A Linux Kernel Scheduler Extension For Multi-Core Systems

Aleix Roca Nonell

Master in Innovation and Research in Informatics (MIRI) specialized in High Performance Computing (HPC)

Facultat d'informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya

Supervisor: Vicenç Beltran Querol
Cosupervisor: Eduard Ayguadé Parra

Presentation date: 25 October 2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Abstract

The Linux Kernel OS is a black box from the user-space point of view. In most cases, this is not a problem. However, for parallel high performance computing applications it can be a limitation. Such applications usually execute on top of a runtime system, itself executing on top of a general purpose kernel. Current runtime systems take care of getting the most of each system core by distributing work among the multiple CPUs of a machine but they are not aware of when one of their threads perform blocking calls (*e.g.* I/O operations). When such a blocking call happens, the processing core is stalled, leading to performance loss. In this thesis, it is presented the proof-of-concept of a Linux kernel extension denoted *User Monitored Threads* (UMT). The extension allows a user-space application to be notified of the blocking and unblocking of its threads, making it possible for a core to execute another worker thread while the other is blocked. An existing runtime system (namely *Nanos6*) is adapted, so that it takes advantage of the kernel extension. The whole prototype is tested on a synthetic benchmarks and an industry mock-up application. The analysis of the results shows, on the tested hardware and the appropriate conditions, a significant speedup improvement.

Acknowledgement

As the author of this thesis I would like to acknowledge all the people who participated in this work. Specially to Vicenç Beltran for giving me the opportunity to join the Barcelona Supercomputing Center, proposing me my so much desired first Linux Kernel related work that I had been searching for almost a year, and for making me realize of all my continuous nonsenses and mistakes during the development of it (a non-underestimated amount). Also to Kevin Marquet for helping to sort and clarify all my ideas the first months, helping so much with the writing of the first paper (this work contains a few reused lines written by him), and for showing me an awesome way to be a researcher. Thanks to Samuel Rodriquez for offering support with the FWI benchmark and unloading me of the task of completely rewriting how FWI manages internal data structures to improve performance and ease parallelization. Thanks to Eduard Ayguade for offering himself to review the work. Thanks to Albert Segura for his contributions before I retook his work after he left the BSC and for leaving the job position opened :-).

And of course, thanks to my family for their infinite patience, my friends and coworkers for their support and encouragement, and, specially, to Thais, the love of my live for being the most comprehensive and cheerful being on earth, even when work and my obstinacy keeps me chained on the desk.

Contents

1 Introduction **7**

1.1 Preamble and Motivation . . . 7

1.2 Related Work . . . 8

1.3 Methodology . . . 9

2 Background **11**

2.1 OmpSs . . . 11

2.1.1 Introduction . . . 11

2.1.2 The OmpSs/Nanos6 Threading Model . . . 11

2.2 Linux Kernel . . . 12

2.2.1 Introduction . . . 12

2.2.2 The Process Scheduler Subsystem . . . 14

2.2.3 The I/O Path . . . 16

2.2.3.1 The Page Cache . . . 17

2.2.3.2 The I/O Block Subsystem . . . 19

2.2.3.3 Single Queue I/O Schedulers . . . 20

2.2.3.4 Multiple Queue I/O Schedulers . . . 21

2.2.3.5 The Storage Devices . . . 22

2.2.4 Kernel/User Space Communication Channels . . . 23

2.2.4.1 System Calls . . . 23

2.2.4.2 Pseudo Filesystem . . . 23

2.2.4.3 Netlink Sockets . . . 24

2.2.4.4 Eventfd . . . 24

3 Design and Implementation **25**

3.1 Proposal Overview . . . 25

3.2 Design Alternatives . . . 26

3.2.1 Kernel Standalone Management (KSM) . . . 26

3.2.2 Kernel Active Management (KAM) . . . 29

3.2.3 Kernel Passive Management (KPM) . . . 33

3.3 User-Monitored Thread (UMT) . . . 35

3.3.1 Overview . . . 35
3.3.2 Linux Kernel Side . . . 37
3.3.2.1 Syscall: kpm_mode_enable() . . . 37
3.3.2.2 Syscall: ctlschedkpm() . . . 38
3.3.2.3 Scheduler Wrapper . . . 41
3.3.3 User-Space Runtime Side . . . 45
3.3.3.1 Leader Thread . . . 45
3.3.3.2 Workers . . . 49

4 Experimental Validation **51**
4.1 Environment, Tools and Metrics . . . 51
4.2 Synthetic Benchmark . . . 52
4.3 Full Waveform Inversion Mock-up (FWI) . . . 55
4.3.1 Introduction . . . 55
4.3.2 FWI Analysis . . . 56
4.3.2.1 FWI OmpSs Porting . . . 56
4.3.2.2 FWI and UMT Analysis . . . 58
4.3.2.3 FWI Versions . . . 61
4.3.2.4 FWI and the Block Layer . . . 62
4.3.3 Results . . . 62
4.4 KPM Overhead . . . 68

5 Conclusions and Future Work **69**
5.1 Future Work . . . 69
5.2 Conclusion . . . 70

Bibliography **73**

Chapter 1

Introduction

1.1 Preamble and Motivation

High performance computing applications usually execute in worker threads that are handled by a userland runtime system, itself executing on top of a general purpose operating system (OS). The main objective of the runtime system is to provide maximum performance by getting the most out of available hardware resources. On a multi-core machine, this translates to distribute the work of applications among the machine's available cores and balance each core workload to ensure that each possible CPU cycle is used on behalf of the application's purpose.

Runtime's balancing capabilities are subject to the underlying OS scheduler policy. This scheduler has to take decisions at various moments: when threads are created, when they finish, when they perform I/O operations etc. The problem is that this general purpose scheduler takes decisions with a low-level knowledge of what is executing in userland. This leads to performance loss. Indeed, when a thread performs a blocking I/O operation against the OS kernel, the core where the thread was running becomes idle until the operation finishes. This problem can lead to huge performance loss as some HPC or high-end server applications perform lots of I/O operations because they heavily deal with file and network requests.

One approach to address this issue is to make the runtime system aware of when blocking and unblocking events happen. In this way, it can chose to execute another worker thread while the first one is blocked. This approach requires special kernel support, and although several solutions exist to do so, their complexity has prevented them to make it into the Linux kernel mainland code.

In this thesis, several approaches to achieve the same goal are studied which conclude with the following proposed contributions:

- A new, **simple and lightweight** Linux kernel extension in which the OS provides user-space feedback by using the *eventfd* interface which allows to use, in user space, a file descriptor as an event wait/notify mechanism.
- A modification for the *Nanos6* task-based runtime of the *OmpSs* [1] programming model so that it takes advantage of the kernel extension.
- An evaluation of the whole prototype using both a custom benchmark and an industry mock-up application.

The remaining of this thesis is organized as follows: The *Background* chapter exposes the most relevant parts of both the OmpSs runtime and the Linux Kernel, which forms the foundation this work relies on.

The *Design and Implementation* chapter highlights the models considered and the implementation details for the integration of the OmpSs runtime and the Linux Kernel. The *Experimentation* chapter illustrates the prototype's test bed and comment on the results obtained. Finally, this thesis concludes with the final remarks and observations in the *Conclusions* section.

1.2 Related Work

Mechanisms to provide feedback from kernel-space to user-space when a blocking operation occurs have already been considered in the context of *user-level threads*. User-level threads provide a mechanism to perform context switches in user-space between threads that belong to the same process, thus minimizing the cost of context switching. This is also known as the N:1 model, in which N user-level threads are mapped to a single kernel thread. The problem with this model is that when a user-level thread performs a blocking operation, all user-level threads associated to the same kernel-level thread block. A palliative approach exists, known as *Hybrid approach* in which a set of user-level threads are mapped to a set of kernel threads. However, if just one of these user-level threads blocks, all other user-level threads associated to the same kernel-level thread also block. The problem is that the kernel is not aware of user-level threads.

Scheduler Activations (SA) [2] provide user-level threads with their own reusable kernel-level context and a kernel to user-space feedback mechanism based on *upcalls* (function calls from kernel-space to user-space). When a user-space thread blocks, a new type of kernel thread known as *activation thread*, is created (or retrieved from a pool) to relieve it. The activation thread upcalls a special user-space function that informs the user-space scheduler of the blocked thread. Then, still on the upcall (from the user-space side), the user scheduler runs and schedules another user-space thread. When the blocking operation finishes, another activation thread is created/retrieved and upcalls another user-space function to inform the user-level scheduler that the user-space thread is ready again.

SA has an important drawback. The user-space scheduler thread cannot safely access shared resources protected with a lock that it has not direct access to. For example, it is possible for a user-level thread to get an internal glibc lock and block without releasing it because of a page fault. Then, the kernel will wake up another user-level thread to handle the blocking event. If the user-level thread code that handles the event also uses the same glibc resource as the blocked thread and tries to acquire the lock, a deadlock happens. This extends to internal kernel locks, such as memory allocation locks.

SA were integrated into production OS's such as NetBSD [3, 4] and FreeBSD [5] (under the name of Kernel Schedule Entities or KSE). An implementation was proposed for the Linux Kernel [6, 7, 8] but the SA concept itself was rejected because the associated complexity was to high [9]. In the NetBSD 5.0 version, support for SA was removed for the traditional 1:1 threading scheme because *"The SA implementation was complicated, scaled poorly on multiprocessor systems and had no support for real-time applications"* [10]. FreeBSD KSE support was also removed since their 7.0 version and changed it for the 1:1 model.

Windows OS has a similar implementation called User-Mode Scheduling (UMS) [11], it is based on the same principle of upcalls, userland context switches and in-kernel unblocked thread retention. The interface is available since the 64 bit version of windows 7 and Windows Server 2008 R2. The locking problem also arises in their implementation as noted in the "UMS Best Practices" section of the cited document above: *"To help prevent deadlocks, the UMS scheduler thread should not share locks with UMS*

worker threads. This includes both application-created locks and system locks that are acquired indirectly by operations such as allocating from the heap or loading DLLs".

The K42 research OS [12, 13] proposed a more sophisticated mechanism to solve a similar problem. The K42 kernel schedules entities called dispatchers, and dispatchers schedule user-space threads. A process consists of an address space and one or more dispatchers. All threads belonging to a dispatcher are bound to the same CPU as the dispatcher is. Hence, to achieve parallelism, multiple dispatchers are required.

When a user-space threads invokes a kernel service[1] to initiate an I/O request, a "reserved" thread from the kernel space is dynamically assigned from a kernel maintained pool of reserved threads to assist it. This thread is in charge of initiate the I/O operation against the underlying hardware and block for it to complete. In the meantime, the kernel returns control to the user-space thread dispatcher so it can schedule another user-space thread if it has one. When the I/O completes, the kernel notifies the dispatcher with a signal-like mechanism so it can schedule the user thread again. Is worth noting that because dispatchers schedule user-space threads, an unblocked thread is not going to run unless there is some explicit interaction from the dispatcher scheduler.

The K42 user-level scheduling of dispatchers is provided by a trusted thread library. This library suffers from the same problem as SA: it cannot share any lock with the user-level threads, otherwise a deadlock would block the process if the dispatcher code to schedule a thread tried to get a lock that was already taken by the blocked user-space thread.

The *User-Monitored Thread* (UMT) model described in the following sections is similar to SA and K42 in the sense that both use a mechanism to notify a user-space thread whenever another thread blocks or unblocks in kernel-space. Also, both use the notification to manage the active threads. The main differences of UMT with SA and K42 are, on one hand, that unblocked threads are not retained anywhere (hence, there is no locking problem with the user-space scheduler) and, on the other hand, the UMT implementation is much more simple and lightweight. However, as a consequence of not retaining the recently unblocked threads, UMT needs to deal with periods of over-subscription. The experimentation chapter of this document carefully examines this drawbacks and shows the results obtained which, under the appropriate work loads, improves performance.

1.3 Methodology

This thesis work follows a design research methodology. In such methodology three main cycles define the workflow: The relevance cycle, the design cycle and the rigor cycle as shown in figure 1.1. The relevance cycle encompasses the problem identification and the analysis of requirements for a possible solution. In the design cycle, prototypes are build and tested based on the previously defined requirements. Finally, in the rigor cycle, new knowledge is generated after studying the results obtained. In design research, the workflow is not linear. Because neither the preliminary objectives nor the procedures to follow might be clear at the beginning, initial plans might be altered as more information is gathered, prototypes are tested and new knowledge is obtained. For this reason, a common design research pattern is to move between the different research stages as the work advances.

[1] All Kernel services are handled by means of Protected Procedure Calls (PPC).

