dirty	bdi (unknown): ino=12414643 state=I_DIR
1096	2	780040.218024	<...>	626007	....	writeback_dirty_inode_start	bdi (unknown): ino=12414643 state=I_DIR
1097	2	780040.218026	<...>	626007	....	writeback_dirty_inode	bdi (unknown): ino=12414643 state=I_DIR
1098	0	780040.218099	<...>	625997	....	writeback_mark_inode_dirty	bdi (unknown): ino=12414643 state=I_DIR
1099	0	780040.218101	<...>	625997	....	writeback_dirty_inode_start	bdi (unknown): ino=12414643 state=I_DIR

Figure 7: Kernelshark screenshot showing an open event capture file

The potential of BCC when combined with Kprobes is huge (see figure 6), but has the problem of still being low level. Examples provided by the project include: counting task switches from/to PIDs, tracing data generated by urandom, inspecting KVM, ...

- **SystemTap**: An integrated framework for profiling and tracing the system. It defines its own scripting language and is able to use many kernel technologies under the hood. Higher level than BCC, although probably not as powerful.

2.4 Resource control & accounting

This section details kernel technologies explored in the development phase of the proof-of-concept. These allow us to impose restrictions on resource consumption from (groups of) processes, which will hopefully be useful in mitigating the adverse effects of writeback caching.

There's also some mention of APIs for resource accounting, which allow us to query resource consumption and will be useful to identify relevant processes and measures to take.

cgroups Short for Control Ggroups, it's the primary mechanism for resource control in Linux. They allow placing hard limits, soft limits or weights to groups of processes regarding CPU, memory, I/O and more. cgroups are hierarchical, making them a good technology for containers. They can also be used for accounting.

They're used via a set of magic filesystems typically mounted in `/sys/fs/cgroup`. Each cgroup is represented by a directory, and cgroups can be created / removed with `mkdir` & `rmdir`. Inside the directory of a cgroup there's many files to query or modify every parameter.

By default, processes are attached to the root cgroup (which can't be deleted). Processes may be moved to another cgroup by writing their PID to the `cgroups.procs` file in the cgroup's directory. The current cgroup of a process may be queried by reading `/proc/<pid>/cgroup`. There's no control for individual tasks, only processes.

A variety of cgroup *controllers* are available, each one offering control for a certain kind of resource. The `memcg` controller manages memory, `blkio` manages block I/O, ... There are also controllers for less tangible resources like I/O latency.

There's two versions of cgroups, v1 & v2, which can partially coexist on a system. In v1, every controller has its own cgroups (and hierarchy) independently from the rest, but in v2, there's a single hierarchy for all controllers. A system may use a certain controller for v1 or v2, but not both: this means we should theoretically support both versions just in case. `systemd` is present in most modern systems and uses `memcg` with v1, so we'll need to use v1 as well. v1 controllers are mounted at `/sys/fs/cgroup/<controller name>`, so to create a memory cgroup one would use:

```
mkdir /sys/fs/cgroup/memory/mycgroup
```

v2 is a single magic fileystem typically mounted at `/sys/fs/cgroup/unified`.

Other APIs can also integrate with cgroups, like taskstats, some I/O schedulers (explained below), or eBPF.

Memory cgroups (memcg) One of the cgroup controllers we'll likely be working with is `memcg`, which lets the user place soft & hard limits on the memory usage of the cgroup. It also offers accounting for various VM related operations such as faults, swapping, ... [12]

At one point [13], dirty page accounting was being implemented and offer a way to change the dirty pages limit for the cgroup. However, it appears to not have reached the mainland kernel.

Block I/O cgroups (blkio) This cgroup controller allows throttling the read or write bandwidth, per block device. It also allows some accounting on the performed I/O and allows setting the weight of the cgroup, which is used by priority-aware I/O schedulers, like BFQ (see below).

BFQ scheduler Short for Budget Fair Scheduler, it is a time (and bandwidth) proportional I/O scheduler, designed to operate in conjunction with cgroups or I/O process priorities, see `ioprio_set`. It works by dividing time in slices and reserving every slice to a particular task or group. The relative amount of time reserved depends on the weight of the process [14].

However BFQ also has multiple queues that are served in strict priority; this can be specified by setting a different I/O class, in addition to the I/O priority value (which ends up mapped to weight linearly). There are three: realtime, best effort and idle. Hierarchical distributions are also accepted.

By default, BFQ automatically detects interactive tasks and raises their weights. As natural BFQ has a somewhat modest overhead compared to simpler schedulers like `mq-deadline`, which is frequently the default.

Process stats APIs To obtain info and statistics at the process level, applications usually operate on procfs (`/proc`). However if we need more specific statistics, then the **taskstats API** seems to offer them.

taskstats is an accounting API that supplies much more detailed info about processes and individual tasks, and also informs the listeners through events when a task dies. Among the provided data is delay accounting, memory usage, disk I/O, syscall counts, ... [15]

taskstats is implemented on top of the Generic Network protocol, and unlike procfs, requires root or the `CAP_NET_ADMIN` capability to be used —this was done due to the sensitivity of the data, which among other things made it easier to guess passwords. Several tools use this API, like `iotop`.

2.5 Other used techologies

UML Short for User Mode Linux, this is a 'virtual' architecture shipped with Linux that allows it to be compiled as a regular executable. The kernel can then be 'booted' by simply executing it, and runs as a regular user-space process inside the host kernel [16].

In essence, it's a light way (in terms of setup) to run a virtual machine, requires *no privileges*, boots quickly and is fairly portable (and also fun). It is not used in production, since it isn't especially secure or performant when compared to using a proper hypervisor that takes advantage of hardware acceleration.

Using UML only requires setting `ARCH=um` when compiling the kernel. `make menuconfig` will show some UML-specific related options, such as whether to compile a 64-bit kernel and the processor type. `make` will then produce a `linux` executable. To boot the kernel, invoke this executable supplying kernel parameters as command-line arguments. Keep in mind these arguments are internally joined with spaces before processing.

Files may be exposed to the kernel as block devices, through the UBD driver: add `ubdX=file` arguments such as `ubda=/tmp/rootfs` and it will appear as `/dev/ubda`. Copy-on-write may also be used by supplying a comma-separated list of files. UML also ships with a 'magic filesystem' that exposes the host filesystem: it's called hostfs and may be used like this:

```
UML$ mount -t hostfs none /host -o /home/dev/uml
UML$ cat /host/foo  # accesses /home/dev/uml/foo in the host
```

This causes problems with permissions and privileged operations when the kernel runs as an unprivileged user, because not even UML root will be able to modify the ownership of files, for instance. To remediate this, an additional filesystem is provided that stores the metadata separately, but it is not relevant to the project.

Other peripherals are provided as well, such as multiple options to provide network (TUN or TAP) and serial lines / consoles. The amount of physical memory used by UML may be adjusted through the `memory` option, i.e. `memory=150M`.

3 Methodology / project development

Carrying out this project required defining an **experiment model**, and the development of different tooling to:

- Interface with the kernel tracer, and get familiar with the available data.
- Simulate the behaviour we're trying to analyze, in the form of both offender processes and innocent processes.
- Automate experiments, and carry them in a controlled environment to make them (more) reproducible.
- Analyze the resulting data and draw conclusions.

This section walks through the reasoning, design and development of the experiment model and tooling. Specific experiments, results or proof-of-concept development are presented later in section 4; however, some of the tools were developed in parallel with the experiments.

3.1 Realtime monitor application

Before starting to design the experiments, it was useful to have a bit more of insight into what's happening in the cache. We explored the **kernel tracer** on (section 2.3), and found the `global_dirty_state` tracepoint to be of interest, as it holds valuous info that can give us a first overview of the cache state (how many dirty pages exist, thresholds, writeback, . . .).

To familiarize ourselves with that data (and also with the kernel tracer) and play around with it, it'll be useful to have some sort of tool that will plot these variables in realtime —allowing us to quickly see the effects of our actions.

The best tool we had at hand (i.e. had experience working with, and seemed to be simple enough to make it feasible) was to write a Node.JS server together with an appropriate library for the browser, and let them communicate via websockets.

Node.JS is a platform for event-driven programming in JavaScript. Since it's based on an event loop and uses the same language as the browser, it was a natural choice for this application. We chose Express for the server, socket.io for the client-server communication, and Smoothie Charts for the real-time plots.

We won't go into much detail on the development here. First the usual server boilerplate code, which is shown in listing 1. Then we need to set up the tracer (clock & events) and start it, as shown in listing 2. Keep in mind that tracefs is mounted on `/sys/kernel/debug/tracing` in old systems, and `/sys/kernel/tracing` on recent ones [17].

And then the hard part is parsing the events. The ideal way would be to parse the event definitions and subscribe to the binary consumer (or per-cpu binary consumers), but implementing this seemed too much work. Instead we'll choose the text-based interface at `./trace_pipe`,

```
const { writeFileSync, createReadStream } = require('fs')
const path = require('path')
const express = require('express')
const readline = require('readline')
const { execSync } = require('child_process')

const app = express()
const server = require('http').createServer(app)
const io = require('socket.io')(server)

app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html')
})
app.use(express.static(__dirname + '/public'))
server.listen(4444)
```

Listing 1: Barebones webserver + socket.io code

```
const traceFsBase = '/sys/kernel/debug/tracing'

writeFileSync(path.join(traceFsBase, 'trace_clock'), 'mono')
writeFileSync(path.join(traceFsBase, 'tracing_on'), '1')

const subscribeToEvent = (name) => writeFileSync(
  path.join(traceFsBase, 'events', name, 'enable'), '1')

subscribeToEvent('writeback/global_dirty_state')
subscribeToEvent('writeback/balance_dirty_pages')
```

Listing 2: Setting up the kernel tracer

```
const eventsPipe = readline.createInterface({
  input: createReadStream(path.join(traceFsBase, 'trace_pipe')),
})

let stateTimer
eventsPipe.on('line', line => {
  // FIXME: parse timestamp and these other things
  const m = /^[^:]+ ([\d.]+): (\w+): (.+)$/.exec(line)
  if (!m) return console.log('Unparseable event:',
    ↪ util.inspect(line))
  const [ _, timestamp, event, info ] = [...m]
  if (event === 'global_dirty_state') {
    const parsedInfo = {}
    info.split(' ').forEach(x => {
        const m = /^(\w+)=(\d+)$/.exec(x)
        parsedInfo[m[1]] = Number(m[2])
    })
    if (stateTimer) return
    io.emit('global_dirty_state', { ...parsedInfo, pageSize })
    stateTimer = setTimeout(() => stateTimer = null, 100)
  }
})
```

Listing 3: Receiving and parsing tracer events in text form

and parse the lines using regular expressions. Once parsed, a `global_dirty_state` event is broadcasted to all client sockets using `io.emit(...)`. Also because of the high-rate of these events, we ended up setting up a timer (`stateTimer`) to filter them. This is shown in listing 3.

We also had to fix a bug in Node.JS that prevented the `writeFileSync` calls from working. After that, we moved to the client. Again, we won't go into much detail but the JavaScript code is shown in listing 4. It initializes a Smoothie Charts instance, listens for events from the server, and adds datapoints to the plot (taking care of converting from pages to MiB).

The application is now finished and may be browsed at `http://localhost:4444`. Figure 8 shows a screenshot. The full source code may be found in the `monitor` directory of the annex.

Important: Because unresponsiveness may freeze the browser itself (this should theoretically not happen to the server, but I didn't verify), it's *highly recommended* to browse the plots from another computer. Ideally, we should parse the event timestamps and use those, but this probably requires clock synchronization and wasn't deemed worthy.

```
const socket = io()
const statusLabel = document.querySelector('.status')
socket.on('connect', () => statusLabel.innerHTML = 'connected')
socket.on('disconnect', () => statusLabel.innerHTML =
  ↪  'disconnected')

const fromPages = (pages, event) =>
  pages / (1024 * 1024 / event.pageSize)

setupChart(document.getElementById('chart1'), {
  millisPerPixel: 58,
  grid: { verticalSections: 4 },
  labels: {},
  delay: 0,
  tooltip: true,
}, [
  {
    init: ts => socket.on('global_dirty_state', x => {
      ts.append(Date.now(), fromPages(x.thresh, x))
    }),
    options: { strokeStyle: 'rgba(255,0,0,0.80)' },
  },
  // ...
  {
    init: ts => socket.on('global_dirty_state', x => {
      ts.append(Date.now(), fromPages(x.dirty + x.writeback, x))
    }),
    options: { lineWidth: 2 },
  },
])

function setupChart(element, options, timeSeries) {
  const chart = new SmoothieChart(options)
  for (const ts of timeSeries) {
    const line = new TimeSeries()
    ts.init(line)
    chart.addTimeSeries(line, ts.options)
  }
  chart.streamTo(element, options && options.delay)
}
```

Listing 4: Receiving tracer events and plotting them in the browser

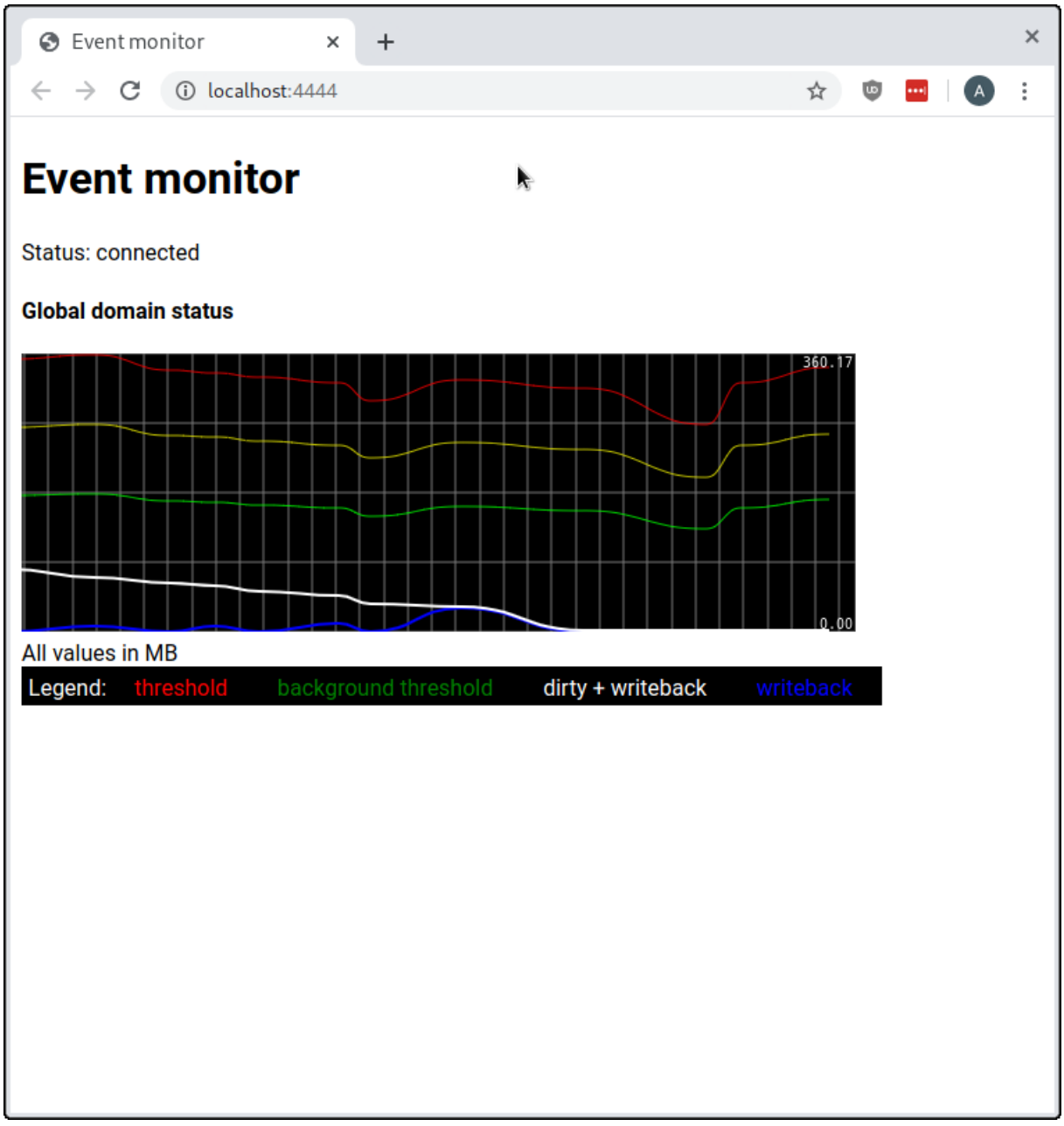


Figure 8: Screenshot of the finished monitor application

3.2 Experiment model

First task is to define the setup we'll use to run experiments. Since there aren't explicit parameters to touch and debugging is difficult, we'd like our experiments to be as *reproducible and clear* as possible while making it easy to play different configurations. We should *log all experiment parameters* to make it easy to identify and re-run previous experiments should we need it.