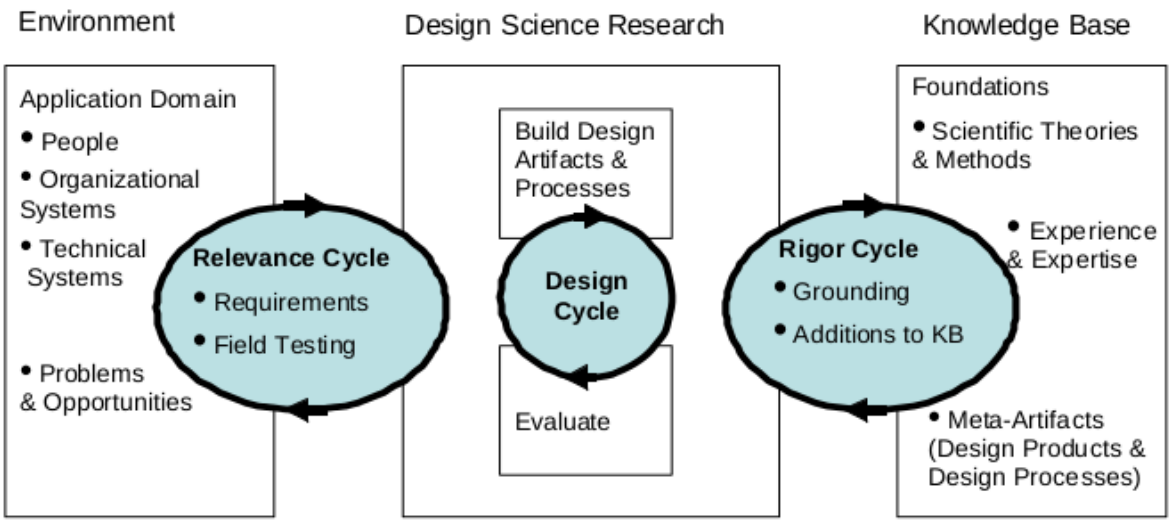

Figure 1.1: Design Research Methodology [14]

This work departs from the objective of improving modern HPC environments settling the working base on the Operating Systems and runtime engines for parallel computation. Based on the initial idea of monitoring threads at kernel level to improve system resource usage, the viability and applicability of such new feature was studied and contrasted with the current state of the art. An initial model was first implemented on a stable Linux Kernel version from which several first conclusions were obtained. Next, two more designs were proposed as the preceding ones were implemented and tested to solve unnoticed flaws or to add better features. Finally, the resulting design and its corresponding implementation were deeply analyzed and documented to obtain and share the contributions with the community.

Chapter 2

Background

2.1 OmpSs

2.1.1 Introduction

OmpSs is a programming model developed at the Barcelona Supercomputing Center (BSC) with the objective of guiding the development of the OpenMP programming model[15, 16]. The main OmpSs focus is both asynchronous parallelism and device heterogeneity to distribute work among different devices such as CPUs, GPUs and FPGAs. OmpSs is a research test bed for new features that is also being used on a production environment at the BSC. The main components of OmpSs are the source to source Mercurium Compiler and the Nanos library runtime. Mercurium translates source code pragmas into Nanos library calls and Nanos manages the application's execution flow at runtime.

OmpSs is a task based programming model, which means that all units of parallelism are expressed as tasks. A task is an enclosed sequence of instructions specified by the developer that must be executed sequentially. Multiple tasks can be executed in parallel as long as all their dependencies have been fulfilled. Dependencies express which data a task require to perform its computation and which data it produces. In fact, dependencies are expressed simply by specifying which variables a task uses as input, output or both. The actual execution sequence of tasks is determined by the Nanos library at runtime.

2.1.2 The OmpSs/Nanos6 Threading Model

In the OmpSs/Nanos6 threading model, everything is a task. The first task created by the runtime is, in fact, a wrapper around the `main()` function in the C programming language before the application starts executing. Tasks are able to create other tasks (its childs) and wait for all or a subset of them to be executed before continuing. The runtime names the list of tasks whose dependencies have been satisfied and can be executed as the *list of ready tasks*. Tasks are continuously being executed as long as their dependencies allows them to.

At startup, Nanos creates a special thread called the *Leader Thread* and a set of *Worker Threads*. The leader thread role is simply to run periodically to perform miscellaneous work such as printing logs. The Worker threads' objective is to execute ready tasks.

Nanos creates a worker thread per CPU. Workers are pinned (bound) to their CPU to prevent the OS from migrating them and polluting the cache. The Worker's main body is called the *Idle Loop*. As long as

there are available ready tasks, workers loop in the Idle Loop and ask the runtime task scheduler for a ready task from the list. When no more tasks left, the current worker adds the CPU where it is running into a list of Idle CPUs, adds itself to a list of idle Workers and voluntarily sleeps.

Once a worker acquires a task, it checks if the task already has a worker assigned (as explained later). If this is the case, the current workers ceases its execution and wakes up the worker which currently owns the task so it can continue running it. If the task's worker is pinned to a different CPU, Nanos pins it to the current CPU before waking it up. If the task does not have owner, the worker starts executing it immediately. Tasks assigned to workers are non-transferable, i.e. tasks are not moved between workers.

After completing a task, the worker checks if another task's dependencies have been satisfied due to the current task having finished. If such condition is met, the affected tasks are unblocked and are placed in the ready list of tasks to be executed.

Whenever a new task is created or an existing one is unblocked, it is checked if there is an idle CPU to run it. If it is the case, both the idle CPU and and idle worker are removed from their respective idle lists and the worker is woken up to run the task on the CPU. Otherwise, the task is queued on the list of ready tasks.

When a worker encounters a *taskwait* clause, the current task is locked and stops executing. However, the worker tied to it continues to execute the child tasks of the sleeping task, if any. Because any worker can execute any task, it is possible for the taskwait worker to not be able to run some of its own child task because other workers have acquired them first. If there are no more child task available for the taskwait worker to run although they are not finished, the taskwait worker sleeps. When another worker executes the last child task, it unlocks (not unblocks) the taskwait worker's task and places it in the task scheduler list of ready tasks (this is why workers can find tasks with already assigned workers in the idle loop). By prioritizing the execution of the child tasks it is intended to advance work in the shortest direction to unblock the current task.

The exact policy to retrieve the next task to execute from the ready list (and the actual list implementation) depends on the currently selected Nanos6 scheduler. Several solutions exists based on the expected workload or the machine architecture. The default scheduler is a simple FIFO based on the assumption that tasks are generally generated in the order that they are needed to be executed and that gives priority to unlocked tasks before new tasks.

2.2 Linux Kernel

2.2.1 Introduction

The Linux Kernel is the core of all GNU/Linux distributions. It is a software layer that sits between the underlying computer hardware and user applications. Its main role is to provide a hardware abstraction layer, manage system resources such as main memory or network connections and schedule processes.

The Linux Kernel is Open Source software, which means that it's possible to study its internals, contribute to its development and use it in projects without almost any restriction. For this reason it has become a test bed for research innovations. New ideas, as the one proposed in this thesis, can be tested on Linux and make public when validated and accepted for the community. The Linux community is active and responsive (when right questions are asked properly). As a result of the collaborative effort, a new version is released regularly every two-three months with hundreds of new features. Because of its highly

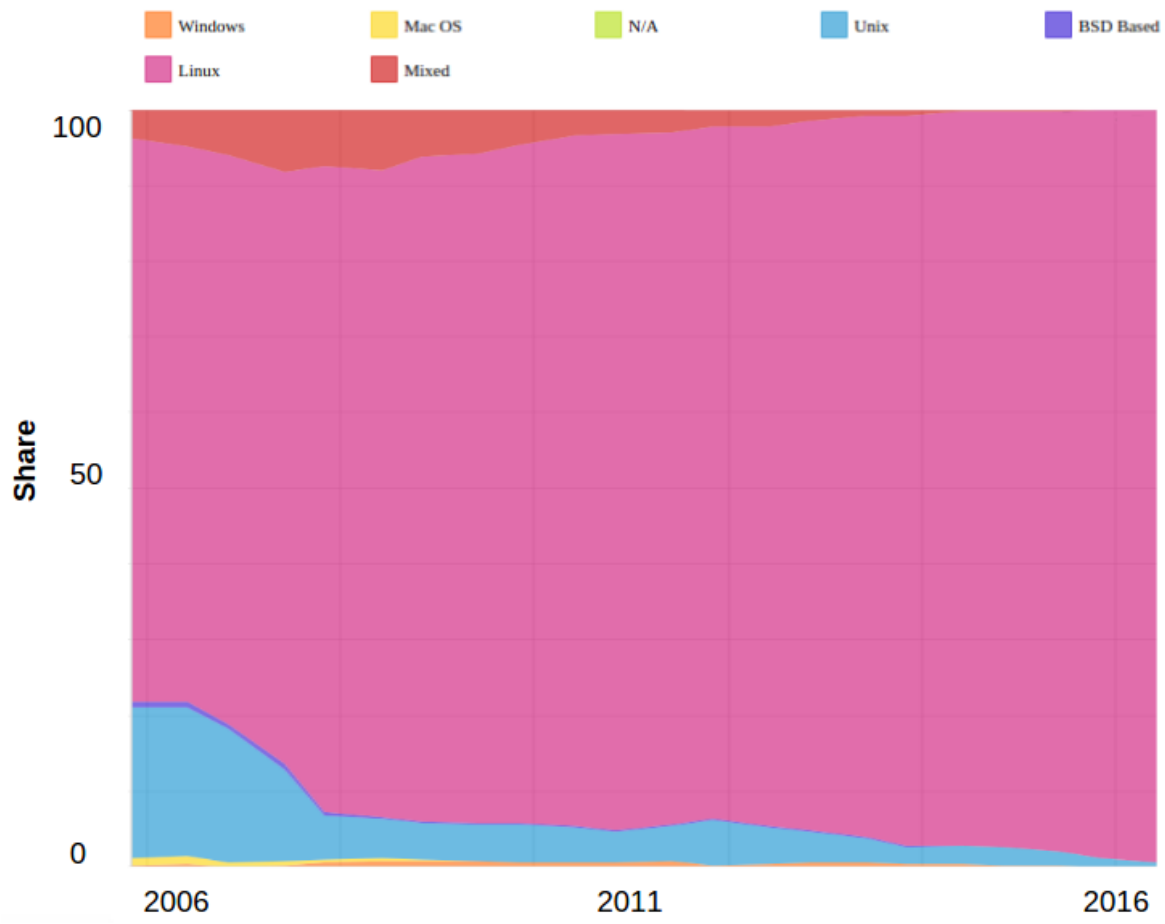

Figure 2.1: Top500 Operating System Family share June 1st, 2017

customizable module-based features, the Linux Kernel has became the standard solution from embedded devices to HPC environments [17]. Figure 2.1 shows the evolution of Operating System family choice in the Top500 supercomputer list as of June 1st, 2017.

Nowadays the Linux Kernel source code contains more than 25 millions of lines. To manage this huge project, it is divided into subsystems which are in turn organized as a set of git[1] trees. Each subsystem is managed by one or more maintainers. Maintainers decide whether a patch is included or not into their subsystem. At the top of the git tree, as the maintainer of the development version of Linux, sits Linus Torvalds, the Linux kernel founder and the last authority in accepting a patch into the mainline kernel.

The Linux Kernel development cycle [18] is organized in time periods named *Merge Windows*. Patches with new features are accepted when the merge window is opened. At this point Linus pulls changes from the underlying git trees and adds them into its development git tree. After typically two weeks, the merge windows closes and for six to ten weeks, only bug fixes are accepted. During this period, a release candidate is released every week for developers to test the current version. When Linus decides, it releases a the new kernel version and the cycle is repeated again by opening the next merge window.

The Linux kernel is a single big binary (monolithic kernel) capable of loading and unloading external modules on-demand. It supports preemptive multitasking for both user and kernel processes (running processes, even kernel processes, can be preempted by other processes to ensure runtime fairness). It also

[1]The git version control system was, in fact, developed by Linus Torvalds (the Linux Kernel creator) for the Linux Kernel project.

supports threading, virtual memory and shared libraries among much other features.

The rest of this section summarizes the main roles of the different subsystems related with the scope of this work: The generic structure of the process scheduler subsystem has been adapted to incorporate this work extension. The kernel to user-space communication channels, in particular, the eventfd has been used as the basis of this work to communicate block and unblock events. The understanding of the Linux Kernel I/O path has been of extreme importance to propose design alternatives and understand the results obtained.

2.2.2 The Process Scheduler Subsystem

In Linux, a process runs on a CPU indefinitely until the process scheduler subsystem decides that it needs to be preempted[2] for another one or the process voluntarily stops running either because it blocks or because it ends. The currently running process invokes the kernel scheduler code at some specific code locations when it is running in kernel mode by calling the `scheduler()` function. More specifically, the scheduler code instruments the following points:

- The return points used by system calls to return from kernel-space to user-space.
- The interrupt's return points in both user and kernel space.
- When the kernel enables preemption after disabling it to protect a critical region.
- When the timer interrupt handler is triggered or a new process is woken up, a flag to indicate that the current process should be preempted as soon as possible in the points mentioned before (this is necessary to avoid redundant checks).

It is interesting to note that because a process cannot preempt another at any point, scheduling a process might suffer delays that highly depend on what the system is working with. This also applies to threads. In Linux, there is almost no distinction between processes and threads. Threads are just processes that share the same address space. Hereinafter in this document, the therm process and thread is used indistinguishably[3] unless otherwise stated.

The **process scheduler subsystem** manages processes among the available computer cores. Its main task is to fairly distribute computational time among all system processes. Computational intensive processes can be preempted for another one if certain conditions are met to avoid starvation. Also, processes can be migrated between cores/sockets based on the load balancing policy of the scheduler. The exact details that determine how this decisions are taken depend on the currently used scheduler.

In Linux, multiple process schedulers coexist although each process can belong to a single one at a time. This infrastructure to manage multiple schedulers is known as "scheduler classes". Each scheduler is a container of processes with its own set of data structures, rules and policies to organize the processes that it contains. To mange multiple schedulers, each one is assigned a unique priority. The next process to be run is chosen from the scheduler with higher priority that has at least one process ready to be run. Once the scheduler is selected, the internal algorithm of the scheduler is used to select the next process to run. The scheduler with smaller priority is always the "Idle Scheduler" that always contains an "Idle Process" ready to run when no other process in the system requires a CPU. If at some point a process

[2] A process being *preempted* means that it is substituted by another processes that also deserves to run although the current worker has not explicitly left the CPU voluntarily. Thereby, preempted processes are block involuntarily.

[3] In the Linux Kernel terminology the therm "task" is used for both processes and threads. However, here it is not used into avoid confusion when mentioning tasks in the context of programming models.

from a higher priority scheduler becomes ready, the current process would be preempted as soon as possible. The scheduler classes framework helps new schedulers such as HPCSched [19] and mainline schedulers to be seamlessly integrated into the kernel by just implementing the core functions required by all schedulers[4]. The Linux Kernel 4.10.5 uses by default four schedulers[5] from highest to lowest priority: Deadline Scheduler, Real Time Scheduler, Completely Fair Scheduler and the Idle Scheduler.

The **Deadline Scheduler** has the most high priority processes in the system. It is an implementation of the Earliest Deadline First (EDF) algorithm. Its usage is limited to applications that need to run for a limited amount of time every certain period. For example, with the Deadline Scheduler it is possible to specify that a process should run for 20ms within 100ms. A user with enough privileges is able to configure three parameters when a process is moved into this scheduler: runtime, deadline and period. As the name suggests, runtime is the expected runtime of the process. Deadline is the time windows when the process should run, and period is the time from the start of one window to the start of the next one. It is a requirement that runtime <= deadline <= period. When a process is moved into the deadline scheduler, the Kernel checks whether is is feasible to accomplish the timing requirements of the new process and the other process in the scheduler. If it is not possible, the Kernel will refuse to add it as a protection measure for the other processes in the scheduler.

The **Real Time Scheduler** contains system critical processes. Examples of this processes are the *watchdog* kernel thread that periodically checks if a process has been running in kernel mode for too long to detect kernel bugs and the *migration* kernel thread used to expel a process that needs to be migrated from its CPU. The Linux kernel Real Time scheduler only guarantees "soft" real time which means that a process that wants to run can be delayed an *undefined* amount of time until it is finally run. Hence, no real determinism behaviour can be expected although the delays are minimized as much as possible[6]. There are 100 real time priorities that can be assigned to processes belonging to the Real Time Scheduler. A process with higher priority than another will always preempt it (hence, a bad usage of this scheduler can compromise the entire system by delaying other important processes). The scheduler's behaviour when multiple processes with the same priority are ready to run is determined by the policy selected by the user: SCHED_RR or SCHED_FIFO. If SCHED_FIFO is selected, processes are served in a First Input First Output basis. Instead, with SCHED_RR processes are run in a round robin fashion.

The **Completely Fair Scheduler** (CFS) is the default scheduler for common processes. Essentially, it keeps track of the runtime spent for each process and when a new processes has to be selected to be run, the one with the smaller time is chosen. This way, even interactive processes than do not run for large periods of time are not starved by batch processes. A new process is chosen to be run when the currently running process voluntarily stops or because the current process needs to be preempted. In CFS, a process needs to be preempted when exists another ready process that has run less time than the current process plus some margin to avoid over-scheduling. The actual runtime of each processes is weighted according to its *niceness* (classic UNIX nice priority) and named "virtual runtime". The idea is that for processes with higher niceness time accounting flows faster and for smaller niceness time flows slower. The list of ready to run processes is kept in a red-black tree[7] sorted by the virtual runtime as can be seen in figure 2.2.

[4] Although the C language does not support Object Oriented (OO) programming, the Kernel uses the abstraction "manually" in several infrastructures such as file descriptors or scheduler classes.

[5] See *man sched* for more information about all types of schedulers.

[6] The RT_PREEMPT[20] kernel patch can be applied to further reduce the delays, although no hard real time is still guaranteed, it depends on the exact system configuration.

[7] The Red-black tree is a kind of self-balancing binary tree.

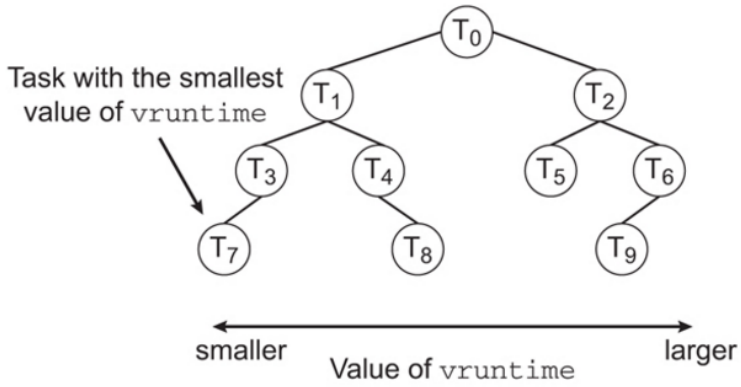

Figure 2.2: Completely Fair Scheduler (CFS) Red Black Tree [21]

Hence, the next process to be run (the one with smaller virtual runtime) is placed at the left-most leaf of the tree and can be found in $O(1)$. Adding and removing a process is done in $O(\log n)$.

CFS supports two policies: SCHED_OTHER and SCHED_BATCH. SCHED_OTHER (also called SCHED_NORMAL) is used for regular desktop workloads. SCHED_BATCH does not preempt nearly as often as SCHED_OTHER, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.

The **Idle scheduler** contains an Idle Thread per CPU whose main task is to do nothing the most efficient possible way. Doing nothing is not as easy as it seems. In fact, the Idle Scheduler belongs to the architecture dependent CPU Idle Subsystem [22] in charge to find a compromise between energy saving and latency spent to schedule a non-idle process after the CPU has entered an energy saving mode.

2.2.3 The I/O Path

The GNU/Linux system I/O path consist of several layers as is depicted in figure 2.3. Each layer tries to hide the latency of writing and reading to the underlying slower layer [23]. Each layer has its own buffering scheme which in some cases might lead to redundant operations if not used correctly. Because all this stacked layers are completely transparent to the user, this section details the main components and kernel subsystems involved: The page cache, the I/O block layer, the I/O schedulers and the disk storage.

I/O requests start their journey at the application level, when a user processes wants to write or read data from/into a memory buffer into/from a storage device. Usually, user space applications rely on libraries such as glibc that perform its own buffering in the calling processes address space to minimize system calls. This is the case for the family of functions `fwrite()`, `fread()` library functions. When the glibc function `fflush()` is called, the library buffers are sent to the Linux Kernel through the write() and read() system calls. The system call data is then buffered in the Linux Kernel page cache. The page cache is a general purpose cache that keeps track of pages written and read to/from devices. When data is to be written to these devices, it is, instead, written to the page cache. Later, a gang of kernel threads named "flusher threads" periodically write or update these pages. Flusher threads send data to the block I/O layer where I/O requests are placed in a set of queues pending to be sent to the device. The I/O scheduler examines the queues and optimizes them by merging and sorting requests. When the I/O request is moved from the I/O block layer queues to the device (through the device driver), it is first received by the device controller. This controllers might cache the data in big volatile memories where more

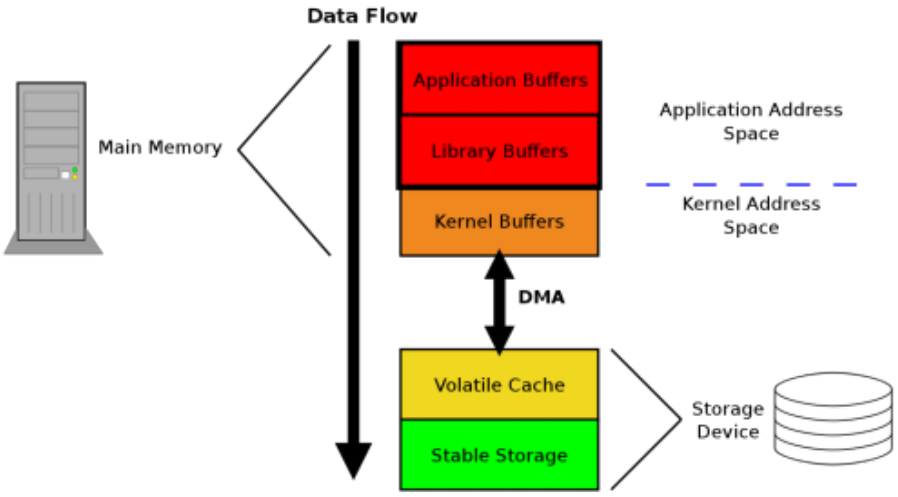

Figure 2.3: Simplified Linux System I/O path [23]

optimizations might be applied by the device's firmware. When the device is able to, it finally writes the data into the non-volatile storage.

A more detailed view of the complete Linux Storage path can be seen in figure 2.4.

2.2.3.1 The Page Cache

The page cache is a general software cache that buffers I/O operations in system memory. The need for the page cache raises form the fact that current non-volatile devices such as hard disks or solid state drives are slower than main memory. Whenever data is read of written to the slower device, it is temporarily stored in the page cache because of the principle of temporal locality. Hence, further references to the same data won't need to access the slow device again. Also, data is read in fixed size blocks named *pages* which might be larger than the requested memory to exploit the principle of temporal locality. In fact, pages track smaller blocks simply named "blocks" which are the minimum data unit used by file-systems. At the same time, blocks are composed of sectors, the smaller data chunk a hardware device work with. Data stored in the page cache can be released at any time when pressure for free memory increases.

Data written through the page cache follows a write-back policy in which writes to the backing storage (the non-volatile device) are deferred to some point in the future. The main advantages of this behaviour are three. In the first place and as commented before, writes become fast and hence, non-blocking. In the second place, multiple writes to the same in-memory sector will need a single write to the backing device instead of one for each write because writes are "merged" in memory. In third place, buffering writes might allow to reorder multiple small sparse sector writes in a single large sequential write of multiple sectors. The main drawback is that data is not "completely" safe until it reaches the backing storage and a power outage might lead to data lost.

When a page is written, it is first read from the backing device and copied into the page cache (if it was not already there). The page is then updated in-memory and marked as *dirty*. In Linux, a gang of threads called *flusher threads* are in charge of storing the dirty pages to the backing device and cleaning the dirty flag, albeit other methods exists. The behaviour of the flusher threads can be configured through the Linux */proc* pseudo file system. In particular, pages are written to the backing device in the following situations:

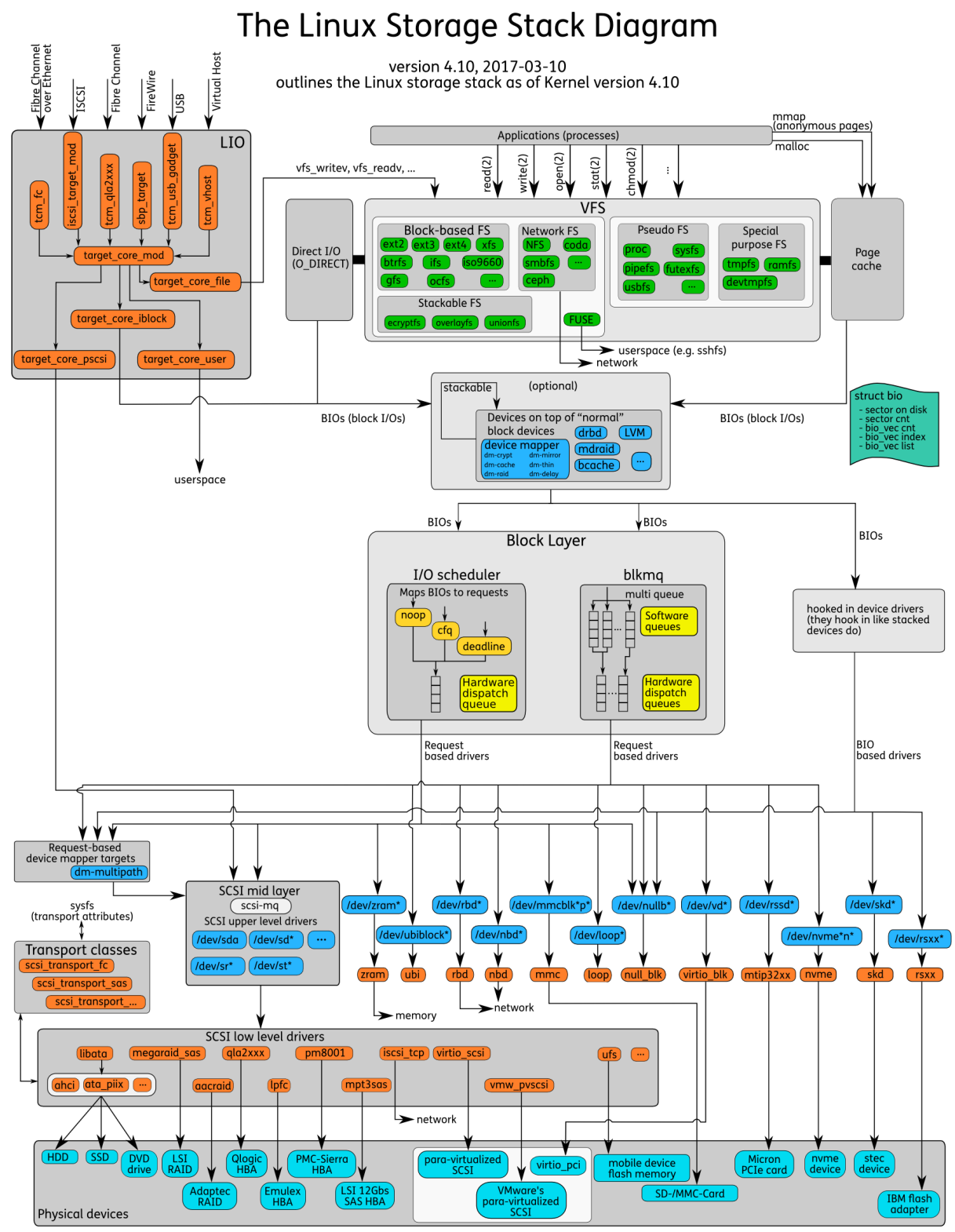

Figure 2.4: Linux 4.10 Storage Diagram [24]

- Pages start being written to disk when the amount of free memory is lower than the value stored in */proc/sys/vm/dirty_background_ratio*. However, subsequent write syscalls do not block yet.
- When the amount of free memory drops below */proc/sys/vm/dirty_ratio* number of pages, the hard-limit is reached and subsequent writes block until the flusher threads have backed up enough pages.
- Dirty pages older than */proc/sys/vm/dirty_expire_centisecs* are also written to disk every */proc/sys/vm/dirty_writeback_centisecs* centiseconds.
- Explicit `sync()` like syscalls. In this case, the thread issuing the syscall performs the write operations instead of the flusher threads.