System unresponsiveness is subjective. But the basic idea is that in the presence of an offending (fast writing) process, other processes appear to be blocked for long amounts of time when submitting I/O. This gives us a straightforward way to proceed: we'll run a series of 'innocent' processes in the background, which will periodically attempt to do I/O and will log the time each attempt takes. If our assumption is true, reducing those pauses should improve responsiveness.

Loads As we said before, we need to code 'innocent' processes that will periodically submit some I/O operations and log the time each one takes. This will be our primary way to observe and validate unresponsiveness.

However, to trigger this unresponsiveness, we'll also need some faux 'offender' processes which will constantly write to disk in an unbounded way. Logically, it would also be helpful to visualize the times that *this* I/O takes, to compare them with the innocent ones.

This gives us a basic structure for each of the processes in our experiment, which we'll call 'loads':

```
times = []
while True:
  # execute load-specific task
  cycle()
  # log current time
  times.append( gettime() )

# once experiment finishes
print(times)
```

For an **innocent load**, `cycle` will consist of an I/O task followed by a sleep:

```
test_file = open('test_file', 'w')

def cycle():
  test_file.write('hello\n')
  sleep(0.1)
```

While for an **offender load**, the sleep will be omitted and the I/O can be submitted in larger blocks to put more pressure:

```
test_file = open('test_file_2', 'w')
block = bytes([ 0xFF ] * 4096)

def cycle():
  test_file.write(block)
```

Then, by inspecting the logged times (their differences), we can detect if any part of the cycle took more than expected, how much, and when it happened.

There is a reason for buffering the times into memory and logging them at the end: to avoid the logging *itself* from blocking (thus altering the results). From the kernel's perspective we want these processes to strictly submit I/O and sleep, nothing more.

It is also important to have a **control load**, which will do zero I/O; its cycle can just be a sleep. This will let us be sure the pauses we observe are strictly because of I/O, and not other system unstabilities (long interrupts, CPU exhausted, etc.) and make sure our experiment runs in good conditions:

```
def cycle():
  sleep(0.1)
```

Coding the loads To make it easy, we'll write a single executable that will implement the many kinds of load (innocent, offender, control) we'll need, and the kind (cycle function to use) will be specified via command-line arguments.

Of course, we'd like all of these parameters (block size, sleep time, . . .) to be configurable for every process. We'll also need them to operate on different files to avoid colliding, as we might wish to have multiple innocent loads of the same kind.

Also, experiments need a stopping condition. We could define a duration, but —for reasons explained later— it was decided to just make the offender load stop after writing a number of bytes. The other loads can then be told to stop by interrupting them (i.e. pressing Control+C or sending SIGINT).

In addition, we'd like the offender load to start after some seconds so we can compare how the system was before applying pressure. We could run the load at a later time, but to simplify the experiment setup as much as possible, we decided to code this into the load, making the first cycle be a sleep for a configurable number of seconds.

The kinds of load that were initially implemented are described in table 2. Note that we used a `load` identifier for the offender kind. It would have been less confusing to use `offender`, but at the point this was realized it was difficult to change.

Also note that there's two kinds of innocent loads, **write** and **multiwrite**. The second one is

Kind	Parameters	Cycle behaviour	Notes
control	idle	sleep for `idle`	
load	f, wait, size, block	write of size `block` to file `f`	first cycle sleeps for `wait` stops after writing `size` bytes
write	f, idle	small write to file `f` sleep for `idle`	
multiwrite	f, idle	open new file `f + suffix` small write to file close file sleep for `idle`	`suffix` is `".0"`, `".1"`, etc.

Table 2: Kinds of load initially implemented for experiments

similar to the first, but writes in a (newly created) different file each time. Therefore, its cycle consists of an `openat` followed by a `write` and sleep. It was implemented after some preliminary tests that seemed to indicate `openat` is more prone to blocking than `write`.

Now that we've defined the behaviour of the load executable and how it is parameterized, it's time to code it. We chose **Python** for the task, because it is high-level and easy to work with, but at the same time offers an extense API that still maps pretty much one-to-one with syscalls. For situations where we'd need precise control, it ships with `ctypes`, a low-level FFI interface. For this reason, low-level exploits are often implemented in Python.

First we need to define and parse the command-line arguments, which will be:

```
./load.py <name> <kind> [<parameter>...]
# example load invocations:
./load.py c1 control '0.05'
./load.py mw1 multiwrite /tmp/mw1 '0.1'
./load.py l1 load /tmp/load1 43 32000000 512
```

And at the end, we'll use `pickle` to serialize the times into a file named after the load's name. This file will contain additional info, such as the kind and parameters, and the PID (so we can later correlate it with kernel debugging info):

```
with open('load.{}.pkl'.format(name), 'wb') as out:
    pickle.dump({
        'pid': os.getpid(),
        'kind': kind,
```

```
            'params': dict(params),
            'times': times,
        }, out)
```

Finally we used `strace` to make sure the loads behaved correctly from kernel's perspective (no extra syscalls, etc.):

```
$ strace ./load.py mw1 multiwrite /tmp/mw1 '1'
[...]
select(0, NULL, NULL, NULL, {tv_sec=1, tv_usec=0}) = 0 (Timeout)
openat(AT_FDCWD, "/tmp/mw1.2", O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC,
  ↪  0666) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
write(3, "test", 4)                        = 4
close(3)                                   = 0
select(0, NULL, NULL, NULL, {tv_sec=1, tv_usec=0}) = 0 (Timeout)
openat(AT_FDCWD, "/tmp/mw1.3", O_WRONLY|O_CREAT|O_TRUNC|O_CLOEXEC,
  ↪  0666) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
write(3, "test", 4)                        = 4
close(3)                                   = 0
select(0, NULL, NULL, NULL, {tv_sec=1, tv_usec=0}
```

Other than the extra `fstat` issued after opening the file, all kinds seemed to behave correctly. The full source code of `load.py` is listed at appendix A.

3.3 Virtual environment

At this point, we just need a main script to run a set of loads and stop them at the end. We'll create a dedicated directory for each experiment we run. Listing 5 shows a base experiment script we prepared, which includes two offenders kicking in at different times, two innocent loads (write & multiwrite) and a control one.

This is good to get started, but we'd like to have a better control of the environment. A virtual machine of some sort would be ideal here, because it would make our experiments much more *reproducible* by offering:

- Precise control of almost every parameter, including system memory, cache thresholds, filesystem type & size, bandwidth throttling. . .
- Kernel flexibility: it makes it easy to test many kernels (or kernel modifications) and gives us precise control of the config; we may enable / disable whatever modules or knobs we might need.

```
#!/bin/bash
set -e

# create experiment directory
EXPDIR="exps/$(date +%Y-%m-%d_%H-%M)"
mkdir "$EXPDIR"; cd "$EXPDIR"
echo "Starting experiment: $EXPDIR"

# start loads
mkdir loads
./load.py c1 control '0.05'
./load.py w1 write loads/write1 '0.05'
./load.py mw1 multiwrite multiwrite1 '0.1'
./load.py l2 load loads/load2 43 12000000  512
./load.py l1 load loads/load1  5 30000000 1024

# wait for main load to end, stop rest, clean up
wait -n; jobs -p | xargs kill -INT; wait
rm -r loads
echo "-- Done --"
```

Listing 5: Base experiment session

- Isolation: only our processes are running, many experiments can run in parallel, it's difficult for an experiment to alter the next, . . . This also means less junk data when we start tracing kernel events.
- Safety: we'll be doing writing high-speed during a long time, which can burn out the disk. A VM lets us simulate one from memory.

While difficult, if we were able to reproduce realist unresponsiveness inside a VM, it would make things *incredibly* easier. So, trying it seemed worthy. From the options we researched, **User Mode Linux** (detailed in section 2.5) seemed the most ideal one. It has:

- Portable & compact setup: Unlike other VM solutions, there's no need for root, dedicated software, loading kernel modules or specific system configuration steps. Everything can run right off from a script.
- Easy to use: With use of some clever techniques, there's no need to prepare a rootfs or disk drive image to boot off. UML also integrates effortlessly with the host filesystem. This also means it's. . .
- Automatable: We'll likely start lots of tests; it would be good to reduce the amount of effort & steps needed to run one. UML is very easy and intuitive to automate, allowing us to reduce this effort to zero!

So, we start by cloning Linux and compiling a stock UML kernel:

```
git clone https://github.com/torvalds/linux.git && cd linux
ARCH=um make defconfig
ARCH=um make
```

We can now run it (`./linux`) and verify it boots correctly. However it needs a root FS. Typically we would download an image from the Internet and load it as a block device, but there's a clever way to avoid this altogether. Remember UML has hostfs which exposes the host files. Well, we can specify this as the root filesystem:

```
./linux rootfstype=hostfs init=/bin/bash
```

It works! This should be considered a hack, because of hostfs drawbacks mentioned in section 2.5, and should **not** be used to boot a real `init` because it will attempt to modify your actual system's files. But in our case, coding a custom `init` is fine! It will require some extra setup & cleanup code, but will give us much more control and will make everything more compact.

So, the idea is to create an `experiment.py` script that will:

1. Launch the kernel, specifying *itself* as `init`.
2. When launched as init inside the kernel (which can be detected by checking PID = 1), set

up the machine.

3. Start the loads.
4. Wait for them to finish; clean up.

This will work as long as there's no spaces or other control characters in the path we're working on. Some init-related tasks we should totally do are: mounting procfs, sysfs and the like, and syncing data before finishing.

A template script that performs all this is presented in listing 6. We chose Python as well, for the same reasons. Also note that we can't just exit when we're done, that's a kernel panic. To do things correctly, we should *power down* the kernel. This is done via the ominous `reboot` syscall, which is only supposed to be invoked by init and not exposed to Python, so we had to use the FFI to call it. Any exceptions result in kernel panic and get correctly propagated to the main script, which is nice.

Putting it together Now that we have the tools we need, we can finally put together the script to launch the UML, set it up and start the loads. We need to create a block device that will hold the filesystem that loads will operate on. It must be in RAM to avoid burning out our disk. We'll use the following approach:

- Create a temporary file at `/var/run/user/` (which unlike `/tmp` is guaranteed to be a tmpfs) which will hold the filesystem on which loads will operate. Allocate 130 MB of space to it. Initialize it with 0xFF bytes, and format it as **ext4**. Pass it as `ubdb=file`.
- When inside the kernel, mount the block device: `mount /dev/ubdb /mnt`
- Through the root blkio cgroup, throttle the write bandwidth to this block device to 1 MB/s. This would be done in Bash as follows:

```
mount -t tmpfs  none /sys/fs/cgroup
mount -t cgroup none /sys/fs/cgroup/blkio -o blkio
echo `stat -c%t:%T /dev/ubdb` 1000000 \
  > /sys/fs/cgroup/blkio/blkio.throttle.write_bps_device
```

We studied other approaches which would behave more closely like a real disk, such as doing the throttling in the host, or writing a fake block device driver, but ultimately decided to go with that one.

Other things to keep in mind:

- Before launching the kernel, create an experiment directory. Pass this directory as an `init` argument into the inner script and change to it.
- We should also remember to adjust the amount of memory when launching the kernel to

```python
from subprocess import run
from os.path import dirname, join, abspath
import ctypes
libc = ctypes.CDLL(None)

def main():
  print('Running kernel...')
  run(check=True, args=['./linux', 'rootfstype=hostfs', 'rw',
                        'init=' + abspath(__file__) ])
  print('Done!')

def inside_kernel():
  print('Inside kernel!')
  run(check=True, args=[ 'mount', '-t', 'proc', 'none', '/proc' ])
  run(check=True, args=[ 'mount', '-t', 'sysfs', 'none', '/sys' ])
  # ...

  # TODO: perform experiment

  # power down
  libc.syncfs(os.open('/', 0))
  libc.reboot(0x4321fedc)

if os.getpid() == 1:
  inside_kernel()
else:
  main()
```

Listing 6: Barebones script that runs itself inside a UML kernel

150 MB, because it's an important parameter since it indirectly determines the writeback cache thresholds. We will leave these thresholds set to their defaults (background 10 %, normal 20 %).

All of these sizes should be configurable parameters of the experiment as well. Before starting the loads, we'll save them in `experiment.json` together with kernel version and some other info, to make it easy to reproduce the experiment in the future:

```
with open('experiment.json', 'w') as f:
  json.dump({
    'kind': 'uml',
    'start': datetime.datetime.now().isoformat(),
    'kernel': os.uname().release,
    'write_bps': write_bps,
    'dev_size': dev_size,
    'memory': memory,
  }, f, indent=4)
  f.write('\n')
```

At this point, and not after a few fixes, we have a fully functional experiment script that prepares the filesystem, boots the kernel, starts the experiment and cleans up. Nice!

Kernel tracing We'd also like to incorporate data from the kernel tracer, which was previously explored in sections 2.3 and 3.1. This time we don't want to visualize it in real time, we just need to log it among with the rest of the experiment's data.

`trace-cmd` can do that for us, as well as setting up the tracer, and using `splice` to make sure the kernel moves the data to the disk without any userspace intervention. However because the tracer's data will be logged to disk, and probably interact with the writeback cache, there is potential for this logging altering the results of the experiment. Ideally, this data should be sent over a set of network sockets (again using `splice`) and written on another machine[4]. However we'll only be logging a few events for now, and the data volume was confirmed to be low. Experiment data is in another filesystem as well.

So, we'll modify `experiment.py` to start up `trace-cmd` before the loads:

```
add_task('trace-cmd record -e balance_dirty_pages -e
  ↪  global_dirty_state')
time.sleep(3) # wait for it to start up
```

And then kill it afterwards, and wait for it to finish. We'll also check for dropped tracer events

[4]At a later time we realized that `trace-cmd` has this feature built in, and with reasonable effort we could probably have implemented it, starting the receiver outside of the UML.

```
base = '/sys/kernel/tracing/per_cpu'
for cpu in os.listdir(base):
    with open(join(base, cpu, 'stats')) as f:
        stats = dict( re.fullmatch(r'(.+?): (.*)',
          ↪  x.rstrip()).groups() for x in f )
    lost = int(stats['overrun']) + int(stats['dropped events'])
    if lost:
        raise Exception('{} lost events on {}'.format(lost, cpu))
```

Listing 7: Checking for dropped tracer events

before exiting, see listing 7. Since we're using `splice` this shouldn't happen, but just in case.

We run a full experiment and verify that we end up with `trace.dat` along with the rest of the expected files:

```
$ ./analysis/experiment.py
Preparing experiment at: /home/alba/Documents/tfg/exp/_06-16_12-17
-- Creating block device --
-- Formatting --
mke2fs 1.45.6 (20-Mar-2020)
/run/user/1000/tmplvhupg6q contains `ISO-8859 text, with very long
  ↪  lines, with no line terminators' data
Es descarten els blocs del dispositiu: fet
S'està creant un sistema de fitxers amb 133120 1k blocs i 33320
  ↪  nodes-i
UUID del sistema de fitxers=0773a3aa-099c-4621-8f81-ee481abc58a8
[...]
Escriptura de la informació dels superblocs i de comptabilitat del
  ↪  sistema de fitxers:fet

-- Launching kernel --
Core dump limits :
    soft - NONE
    hard - NONE
Checking that ptrace can change system call numbers...OK
Checking syscall emulation patch for ptrace...OK
Checking advanced syscall emulation patch for ptrace...OK
Checking environment variables for a tempdir...none found
```

```
Checking if /dev/shm is on tmpfs...OK
Checking PROT_EXEC mmap in /dev/shm...OK
Adding 6578176 bytes to physical memory to account for exec-shield
   ↪  gap
Linux version 5.6.0-rc7-alba1-01120-g83fd69c93340 (alba@alba-tpi)
   ↪  (gcc version 9.3.0 (Arch Linux 9.3.0-1)) #3 Sat Jun 13
   ↪  01:38:53 CEST 2020
Built 1 zonelists, mobility grouping on.  Total pages: 39380
Kernel command line: mem=150M root=/dev/root rootfstype=hostfs rw
   ↪  ubdb=/run/user/1000/tmplvhupg6q
   ↪  init=/home/alba/Documents/tfg/analysis/experiment.py --
   ↪  /home/alba/Documents/tfg/exp/_06-16_12-17
Dentry cache hash table entries: 32768 (order: 6, 262144 bytes,
   ↪  linear)
Inode-cache hash table entries: 16384 (order: 5, 131072 bytes,
   ↪  linear)
mem auto-init: stack:off, heap alloc:off, heap free:off
Memory: 144708K/160024K available (3282K kernel code, 1153K rwdata,
   ↪  1116K rodata, 145K init, 175K bss, 15316K reserved, 0K
   ↪  cma-reserved)
NR_IRQS: 16
[...]
printk: console [tty0] enabled
Initializing software serial port version 1
printk: console [mc-1] enabled
Failed to initialize ubd device 0 :Couldn't determine size of
   ↪  device's file
VFS: Mounted root (hostfs filesystem) on device 0:14.
devtmpfs: mounted
This architecture does not have kernel memory protection.
Run /home/alba/Documents/tfg/analysis/experiment.py as init process
random: python3: uninitialized urandom read (24 bytes read)
-- Inside kernel --
-- Mounting things --
-- Throttling bandwidth --
-- Mounting FS --
EXT4-fs (ubdb): mounting ext2 file system using the ext4 subsystem
EXT4-fs (ubdb): mounted filesystem without journal. Opts: (null)
ext2 filesystem being mounted at /mnt supports timestamps until
   ↪  2038 (0x7fffffff)
-- Preparing --
```

```
-- Starting experiment --
Hit Ctrl^C to stop recording
random: python3: uninitialized urandom read (24 bytes read)
random: python3: uninitialized urandom read (24 bytes read)
random: python3: uninitialized urandom read (24 bytes read)
random: fast init done
------------[ cut here ]------------
WARNING: CPU: 0 PID: 0 at lib/refcount.c:28
  ↪  refcount_warn_saturate+0x13f/0x141
refcount_t: underflow; use-after-free.
[...]
---[ end trace d6f8d67503c36ee4 ]---
CPU0 data recorded at offset=0xb2000
    5746688 bytes in size
-- Powering off --
reboot: System halted

-- Completed successfully --
Results at: 06-16_12-17

$ ls exp/06-16_12-17
experiment.json  load.l1.pkl  load.mw1.pkl  trace.dat
load.c1.pkl      load.l2.pkl  load.w1.pkl
```

The full source code of `experiment.py` is listed at appendix A and may be found in the submitted annex as well, under the `analysis` folder.