More details on this kernel variables can be found in the Linux Kernel source code repository file *Documentation/sysctl/vm.txt*. Finally, is worth noting that the page cache might not be always beneficial depending on the application needs. Because of the principles of spacial and temporal locality is likely to be useful. However application's performance using its own cache, not reusing data, not accessing sequential data or working with more data than the main memory can hold, might worsen because of the page cache overhead. For this reason, there are mechanisms in the Linux Kernel to avoid using the page cache such as the O_DIRECT flag of the `open()` syscall. See again figure 2.4 for visual details in how the page cache is bypassed with O_DIRECT.

2.2.3.2 The I/O Block Subsystem

The Linux Kernel I/O block Subsystem provides a general abstraction for random access memory devices such as hard drives, or solid state devices. Processes issuing read or write requests to these devices transparently use the I/O block subsystem to manage their petitions. Essentially, it provides queues of I/O requests that are serviced to the hardware device following the currently selected I/O scheduler policy. There are currently two implementations of the I/O block subsystem as can be seen in figure 2.4, the traditional single queue implementation and the new multi-queue (blk-mq) implementation.

The **single queue implementation** maintains a single request queue shared for all cores and processes. The I/O schedulers can easily traverse the queue and optimize it according to all system's pending requests. This is the default option in use, although this might change for the blk-mq. The main disadvantage is the queue's locking on a multi-core system. Adding, removing or optimizing the queue requires to acquire a lock. When multiple processes in multiple cores try to acquire the same lock in parallel, the memory line where the lock resides moves from one core cache to the other, forcing to invalidate the cache line in the other cores. This might create a huge contention, specially in Non-Uniform Memory Access (NUMA) systems where the cache line needs also to move between sockets. The I/O schedulers supported fort this approach are: The Deadline I/O Scheduler, the Completely Fair Queuing I/O Scheduler and the Noop I/O Scheduler.

The **blk-mq** implementation supports multiple software requests queues, one for each CPU (or socket, as specified in the configuration), and multiple hardware queues (if the device supports it) as detailed in [25]. A software request queue is maintained per CPU to avoid the locking problems mentioned in the single queue approach. If the underlying hardware device only accepts a single requests queue (hardware queue), the software queues need to be merged and the contention might occur again. For this reason, modern hardware storage devices such as NVMe devices also support multiple hardware queues so mapping a software queue to a hardware queue can be done directly in a 1:1 mapping if the number

of software and hardware queues are the same (ideal case) or in a M:N mapping otherwise[8]. Currently only two I/O schedulers are supported for this approach: The Budget Fair Queuing I/O Scheduler and the Kyber I/O Scheduler.

It is also possible for driver implementations to bypass the block I/O layer, however, this is discouraged because the driver needs to do its own queue processing instead of using the generic one. Some high performance drivers have used this options in order to bypass the contention of the single queue block I/O layer. However, this should change with the new blk-mq implementation [26].

Regardless of the block layer implementation, the objects stored in the requests queues are of type *struct request_queue*. Each requests is a set of at least one *struct bio*. Each bio structure contains a set of <page,address,length> that describe several contiguous segments to be read or written. Overall, the data written on a request must be physically contiguous on the storage device although is not necessary to be contiguous in-memory (scatter gather operation). A request can hold multiple bio structures because merging requests is then simplified by just adding to one of the requests the bio structures of the other instead of merging the bio structures itself.

As mentioned before, each block layer implementation as its own set of I/O schedulers. The **I/O scheduler** purpose is to examine the software *requests queues* before they are sent to the hardware queues to perform two basic operations: reorder and merge requests. Reorder means changing the order in which the requests are sent to the storage device controller while merge is the action of joining multiple requests into a single one. Several strategies are implemented in the form of multiple I/O schedulers based on this two simple operations. However, only one I/O scheduler can be set per device, although different schedulers can be set for different devices.

I/O schedulers are deeply influenced by the low level characteristics of the underlying storage device, in particular for how traditional hard drives work. For a rotational device, the seek time (time spent to move the disk head) adds a significant overhead to the I/O operation, and specially for direction changes. For this reason, it is of interest for the I/O scheduler to reorder requests in order to take advantage of straight linear movements instead of performing irregular sweeps. Modern SSDs do not rely on mechanical components and do not suffer from this penalty. However, they do still benefit from merging requests.

The rest of this section summarizes the strategies implemented by the most common I/O schedulers.

2.2.3.3 Single Queue I/O Schedulers

The **Noop I/O Scheduler** is the simplest of the current I/O schedulers. The block I/O request queue is implemented as a FIFO and its only task is to scan it to merge multiple adjacent requests into a single one. This is useful for SSDs because they do not have special restrictions in the order in which operations are issued.

The **Deadline I/O Scheduler** applies special policies to avoid starvation of both read and write operations. Read operations are usually more critical than write operations. When a processes issues a read operation it usually cannot continue until the data is ready. Unlike reads, writes can be deferred and the process can continue. Hence, reads are synchronous operations while writes asynchronous operations. If the scheduler does not differentiate reads form writes, there is the risk of starving a few critical reads when a big write operation is in course. To avoid this, the deadline I/O scheduler

[8]Even if the number of software queues is larger that the number of hardware queues, contention is minimized by merging a subset of software queues into a single hardware queue.

maintains three queues: A FIFO for only writes, another FIFO for only reads and a third queue sorted to minimize disk seeks with both read and writes. When requests arrive to the block layer, a default timeout of 2ms for reads and 10ms for writes is assigned to it. While no request timeout has expired, the deadline scheduler services requests from the sorted queue. When a timeout expires of either the read or write FIFO, it switches to that queue and starts servicing request from there. Hence, this mechanism minimizes seeks by default, but also ensures that no request is indefinitely deferred, specially read operations because of their substantially smaller timeout. The exact parameters of the Deadline scheduler can be tuned at runtime through the */sys/devices/<device-path>/queue/iosched/* kernel directory, where <device-path> is the chain of hardware buses and devices to a particular storage device such as *pci0000:00/0000:00:1f.2/ata2/host1/target1:0:0/1:0:0:0/block/sda*. For instance, the file *read_expire* and *write_expire* can be used to set the timeout for reads and writes respectively. Also the file *fifo_batch* allows to determine how many requests are processed in a batch before checking for expired timeouts, this feature prevents the scheduler from switching between read and write queues too often. More details can be found in the kernel source tree file *Documentation/block/deadline-iosched.txt*

The **Completely Fair Queuing I/O Scheduler** (CFQ) (not to be confused with the CFS process scheduler) is based on the idea that all processes should have its fair share of I/O bandwidth. To do so, it maintains a sorted queue per processes issuing synchronous requests. Then, requests are selected from queues using round robin taking into consideration each processes priority. The number of requests taken from each queue can be configured at runtime. Asynchronous requests for all processes are classified by their processes priority in a queue per priority instead of a queue per processes.

CFQ supports idling[9] as a method for buffering requests: when the scheduler has finished processing a batch of requests of a process, it waits some time on the empty queue even if there are other requests pending in another queue. The rationale for this is that a processes uses to issue requests to physically close locations. Hence, waiting a bit on the current queue might minimize seeks if another request is about to come. If the processes does not issue any other request while idling, the time is wasted. However, if seeks are slow, the performance generally improves. For fast SSDs, this feature is disabled by default because there is no seek penalty. More details can be found in the kernel source tree file *Documentation/block/cfq-iosched.txt*.

2.2.3.4 Multiple Queue I/O Schedulers

The multi-queue block layer supports two dedicated I/O schedulers that were included in the Linux Kernel 4.12. The single queue deadline I/O scheduler was ported to the multi-queue approach in the Linux Kernel 4.11, however, it was ported as a proof of concept, instead of a real solution [27] and hence, the adaptation is not explained here[10].

The **Budget Fair Queuing** (BFQ) I/O Scheduler is intended to be a general scheduler for both HDDs and SSDs and designed to achieve good interactive response. BFQ was forked from the CFQ scheduler. Such as CFQ, it creates a request queue for each process in the system. For each queue, a budged is assigned which determines for how much time requests are sent to the storage device from the current

[9] The idling feature was the main feature of the Anticipatory I/O scheduler.This scheduler was removed from the kernel tree because CFQ could behave similarly.

[10] In fact, neither the dedicated I/O schedulers nor the deadline scheduler were present the last time the Linux Kernel 4.10.5 source code was forked from the Linus tree to implement the changes proposed in this thesis. However, the multi-queue schedulers are briefly detailed for completeness in this section.

process. The budget is determined by the scheduler itself and is based on the process priority and observations of the process's I/O behaviour. It also features a series of heuristics to improve performance under specific situations such as merging queue requests of processes working on the same drive area, idling on a queue if it becomes empty before the budget expires, specially give more priorities to realtime processes and much others.

The **Kyber** I/O Scheduler is a quite simple scheduler specialized for SSD devices. The main idea implemented in Kyber is to keep the queues that feed requests directly to the device short. Hence, the time requests spent in the queue is small and high priority requests can overtake common requests. Kyber adjusts the size of the queues to try to achieve the completion latency specified by the user in the Kyber configuration sysfs files.

2.2.3.5 The Storage Devices

Hard Disk Drives (HDD) store their data in magnetic disks that rotate at speeds between 4000rpm and 15000rpm. Data is stored in parallel circular tracks, arranged in blocks called sectors as shown in figure 2.5. A magnetic head mounted on a mechanical actuator arm moves radially over the disk to read and write blocks. Sequential data is read/written much faster than random data because the head does not needs to move (it is stored in the same circular track). Multiple platters might be used to read or write data stored in the different disks in parallel. The mechanical nature of HDDs causes I/O operations to be highly variable (depending on where the head is and where the data is). For this reason, manufacturers report I/O speed metrics based on benchmarks' average values.

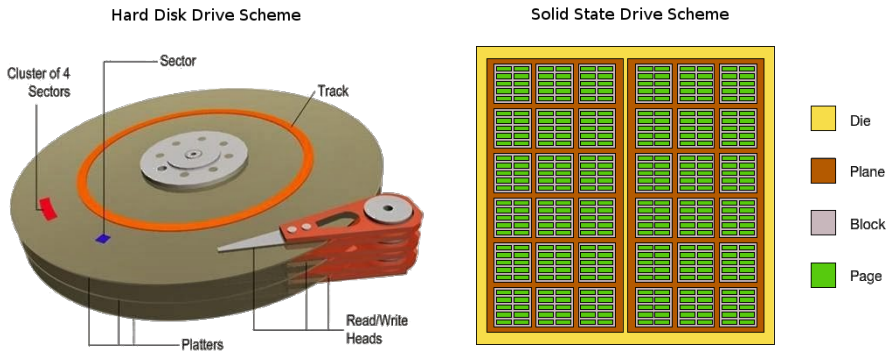

Figure 2.5: Hard Disk Drive and Solid State Drive scheme comparison.

Solid State Devices (SSD) rely on flash technology and do not require mechanical parts. They do not suffer a seek penalty and moreover, are able of servicing multiple requests in parallel. The main drawback is that data cannot be overwritten, it must be first erased. Internally, SSD organize data in cells that are grouped in pages of 4KiB to 16Kib which are again grouped in blocks of 128 or 512 pages as shown in figure 2.5. Data can be written at the page level, but it can only be erased at the block level. For this reason, when data needs to be updated, the SSD controller searches and writes the new data in an empty location then, marks the old data as invalid and defers to a garbage collector its deletion. If a write is to be done when no erased blocks left, the SSD must erase then, delaying the write operation. To alleviate this problem, the OS filesystem communicates the SSD the pages that are no longer used using a "TRIM" operation to clear blocks.

2.2.4 Kernel/User Space Communication Channels

The Linux Kernel provides several mechanism to communicate with user space processes. This section reviews the most used options such as syscalls, pseudo file systems, Netlink sockets and relatively new eventfd interface.

2.2.4.1 System Calls

System Calls or syscalls are the most known communication method for developers. System calls are used for user processes to requests kernel services. Issuing a system call involves a special hardware instructions that perform a mode switch (change from kernel to user space or vice versa) such as "syscall/sysret" in the x86 architecture. The system's call arguments are passed through hardware registers and hence, they are limited depending on the architecture. If more parameters than registers are needed, pointers to user space structures are passed instead.

When a user process invokes the kernel through a system call, a Linux kernel thread executes kernel code on behalf of the user space application at a specific code entry. Using the syscall number, the syscall entry point is redirected to the appropriate system call handler. The Kernel then copies the required data to/from the user-space processes if required and services the user request.

The Linux Kernel has kept the number of system calls small (367 for the 4.10.5 kernel), although some special syscalls offer a multiplexer like behaviour such as ioctl(), fcntl() and prctl(). fcntl() is used to get/set values for a particular file descriptor and prctl() is used to get/set values for processes. The ioctl() system call is used for filesystem-like objects that require more functionalities than read()/write() like system calls.

When a new system call is added, it needs to be maintained indefinitely. Removing a system call would break backwards compatibility with user space applications that rely on it and hence, special care must be taken before choosing this option.

2.2.4.2 Pseudo Filesystem

Pseudo file system are used to expose kernel runtime information to user space through a file system like infrastructure. The most known pseudo file systems are sysfs and proc which are mounted by default in */sys* and */proc* respectively. When the filesytem is mounted, the kernel populates the directory and creates in-memory files than can be read or written as usual (if the user has enough privileges to). When read() or write() operation is done against this files, the appropriate kernel file handler function is called to perform the desired operations. Generally, reading is done to get kernel runtime parameters while writing has the effect of configuring kernel parameters at runtime to modify its behaviour.

Because of the file layout, all communications through these files need to be adapted to the generic filesystem family of syscalls such as `open()`, `write()` and `read()`. Also, support for `select()`, `poll()` and `epoll()` syscalls can be implemented for the pseudo filesystem to enable kernel to user space notifications. The behaviour for this three system calls is similar: A user space process opens a file, then informs the kernel using a system call that wants to be notified whenever data to the file is ready

to be read and/or it is possible to write data, next `select()/poll()/epoll()`[11] is called and the processes is blocked until any of the previously configured events occur, when this happens the kernel wakes up the process and the application continues. Other functionalities that do not map to these syscalls might be implemented by using `ioctl()` as explained before.

Support for pseudo filesystem can be added or removed dynamically by loading or unloading kernel modules as needed. However, because the filesystem needs to be mounted, it might not be available in all environments such as namespaced, sandboxed or chrooted environment.

Depending on the needs, instead of creating a new pseudo filesystem it is possible to simply add an entry to an existing one, such as */sys* or */proc*.

2.2.4.3 Netlink Sockets

The Netlink interface[28] was designed to communicate user space processes with the Linux Kernel to transfer miscellaneous network packets, although any data can be sent through it. A Netlink socket is a full duplex communication channel, meaning that both ends can initiate a connection (the other methods always required a user space application to start the communication). Its usage is preferred instead of system calls or pseudo filesystems to avoid polluting the Linux Kernel interface with features that are likely to change in the future, such as specific kernel module driver to user space communications. With Netlink sockets, a standard socket communication channel is opened so both ends -kernel and user space- can interchange messages as needed. Netlink sockets, unlike system calls, can be loaded and unloaded, allow to multicast a packet to several processes and are asynchronous. However, for a simple communication channel it might add too much complexity and communication overhead.

2.2.4.4 Eventfd

An eventfd is a simplified pipe that was designed as a lightweight inter-process synchronization mechanism. Internally, an eventfd holds a 64 bit counter. A `write()` system call can be used to increment the internal counter with a given value. A `read()` to an *eventfd file descriptor* (EFD hereafter) blocks if its internal counter is zero. It also blocks if an overflow was to happen after a `write()`. A `read()` operation returns the counter value and sets its internal value to zero. It is also possible to configure an EFD as a semaphore (see `man eventfd` for more details). In this case, each `read()` decrements the internal counter by one instead of setting it to zero.

The main advantage of eventfd is its simplicity. It provides a clean interface to communicate events between processes or between a user space processes and the Linux Kernel that do not require to exchange complex data structures. Because it is integrated with the select()/poll()/epoll() family of system calls, it is possible for processes to block on multiple eventfd objects. However this interface is currently only available in Linux.

[11] The difference between `select()`, `poll()` and `epoll()` calls are mostly related with how portability, limits on file subscriptions and how file event subscriptions are managed. `select()` is the most limited but also the most portable solution, `poll()` solves most of the select problems but is less portable and `epoll()` works with any number of subscriptions but is only available on Linux.

Chapter 3

Design and Implementation

In this chapter, it is detailed the design and prototype implementation. First, three different design alternatives for the kernel part of the prototype are analyzed keeping in mind the runtime needs. Then, the final implementation details of both the Linux Kernel and the user-space OmpSs runtime are presented with the most relevant code examples.

3.1 Proposal Overview

This work main idea consist in detecting at kernel level blocking and unblocking events of threads to wake up replacement threads with the objective of avoid wasting system resources.

This feature is of special interest for user-space runtimes that need to monitor the status of threads (called workers) to execute in parallel an application with high performance requirements. Indeed, when a worker blocks, it can spend an undefined amount of time waiting for some requested data or service while the CPU where it was running sits idle.

The need for such extension comes form the fact that the current Linux Kernel does not provide any mechanism to **notify** user-space threads of when a thread blocks or unblocks, it is only possible to poll for the thread internal status, which requires to perform a system call in a busy loop. For the runtime to be able to react on a blocking event, it is likely to need a too fine grained resolution as to implement a busy loop.

Instead, the runtime could ask the kernel to monitor a number of his workers to receive notifications whenever one of them either blocks or unblocks, or it could just ask the kernel to automatically wake up an idle worker when it detects a blocking event on a monitored thread. In any case, a new worker would run on the idle CPU to either advance computational work or queue another data request or kernel service. Also, having multiple requests in flight might even lead to a better kernel scheduling which improves general throughput.

Because this idea might be applied in a number of ways, the following sections enumerate the different options considered and enters in details with an actual implementation and testing.

3.2 Design Alternatives

This section describes in detail the different approaches considered to implement this thesis main idea by exposing the problems, enumerating the options for each problem and concluding why each solutions is selected.

Three different Linux kernel extension have been considered to monitor threads keeping in mind a basic user-space runtime needs. Following this work main idea, the options studied are: "Kernel Standalone Management" (KSM), "Kernel Active Management" (KAM) and "Kernel Passive Management" (KPM).

- In KSM, the kernel automatically wakes up workers from a pool when a monitored worker blocks and retains workers when they unblock. Note that there is no communication with user space, all thread management is done at kernel level.
- In KAM, the Kernel actively retains threads immediately after being unblocked until the user-space application instructs it to release them (such as in the Scheduler Activations approach).
- In KPM, instead, the kernel has a pure passive role in which it only informs the user-space application of threads being voluntarily blocked (i.e. not preempted) or unblocked.

All approaches use eventfd's objects (see section 2.2.4.4) as the main communication channel between processes, although each approach uses it in a completely different manner. The EFD has been chosen among the other options presented because of its simplicity and flexibility.

For all cases, two fundamental properties have been defined to ensure that the models proposed respect a basic runtime needs:

- Selective wake up: Workers holding a task ready to run after being blocked should have a chance to run before other workers without a task. It is a good practice to finish already started tasks before starting new tasks because at some point, it is possible that the runtime decided that the already started task should run first. Also, an started task is likely to end sooner than a new one, which might unlock other tasks after its dependencies are satisfied.
- Worker fairness: Workers must not starve other workers. Workers should be able to stop running after finishing a task to allow other workers to run. Otherwise, a worker already holding a task might take too long to execute. To do so, it needs to be possible for workers to implement a mechanism to stop in behalf of others.

3.2.1 Kernel Standalone Management (KSM)

In KSM the user-space application is not aware of when a process gets blocked, but the kernel automatically chooses one worker (if available) to execute.

KSM is based on the abstraction of *tokens*. Only workers owning a token can run on a CPU. Initially there are as many tokens as CPUs. When a running worker blocks, it passes its token to another worker so it can substitute it while the first worker resolves its blocking state. When a worker tries to wake up, the kernel forces it to ask for a token and if there are not available tokens, it blocks again. In fact, a pool of tokens is shared among workers. When a worker blocks, the token is passed to the pool. When a worker tries to wake up, it checks the pool for a token. Workers being preempted do not release their token. Figure 3.1 depicts the KSM workflow.

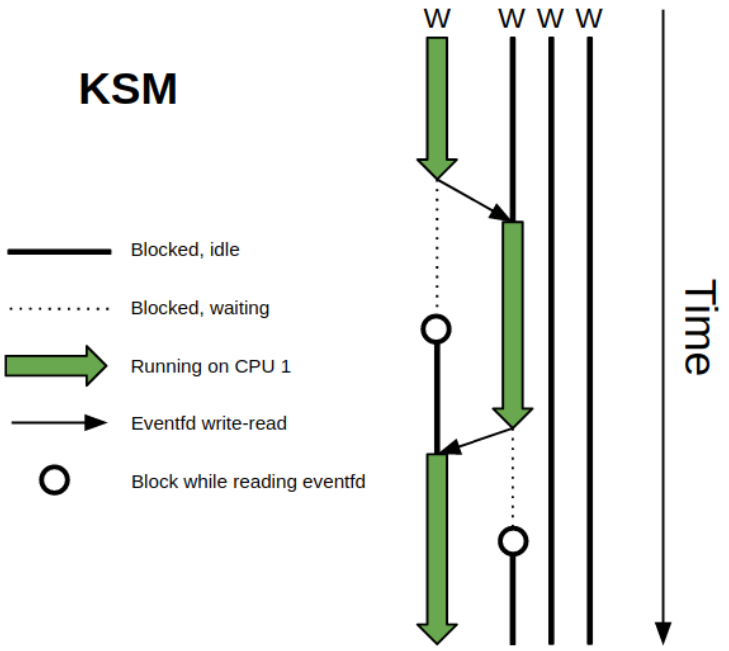

Figure 3.1: Kernel Standalone Management (KSM) example workflow. Each vertical line shows the state of a worker as time advances. Bold lines represent blocked workers that are ready to be run at any time when some other processes unblocks them explicitly. Dotted lines are workers blocked that are waiting for some event such as data from a storage device. Big colored arrows show workers running on a CPU. Small black arrows depicts an EFD being written at the tail of the arrow and being read at the arrowhead. White circles denote a worker being scheduled on a CPU, reading an EFD and blocking on the EFD. The workflow is as follows: The left-most worker voluntarily blocks and writes its token. Next, the second worker is waken up because of the EFD being written and starts running. While the second worker is running, the first worker resolves its blocking state, is scheduled on the CPU (which means that the second worker is momentarily preempted although it is not shown for simplicity) and blocks on the EFD because there is no token available. When the second worker voluntarily blocks, it returns the token to the pool and the first worker is woken up as a consequence. Then, the first worker continues to run normally.

Tokens and its transactions are implemented using an EFD configured to work as a semaphore. Workers share a single EFD object. Initially, the EFD contains the number of available tokens. When a worker tries to acquire a token, essentially it reads the EFD object. If there is at least one token in the EFD, the read call returns successfully and the EFD counter of tokens is decremented by one. If no more tokens left, the read call blocks until new tokens are available. When a worker no longer needs its token, it writes the EFD increasing its token counter by one and waking up blocked workers pending for a token if any.

The EFDs instrumentation is placed around the Linux Kernel schedule function. The schedule function performs a context switch on the calling processes by the next process to be run which is chosen inside the same schedule function by calling the Linux processes scheduler routines. More precisely, a worker lends its token (writes the EFD) just before calling the schedule function due to the process being blocked. A workers asks for a token (reads the EFD) just after returning from the schedule function.

As noticed before, when a worker is waken up after being blocked, it will have to be scheduled to check if there is a token available. If not, the worker will block again. This means that this approach leads to unnecessary context switches.

Ideally workers should be waken up only when there is a token available for them. To do so, when the

worker is about to be woken because its blocked state has been resolved (for example, requested data is available) the generic kernel wake up function should check if there is a ready token for it before waking it up. If not, the worker should be added to a special queue of unblocked workers that are ready to be run and should remain blocked. Then, when a worker holding a token is about to be blocked, the token should be transferred to one of the workers in the queue and woken up.

Regardless of the proposed simplification, the potential of this approach can be analyzed without affecting the principal workflow. It is left has a future work task to improve this detail.

The number of EFD objects used depends on the number of worker pools. To implement the pool of workers, two options have been considered: *floating workers* and *pinned workers*:

Floating workers is the simplest option. A global pool of workers is maintained for all CPUs than can be used to run a worker in any core. This means that there is a unique EFD with as many tokens as CPUs. Then, workers read and write it from the kernel space.

In the pinned workers option a set of pools per CPU or per socket is maintained, where all threads belonging to a pool are pinned to the same CPU or socket and share the same EFD. The reason for this option is to avoid polluting the cache of the core and minimize unnecessary migrations. However this might lead to an unbalance problem if a core's pool contains idle workers while another core's pool is empty. Additional management would be needed.

An important problem is that there is no intuitive way to satisfy the selective wake up property to wake up workers holding a task prior to idle workers. The kernel needs somehow to know which currently blocked processes holds a task and which not. However, when a process blocks, the kernel does not have any way to know if the process was blocked while executing a task or because the worker does not have more work to do.

In the later case, the problem is still worse. If a worker who was running on a CPU finishes its task and there are no more tasks to execute, it could intentionally block until there is more work to do. Because this worker would write the EFD before blocking, this means than another worker will wake up although it's known that there is no more work to do. This process would repeat for all workers indefinitely.