Other considerations As can be seen in the previous log, a trace appears in the middle of the experiment, and it seems to be related to the UBD driver:

```
------------[ cut here ]------------
WARNING: CPU: 0 PID: 0 at lib/refcount.c:28
  ↪  refcount_warn_saturate+0x13f/0x141
refcount_t: underflow; use-after-free.
Modules linked in:
CPU: 0 PID: 0 Comm: swapper Not tainted
  ↪  5.6.0-rc7-alba1-01120-g83fd69c93340 #3
Stack:
 6047eed0 6033c60b 00000009 6047ef60
 00000000 6025a5fd 6047eee0 6033c650
 6047ef40 6003a8d7 6047ef30 1c6004f0b1
```

```
Call Trace:
 [<6006dc16>] ? printk+0x0/0x94
 [<6001f67a>] show_stack+0x13b/0x155
 [<6033c60b>] ? dump_stack_print_info+0xdf/0xe8
 [<6025a5fd>] ? refcount_warn_saturate+0x13f/0x141
 [<6033c650>] dump_stack+0x2a/0x2c
 [<6003a8d7>] __warn+0x107/0x134
 [<6022b1bb>] ? blk_queue_max_discard_sectors+0x0/0xd
 [<6003ada9>] warn_slowpath_fmt+0xd1/0xdf
 [<6003acd8>] ? warn_slowpath_fmt+0x0/0xdf
 [<6001f739>] ? timer_read+0x10/0x1c
 [<6024a1ee>] ? bfq_put_queue+0x40/0x1f9
 [<600789a6>] ? raw_read_seqcount_begin.constprop.0+0x0/0x15
 [<6003192f>] ? os_nsecs+0x1d/0x2b
 [<6025a5fd>] refcount_warn_saturate+0x13f/0x141
 [<6022f676>] refcount_sub_and_test.constprop.0+0x32/0x3a
 [<602302ca>] blk_mq_free_request+0xf6/0x112
 [<602303f2>] __blk_mq_end_request+0x10c/0x114
 [<6002b1bc>] ubd_intr+0xb5/0x169
 [<6006ea31>] __handle_irq_event_percpu+0x6b/0x17e
 [<6006eb6a>] handle_irq_event_percpu+0x26/0x69
 [<6006ebd3>] handle_irq_event+0x26/0x34
 [<6006ebad>] ? handle_irq_event+0x0/0x34
 [<60072160>] ? unmask_irq+0x0/0x37
 [<600727c0>] handle_edge_irq+0xbc/0xd6
 [<6006e314>] generic_handle_irq+0x21/0x29
 [<6001ddd4>] do_IRQ+0x39/0x54
 [<6001de96>] sigio_handler+0xa7/0x111
 [<60030db5>] sig_handler_common+0xa6/0xbc
 [<6001f7a0>] ? timer_handler+0x0/0x48
 [<60031137>] ? block_signals+0x0/0x11
 [<60030e27>] ? timer_real_alarm_handler+0x5c/0x5e
 [<602576ae>] ? find_next_bit+0x1b/0x1d
 [<6007638e>] ? __next_timer_interrupt+0x78/0xd0
 [<600773d1>] ? arch_local_irq_save+0x22/0x29
 [<60077d8c>] ? hrtimer_get_next_event+0x59/0x61
 [<60031137>] ? block_signals+0x0/0x11
 [<600311c4>] unblock_signals+0x7c/0xd7
 [<60039e2b>] ? kernel_thread+0x0/0x4e
 [<6001e928>] arch_cpu_idle+0x54/0x5b
 [<60031137>] ? block_signals+0x0/0x11
```

```
 [<60350013>] default_idle_call+0x32/0x34
 [<6005f26f>] do_idle+0xaa/0x127
 [<6034d8e9>] ? schedule+0x99/0xdd
 [<6005f1c5>] ? do_idle+0x0/0x127
 [<6005f540>] cpu_startup_entry+0x1e/0x20
 [<6005f522>] ? cpu_startup_entry+0x0/0x20
 [<60053cf2>] ? find_task_by_pid_ns+0x0/0x2d
 [<6034c675>] rest_init+0xc2/0xc7
 [<6003121f>] ? get_signals+0x0/0xa
 [<60345994>] ? strlen+0x0/0x11
 [<600016bc>] arch_call_rest_init+0x10/0x12
 [<60001d26>] start_kernel+0x663/0x672
 [<600036be>] start_kernel_proc+0x49/0x4d
 [<6006b3d1>] ? kmsg_dump_register+0x70/0x78
 [<6001e3e4>] new_thread_handler+0x81/0xb2
 [<60003673>] ? kmsg_dumper_stdout_init+0x1a/0x1c
 [<60021049>] uml_finishsetup+0x54/0x59

---[ end trace d6f8d67503c36ee4 ]---
```

After some amount of effort, we were able to bisect this bug to patch `ecb0a83e3` which first appeared in Linux 4.20-rc1. To reproduce, `CONFIG_REFCOUNT_FULL` has to be enabled on old versions before it was made default and the option removed.

We notified the linux-um mailing list with that info. Nevertheless, the system continues working correctly after the trace and data is flushed to the block device, so we decided to move on.

3.4 Data analysis

After we can run experiments, we need a way to visualize the resulting data and see what's going on. We'll build a tool to render a **timeline** showing where the pauses occur on each load and how much they took. It should also plot the data from the kernel tracer, in a similar way to what we did in the monitor application (section 3.1). We'll use a Jupyter + NumPy + Matplotlib setup, which is typical in data science.

For the impatient, the finished product can be seen later in the analysis section, see figure 9 for an example. We'll now present the structure of the code in a simplified way, see the full source code for reference.

Basic timeline Let's omit the tracer data for now. Listing 8 shows code for loading data from a particular experiment, given the directory name of the experiment (timestamp). It populates `expdata` and a `loads` dictionary.

```python
import os
from os.path import join
import re
import json
import pickle
import numpy as np

expname = '04-11_05-01' # experiment to load

expbase = join('../exp', expname)
with open(join(expbase, 'experiment.json')) as f:
  expdata = json.load(f)
expfiles = os.listdir(expbase)
loads = {}
for x in expfiles:
  if m := re.fullmatch(r'load\.(.+)\.pkl', x):
    with open(join(expbase, x), 'rb') as f:
      load = pickle.load(f)
      load['times'] = np.array(load['times'])
      loads[m.group(1)] = load
```

Listing 8: Loading & parsing experiment data

Then the visualization code. First we need to extract the experiment's start and end timestamps; we can do that by looking at the initial and final times of one of the loads:

```
start, end = loads['c1']['times'][[0,-1]]
# all plotted timestamps will be relative to start
```

One way to visualize cycle times is to plot steps, one for each cycle. Each step will start & end when the cycle begins or ends, and the height of the step will also correspond to the cycle time. In other words, we'll be drawing a sequence of *squares* whose sides are the cycle times (of course, X and Y axis will be scaled differently so they will render as rectangles). We made a helper for that:

```
def render_segments(ax, edges, values, *args, **kwargs):
  if len(edges) != len(values)+1:
    raise Exception('Got {} edges, {} values'.format(len(edges),
      ↪ len(values)))
  res = ax.fill_between(np.repeat(edges, 2)[1:-1],
    ↪ np.repeat(values, 2), -1, *args, **kwargs)
  ax.set_ylim(0, ax.get_ylim()[1])
  return res

# ...
ax.set_title('cycle times of the c1 load')
ts = loads['c1']['times'] - start
render_segments(ax, ts, np.diff(ts))
```

Since our timeline will have different *panes* with a shared X (temporal) axis, and because many of these panes will share the same logic, we'll define functions to render each kind of pane given the `Axes` object and some parameters. This allows us to add, remove or reorder them easily. Listing 9 shows the panes for a control load, offender load, and innocent load respectively. On innocent loads, we subtract the idle parameter (sleep) from the cycle time. On offender loads, we discard the first cycle since it's a big sleep.

Once this is done, we can define a list of panes (their height, function and parameters) and create a blank figure using `subplots`. Then invoke the functions to render each axis, some other code to set up ticks and labels, and we get the result. This is shown in listing 10.

```
def p_control_task(ax, ln='c1'):
  load = loads[ln]; ps = load['params']; ts = load['times'] - start
  ax.set_title('{} control task (cycle time)'.format(ln))
  render_segments(ax, ts, np.diff(ts))
  ax.set_ylim(ps['idle']*(1-.05), ps['idle']*1.25)

def p_load_task(ax, ln='l1'):
  load = loads[ln]; ps = load['params']; ts = load['times'] - start
  ax.set_title('{} {} task (cycle time)'.format(ln, load['kind']))
  ts = ts[1:]
  render_segments(ax, ts, np.diff(ts))

def p_idle_task(ax, ln='w1'):
  load = loads[ln]; ps = load['params']; ts = load['times'] - start
  ax.set_title('{} {} task (cycle time minus idle)'.format(ln,
    ↪  load['kind']))
  render_segments(ax, ts, np.diff(ts) - ps['idle'])
```

Listing 9: Pane logic for load cycle times

```
panes = [
  (0.5, p_control_task, dict()),
  (1, p_load_task, dict(ln='l1')),
  (1, p_idle_task, dict(ln='mw1')),
  (1, p_load_task, dict(ln='l2')),
  (1, p_idle_task, dict(ln='w1')),
]
fig, axs = subplots(sharex=True, figsize=(10,8), tight_layout=True,
  ↪  nrows=len(panes), gridspec_kw=dict( height_ratios=[ x[0] for
  ↪  x in panes ] ))
axs[-1].set_xlim(0, end - start)
axs[-1].set_xlabel('time [s]')
axs[-1].xaxis.set_major_locator(mpt.MultipleLocator(10))

for ax, (_, fn, kwargs) in zip(axs, panes):
  ax.yaxis.set_major_formatter(mpt.EngFormatter(sep='', unit='s'))
  fn(ax, **kwargs)

for ax in axs:
  for load in loads.values():
    if load['kind'] == 'load':
      ax.plot([ load['times'][1] - start ]*2, ax.get_ylim(), lw=1,
        ↪  ls=(0, (4,4)), color='black')
```

Listing 10: Rendering the whole timeline

Tracer data Now it's time to add in the info from the tracer events. As mentioned in section 2.3, `trace-cmd` includes a Python interface. It's not very well maintained and we were unable to determine clearly if the intent of this interface was to provide a means to write plugins, or to be used as a library in regular Python scripts (our case), or both [18].

In any case, the Python bindings ended up installed at `/usr/lib/trace-cmd/python` but when we tried to load them, we got linking errors:

```
$ export PYTHONPATH=/usr/lib/trace-cmd/python
$ python
Python 3.8.1 (default, Jan 22 2020, 06:38:00)
[GCC 9.2.0] on linux
Type "help", "copyright", "credits" or "license" for more
   ↪  information.
>>> import ctracecmd
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: /usr/lib/trace-cmd/python/ctracecmd.so: undefined
   ↪  symbol: tracecmd_append_cpu_data
>>>
$ ldd /usr/lib/trace-cmd/python/ctracecmd.so
    linux-vdso.so.1 (0x00007fffbbd57000)
    libc.so.6 => /usr/lib/libc.so.6 (0x00007f31765a2000)
    /usr/lib64/ld-linux-x86-64.so.2 (0x00007f317685a000)
```

To get the library to load, we needed to look up the missing symbols and remove them from the source code (it looks as if the definition was removed, but some undefined references were left). We also needed to modify the Makefile so it would build correctly.

That wasn't enough, because the code was for a very old version of Python 2 that used things like `DictMixin`. After this and several other fixes, we got the code working and were able to open the `trace.dat` file and parse it. Horray!

```
import tracecmd
trace = tracecmd.Trace('path/to/trace.dat')
trace.read_next_event()
# => 3502.111919739 CPU0 global_dirty_state: pid=15091 comm=python3
   ↪  type=801 nr_dirty=b'\\xed\\...
```

The resulting patches to `trace-cmd` codebase can be found in `misc/trace-cmd.patch` in the submitted annex. The resulting (compiled) Python bindings are also included in `misc/tracecmd`. Then we need to add the appropriate code to the notebook to parse the events, see listing 11.

```python
import sys
sys.path.append('../misc/tracecmd')
import tracecmd

trace = tracecmd.Trace(join(expbase, 'trace.dat'))
events = []
while (event := trace.read_next_event()) != None:
    events.append(event)
events = sorted(events, key=lambda e: e.ts)
gds_events = [e for e in events if e.name == 'global_dirty_state']
bdp_events = [e for e in events if e.name == 'balance_dirty_pages']
```

Listing 11: Parsing the `trace.dat` file from Python

After that, we implemented two new panes in the timeline: one that shows the cache state, like the monitor application, and one that measures how many events per second are found of every event type. The implementations for these panes are shown in listing 12 and are rendered in the second and third positions of figure 9. The legend is a bit obstructing, so we'll remove it on most timelines.

We could also verify that the timestamps of the kernel tracer were synchronized with those of the loads, so they use `CLOCK_MONOTONIC` time as well. Now we have a decent amount of info in the timeline and we may move to analyzing the data and investigating what's going on.

Other changes The full source code (notebook file) is included in the submitted annex, see `analysis/timeline.ipynb`. We won't go into them, but several changes & improvements were made to the code at a later time in order to implement:

- Live experiments (these have no offender loads, see page 60)
- Close-up timelines (see figure 12)
- Experiment comparison (see figure 16)

```
def p_event_density(ax):
  ax.set_title('tracing events count')
  ax.set_ylabel('events / s')
  def plot_times(events, *args, **kwargs):
    if not len(events): return
    ts = np.array([ e.ts / 1e9 for e in events ]) - start
    bins = min(int((end - start) / .4), 300)
    weights = [bins / (end - start)]*len(ts)
    ax.hist(ts, range=ax.get_xlim(), weights=weights, bins=bins,
      ↪  log=True, histtype='step', *args, **kwargs)
  plot_times(gds_events, label='state')
  plot_times(bdp_events, label='balance')
  ax.set_ylim(1,100e3)
  ax.yaxis.set_ticks((1,1e1,1e2,1e3,1e4,1e5))
  ax.yaxis.set_major_formatter(mpt.EngFormatter(sep=''))
  ax.legend()

def p_dirty_state(ax):
  ax.set_title('dirty state overview')
  from_pages = lambda pages: pages * 4096
  ts = np.array([ e.ts / 1e9 for e in gds_events ]) - start
  ax.plot(ts, [ from_pages(int(e['background_thresh'])) for e in
    ↪  gds_events ], lw=1, ls=(0,(3,2)), color='green',
    ↪  label='background thresh.')
  ax.plot(ts, [ from_pages(int(e['dirty_thresh'])) for e in
    ↪  gds_events ], lw=1, ls=(0,(3,2)), color='brown',
    ↪  label='dirty thresh.')
  ax.plot(ts, [ from_pages(int(e['dirty_limit'])) for e in
    ↪  gds_events ], lw=1, ls=(0,(3,2)), color='red', label='dirty
    ↪  limit')
  ax.plot(ts, [
    ↪  from_pages(int(e['nr_dirty'])+int(e['nr_writeback'])) for e
    ↪  in gds_events ], color='black', label='dirty + writeback',
    ↪  zorder=-1)
  ax.plot(ts, [ from_pages(int(e['nr_dirty'])) for e in gds_events
    ↪  ], color='blue', lw=1, label='dirty', )
  ax.yaxis.set_major_formatter(mpt.EngFormatter(sep='', unit='B'))
  ax.set_ylim(0, ax.get_ylim()[1])
  ax.legend()
```

Listing 12: Pane logic for kernel tracer data

4 Experiments and results

With proper reasearch & tooling in place, we now walk through the performed experiments, results and the conclusions or theories drawn from them: this is the analysis phase (section 4.1). We'll then investigate ways to mitigate the side effects and develop a PoC (section 4.3).

It is advisable to read section 3 first, to be familiar with the experiment model and environment.

4.1 Analysis phase

UML tests We started performing experiments inside our UML kernel and analyzing them. We did several iterations which can be summarized in an experiment shown in figure 9. This experiment uses the configuration shown in table 3, and we can see:

- At $t = 0$ s: only the control & innocent loads are running. We can observe almost no I/O, and the cache growing very slowly.
- At $t = 5$ s: the first offender load starts kicking in, trying to write as much as possible. The dirty pages grow and reach the `dirty_limit` in a couple of seconds. We start observing **long pauses** (several seconds) interleaved by periods of zero throttling, on both the offender load and the multiwrite load, but not on the write load.
- At $t \approx 25$ s: The kernel starts lowering the `dirty_limit`, which alters the global throttling curve[5] (see figure 4) to apply more throttling to the processes. Long pauses stop and we start observing rate-limiting on the offender load (this can be seen through constant, but small, pauses on the `l1` pane; we can also observe how dirty pages grow much slower than before).
- At $t = 43$ s: The second offender load kicks in, but has zero impact on the rest of the system. It quickly gets rate-limited. The rate-limiting seems to be shared on both tasks, and is increased a bit to account for the new pressure.

We can also observe a 1 s pause at $t = 26$ s, but since this pause also affected the control load and event count panes, it can be discarded as it was probably caused by the UML kernel being stopped for a while at host level.

Interpretation There's some remarkable results to note in this experiment:

- We were able to reproduce long, unfair pauses in an innocent load caused by the presence of an offender process. Horray!
- These unfair pauses don't almost affect the `write` load, but do affect the `multiwrite` one. Further investigation reveals the `mw1` load is almost always being blocked on `openat`

[5]Note that our kernel was compiled with `CONFIG_BLK_WBT` disabled, and as such there should be no device-specific curve, only the global one.

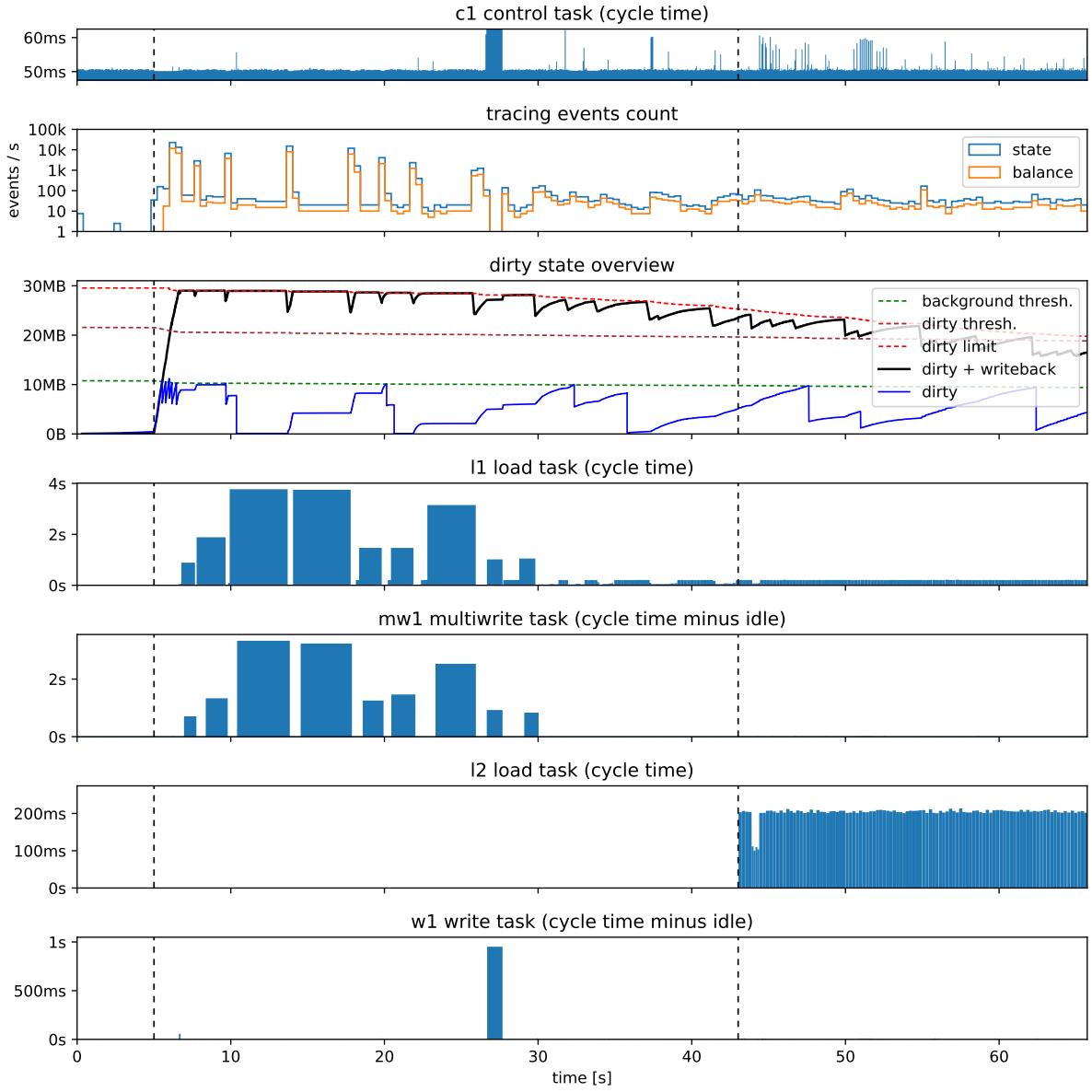


Figure 9: Timeline from UML base experiment

Parameter	Value
Kernel version	5.6.0-rc7
Write throughput	1 MiB/s
Filesystem size	130 MiB
Memory size	150 MiB

(a) Global environment parameters

Name	Kind	Sleep period	Start time	Total size	Block size
`c1`	control	50 ms			
`w1`	write (innocent)	50 ms			
`mw1`	multiwrite (innocent)	100 ms			
`l1`	load (offender)	–	5 s	65 MiB	1024 B
`l2`	load (offender)	–	43 s	26 MiB	512 B

(b) Load processes present

Table 3: Base configuration for UML experiments

syscalls (i.e. when opening each file). However, we *did* observe pauses on `w1` as well in some experiments, it's just much less probable.

- It takes a long time (20 s) for soft-throttling to kick in correctly, but after that **there are no unwanted pauses**. Innocent loads aren't blocked at all, not even soft-throttled. Maybe the throttling does have some amount of fairness?
- The `dirty_limit` throttling parameter doesn't (only) depend on configured ratios or free memory, but is adjusted through some unknown process, probably depending upon past I/O in a certain amount of time. We can see how the dirty threshold and background dirty threshold do not change.
- The `dirty_limit` starts actually higher than the dirty threshold, at about 135 % of the dirty threshold in this case... so the dirty threshold is *not* the actual limit.

We then performed several variations of the experiment to control for other factors. One of them was repeating the experiment but mounting the filesystem in `sync` mode. This disables the writeback cache, and allows us to make sure the long pauses are at least related to it. This didn't give the clearest results (figure 10) because there were much frequent machine-global pauses. Still, we get no unwanted pauses but increased latency due to synchronous operation.

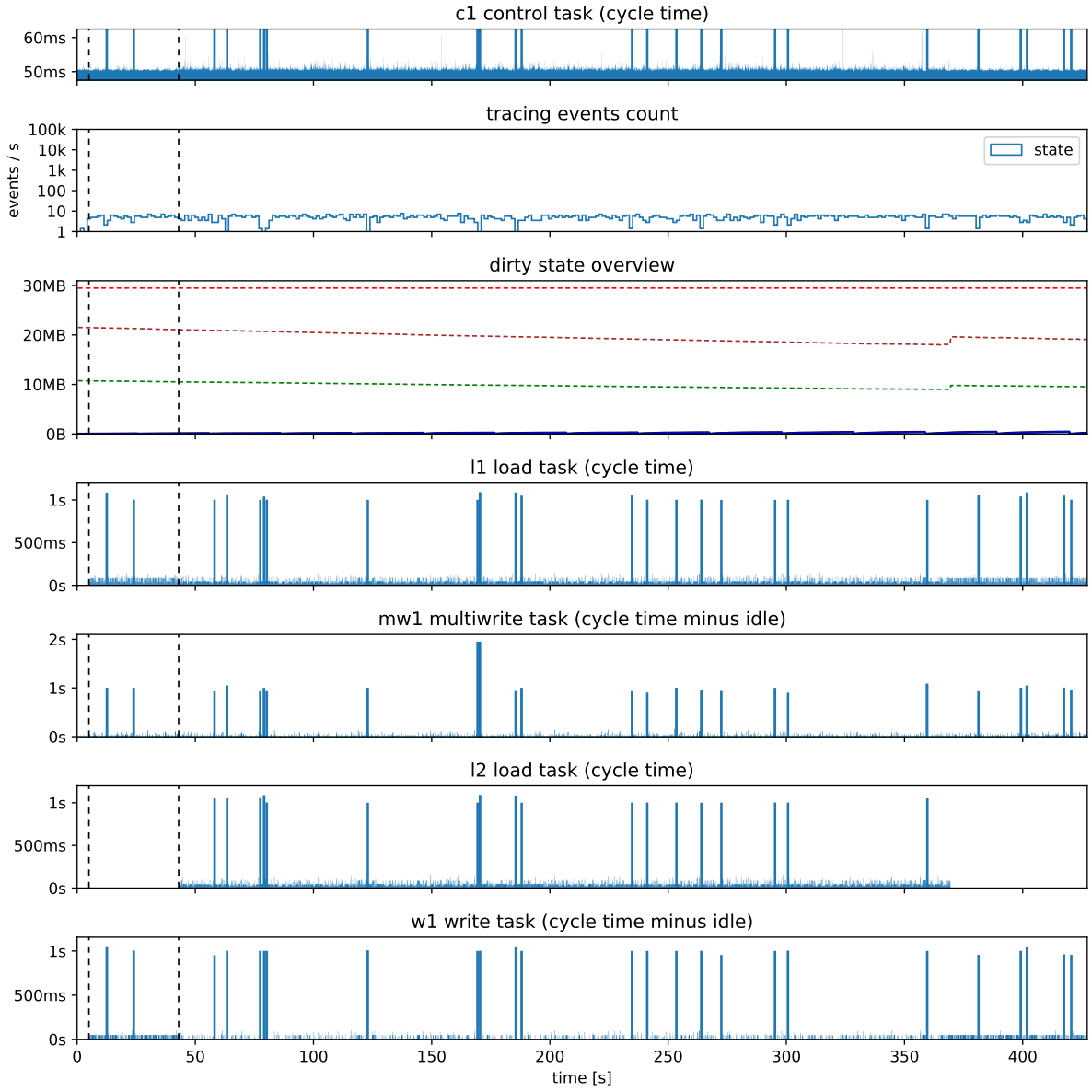


Figure 10: Timeline from UML experiment, but disabling the cache

Parameter	Value
Kernel version	5.6.4-arch1-1
Disk type	SSD
Disk throughput	(measured) 70 MiB/s
Filesystem size	130 MiB
Memory size	8 GiB
Main command	`pacman -Syu`

(a) Global environment parameters

Name	Kind	Sleep period
`c1`	control	50 ms
`w1`	write (innocent)	50 ms
`mw1`	multiwrite (innocent)	100 ms

(b) Load processes present

Table 4: Base configuration for live experiments

Live experiments This looked like a promising reproduction of the problem, but the environment —while highly controlled— is still different from a real one in terms of sizes: the cache is very small, the bandwidth is also small, ... Also, the offender load could be doing something different to what a normal load would do. Thus, we wanted to perform a more realistic experiment to see if it matched what we saw inside UML.

So, a stripped down version of `experiment.py` was made, which runs the loads directly on the host. Also, instead of running offender loads it starts an actual command (which in our tests was `pacman -Syu` to upgrade the system). This script is called `live_experiment.py` and its source code may be found in appendix A as well as in the `analysis` folder of the submitted annex.

Again we needed many iterations of the experiment to apply fixes and get usable data, one of which is shown in figure 11 and table 4 for the configuration. We won't go into a temporal description of the events like before, but there's an important thing to note: **no throttling is being applied, yet there are similar unwanted pauses**.

To verify that no throttling is ocurring, we can look at how no `balance_dirty_pages` events occur in the second pane, unlike in figure 9. And it's expected, since the dirty pages barely get to grow past the dirty background limit. Other things to note are: a much larger cache (1 GB), and that a lot more of these unwanted pauses affect the `w1` load as well.

So, on one hand we have confirmed how (in a more real environment) we observe these unwanted pauses, during which (as the lack of kernel tracer event shows) there is *almost no I/O* in the system. In fact, these pauses themselves reduce the offender's throughput enough that there's no need for actual throttling! But on the other, this experiment differs from our UML ones and seems to indicate the pauses have a different cause.

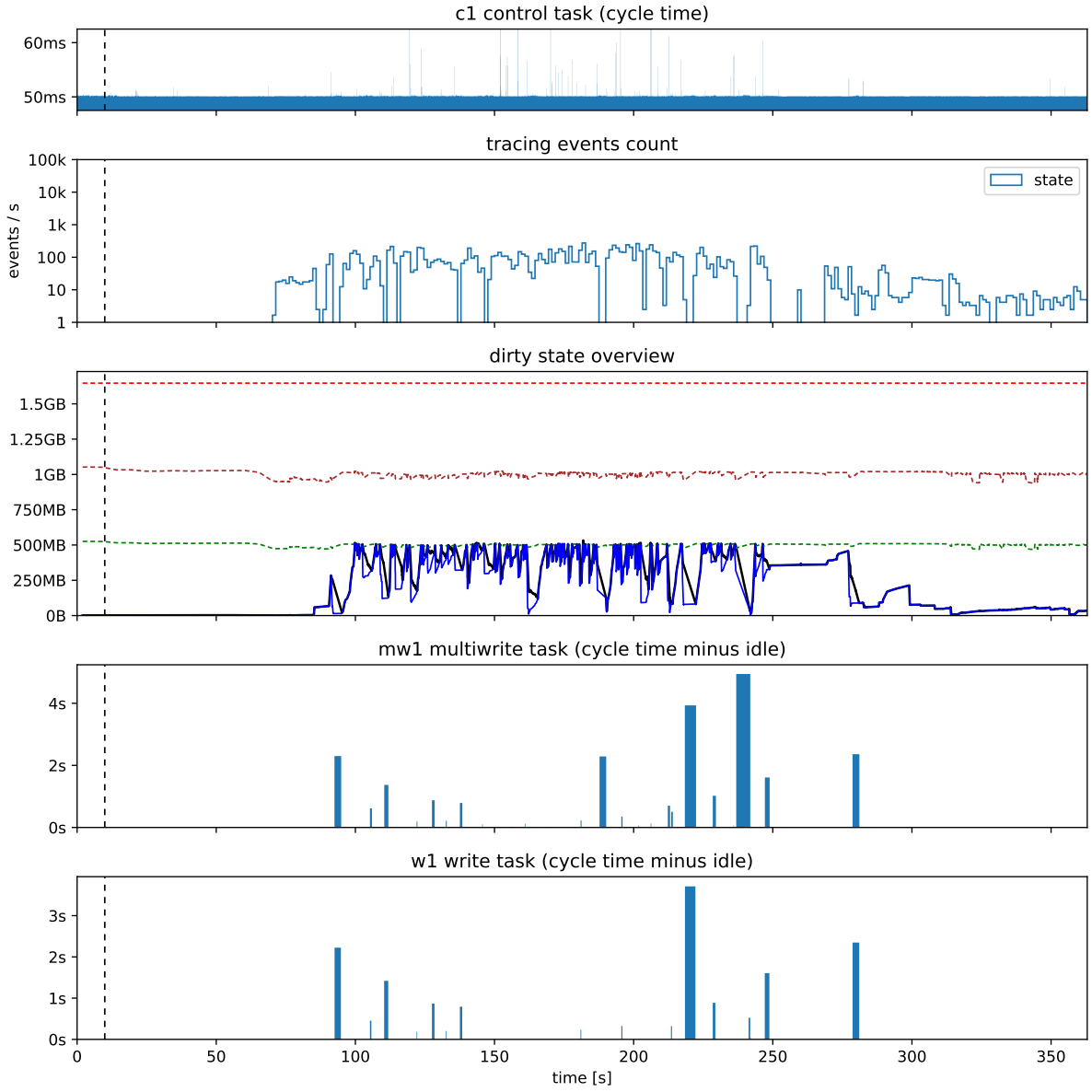


Figure 11: Timeline from a live experiment (system upgrade)

Looking closer We then attempted to get more data on what was happening in the live experiment. We decided to capture more data from the tracer, namely:

- All events in the whole `writeback` subsystem
- `syscalls` events to precisely track when possibly blocking syscalls (`write`, `openat`, `close`) start & end.