Hence, in some situations it is required to avoid reading/writing the associated EFD. There are various possible ways to implement this special blocking mechanism to differentiate idle workers from workers holding a task.

An option is to implement a new system call (or another communication method) to modify an internal flag of the worker thread to tag himself as an idle worker. After setting up the flag, the worker would block using a conventional user-space mutex while waiting for new tasks but internally the kernel would know that it won't need to read/write from its associated EFD. When the worker wakes up again and after fetching a new task to execute, it can use the system call to disable that flag. The problem with this approach is that the kernel should have to wake up the thread from the mutex in kernel-space, which it is not conventional given that the thread would have not been awaken from a user-space syscall.

Another option, similar to the preceding one, is to implement a different blocking mechanism for idle workers such as a new system call that blocks the process in a special kernel queue. The kernel would easily manage idle workers form a single queue. Also the syscall could block a worker while holding the EFD token with him to prevent waking up other workers. Then, when more tasks are generated, the runtime can wake up these idle workers to continue running. The only drawback is that it breaks the EFD "token" consistency, i.e. the worker that has the token is always running.

In any case, the main problem of the KSM approach is also its main advantage. Because the thread management is done at kernel level to get optimal performance, special scheduling needs to determine which worker holding a task should run next have to be implemented at kernel level (this is a case of the worker fairness property). Essentially this means either moving part of the runtime into the Kernel or keep a naive approach. For this reason and the kernel complexity involved to implement KSM, the development of this work was moved to the KAM approach.

3.2.2 Kernel Active Management (KAM)

In KAM, the kernel retains workers in kernel-space immediately after being unblocked. Tokens are used in conjunction with EFDs to regulate which workers can run. The difference now is that in-kernel retained workers are not unblocked until it is explicitly requested by a user-space processes named "Leader Thread" which keeps track of the state of workers.

In this approach, two EFD per worker are needed[1] as shown in figure 3.2. One EFD named EFD_CTL is used to block the worker in kernel-space. The EFD_INF is used to send block and unblock notifications from kernel-space to user-space. The EFD is a simple counter, multiple writes are added to the internal counter. To distinguish the block from the unblock event, the EFD is used as a bit mask. Both the block (WORKER_BLOCK flag) and unblock (WORKER_UNBLOCK flag) are different power of two numbers, hence, it is possible to write both of them before they are read and still be able to distinguish which events have occurred. Writing the same event two times without the EFD_INF being read would corrupt the counter, however, the nature of the mechanism does not allows it. A more detailed explanation follows:

The leader thread monitors all workers' EFD_INF using `epoll()` syscall. Hence, the leader threads sleeps on the `epoll()` call until any of the workers associated to the monitored EFD generates an event. When a running worker is about to get blocked, it first writes the flag WORKER_BLOCK to the EFD_INF at kernel level to notify the user-space leader thread that is going to block. When a worker is ready to run after being created or unblocked, it writes the flag WORKER_UNBLOCK in the EFD_INF at kernel level to notify the leader thread that it is ready again. Immediately after, it reads the EFD_CTL at kernel level and gets blocked waiting for permission from the leader thread to run. After each write in the EFD_INF the leader thread wakes up from the `epoll()` sleep and keeps track of the worker status associated with the EFD. Then, when it considers it, it chooses a worker from a user-space pool of EFD_CTL blocked workers and writes to its EFD_CTL. The selected worker wakes up and starts executing its task.

The selective wake up property is maintained in KAM because the leader thread can prioritize workers holding a task to be run as a consequence of keeping track of processes when they get blocked while executing. Also the worker fairness property can be satisfied by the runtime for the same reason.

Similarly to the KSM approach, idle worker management requires to make an extra effort. Idle worker thread could be blocked in-kernel or on a user space mutex.

[1] In fact, a single EFD could be used. For this approach, it is needed that both the kernel and the user side of a processes get blocked when writing and reading the EFD respectively. Reads block when the internal EFD counter is zero and writes block when the written value plus the contents of the counter overflows the counter. Hence, it could be possible for the processes in the kernel side to write a big enough number to get blocked in the kernel. However it is not a good practice to use such feature because of two reasons: A blocking write on an EFD is not possible with a single call, it has to be done at least in two calls because it is not allowed to write such a big number at once (also it is not a valid operation to write a negative value to produce an underflow as well). Second, when a blocking write wakes up, it finishes the write (it writes the remaining of the number that could not be written), which means that the counter will not be zero. Therefore, it needs to be read again.

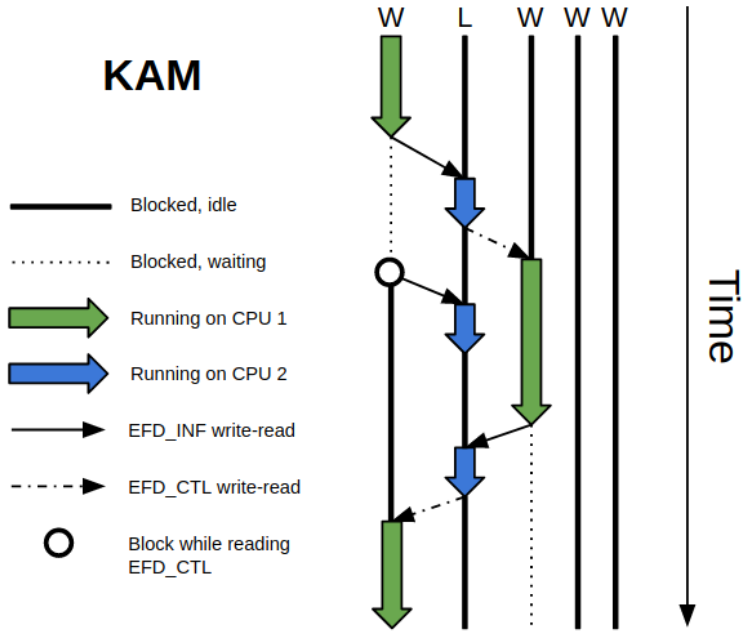

Figure 3.2: Kernel Active Management (KAM) example workflow. Each vertical line shows the state of a worker as time advances. Bold lines represent blocked workers that are ready to be run at any time when some other processes unblocks them explicitly. Dotted lines are workers blocked that are waiting for some event such as data from a storage device. Big colored arrows show workers running on a CPU. Small black arrows depicts an EFD (either INF or CTL as shown in the legend) being written at the tail of the arrow and being read at the arrowhead. White circles denote a worker being scheduled on a CPU, reading an EFD and blocking on the EFD. The workflow is as follows: The left-most worker blocks waiting for data and writes the WORKER_BLOCK flag to its EFD_INF. Next, the Leader Thread is waken up because of EFD_INF being monitored by an epoll syscall. The Leader thread writes a "1" to the EFD_CTL of the second worker and wakes it up to run on the idle CPU. While the second worker is running, the first worker resolves its blocking state, is scheduled on the CPU (which means that the second worker is momentarily preempted although it is not shown for simplicity), it writes WORKER_UNBLOCK on its EFD_INF and blocks on reading its EFD_CTL because its internal counter is zero. The Leader Threads wakes up to read the event but it does nothing because all CPUs are still busy. When the second worker voluntarily blocks it writes its EFD_INF and the Leader thread is woken up again. Then it immediately wakes the first worker by writing a "1" to its EFD_CTL. Then, the first worker continues to run normally.

To block the worker in-kernel it could be possible to use the `sched_yield()` syscall. This syscall forces a call to the Linux kernel scheduler. However, from the scheduler point of view, the `sched_yield()` is seen as a preemption, and preemptions do not write nor read the EFD's so it will not block on the EFD. A simple option would be to modify the `sched_yield()` syscall to add an internal flag to avoid bypassing the EFD mechanisms.

If a worker blocks in a user-space mutex on the idle loop because there are not any more tasks left, the Leader thread will receive a block notification and will not really know that it comes from an idle worker. Nevertheless, this is not a problem. Once the block notification is received, the Leader thread will try to unblock another worker in the idle CPU. However, it can check first if there is work to execute. If there is not, then, no worker is woken. Otherwise it means that new work has been created in the period between the idle worker going to sleep and the Leader thread waking up, meaning that a worker can be scheduled on the CPU.

When the idle worker is unblocked, first it needs to unlock the EFD_CTL and then the mutex. When it starts running, it will generate an unblock event and the Leader thread might try to add it to the list of unblocked workers pending to be woken up, which is an error because the worker is already running. Hence, the Leader thread needs to keep track of the idle worker identifier that has just woken up to ignore the incoming unblock event.

Another option, such as in KSM, is to add a new syscall to set a flag to enable or disable the EFD monitoring of the current thread. Hence, from user-space, an idle thread pretending to block on the idle loop would first disable the management and then signal somehow the leader thread to inform it that it is now idle. When the idle thread is woken up again, it would enable the management before starting to process new tasks.

From the three options studied, blocking on a user-space mutex without further modifications seems the most interesting option because of its simplicity compared to the other ones. The only minor drawback are the small spurious events that the Leader thread needs to ignore.

The main problem with KAM, just as in Scheduler Activations is that the leader thread and a worker cannot share resources. It could happen that the worker gets a lock to a shared resource and blocks because of a blocking syscall or a page fault (preemption is not considered here because preempted processes ignore the EFD mechanism) before releasing it. Then, if the leader thread tries to acquire the same lock, it will deadlock. The worker thread will never release the lock because the leader thread is not able to schedule it by writing its EFD. Some possible solutions follow:

- A naive option is to disable and enable the EFD management before and after entering a critical region of resources shared with the leader thread. The problem is the effort of modifying the critical user code regions run by workers and the overhead due to performing two system calls for every critical region.
- Another solution is to restrict the usage of resources to either the leader or the worker but not both. This, however, limits performance. For instance, it's interesting that a worker can wake up another worker when a new task is created. To do so, the worker needs to use the lists of idle and unblocked workers which contain workers that are ready to be executed. The leader also needs this list to wake up workers, and hence this lists should be shared. Instead, workers could signal the leader to perform the wake up operation but this introduces scheduling lags.
- Another solution is to avoid all possible blocking operations inside a critical region. This includes page faults and any blocking function calls such as malloc().

  A page fault can occur inside a critical region if a line of code of that region resides in a page not loaded to memory because it has not been loaded yet or because it was swapped out. Also a page fault can occur if more memory is needed for the stack or the heap. For instance, pushing a value into a list which was full, might imply a malloc() to increase the array size which might need another page.

  To avoid any page fault, the memory of the process can be locked in ram by calling mlock() or mlockall(). The process stack can be pre-faulted by creating a huge array and writing to it to force the kernel to map the pages in main memory.

  However, even if all memory is locked into ram it might happen that still more memory is needed. mlockall() is able to lock future pages but first, they suffer a page fault. To avoid blocking in critical areas due to lacking memory, there are two options:

- Avoid using any operation that might allocate memory inside a critical region.
- Use a pool allocator. This consist in pre-allocation objects on a pool of memory. Hence, when an object is needed, it is retrieved from the list instead of explicitly allocating memory for it. When the object is destroyed, is simply freed from the pool, but the memory is still kept by the process. This means, however, that the amount of of memory must be fixed at startup time. If more memory is needed, it might be possible to define a safe-point where it is safe to allocate more memory by temporary stopping all workers.

Any of the solutions proposed above might solve the problem in userland. However, it does not solve the problem in kernelland. A process that has multiple threads share its memory with all of them. Inside the kernel, this is achieved by sharing the object mm inside the `struct task_struct`[2] of all threads. This object represents the virtual memory of the process and is protected with a read-write semaphore. This object allows multiple readers to be in the critical region to perform read to this object but it only allows one writer in the region at a time and without readers. This leads to the following situation: A worker performs a `malloc()`. If there is not enough memory on main memory, a page fault occurs and the worker is blocked while it is served while holding the mm lock. When the worker returns from the schedule call, it blocks on the EFD while still holding the kernel lock. Then, if the leader or another worker does calls `malloc()` or has a page fault, it will try to acquire the mm lock and it will deadlock. To solve the kernelland problem, two options have been studied:

- The first option is to lock all memory and use a pool allocator approach, just like the solution for the userland problem. But this do not prevent from blocking while holding other locks from other shared structures of the kernel that we are not aware.
- The second option is to modify the EFD blocking mechanism inside the kernel to avoid blocking tasks holding a kernel lock. To do so, there are two options:
  - The first option is to block a user managed thread not just after it returns from a context switch but just before returning to user space. This way, any held lock will be freed. The problem of this approach is that are multiple return points to user space that should be instrumented. The final kernel extension would then be invasive.
  - The second one is to rely on the `CONFIG_LOCKDEP`[3] option which adds to the `struct task_struct` the field `lockdep_depth` that keeps the count of held locks by each process. However, this is not the only information kept. This config option keeps specific information of each lock, and hence, each time a lock is acquired or released (which is something that happens quite often), some data structure manipulation happens, decreasing performance. This option is intended for debugging, and not for production environments. In the long term, the CONFIG_LOCKDEP option could be split into two configurations, one to just keep the number of held locks per processes, and the other to enable the rest of the CONFIG_LOCKDEP features.

[2]The `struct task_struct` object contains all the data structures required for each process or thread in the system. It is the Linux Kernel implementation of the classic Process Control Block (PCB) in OS theory.