However these are a *lot* of events, especially on a real-life system. Since we are probably happy with just capturing a handful of pauses, we proceeded by manually turning the tracer on & off through the `tracing_on` file while `trace-cmd` was recording. We waited for a pause to happen, then turned the tracer on, and after around 20 s we got another pause. We then turned the tracer off.

This gave us a `trace.dat` dump of about 720 MB. With help of our visualization tool, we noted the timestamps of our pauses and cropped the `trace.dat` file to those timestamps, leaving some padding:

```
trace-cmd split -o trace-pause.dat 811900 811910
```

This gave us a more manageable 110 MB dump.

We first produced a zoomed-in version of the timeline to have some overview of the period, which is shown in figure 12. The cycle times of `mw1` are now represented by dashed vertical lines on every pane. The following seems to happen on most pauses:

1. The number of dirty pages reach the dirty background threshold.
2. The cache moves a lot (around 100 MB) of pages into writeback state (difference between black and blue line).
3. The innocent load gets out of sleep, attempts to do I/O and gets blocked (pause).
4. There is a period of almost no I/O, judging by the events count pane and the little growth in dirty pages. That also means we have less samples, but we can at least see how. . .
5. These writeback pages get fully written (the lines touch again)
6. I/O resumes, and *after some time*, our innocent load gets unblocked.

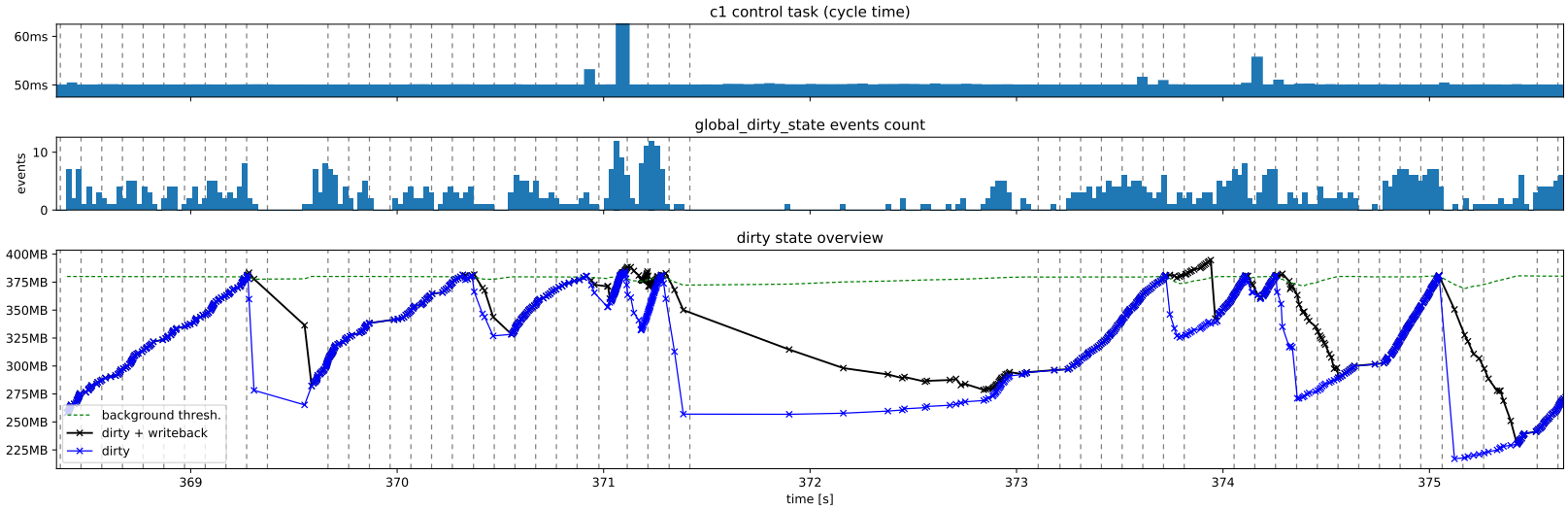


Figure 12: Timeline close-up when no measures are taken

253290	0	811901.172911	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195765 dirtied 4538446701 state mode 0>o<
253291	0	811901.172915	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195782 dirtied 4538446702 state mode 0>o<
253292	0	811901.172919	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195771 dirtied 4538446701 state mode 0>o<
253293	0	811901.172924	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195772 dirtied 4538446701 state mode 0>o<
253294	0	811901.172927	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195773 dirtied 4538446701 state mode 0>o<
253295	0	811901.172931	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195775 dirtied 4538446701 state mode 0>o<
253296	0	811901.172934	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195780 dirtied 4538446701 state mode 0>o<
253297	0	811901.172939	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195783 dirtied 4538446702 state mode 0>o<
253298	0	811901.173850	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195506 dirtied 4538446683 state mode 0>o<
253299	0	811901.175043	<...>	761863	d..2	sb_clear_inode_writeback	dev 254,2 ino 4195505 dirtied 4538446673 state mode 0>o<
253300	1	811901.178700	<...>	764162	...1	sys_exit_openat	0x3
253301	1	811901.178727	<...>	764162	...1	sys_enter_write	[FAILED TO PARSE] __syscall_nr=1 fd=3
253302	1	811901.178734	<...>	764162	....	writeback_mark_inode_dirty	bdi 254:2: ino=8808668 state= flags=I_DIRTY_SYNC|I_DIRTY_TIME

60837	2	811904.617810	gnome-shell	1255	...1	sys_exit_write	0x8
60838	3	811904.617887	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029394 dirtied 4538447541 state mode 0>o<
60839	3	811904.617892	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029396 dirtied 4538447549 state mode 0>o<
60840	3	811904.617896	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029350 dirtied 4538447473 state mode 0>o<
60841	3	811904.617905	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029353 dirtied 4538447479 state mode 0>o<
60842	3	811904.617911	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029356 dirtied 4538447484 state mode 0>o<
60843	3	811904.617916	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029360 dirtied 4538447489 state mode 0>o<
60844	3	811904.617920	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029397 dirtied 4538447550 state mode 0>o<
60845	0	811904.618011	<...>	762814	d..2	sb_clear_inode_writeback	dev 254,2 ino 6029373 dirtied 4538447516 state mode 0>o<
60846	1	811904.618592	<...>	764162	...1	sys_exit_openat	0x3
60847	1	811904.618616	<...>	764162	...1	sys_enter_write	[FAILED TO PARSE] __syscall_nr=1 fd=3
60848	1	811904.618622	<...>	764162	....	writeback_mark_inode_dirty	bdi 254:2: ino=8808686 state= flags=I_DIRTY_SYNC|I_DIRTY_TIME

Figure 13: Tracer events prior to finish of a long pause

This seems to imply that the cause for these pauses is the writeback *itself*, not the throttling. Remember that the VM layer works in terms of pages, but the VFS layers works with inodes, and the writeback cache must reconcile both. We inspected the `trace-pause.dat` file in Kernelshark, and among other things noticed that **the innocent load is always woken up just after inodes are cleared** according to the event log, see figure 13. The selected event is when the `openat` syscall in `mw1` exits.

It is also important to look at how there is a period (point 6) where high-rate I/O starts again, yet our innocent load remains blocked. Maybe the offender load has been unblocked first and starts writing again, but it is not obvious why this would happen.

The cause Indeed, a look at the kernel's source code seems to confirm that inodes are locked while in writeback state [19]:

> Inode state bits. Protected by `inode->i_lock` [. . .]
> Two bits are used for locking and completion notification, `I_NEW` and `I_SYNC`.
>
> [. . .]
>
> **I_SYNC**: Writeback of inode is running. The bit is set during data writeback, and cleared with a wakeup on the bit address once it is done. The bit is also used to pin the inode in memory for flusher thread.

And also [20]:

```
static void inode_sync_complete(struct inode *inode)
{
  inode->i_state &= ~I_SYNC;
  /* If inode is clean an unused, put it into LRU now... */
  inode_add_lru(inode);
  /* Waiters must see I_SYNC cleared before being woken up */
  smp_mb();
  wake_up_bit(&inode->i_state, __I_SYNC);
}
```

Knowing all this, we draw the conclusion that these unwanted pauses are a direct consequence of (a) having a large cache, and (b) inodes being locked while waiting for writeback. Large caches make it possible (and frequent!) for large amounts of data to be queued for writeback when the cache is flushed. This means some of those inodes can be locked for a long time before they're finally written, and this is probably what's blocking our innocent load too.

To be clear: the writeback cache is a complex component and the reality is probably a bit more complicated, but that conclusion means that any offender will cause long pauses not only on itself, but any system process that writes to the filesystem at that time.

The relevant variable here is the amount of time it takes to flush the cache, which gives a high bound for the pause time:

$$T_{flush} = \frac{\text{dirty background threshold}}{\text{disk throughput}} \simeq \frac{375\,\text{MB}}{70\,\text{MB/s}} \simeq 5.4\,\text{s}$$

Over the last couple of decades, RAM size has increased almost logarithmically [21]. Disk sizes & densisties have also increased, but disk *throughput* has remained almost the same. So around 2005, flush times were probably on the order of *tenths* of a second and didn't hurt responsiveness that much.

While doing initial research, we found people complaining about large caches on the Linux kernel mailing list around late 2013 [6]:

> My feeling is that vm.dirty_ratio/vm.dirty_background_ratio should *not* be percentage based, 'cause for PCs/servers with a lot of memory (say 64GB or more) this value becomes unrealistic (13GB) and I've already had some unpleasant effects due to it.

And this was Linus Torvald's response, which includes a rationale about the percentage ratios:

> Right. The percentage notion really goes back to the days when we typically had 8-64 *megabytes* of memory So if you had a 8MB machine you wouldn't want to

> have more than one megabyte of dirty data, but if you were "Mr Moneybags" and could afford 64MB, you might want to have up to 8MB dirty!!
>
> Things have changed.
>
> So I would suggest we change the defaults. Or pwehaps make the rule be that "the ratio numbers are 'ratio of memory up to 1GB' ", to make the semantics similar across 32-bit HIGHMEM machines and 64-bit machines.
>
> The modern way of expressing the dirty limits are to give the actual absolute byte amounts, but we default to the legacy ratio mode..
>
> Linus

Theodore Ts'o suggested to make it possible to specify the thresholds in terms of *seconds* (T_{flush} in the formula above) rather than absolute sizes:

> What I think would make sense is to dynamically measure the speed of writeback, so that we can set these limits as a function of the device speed. It's already the case that the writeback limits don't make sense on a slow USB 2.0 storage stick; I suspect that for really huge RAID arrays or very fast flash devices, it doesn't make much sense either.

This wasn't implemented, which is not surprising given there's no straightforward way to do it —there's only one cache, which may serve different disks, and determining or even defining the throughput of a disk isn't easy.

It *was* made possible to specify the thresholds in bytes, rather than integer percentages (see page 21), but it is not enabled by default and we are not aware of any distributions that take care of setting them up.

4.2 Mitigation phase

Once we have a clear enough understanding of the problem, we'll try to investigate ways to alleviate the problem. Note that the pauses reproduced inside the UML kernel (figure 9) seem to have a different nature than the pauses reproduced in the live experiments (figure 12), where no throttling is even triggered.

The first objective involves altering our tests from the analysis phase, trying different approaches until we measure a consistent reduction in pauses experienced by innocent loads.

Once (and if) we find a mechanism to reduce these pauses, we'll then automate it into a *userspace daemon* to apply it automatically on a live system, whenever it detects an offender task.

Initial efforts It's important to note that this thesis isn't entirely in chronological order. Due to time / workplan constraints, this PoC development phase was performed partially in parallel

with the analysis phase. Before the cause of the long pauses was pinned down we tried many approaches, including:

- CPU-limiting the offender loads.
- Limiting their memory consumption using `memcg` cgroups (which should theoretically limit their allowed amount of dirty pages, indirectly).
- Throttling their block I/O bandwidth using `blkio` cgroups.

As well as combinations of multiple restrictions. All this didn't seem to have statistically significant effects on pauses.

Later, when it was discovered that the origin of those pauses (in live experiments) was due to cache flushes, we moved our efforts into live experiments. After a bit of more research into the block I/O layer, we tried using the BFQ I/O scheduler.

BFQ experiments As explained in section 2.4, BFQ is a fair I/O scheduler. It works by reserving the disk to one task for some time, then switching to the next. By adjusting the proportion of time assigned to each task using *weights*, both bandwidth and latency can be theoretically controlled. BFQ also supports *hierarchical distributions* and different *queues* which are served in strict priority order.

Thus, we switched from our default `mq-deadline` to using BFQ:

```
echo bfq > /sys/block/sda/queue/scheduler
```

Note: Not all block device drivers use I/O schedulers; some of them handle BIOs directly. In our case, the root filesystem was mounted on an LVM volume, which uses the *device mapper* and is an example of such a device. Since BIOs are eventually forwarded to the disk (`sda`) device, the effect should be the same as if the FS was directly mounted there. Our understanding of the block I/O layer is still limited though, so we could be wrong.

Given the origin of our pauses, it seems intuitive that lowering the priority of the offender tasks should allow flushed writes from other tasks to be processed first, thus reducing pauses. This is what we tried next.

According to BFQ's documentation, the scheduler automatically detects interactive tasks and assigns higher weights to them by default [14]. This seems to be in line with what we're looking for; however, we're not just looking for higher weights for our innocent tasks, but for their requests to be served *first*. Some preliminary experiments confirmed that simply switching to the scheduler didn't seem to reduce pauses substantially.

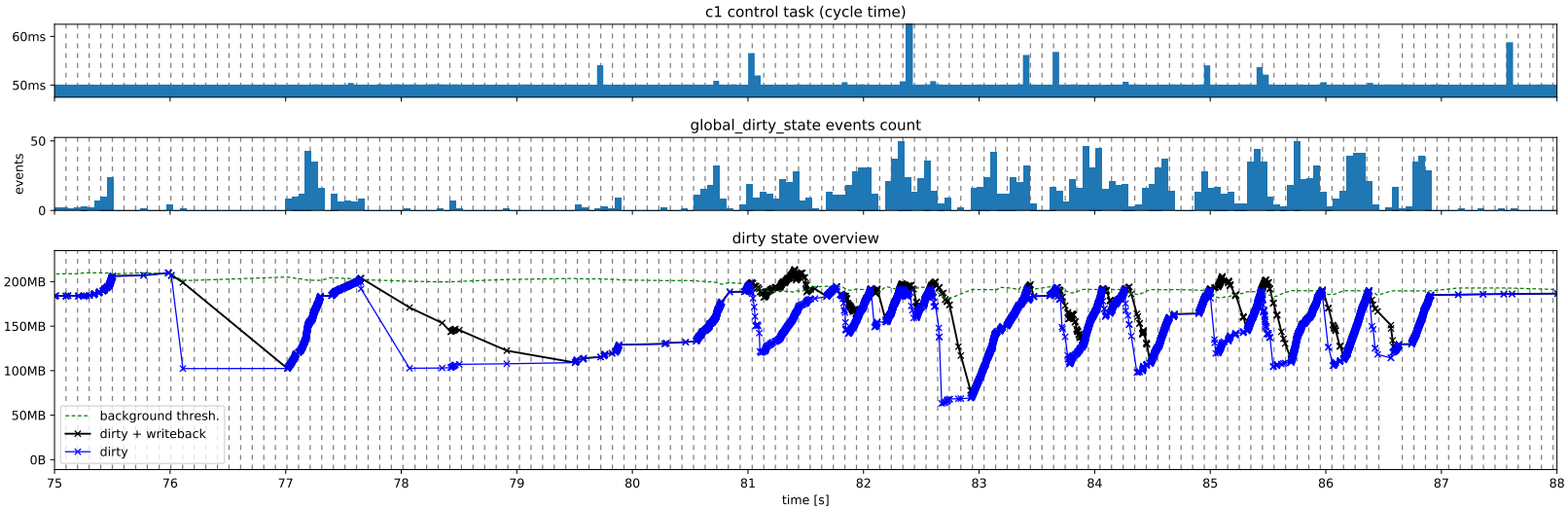


Figure 14: Timeline close-up when using BFQ and the `idle` class

BFQ distributions can be set through `blkio` cgroups, like we did before, but it can also be controlled through per-process I/O priorities. These priorities can be managed through the `ionice` command or programmatically using `ioprio_get` and `ioprio_set`. It's important to note, though, that these "I/O priorities" usually control the *weight* of the task, not its actual priority queue. To place the task on a different queue, we have to supply a different I/O class when setting the priority to the process.