[3]CONFIG_LOCKDEP is a Linux kernel debugging mechanism to detect deadlocks at runtime and provide useful information to the kernel developer

The KAM approach was implemented and tested in conjunction with a small task-based runtime developed for this project. The CONFIG_LOCKDEP option was used to prevent Linux Kernel locks successfully. However, the user space locking part lead the development continue to the next approach: KPM.

3.2.3 Kernel Passive Management (KPM)

In KPM, the kernel does not retain monitored threads just after being unblocked. Instead, it only adopts a passive role in which it sends notifications to user-space. Because of this relaxed approach nature, there are no more fundamental complex locking problems, although the runtime does not have so much control of the execution flow. This idea has been studied in two scenarios: assign an EFD per monitored worker or per CPU.

In the first case, each monitored worker has two EFD that the kernel uses to send events. When a worker blocks, it writes a "1" to the EFD_BLOCK, when it unblocks it writes a "1" to the EFD_UNBLOCK. If the worker blocks and unblocks several times before the leader thread reads its EFDs, the value in the EFDs increases, effectively counting the number of times each event occurred. When the leader thread reads the values, the EFD internal counters are reset to zero.

The leader thread monitors all EFDs of all workers with `epoll()` and keeps track of its states. This global view of the execution status allows the leader thread to identify which CPUs are idling by comparing the amount of blocked workers against the total amount of workers assigned to the CPU. To calculate the number of ready workers per CPU, the leader thread keeps a user-space counter of ready workers per CPU that updates with the readings of the EFD. After returning from each epoll() call, the two EFD of a worker are read and subtracted (unblocked[4]- blocked) and then added to the user-space counter. Once an Idle CPU has been identified, the runtime decides if it is worth waking up another worker there.

However the runtime does not really needs to keep track of the status of individual workers but of CPUs. In the second option, instead of assigning two EFD per workers, the kernel maintains a single EFD per CPU as seen in figure 3.3. Whenever the kernel scheduler identifies a monitored thread blocking or unblocking, it writes the EFDs of the CPU where the worker is executing. The leader thread then operates as in the last approach and keeps a user-space counter of ready workers per CPU.

If the EFD counters for blocking and unblocking events are kept in separate EFDs, there is the risk of not getting the real number of blocked workers. It is possible for the leader thread to be preempted between reading the two EFD of a CPU. If a worker gets blocked or unblocked while the leader thread is preempted, the EFD internal values will change and when the leader thread is able to run again, it will not have the real picture of the CPU. In fact, even if the leader thread is not preempted, the EFDs might change in the time lapse between the reading of the two[5].

To prevent this situation, KPM uses a single EFD per CPU. Because EFD are implemented as single 64 bits unsigned integers, the blocked and unblocked counters are stored in the first 32 bits and the next 32 bits respectively. In this proof-of-concept eventfd overflow problems are not being considered. Namely, 2^{32} thread blocks without reading the EFD will unnoticeably overflow the blocked counter and corrupt the unblocked counter. Also, 2^{64} total writes or 2^{32} unblocks will overflow the EFD maximum storage. Ideally, instead of an EFD a more complex structure should be used that could keep track of the number

[4]The first time a thread is run after being created, it increases its unblocked counter.
[5]This issue was observed to occur quite often while testing the feature with a simple example.

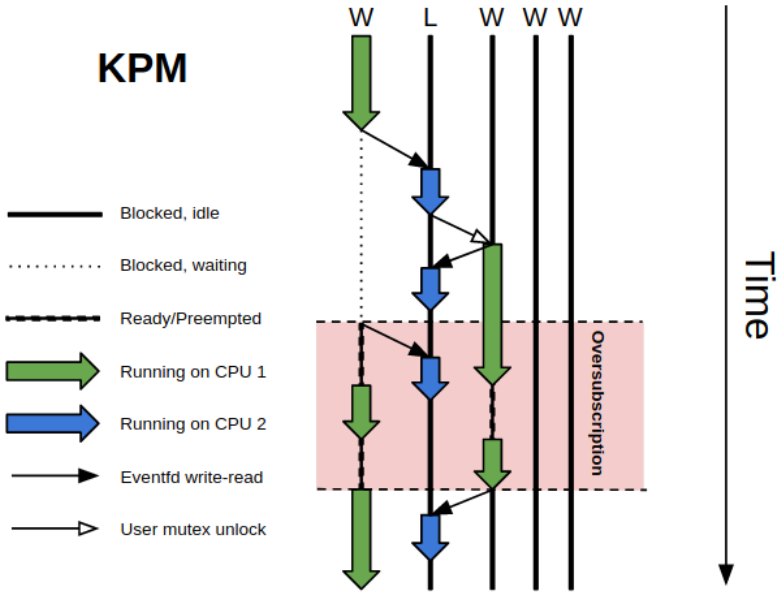

Figure 3.3: Kernel Passive Management (KPM) example workflow. Each vertical line shows the state of a worker as time advances. Bold lines represent blocked workers that are ready to be run at any time when some other processes unblocks them explicitly. Dotted lines are workers blocked that are waiting for some event such as data from a storage device. Big colored arrows show workers running on a CPU. Small black arrows depicts an EFD being written at the tail of the arrow and being read at the arrowhead. Small white arrows illustrate a worker performing an unblock action at the tail of the arrow against the worker at the arrowhead that was blocked on a user-space mutex. The workflow is as follows: The left-most worker blocks waiting for data and writes the WORKER_BLOCK flag to its CPU's EFD. Next, the Leader Thread is waken up because of the EFD being monitored by an epoll syscall. The Leader thread updates the state of the CPU corresponding to the EFD that originated the event and unblocks an idle worker blocked on a user-mutex to run on the idle CPU. The second worker wakes and writes the WORKER_UNBLOCK event on its CPU's EFD and continues running normally. The Leader thread wakes on the event and updates the CPU state. While the second worker is running, the first worker resolves its blocking state, writes the WORKER_UNBLOCK on its CPU's EFD, it is scheduled on the CPU preempting the second worker and the oversubscription period beings. The Leader threads wakes to process the event and sleeps again because there is nothing that it can do. Now both the first and the second workers are in the ready state and bound to the same CPU. The Linux process scheduler chooses one of the two processes and allows it to run for a period of time until it decides that the other worker needs to run and a preemption occurs. Note that as workers are preempted, their CPU's EFDs are not being written. The oversubscription period ends when, in this case, the second worker finishes executing a task and decides to stop running after detecting the oversubscription problem by inspecting the CPU state that the Leader thread has kept updated. When leaving the CPU, the worker generates a WORKER_BLOCK event which wakes up the Leader thread again to update the CPU sate. In the meantime, the first worker continues to run normally.

of ready threads instead of both the number of blocked and unblocked threads. However, the design is simplified using the already in-kernel-tree eventfd structure. In any case, a counter overflow is unlikely to occur given that the leader thread should be scheduled much before than $4,294,967,296$ (2^{32}) worker thread block.

It is interesting to note that the EFDs must be created as non blocking file descriptors (the epoll syscall still blocks on the EFDs until an event occurs). It could happen that multiple runtime threads monitor the EFDs from user-space. If multiple threads try to read the same EFD at the same time, only one will actually read its internal counter, while the others will block until new data arrives. This is a problem, because the thread would bock in the read syscall of a single EFD instead of the epoll syscall that monitors all EFDs. Preventing blocking reads, the second syscall would return immediately with a return value of zero.

In KPM, both the selective wake up and worker fairness properties are satisfied given that the entire scheduling of workers is done by the user-space runtime, which is assumed to be fair.

The main KPM drawback is oversubscription. Because unblocked threads are not retained in-kernel, multiple threads bound to the same CPU might try to run at the same time, causing cache pollution and unnecessary preemptions. This situation can happen if a blocked thread is unblocked on a CPU where the runtime had waken up a thread. Is left to the runtime the task of periodically check the state of per CPU ready workers counters to determine if a worker should voluntarily leave the CPU to reduce oversubscription. A more detailed example is given in the 3.3 and the overhead of it is analyzed in the 4 chapter.

3.3 User-Monitored Thread (UMT)

The flagship of this thesis design is the *User-Monitored Thread* (UMT) model. UMT is based on the KPM kernel extension presented before (see section 3.2.3) plus the user-space runtime layer that makes use of KPM.

This sections presents a UMT overview and then exposes the implementation details of both kernel and user-space runtime code. The proposed design and implementation are based on several relaxed assumptions that simplify the design and do not particularly compromise performance. See the 5.1 section for more details on the planned work to fix this details.

In UMT, the Linux kernel uses a communication channel to notify a user-space application of blocking and unblocking events among their threads. Hence, UMT includes a mechanism for registering threads in kernel space to be monitored, a kernel mechanism to actually monitor the threads and a channel to communicate kernel and user-space.

3.3.1 Overview

The details of the UMT functionality are given in the following Sections, but it is sketched in figure 3.4. In this figure, the W_i are worker threads and L denotes the application's *Leader Thread* whose role is to monitor the communication channel. Basically:

- At time T1, four workers W1, W2, W3 and W4 are bound to CPU's C0, C1, C2 and C3 respectively. The Leader Thread is not bound to any CPU and is waiting for UMT events. A pool of idle workers remain blocked until they are needed.

- At time T2, the worker W1 blocks because of an I/O operation and the Leader Thread is notified of the event.
- At time T3, the Leader Thread wakes an idle worker from the pool and waits again for more events. (When W5 wakes, it would also generate an unblock event which is omitted for simplicity). Worker W5 is now running on a CPU; without the proposed mechanism, it would have been idle.
- At time T4, W1 is unblocked after the I/O operation finishes. An unblocking event is generated and the Leader Thread wakes up. Because there is not any free CPU at the moment, the Leader Thread waits until it momentously preempts another worker. Once it does so, it reads the UMT events and registers that multiple workers (W1 and W5) are running on the same CPU (C0).
- At time T5, after the W5 worker finishes executing tasks, it checks the Leader Thread registers and realizes that there is an oversubscription problem affecting its current CPU. To fix the problem, the worker self surrenders and returns to the pool of idle workers. This generates another event that wakes up the Leader Thread and updates the register of events.
- At time T6, the oversubscription problem has ended and the four workers are running normally.

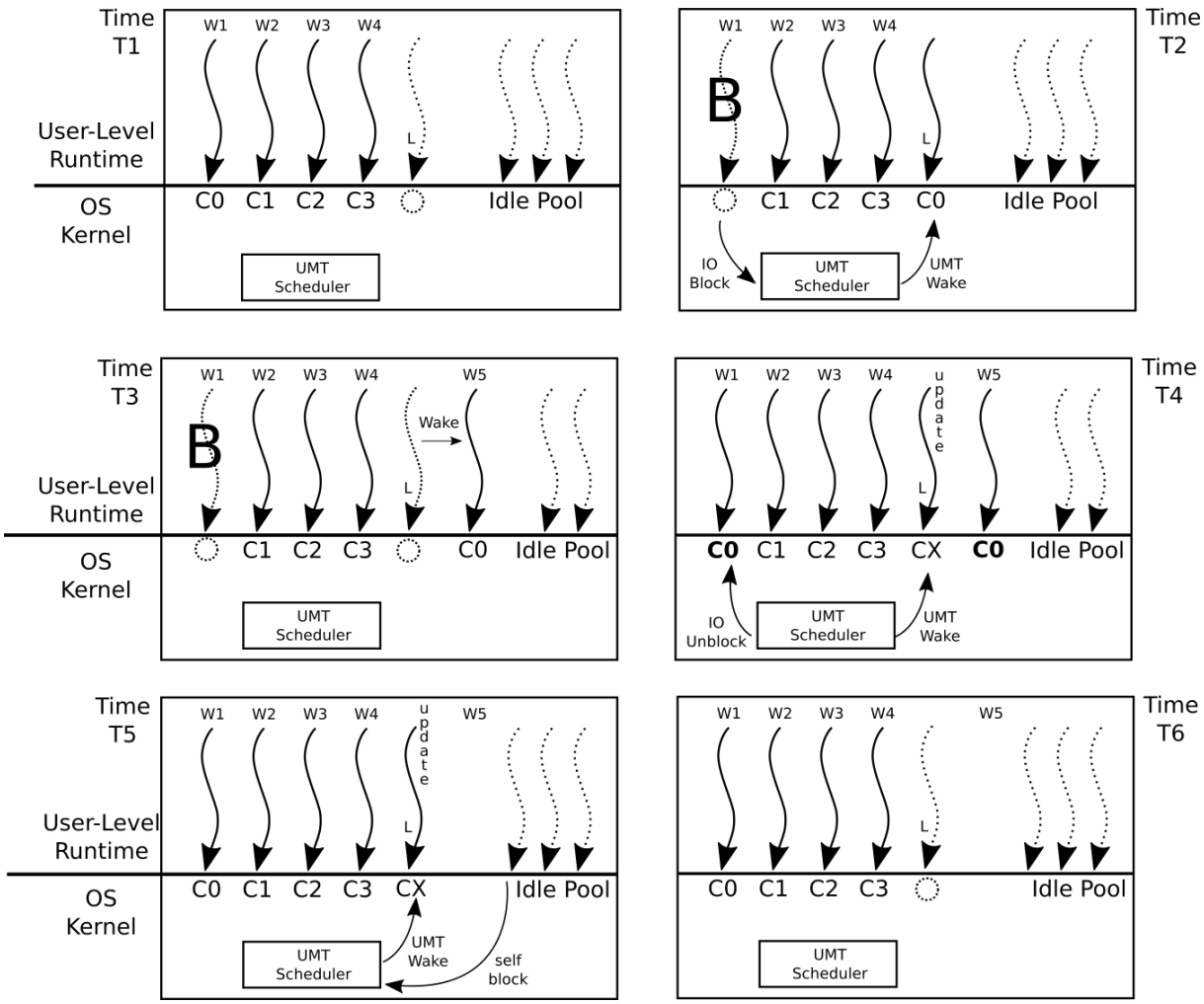

Figure 3.4: UMT model overview example

3.3.2 Linux Kernel Side

This section exposes the implementation details of the KPM extension into the Linux Kernel. However, is worth noting that the infrastructure described here is provisional, and only provides the minimum set of features to operate with KPM and test it. It is part of the future work to clean and improve this interface.