Thus, in addition to using the BFQ scheduler, we also tried starting the offender load in the `idle` class. This should cause their requests to be the last ones to be processed, hopefully reducing innocent pauses. We modified `live_experiment.py` to launch the main command prefixed with `ionice -c idle`, and then performed some experiments.

The results seemed to show that **pauses were greatly reduced**. Figure 14 shows a close up of one of the live experiments. Compared to figure 12, we can appreciate how there's still periods of no I/O in the system (at 76 s to 77 s, at 77.7 s to 79.5 s, and at 82.7 s) but our innocent load still runs unaffected, with its operations completing immediately!

There *is* a 170 ms pause near the end of the timeline though, but these were rare. Figure 15 shows the overview of another experiment, and again, the biggest pause we can see is 585 ms long, followed by 195 ms and 127 ms. Disks have internal caches, background processes, may reorder requests, etc. and this is a probable reason why pauses do not entirely go away.

Final results To better measure & compare the effects of different approaches to reducing pauses, we ran these tests in a different machine. This machine has a *much slower disk* which, together with the default cache size, results in pauses that can easily reach 40 s. It is an extremely large cache and should give us a clearer comparison of both approaches. When performing comparisons, it's important to account for cache size and written data.

After running the experiments, we created an histogram plotting the longest pauses experienced in every experiment, together with some parameters. One of them is the **wait time** which measures, given a random instant, how much we have to wait for the current pause to end. The results can be seen in figure 16.

We can see how simply enabling BFQ *does* appear to reduce the longest pauses a bit, but setting the `idle` class on the offender is needed for substantial effects: it brings the longest pause to half the time, achieves **210 % reduction on wait time** and also a 30 % reduction on the total time the innocent load stays paused for.

We also attempted to use the machine while the experiments were running, and can confirm that it was barely possible except when putting the offender in the `idle` class. Thus, while far from a perfect solution, we can verify that lowering I/O priorities substantially improves system responsiveness.

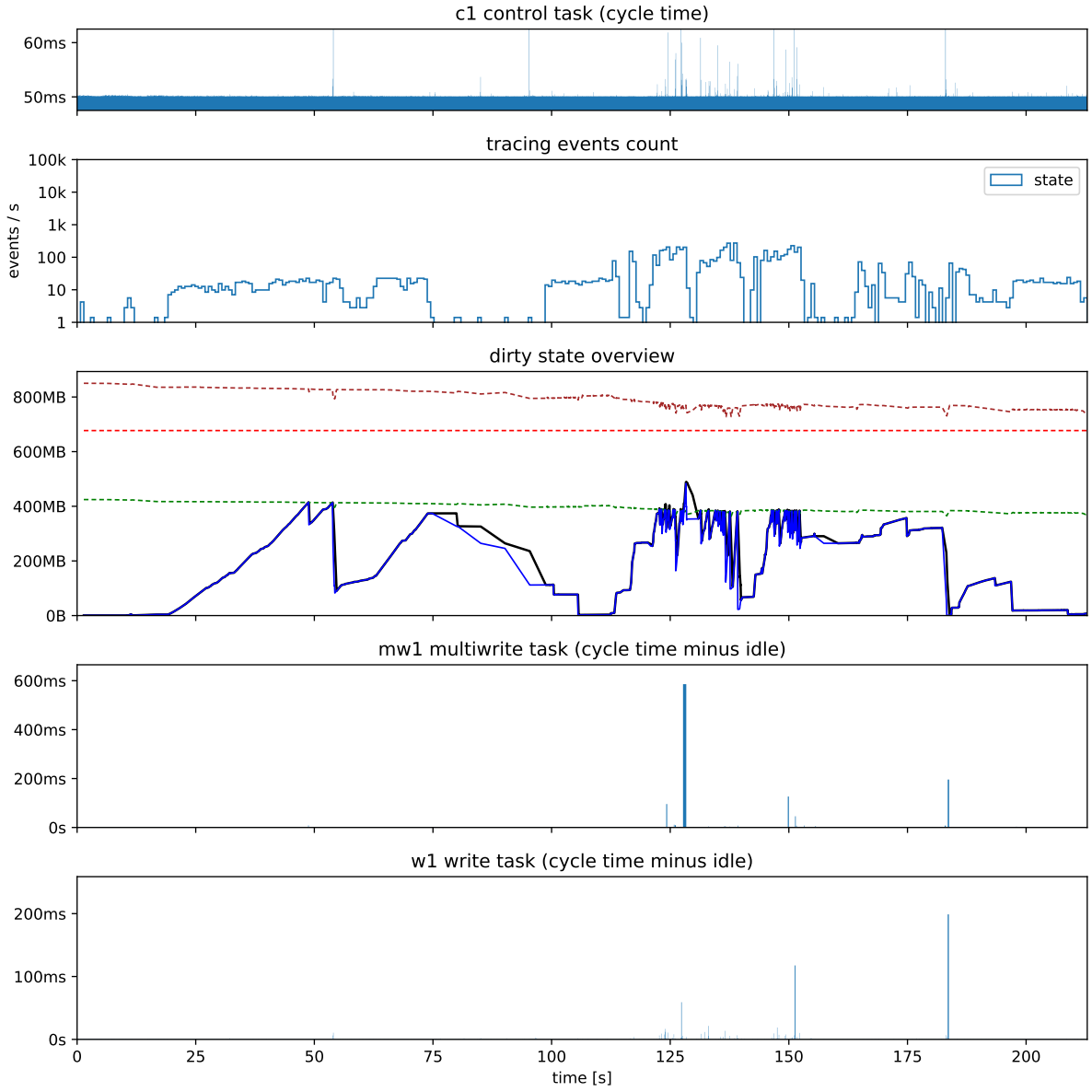


Figure 15: Timeline from a live experiment using BFQ and the `idle` class

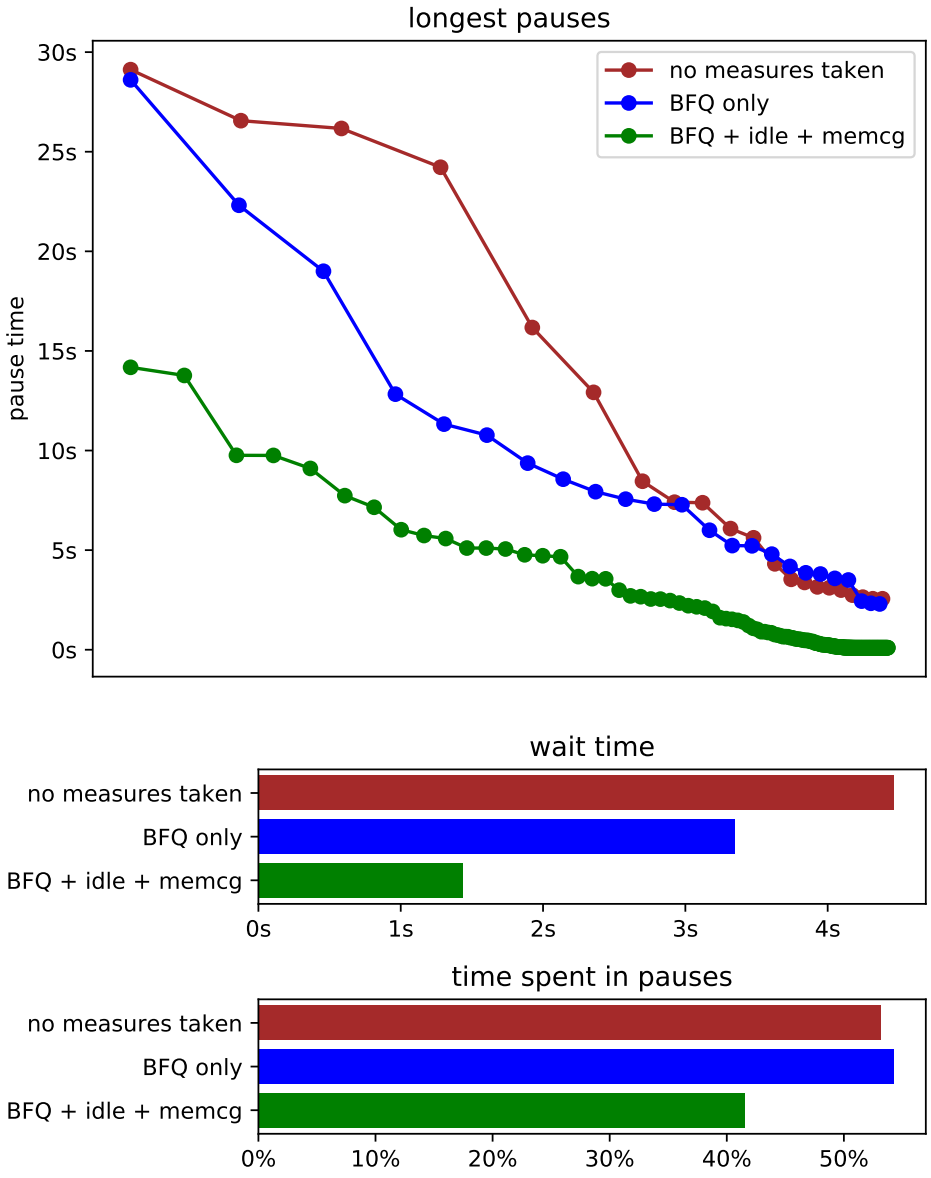


Figure 16: Pause comparison before / after enabling BFQ, on a large cache

4.3 PoC development

The idea is now to create a daemon that will constantly monitor tasks in the system, detect offenders among them (according to some criteria) and lower their I/O priority to reduce the adverse effects on the rest of the system.

Initial design Since an event loop seems ideal for this task, we'll choose Node.JS to code the daemon as we did in the realtime monitor application (section 3.1).

First we need to detect which tasks are offenders. We can use the **taskstats interface** presented in section 2.4, which among other things lets us track the total read & write I/O performed by each process in real time. We'll use our previous node_netlink project to access this interface from Node.

The intuitive way to proceed is to choose a window (of, let's say, 5 s) and filter the write I/O of every process through that window. If this filtered number surpasses a certain threshold (sustained I/O) then we identify this process as an offender. Then, when an offender drops below another threshold (leaving an hysteresis margin) we'd no longer consider it an offender.

However there's many pitfalls to watch over:

1. We don't know what filesystem these writes are going to, or if they are going through the page cache. They *do* seem to report disk I/O, after looking through `iotop` source code and doing some quick tests.
2. Process-level detection may not be accurate. One process may start many workers to do the I/O.
3. An offender may also may be constantly starting subprocesses to do I/O, which would never be detected by our long window.

Pitfalls (2) and (3) can be solved by recursively aggregating stats from children into their parents, with a decay factor (otherwise we'd end up picking the root node in the tree). taskstats informs us of processes statistics when they die, so this aggregation can be done without races.

To solve pitfalls (1) and (2) we could use the kernel tracer, the blktrace API, or hook up a BPF program (which would also be a much more efficient way to collect the info) to the writeback tracepoints. Due to our limited scope, we ended up continuing with the taskstats approach.

We'll use exponential smoothing instead of a FIR window because it's easy to implement and requires no extra memory. We'll choose a time constant of $\tau = 8$ s, and will obtain samples every $T_s = 1$ s. We can then obtain the decay coefficient $\alpha = 1 - e^{-T_s/\tau}$.

As per the thresholds, the user will configure the device bandwidth and we'll have two factors to derive both thresholds. The threshold to be detected as offender will be $k_1 = 70\,\%$ of the configured bandwidth, and the threshold to be restablished again will be $k_2 = 20\,\%$.

```ts
import { createTaskstats, Taskstats } from './taskstats'

const socket = await createTaskstats()

await socket.getTask(process.pid) // Get our own stats

// Listen for task exits
socket.on('taskExit', (t: Taskstats, p?: Taskstats) => {
    console.log('task exited:', {
        tid: t.acPID, comm: t.acComm, uid: t.acUID,
        readBytes: t.readBytes, writeBytes: t.writeBytes })
    if (p) console.log('belonging to process', p.acPID)
})

const cpus: number = os.cpus().length
console.log(`Tracking ${cpus} cpus`)
await socket.registerCpuMask(`0-${cpus-1}`)
socket.socket.socket.ref()
```

Listing 13: Code demonstrating use our taskstats bindings

We also need to perform corrective actions whenever we find (non-)offenders. As said above, we could use either the cgroup interface or set I/O priority on every process. For simplicity we'll use the latter, since cgroup brings its own range of problems (you need to cooperate with the version of cgroup in use, the existing hierarchy, . . .).

Development We'll now walk through the userspace daemon's code in a simplified way. We won't list the full source code here; refer to B and the `daemon` directory in the submitted annex for reference.

First, we need access to the taskstats interface. This basically involves:

- Creating a Generic Netlink socket.
- Calling appropriate commands to query info and register interest when a task exits [15].
- Parsing attributes and the taskstats struct fields according to `linux/taskstats.h` [22].

After some amount of effort, we had functional bindings and could successfully access the data! It's too much code to reproduce here, but listing 13 & figure 17 show a test program to demonstrate use of the interface. However, we had to deal with certain undocumented aspects of the API and, among other things, process-level taskstats don't have `version` and many other

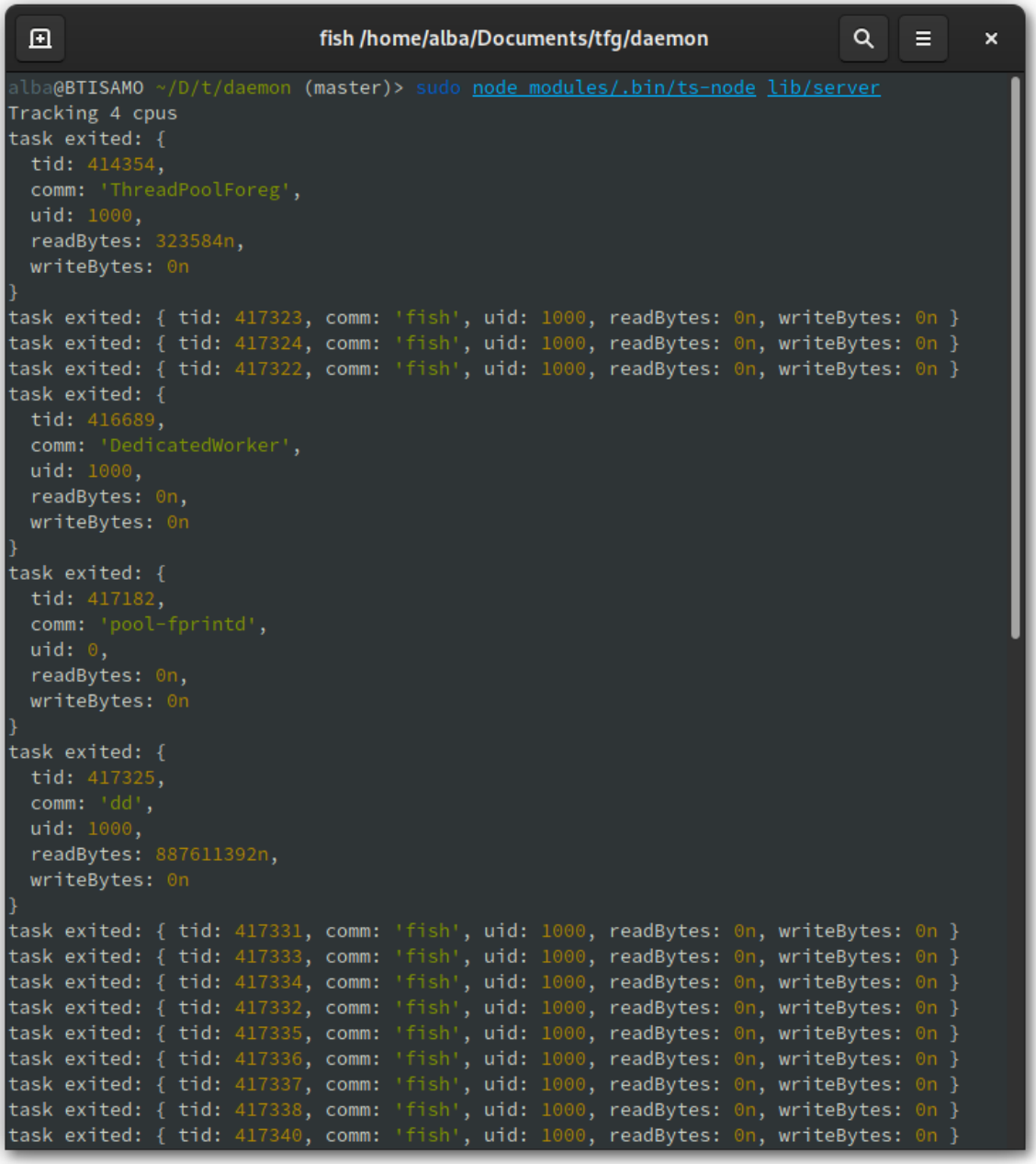


Figure 17: Test program consuming data from the taskstats API

```ts
// Pseudo-filesystem, shouldn't block the loop...
const listPids = (): number[] =>
    readdirSync('/proc').filter(x => /^\d+$/.test(x)).map(x =>
    ↪ Number(x))
const listTids = (pid: number): number[] => {
    try {
        return readdirSync(`/proc/${pid}/task`).map(x => Number(x))
    } catch (e) {
        if ((e as any).code === 'ENOENT')
            return []
        throw e
    }
}
const readPPID = (pid: number) => {
    try {
        return Number(/^\d+ \([\s\S]+\) . (\d+)/.exec(
            readFileSync(`/proc/${pid}/stat`, 'utf8'))![1])
    } catch (e) {
        if ((e as any).code === 'ENOENT')
            return
        throw e
    }
}
```

Listing 14: Fetching info from procfs

fields set on events.

This includes the data we need, so we have no option left but to aggregate it ourselves from individual tasks, which the docs mention as "inefficient and potentially inaccurate (due to lack of atomicity)". For scope & complexity reasons we'll ignore the task exits, even if it makes us vulnerable to pitfall (3).

Another problem is that the taskstats API does not have a way to list all tasks, so we'll resort to accessing `/proc` and `/proc/<x>/task` (which is what `iotop` does). We'll also use `/proc/<x>/stat` occasionally. Listing 14 shows the helper code for this. It's important to note that **everything we do is subject to races**, and a process can disappear at any time after we list it. To be robust, we detect `ENOENT` (when operating on `/proc`) and `ESCRH` (when using `ioprio_set` or the taskstats interface) and ignore the task / process if it pops up:

```
const ignoreEsrch = (e: Error) =>
    (e.message === 'Request rejected: ESRCH'
      || (e as any).code === 'ESRCH') ? null : Promise.reject(e)
```

Then, we need to periodically fetch statistics for all tasks and aggregate them using the decay coefficient. Issuing thousands of concurrent queries could easily exhaust the Netlink receive buffer, so we'll use a promise pool to limit concurrency to 100 by default. Finally, we'll aggregate data into processes (task groups) before passing it down to the rest of the code. This is shown in listing 15.

Once we have the infrastructure in place, we can store the information in our tree and perform the exponential smoothing:

```
const gone = new Set(tree.keys())
await fetchProcesses(p => {
    gone.delete(p.pid)
    const node = tree.get(p.pid)!
    if (!node)
        return initializeNode(p)
    node.parent = p.parent

    const delta = p.writtenBytes - node.writtenBytes
    node.writtenBytes = p.writtenBytes

    const bandwidth = (Number(delta) / 1e6) / (cycle / 1e3)
    node.writeBw += decayCoeff * (bandwidth - node.writeBw)
})
```

Because polling the statistics for *each and every thread in the system* one by one and then aggregating the results is a heavy operation (each cycle takes about 150 ms), we ended up taking samples every $T_s = 2$ s.

After this, some housekeeping is needed to clean up processes that have died since the last iteration. Then we traverse the tree, adding up the children into their parents with the corresponding factor. We track the node with the most aggregated bandwidth, and return it:

```
type NodeScore = { pid: number, bw: number }
function findOffender(pid: number) {
    const node = tree.get(pid)!
    let max: NodeScore | null = null
```

```
interface ProcessData {
    pid: number
    command?: string
    parent: number
    writtenBytes: bigint
}

function aggregateData(pid: number, tasks: Map<number, Taskstats>):
  ↪  ProcessData {
    let writtenBytes: bigint = 0n
    let parent: number | null = null
    tasks.forEach(t => {
        writtenBytes += getWrittenBytes(t)
        parent = t.acPPID
    })
    return {
        pid, command: tasks.get(pid)?.acComm, parent: parent!,
          ↪  writtenBytes
    }
}

const fetchProcesses = (cb: (p: ProcessData) => any) =>
    new PromisePool(function *() {
        for (const pid of listPids()) {
            const tasks: Map<number, Taskstats> = new Map()
            const tids = listTids(pid)
            let done = 0
            for (const tid of tids) {
                yield fetchSocket.getTask(tid)
                    .then(t => tasks.set(tid, t), ignoreEsrch)
                    .then(() => {
                        if (++done === tids.length && tasks.size)
                            cb(aggregateData(pid, tasks))
                    })
            }
        }
    }() as any, fetchConcurrency).start()
```

Listing 15: Fetching info from taskstats & aggregating into processes

```cpp
#include <nan.h>
#include <errno.h>
#include <unistd.h>
#include <sys/syscall.h>

NAN_METHOD(SetIoprio) {
    int which = Nan::To<int>(info[0]).FromJust();
    int who = Nan::To<int>(info[1]).FromJust();
    int ioprio = Nan::To<int>(info[2]).FromJust();
    int res = syscall(SYS_ioprio_set, which, who, ioprio);
    if (res < 0)
        Nan::ThrowError(Nan::ErrnoException(errno, "ioprio_set",
          ↪ "Could not set I/O priority"));
}

NAN_MODULE_INIT(Init) {
    Nan::SetMethod(target, "setIoprio", SetIoprio);
}

NODE_MODULE(native_binding, Init)
```

Listing 16: Native binding to set I/O priority (C++ part)

```
    const maxOf = (x: NodeScore) => (max && max.bw >= x.bw) ? max :
      ↪ x

    let childrenBw = 0
    for (const child of node.children) {
        const result = findOffender(child)
        max = maxOf(result)
        childrenBw += result.bw
    }
    const bw = node.writeBw + childrenBw * parentFactor
    return maxOf({ pid, bw })
}
```

Next up is the logic to restrict offenders (and unrestrict them afterwards). To change the I/O priorities it may be tempting to just spawn the `ionice` command, but this could block since it involves reading the executable from the disk into memory. Thus, creating a native binding to call `ioprio_set` would be a good option, even if it brings complexity up. Listings 16 and 17

```typescript
const native = require('../build/Release/native_binding')

// ...
export const CLASS_SHIFT = 13
export const PRIO_MASK = (1 << CLASS_SHIFT) - 1
// ...
export const PRIO_VALUE = (class_: number, data: number) =>
    (class_ << CLASS_SHIFT) | data

// ...
export enum Class {
  NONE,
  RT,
  BE,
  IDLE,
}

// ...
export enum Who {
  PROCESS = 1,
  PGRP,
  USER,
}

// ...

/** Set ioprio on a process */
export function set(pid: number, class_: Class, data: number) {
    native.setIoprio(Who.PROCESS, pid, PRIO_VALUE(class_, data))
}
```

Listing 17: Native binding to set I/O priority (TypeScript part)

show the relevant code.

We can now put everything together to (un)-restrict offenders we find. Several tests confirm the daemon works correctly, despite a sustained 7 % CPU consumption which is higher than what we'd hoped for.

5 Budget

Since this is a software project, and we've exclusively used open-source tools to produce this work, the project cost comes exclusively from the hours of work and equipment. Regarding the latter, while it is true that there was a risk of burning out the real disk drives on the computers, care was taken to perform all experiments in RAM to avoid incurring any additional hardware costs. Equipment consists of the machine we used to develop and run experiments, which costed 1200 € performing amoritzation over 5000 h of use, this gives us a rate of 0.24 €/h. We'll also account for 20 W of electricity, which corresponds to 8.76 €/h.

We'll consider a single role for this project, but given the variety of technologies we've had to work with we'll choose a higher reference than normal, 20 €/h. The approximate time spent on this project accounted to around 440 h.

Taking everything into account, we have 8800 € in terms of hours worked, and 3960 € regarding equipment & supplies. This adds up to a **total cost of 12 760 €**.

6 Conclusions

This project turned out to be more work than expected, but we still managed to achive the specified mandatory requirements:

- Tooling to perform experiments, non-invasive measurements and data analysis was successfully developed.
- We got consistent results from both the analysis part and the proof-of-concept.
- While far from a perfect solution, we successfully developed a PoC that is able to greatly reduce system unresponsiveness, even in the presence of large caches, in exchange for a minor performance drop.
- Other than switching to the BFQ scheduler, the PoC operates in a non-invasive way: this makes it unlikely to introduce noticeable side effects, and appropriate for general use.

It wasn't an easy job due to the number of components involved and the nature of the problem, but it was fun and allowed us to work with a variety of amazing kernel technologies.

Like email or some filesystem aspects, the writeback cache is one more example of a complex, yet critical, component that was designed on a different context than today given how technology has evolved over the years.

Part of the incidentals were caused by a wrong assumption that these pauses came from the cache's throttling, which turned out not to be the case. It would have helped to be a bit more conservative when measuring the depth of the project, however these kind of risks are an inherent part of research.

All in all, we'd say this worked out as a productive and fun project, and hope it is useful to other people —not only in providing insight about this particular problem in Linux, but also as a learning tool (together with the application & scripts) to anyone seeking to learn how the kernel works and especially the page cache.

7 Future work

The immediate next steps would involve conducting a more precise analysis of the kernel subsystems to confirm our theory of unwanted pauses due to large flushes & locked inodes and get some context on the matter. Again, the page cache is a complex component which involves three subsystems, so there can (and probably will) be additional interactions we've not been able to discover in this thesis.

It would also be beneficial for distributions to manage the cache thresholds and / or allow users to adjust them, or do it unassisted based on disk throughput.

As mentioned in section 4.3 inode locking is a natural part of the cache, however this doesn't remove from the fact that it is deeply unfair, and we shouldn't have to reduce cache size or mess with scheduling to alleviate those problems.

The cache was designed long ago when these problems didn't manifest themselves, so maybe it would be feasible to redesign (part of) it today so that data could be scheduled for I/O *without* being locked (i.e. lock it when the BIO is being processed, for instance). However this would probably require deep changes to the BIO layer, schedulers & drivers. Another possibility would be to perform some kind of *double buffering* to allow for a second version of the inode to be modified while the first is in writeback state.

Our knowledge of the relevant subsystem (and the kernel in general) is still limited so we can't guarantee these things are entirely possible, but it seems worth to investigate them.

References

[1] Georg Schönberger Werner Fischer. Linux storage stack diagram, October 2014. URL https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[2] Linux source code: `mm/page-writeback.c`, . URL https://github.com/torvalds/linux/blob/v5.6/mm/page-writeback.c. Line 829.

[3] Wu Fengguang. Linux patch `143dfe861`: *writeback: IO-less balance_dirty_pages()*, August 2010. URL https://github.com/torvalds/linux/commit/143dfe8611a63030ce0c79419dc362f7838be557.

[4] Johannes Weiner. Linux patch `a1c3bfb2f`: *mm/page-writeback.c: do not count anon pages as dirtyable memory*, January 2014. URL https://github.com/torvalds/linux/commit/a1c3bfb2f67ef766de03f1f56bdfff9c8595ab14.

[5] Linux documentation: Control Group v2, . URL https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#writeback. "Writeback" section.

[6] Linux kernel mailing list thread: *Disabling in-memory write cache for x86-64 in Linux II*, October 2013. URL https://lore.kernel.org/lkml/20131029203050.GE9568@quack.suse.cz/.

[7] Linux documentation for `/proc/sys/vm/`, . URL https://www.kernel.org/doc/html/latest/admin-guide/sysctl/vm.html.

[8] Linux documentation on tracepoints, . URL https://www.kernel.org/doc/html/latest/trace.

[9] Steven Rostedt. Using the `TRACE_EVENT()` macro, March 2010. URL https://lwn.net/Articles/379903/.

[10] Linux documentation: Kprobe-based Event Tracing, . URL https://www.kernel.org/doc/html/latest/trace/kprobetrace.html.

[11] Matt Fleming. A thorough introduction to ebpf, December 2017. URL https://lwn.net/Articles/740157/.

[12] Linux documentation on memory cgroups (v1), . URL https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html.

[13] Linux kernel mailing list thread: *[PATCH v5 0/9] memcg: per cgroup dirty page accounting*, February 2011. URL https://lore.kernel.org/lkml/1298669760-26344-1-git-send-email-gthelen@google.com/.

[14] Linux documentation on the BFQ scheduler, . URL https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[15] Linux documentation on the taskstats interface, . URL https://www.kernel.org/doc/html/latest/accounting/taskstats.html.

[16] Linux documentation on User Mode Linux, . URL https://www.kernel.org/doc/html/latest/virt/uml/user_mode_linux.html.

[17] Linux documentation on configuring the event tracer, . URL https://www.kernel.org/doc/html/latest/trace/events.html.

[18] Johannes Berg. trace-cmd python plugin documentation, May 2010. URL https://github.com/rostedt/trace-cmd/blob/master/Documentation/README.PythonPlugin.

[19] Linux source code: include/linux/fs.h, . URL https://github.com/torvalds/linux/blob/v5.6/include/linux/fs.h. Line 2093.

[20] Linux source code: fs/fs-writeback.c, . URL https://github.com/torvalds/linux/blob/v5.6/fs/fs-writeback.c. Line 1195.

[21] PC Matic Research. Memory trends, 2020. URL https://techtalk.pcmatic.com/research-charts-memory/.

[22] Linux documentation on the taskstats struct, . URL https://www.kernel.org/doc/html/latest/accounting/taskstats-struct.html.

Appendices

A Experiment code

This appendix only reproduces the code for running experiments, see the `analysis` directory in the submitted annex for the data analysis notebook and other code.

`load.py` script

```
#!/usr/bin/env python3
import os
import sys
import time
import tempfile
import json
import pickle

def make_control(idle):
    def cycle():
        time.sleep(idle)
    return { 'cycle': cycle }

def make_write(f, idle):
    f = open(f, mode='wb', buffering=0)
    i = 0
    def cycle():
        nonlocal i
        f.write('{}'.format(i).encode())
        i += 1
        time.sleep(idle)
    return { 'cycle': cycle }

def make_load(f, wait, size, block):
    f = open(f, mode='wb', buffering=0)
    remaining = size // block
    block = bytes([0]*block)
    waited = False
    def cycle():
        nonlocal waited, remaining
        if not waited:
            time.sleep(wait)
```

```
                waited = True
                return
            if remaining <= 0:
                return True
            f.write(block)
            remaining -= 1
    return { 'cycle': cycle }

def make_multiwrite(f, idle):
    i = 0
    def cycle():
        nonlocal i
        with open(f + '.' + str(i), 'wb', buffering=0) as ff:
            ff.write(b'test')
        i += 1
        time.sleep(idle)
    return { 'cycle': cycle }

loads = {
    'control': (make_control, [ ('idle', float) ]),
    'load': (make_load, [ ('file', str), ('wait', float), ('size', int),
      ↪  ('block', int) ]),
    'write': (make_write, [ ('file', str), ('idle', float) ]),
    'multiwrite': (make_multiwrite, [ ('file', str), ('idle', float) ]),
}


args = sys.argv[1:]
if len(args) < 2:
    print('Usage: ./load.py <name> <kind> [<parameter>...]',
      ↪  file=sys.stderr)
    print('Available loads:', file=sys.stderr)
    for kind, (_, params_def) in loads.items():
        tokens = [kind] + ['<{}>'.format(n) for n, f in params_def]
        print(' {}'.format(' '.join(tokens)), file=sys.stderr)
    exit(1)
name, kind, *params = args
if kind not in loads:
    print('Error: Invalid load {}'.format(repr(kind)), file=sys.stderr)
    exit(1)
maker, params_def = loads[kind]
if len(params) != len(params_def):
    print('Error: Expected {} params, found {}'.format(len(params_def),
      ↪  len(params)), file=sys.stderr)
```

```
    exit(1)
params = [ (n,f(x)) for (n,f), x in zip(params_def, params) ]

gettime = lambda: time.clock_gettime(time.CLOCK_MONOTONIC)
load = maker(*(x[1] for x in params))
times = [ gettime() ]
try:
    while not load['cycle']():
        times.append(gettime())
except KeyboardInterrupt:
    pass
with open('load.{}.pkl'.format(name), 'wb') as out:
    pickle.dump({
        'pid': os.getpid(),
        'kind': kind,
        'params': dict(params),
        'times': times,
    }, out)
```

experiment.py script

```
#!/usr/bin/env python3
# Dependencies: python 3, trace-cmd
# This should be in a path WITHOUT SPACES or weird control characters
# Do NOT run as root!!

import time
import datetime
import ctypes
import tempfile
import sys
import os
import signal
import shutil
from os.path import dirname, join, abspath
from subprocess import run, Popen, DEVNULL
import json
import re
libc = ctypes.CDLL(None)

def checked_wait(task, timeout=None):
```

```
    rc = task.wait(timeout)
    if rc:
        raise Exception('Task {} terminated with exit code
          ↪ {}'.format(task.args, rc))

def check_lost_events():
    base = '/sys/kernel/tracing/per_cpu'
    for cpu in os.listdir(base):
        with open(join(base, cpu, 'stats')) as f:
            stats = dict( re.fullmatch(r'(.+?): (.*)',
              ↪ x.rstrip()).groups() for x in f )
        if lost := int(stats['overrun']) + int(stats['dropped events']):
            raise Exception('{} lost events on {}'.format(lost, cpu))

base = dirname(dirname(abspath(__file__)))

## EXPERIMENT PARAMETERS ##
kernel_path = join(base, 'linux', 'linux')
memory = '150M'
write_bps = 1 * 1024 * 1024
dev_size = 130 * 1024 * 1024

def main():
    # Create experiment directory
    expname = datetime.datetime.now().strftime('%m-%d_%H-%M')
    experiment_base = join(base, 'exp', '_' + expname)
    os.makedirs(experiment_base)
    print('Preparing experiment at:', experiment_base)

    # Create device file, filled with 0xFF
    print('-- Creating block device --')
    dev_file = tempfile.NamedTemporaryFile(
      ↪ dir='/run/user/{}'.format(os.getuid()))
    block = bytes([ 0xFF ] * 4096)
    missing = dev_size
    while missing: missing -= dev_file.write(block[:missing])
    dev_file.flush()

    # Make filesystem
    print('-- Formatting --')
    run(check=True, args=[ 'mkfs.ext2', '-F', dev_file.name ])

    # Launch kernel!
    print('\x1b[1m-- Launching kernel --\x1b[m')
```

```
    run(check=True, args=[ kernel_path, 'mem=' + memory,
            'root=/dev/root', 'rootfstype=hostfs', 'rw',
            'ubdb=' + dev_file.name,
            'init=' + abspath(__file__), '--', experiment_base ])

    os.rename(experiment_base, join(base, 'exp', expname))
    print('\x1b[1m\x1b[32m-- Completed successfully --\nResults at:
      ↪ {}\x1b[m'.format(expname))

def inside_kernel():
    print('\x1b[1m-- Inside kernel --\x1b[m')
    os.chdir(sys.argv[1]) # change to experiment dir

    dev_file = '/dev/ubdb'
    dev_number = os.stat(dev_file).st_rdev
    dev_number = '{}:{}'.format(os.major(dev_number),
      ↪ os.minor(dev_number))

    # Mount common filesystems
    print('-- Mounting things --')
    blkio_base = '/sys/fs/cgroup/blkio'
    run(check=True, args=[ 'mount', '-t', 'proc', 'none', '/proc' ])
    run(check=True, args=[ 'mount', '-t', 'sysfs', 'none', '/sys' ])
    run(check=True, args=[ 'mount', '-t', 'tmpfs', 'none',
      ↪ '/sys/fs/cgroup' ])
    os.mkdir(blkio_base)
    run(check=True, args=[ 'mount', '-t', 'cgroup', '-o', 'blkio', 'none',
      ↪ blkio_base ])
    run(check=True, args=[ 'mount', '-t', 'tracefs', 'none',
      ↪ '/sys/kernel/tracing' ])

    # Throttle write bandwidth
    print('-- Throttling bandwidth --')
    blkio_write_bps = join(blkio_base, 'blkio.throttle.write_bps_device')
    with open(blkio_write_bps, 'w') as f: f.write('{}
      ↪ {}'.format(dev_number, write_bps))

    # Mount our block device
    print('-- Mounting FS --')
    run(check=True, args=[ 'mount', dev_file, '/mnt' ])

    # Preliminary things
    print('-- Preparing --')
    with open('experiment.json', 'w') as f:
```

```
        json.dump({
            'kind': 'uml',
            'start': datetime.datetime.now().isoformat(),
            'kernel': os.uname().release,
            'write_bps': write_bps,
            'dev_size': dev_size,
            'memory': memory,
        }, f, indent=4)
        f.write('\n')

    # Start experiment
    print('-- Starting experiment --')
    tasks = []
    add_task = lambda x: tasks.append(Popen(x, shell=True))
    def add_load(name, kind, *p):
        tasks.append(Popen([ join(base, 'analysis', 'load.py'), name,
          ↪  kind, *map(str, p) ]))

    add_task('trace-cmd record -e balance_dirty_pages -e
      ↪  global_dirty_state')
    time.sleep(3) # wait for it to start up

    add_load('c1', 'control', 0.05)
    add_load('w1', 'write', '/mnt/write1', 0.05)
    add_load('mw1', 'multiwrite', '/mnt/multiwrite1', 0.1)
    add_load('l2', 'load', '/mnt/load2', 43, int(dev_size*.2), 512)
    add_load('l1', 'load', '/mnt/load1',  5, int(dev_size*.5), 1024)

    checked_wait(tasks.pop())
    check_lost_events()
    for task in tasks: task.send_signal(signal.SIGINT)
    for task in tasks: checked_wait(task, 4)

    # at the end, we need to invoke the 'reboot' syscall to power off
    print('-- Powering off --')
    libc.syncfs(os.open('/', 0)) # force hostfs to sync data... poweroff
      ↪  unmounts forcefully
    libc.reboot(0x4321fedc)

if os.getpid() == 1:
    # we're init inside the VM! omfg!
    inside_kernel()
else:
    main()
```

`live_experiment.py` script

```
#!/usr/bin/env python3
# Dependencies: python 3, trace-cmd
# Like experiment.py, but runs the tasks directly on the host
# and uses the specified command as the load.

import time
import datetime
import ctypes
import tempfile
import sys
import os
import signal
import shutil
from os.path import dirname, join, abspath
from subprocess import run, Popen, DEVNULL
import json
import re
libc = ctypes.CDLL(None)

def checked_wait(task, timeout=None):
    rc = task.wait(timeout)
    if rc:
        raise Exception('Task {} terminated with exit code
          ↪  {}'.format(task.args, rc))

def check_lost_events():
    base = '/sys/kernel/tracing/per_cpu'
    for cpu in os.listdir(base):
        with open(join(base, cpu, 'stats')) as f:
            stats = dict( re.fullmatch(r'(.+?): (.*)',
              ↪  x.rstrip()).groups() for x in f )
        lost = int(stats['overrun']) + int(stats['dropped events'])
        if lost:
            raise Exception('{} lost events on {}'.format(lost, cpu))

base = dirname(dirname(abspath(__file__)))

def main():
    if os.getuid() != 0:
        print('Needs to be run as root.', file=sys.stderr)
        exit(1)
```

```
# Create experiment directory
expname = datetime.datetime.now().strftime('%m-%d_%H-%M')
experiment_base = join(base, 'exp', '_' + expname)
os.makedirs(experiment_base)
print('Preparing experiment at:', experiment_base)
os.chdir(experiment_base) # change to experiment dir

# Preliminary things
print('-- Preparing --')
with open('experiment.json', 'w') as f:
    json.dump({
        'kind': 'live',
        'start': datetime.datetime.now().isoformat(),
        'kernel': os.uname().release,
    }, f, indent=4)
    f.write('\n')
os.mkdir('loads')

# Start experiment
print('-- Starting experiment --')
tasks = []
add_task = lambda x: tasks.append(Popen(x, shell=True))
def add_load(name, kind, *p):
    tasks.append(Popen([ join(base, 'analysis', 'load.py'), name,
      ↪  kind, *map(str, p) ]))

add_task('trace-cmd record -e balance_dirty_pages -e
  ↪  global_dirty_state')
time.sleep(3) # wait for it to start up

add_load('c1', 'control', 0.05)
add_load('w1', 'write', 'loads/write1', 0.05)
add_load('mw1', 'multiwrite', 'loads/multiwrite1', 0.1)

time.sleep(10)
main_command = 'pacman -Syu'
main_start = time.clock_gettime(time.CLOCK_MONOTONIC)
main = Popen(main_command, shell=True)
main.wait()

check_lost_events()
for task in tasks: task.send_signal(signal.SIGINT)
for task in tasks: checked_wait(task, 4)
```

```
    # Finish things
    shutil.rmtree('loads')
    with open('main.json', 'w') as f:
        json.dump({
            'start': main_start,
            'command': main.args,
            'pid': main.pid,
            'exit_code': main.returncode,
        }, f, indent=4)
        f.write('\n')
    os.rename(experiment_base, join(base, 'exp', expname))
    print('\x1b[1m\x1b[32m-- Completed successfully --\nResults at:
      ↪  {}\x1b[m'.format(expname))

main()
```

B PoC code

This appendix only reproduces the main script of the deamon, see the `daemon` directory in the submitted annex for the rest of the code, including the bindings to taskstats and ioprio.

`lib/server.ts`

```
import { promisify } from 'util'
import PromisePool from 'es6-promise-pool'
import { createTaskstats, Taskstats } from './taskstats'
import * as ioprio from './ioprio'
import * as os from 'os'
import { readdirSync, readFileSync } from 'fs'
const delay = promisify(setTimeout)

// System parameters
const diskBandwidth = 70 // in MBps

// Daemon parameters
const sampleInterval = 2e3
const decayTime = 8e3
const decayCoeff = 1 - Math.exp(- sampleInterval / decayTime)
const offenderThreshold = .70
const innocentThreshold = .20
const parentFactor = .7
const warnThreshold = 1.5
const tsBufferSize = 1024 * 1024
const fetchConcurrency = 100

; (async function main() {

    const getWrittenBytes = (p: Taskstats) => p.writeBytes -
      ↪ p.cancelledWriteBytes

    // Pseudo-filesystem, shouldn't block the loop...
    const listPids = (): number[] =>
        readdirSync('/proc').filter(x => /^\d+$/.test(x)).map(x =>
          ↪ Number(x))
    const listTids = (pid: number): number[] => {
        try {
            return readdirSync(`/proc/${pid}/task`).map(x => Number(x))
        } catch (e) {
            if ((e as any).code === 'ENOENT')
```

```
                return []
            throw e
        }
    }
    const readPPID = (pid: number) => {
        try {
            return Number(/^\d+ \([\s\S]+\) . (\d+)/.exec(
                readFileSync(`/proc/${pid}/stat`, 'utf8'))![1])
        } catch (e) {
            if ((e as any).code === 'ENOENT')
                return
            throw e
        }
    }

    const ignoreEsrch = (e: Error) =>
        ((e as any).code === 'ESRCH' || e.message === 'Request rejected:
          ↪ ESRCH')
            ? null : Promise.reject(e)

    // FETCH CODE

    interface ProcessData {
        pid: number
        command?: string
        parent: number
        writtenBytes: bigint
    }

    function aggregateData(pid: number, tasks: Map<number, Taskstats>):
      ↪ ProcessData {
        let writtenBytes: bigint = 0n
        let parent: number | null = null
        tasks.forEach(t => {
            writtenBytes += getWrittenBytes(t)
            if (parent !== null && parent !== t.acPPID)
                throw Error('parent mismatch')
            parent = t.acPPID
        })
        return {
            pid, command: tasks.get(pid)?.acComm, parent: parent!,
              ↪ writtenBytes
        }
    }
```

```
const fetchProcesses = (cb: (p: ProcessData) => any) =>
    new PromisePool(function *() {
        for (const pid of listPids()) {
            const tasks: Map<number, Taskstats> = new Map()
            const tids = listTids(pid)
            let done = 0
            for (const tid of tids) {
                yield fetchSocket.getTask(tid)
                    .then(t => tasks.set(tid, t), ignoreEsrch)
                    .then(() => {
                        if (++done === tids.length && tasks.size)
                            cb(aggregateData(pid, tasks))
                    })
            }
        }
    }() as any, fetchConcurrency).start()

// TREE CODE

interface TreeNode {
    parent: number
    children: number[]
    writtenBytes: bigint
    writeBw: number
    command: string
}

function initializeNode(p: ProcessData) {
    const node: TreeNode = {
        parent: p.parent,
        children: [],
        writtenBytes: p.writtenBytes,
        writeBw: 0,
        command: p.command || '',
    }
    tree.set(p.pid, node)
    return node
}

async function recursivelyAct(
    callback: (task: number) => Promise<void>,
    ...pids: number[]
): Promise<void> {
```

```
        while (pids.length) {
            // To mitigate the effects of races, first act on roots
            await Promise.all(pids.map( x =>
               ↪  callback(x).catch(ignoreEsrch) ))
            // Then list their children and recurse, breadth first
            const parentOf = (pid: number) => {
                const node = tree.get(pid)
                return node ? node.parent : readPPID(pid)
            }
            pids = listPids().filter(pid => pids.includes(parentOf(pid) ||
               ↪  -1))
        }
    }

    type NodeScore = { pid: number, bw: number }
    function findOffender(pid: number) {
        const node = tree.get(pid)!
        let max: NodeScore | null = null
        const maxOf = (x: NodeScore) => (max && max.bw >= x.bw) ? max : x

        let childrenBw = 0
        for (const child of node.children) {
            const result = findOffender(child)
            max = maxOf(result)
            childrenBw += result.bw
        }
        const bw = node.writeBw + childrenBw * parentFactor
        return maxOf({ pid, bw })
    }


    // Use separate sockets, for receiving stats & fetching them
    const fetchSocket = await createTaskstats()
    const eventSocket = await createTaskstats()

    // Increase the buffer size to avoid losing stats
    eventSocket.socket.socket.setRecvBufferSize(tsBufferSize)

    // Initialize the tree
    log('Populating the tree')
    const tree: Map<number, TreeNode> = new Map()
    let currentOffender: number | null = null
    let time = process.hrtime.bigint()
    await fetchProcesses(initializeNode)
```

```
// Listen for task exits
eventSocket.on('taskExit', (t: Taskstats, p?: Taskstats) => {
    // FIXME: aggregate stats from exited tasks
})
const cpus: number = os.cpus().length
log(`Tracking ${cpus} cpus`)
//await eventSocket.registerCpuMask(`0-${cpus-1}`)

// Begin operating
log('Daemon ready')
while (true) {
    await delay(sampleInterval)

    // Monitor cycle time
    const endTime = process.hrtime.bigint()
    const cycle = Number(endTime - time) / 1e6
    if (cycle > sampleInterval * warnThreshold)
        log(`Loop cycle took ${cycle - sampleInterval}ms more`)
    time = endTime

    // Update counters & also bandwidths
    const gone = new Set(tree.keys())
    await fetchProcesses(p => {
        gone.delete(p.pid)
        const node = tree.get(p.pid)!
        if (!node)
            return initializeNode(p)
        node.parent = p.parent

        const delta = p.writtenBytes - node.writtenBytes
        node.writtenBytes = p.writtenBytes

        const bandwidth = (Number(delta) / 1e6) / (cycle / 1e3)
        node.writeBw += decayCoeff * (bandwidth - node.writeBw)
    })

    // Remove processes that no longer exist
    gone.forEach(pid => tree.delete(pid))
    if (currentOffender !== null && !tree.has(currentOffender)) {
        // offender died
        currentOffender = null
    }
```

```
        // Update children so we can traverse the tree
        tree.forEach(node => node.children = [])
        tree.forEach((node, pid) => {
            const parent = tree.get(node.parent)
            parent && parent.children.push(pid)
        })

        // Detransition offender if it matches requirements
        if (currentOffender !== null) {
            if (findOffender(currentOffender).bw < innocentThreshold *
              ↪ diskBandwidth) {
                log('Unrestricting back')
                await recursivelyAct(async x => ioprio.set(x,
                  ↪ ioprio.Class.BE, 4), currentOffender)
                currentOffender = null
            }
        } else {
            // Traverse from PID 1, picking an offender
            let offender = findOffender(1)
            if (offender.bw > offenderThreshold * diskBandwidth &&
              ↪ offender.pid !== 1) {
                const node = tree.get(offender.pid)!
                log(`Restricting process ${offender.pid}
                  ↪ ${JSON.stringify(node.command)}`)
                await recursivelyAct(async x => ioprio.set(x,
                  ↪ ioprio.Class.IDLE, 0), offender.pid)
                currentOffender = offender.pid
            }
        }
    }

})()

function log(msg: string) {
    const ts = new Date().toISOString().replace('T', ' ')
    console.log(`${ts}: ${msg}`)
}
```

Glossary

BFQ Budget Fair Queueing, a proportional-share I/O scheduler of the Linux kernel

BIO Block I/O, a request to operate on a block storage device; also refers to the subsystem that manages block storage

blkio the controller (or kind of cgroup) that governs bandwidth & latency regarding block I/O

BPF Berkeley Packet Filter, a language & virtual machine definition

cgroups Control Groups, the primordial feature of the Linux kernel for restricting, distributing & auditing access to system resources among processes

dentry Directory Entry, a structure holding the name, kind and pointed inode for an entry in a directory

FCFS First Come First Serve, a simple scheduling policy that executes requests in order of arrival

FFI Foreign Function Interface, a mechanism that allows a programming language to dynamically call routines from another

GPOS General Purpose Operating System

inode a structure holding metadata about a node in a filesystem (a regular file, directory, symlink, device, pipe or socket)

memcg the controller (or kind of cgroup) that governs usage of memory

PoC Proof of Concept

task subject of CPU scheduling; this generally refers to user-space threads, but in the general sense includes kernel worker tasks as well

UML User Mode Linux, a mechanism (or special architecture) that allows a kernel to run as a regular user-space application within another kernel

VFS Virtual File System, a subsystem of the Linux kernel that manages mounted filesystems and operations upon files