Two new syscalls have been added to manage monitored threads from user-space as can be seen below. The processes to add a new syscall is documented in the official kernel website [29] and hence, it is not explained here.

```
1 int kpm_mode_enable(int * cpufds)
2 void ctlschedkpm(unsigned char func, unsigned char activate, unsigned char * get)
```

3.3.2.1 Syscall: kpm_mode_enable()

A processes that wants to monitor its threads must use the `kpm_mode_enable()` syscall to enable KPM mode. Inside the syscall, the kernel initializes the necessary data structures to manage the threads and returns the number of CPUs being monitored. For each monitored CPU, an entry in the `cpufds` argument corresponding to the CPU ID is filled with the EFD file descriptor for the CPU. The syscall returns the number of monitored CPUs.

The complete code for the syscall is shown on the listing 3.1. The SYSCALL_DEFINE macro is used to define a new system call. The last number at the end of the macro is the number of allowed arguments, the first argument must be the system call name and the rest of the parameters must be passed afterwards, first the type and then the argument name. The special macro *_user* declares the corresponding argument as a user space pointer which is used by a preprocessing kernel compilation step to detect whether the pointer is used correctly as is explained later in this section.

The first action taken by the syscall is to get a reference to the processes (not thread) `task_struct` of the calling thread in line 8. To do so, all threads contain the pointer `group_leader` into its `task_struct` that directly points to its parent (in the case of the parent, the pointer points to itself). The `task_struct` of the current processes/thread is always accessible through the `current` variable.

Then, a vector is allocated in the parent's `task_struct` to keep the EFD file descriptors for each CPU at line 10. The memory is allocated using the `kmalloc()` Linux Kernel function to allocate kernel memory. The GFP_KERNEL flag indicates that the memory allocated does not need any special consideration[6]. The actual number of CPUs is kept in the global kernel variable `nr_cpu_ids`. If there is not enough memory, the syscall returns the standard error code ENOMEM.

Next, the EFDs are created by calling the eventfd2 syscall from inside the kernel. A syscall can be called from inside the kernel by prepending the "sys_" keyword on the syscall name as seen in line 16. The resulting function name is the actual syscall function that gets called from user-space after the transition to kernel mode and the general syscall handler code. The EFDs are created with the EFD_NONBLOCK which prevents reads to the file descriptor to block as explained in the section 3.2.3. The syscall is called once per CPU unless an error occur.

In case of error, a goto statement jumps to the end of the syscall where the previous actions are undone (in this case, allocate memory and close opened EFD if any). This is a common kernel pattern

[6] An example of "special consideration" is the flag GFP_DMA that can be used instead to allocate memory suitable for DMA transefers, or GFP_NOWAIT that avoid blocking in kmalloc (by default kmalloc can sleep)

for managing errors in an elegant fashion. Inside the kernel, special care must be taken to avoid memory leakages, in this case, returning the syscall before freeing the memory or not closing the EFD would cause a permanent memory leak in kernel memory because the objects' allocated memory references would have been lost. To notify errors, the generic Linux Kernel `printk()` function is used to add a log in the kernel ring buffer available from user-space when running the *dmesg* command or when examining */var/log/messages* if the syslogd or klogd deamons are running.

The kernel does not need to work with file descriptors, instead the function `eventfd_ctx_fdget()` is used at line 24 to get the `struct eventfd_ctx` which is the relevant data structure really used to actually write and read from an specific EFD.

Finally, the EFDs file descriptors are copied from kernel memory to user memory using the common kernel utility `copy_to_user()` in line 32. This function is used to perform some common checks regarding the correctness of the user-space pointer. The calling user-space processes might have passed to the system call an erroneous or malicious pointer that could originate a memory error or corruption if the kernel tried to write to it. The same applies when copying data from user-space, although in this case is not needed.

3.3.2.2 Syscall: ctlschedkpm()

Once `kpm_mode_enable()` returns, the calling process now has the necessary EFDs to monitor the CPUs. However, threads will not write to the CPUs EFD until they have explicitly requested that want to be monitored. To do so, each thread pretending to be monitored must call the new `ctlschedkpm()` syscall. This syscall is used for both set and get the monitored thread status (enabled or disabled). The listing in 3.2 shows the syscall code.

The new syscall `ctlschedkpm()` requires three arguments. The first one, `func`, expects either the macro "SCHED_UM_ACT_SET" to enable or disable monitorization or "SCHED_UM_ACT_GET" to retrieve the monitoring status of the calling thread. The second argument, activate, is the actual value that will be set if "SCHED_UM_ACT_SET" is chosen for `func`. Valid values for activate are 0 or 1. The last argument is a pointer that the kernel will fill with the current thread status if "SCHED_UM_ACT_GET" was chosen for `func`.

The new syscall first action is to check whether the parent process of the calling thread has called the `kpm_mode_enable()` syscall at line 5, otherwise, it returns an error. If the user has specified the SCHED_UM_ACT_GET flag, the internal status of the thread is copied to user-space in line 7. If the user has specified SCHED_UM_ACT_SET, it is checked if the `activate` parameter value is correct and then it proceeds to enable monitorization.

To do so, it is first necessary to disable kernel preemption for the current CPU using the function `preempt_disable()` which can then be enabled again with `preempt_enable()`. Disabling preemption is necessary because the activation of monitorization consists of several steps that must be done atomically before the kernel scheduler is called. Because the Linux kernel allows preemption on both user and kernel space as explained in the background section 2.2.2, the syscall code can be preempted at any point (by the return point of a timer interrupt, for example) unless preemption is explicitly disabled. Disabling preemption guarantees that the code will not be preempted by any other process.

The initialized variables are `um_activate` to enable EFD operations and `um_oldcpu` to keep track of the last CPU the thread was running on. To get the current cpu, the `smp_processor_id()`

```c
SYSCALL_DEFINE2(kpm_mode_enable, int __user *, cpufds) {
  unsigned int kcpusfd[nr_cpu_ids];
  struct eventfd_ctx * ctx;
  int ret;
  int i;
  struct task_struct * gldr;
  ret = nr_cpu_ids;
  gldr = current->group_leader;
  // Allocate memory for eventfd structures per CPU;
  gldr->se.um_cpufds=kmalloc(sizeof(struct eventfd_ctx *)*nr_cpu_ids,GFP_KERNEL);
  if (!gldr->se.um_cpufds) {
    ret = -ENOMEM;
    goto exit;
  }
  for (i = 0; i < nr_cpu_ids; i++) { // create a an EFD for each CPU
    kcpusfd[i] = sys_eventfd2(0, EFD_NONBLOCK);
    if (kcpusfd[i] < 0){
      printk(KERN_WARNING "kpm_mode_enable: couldn't create EFD\n");
      ret = kcpusfd[i];
      goto efd_free_partial;
    }
  }
  for (i = 0; i < nr_cpu_ids; i++) { // acquire fd context for each given user fd
    ctx = eventfd_ctx_fdget(kcpusfd[i]);
    if (IS_ERR(ctx)) {
      ret = PTR_ERR(ctx);
      goto efd_free;
    }
    gldr->se.um_cpufds[i] = ctx;
  }
  // copy the cpu fds to userspace
  if (copy_to_user(cpufds, kcpusfd, sizeof(int)*nr_cpu_ids) != 0) {
    printk(KERN_WARNING "kpm_mode_enable: couldn't copy from userspace\n");
    ret = -EINVAL;
    goto efd_free;
  }
  return ret;

efd_free:
  i = nr_cpu_ids;
efd_free_partial:
  i--;
  for (; i >= 0; i--) {
    sys_close(kcpusfd[i]);
  }
  kfree(gldr->se.um_cpufds); // free cpufds array
  gldr->se.um_cpufds = NULL;
exit:
  return ret;
}
```

Listing 3.1: Source code for the Linux kernel kpm_mode_enable system call

```
1  SYSCALL_DEFINE3(ctlschedkpm, unsigned char, func, unsigned char, activate,
2                  unsigned char __user *, get) {
3    int i;
4    int err = 0;
5    if (current->group_leader->se.um_cpufds != NULL) {
6      if (func & SCHED_UM_ACT_GET) {
7        if(copy_to_user(get, (char *) &current->se.um_active, sizeof(char)) != 0) {
8          printk(KERN_WARNING "ctlschedkpm: couldn't copy to userspace\n");
9          err = -1;
10       }
11     } else if ( func & SCHED_UM_ACT_SET) {
12       if ((activate == 0) || (activate == 1)) {
13         preempt_disable();
14
15         int cpu = smp_processor_id();
16         current->se.um_oldcpu = cpu;
17         current->se.um_enabled = 1;
18         current->se.um_active = activate;
19
20         preempt_enable_no_resched();
21
22         trace_kpm_active(current->se.um_active, activate);
23       } else {
24         printk(KERN_WARNING "ctlschedumfd: %hhd not valid activate argument\n",
25                activate);
26         err = -1;
27       }
28     } else {
29       printk(KERN_WARNING "ctlschedkpm: %hhd option not supported\n",func);
30       err = -1;
31     }
32   } else {
33     printk(KERN_WARNING "ctlschedkpm: called for a non user managed thread\n");
34     err = -1;
35   }
36
37   return err;
38 }
```

Listing 3.2: Source code for the Linux kernel ctlschedkpm system call

kernel function is used.

After all data structures are initialized, `preempt_enable_no_resched()` function at line 20 enables preemption again. This function, in contrast with `preempt_enable()` does not has the side effect of checking whether the current processes should be preempted. The *_no_resched* variant is used because at this point the syscall has almost finished and preemption will be checked again at the user space return point.

A generic Linux Kernel TRACE_EVENT tracepoint [30] has been included line to ease the debugging processes. When the Linux Kernel TRACE_EVENT feature is disabled system wide, the system performance is almost not affected. This is because the tracepoint CALL instruction within the kernel code is overwritten by NOPS instructions. When the user enables the tracepoints at runtime, Linux rewrites its own code to add back the CALL instructions. To do so, the kernel simply keeps a list of the memory locations of all tracepoints. For this reason, TRACE_EVENTS is compiled but disabled by default in most Linux distributions.

```
1 static void __sched __schedule(bool preempt) {
2         kpm_pre(current);
3         main_schedule(preempt);
4         kpm_post(current);
5 }
```

Listing 3.3: Source code for the Linux Kernel scheduler wrapper

3.3.2.3 Scheduler Wrapper

The actual EFD writing points instrumentation has been placed into a wrapper around the main Linux Kernel `__schedule()` function. This function is the common entry point for all possible paths that lead to a context switch as explained in section 2.2.2. The genuine `__schedule()` function is now substituted by a wrapper shown at listing 3.3 that writes the EFD blocked counter into `kpm_pre()` before calling the original `__schedule()` (now named `main_schedule()`) and writes the unblocked EFD counter inside `kpm_post()` when `__schedule()` returns. The argument passed to both `kpm_pre()` and `kpm_post()` is the current processes `task_struct`.

Writing a block event is done by the `kpm_pre()` function shown in listing 3.4. At this point, the current thread is about to call the main scheduler function which might cause its execution to cease either because it is being preempted, blocked or destroyed. Preemption is disabled all function long to avoid data races as explained before. This function first checks whether the calling thread has to be monitored by checking the `um_active` flag at line 3. Next, it is checked whether the current thread is being preempted or blocked by inspecting the current thread state. Preempted threads are not considered in the EFD count. Whenever a thread is preempted, its CPU does not become idle, but another thread starts running on it. Because one of the main objectives is to minimize the amount of wasted CPU time, it is not necessary to inform user-space of when a thread is preempted but only when it is blocked.

In the Linux Kernel, each thread has a `state` variable stored in its `task_struct` which represents the current execution status. Table 3.1 shows some of the possible values and figure 3.5 shows the flow chart of states. The most relevant ones are TASK_RUNNING, TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE. As long as a thread is in the scheduler's run queue (the ready list) regardless of it is actually running or not on a CPU, its status is TASK_RUNNING. If the thread pretends to block it changes its state to either TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE to indicate that it might be woken up prematurely to process signals or not respectively. When the scheduler sees any of this two states, it removes the current thread from the run queue. When a thread is preempted by another, its state is not changed and remains as TASK_RUNNING. Hence, to identify a preemption at the schedule level function, the `state` is checked to be zero (the value of TASK_RUNNING macro) as can be seen at line 9. Also, the current state is saved for future use in the `um_prev_state` variable as it is explained later.

The EFD struct corresponding to the current CPU is retrieved from the process of the current thread at line 14. Then, the `eventfd_signal()` from the eventfd asynchronous API is used at line 16 to write without blocking in case of overflow. If the returned value is not the same as the value written, a non-blocking overflow occurred and the kernel is stalled with the kernel BUG_ON macro[7].

[7]This macro should not be used here because overflowing this counter will not compromise the entire system, only the user-space runtime. In any case, it is used for debugging purposes

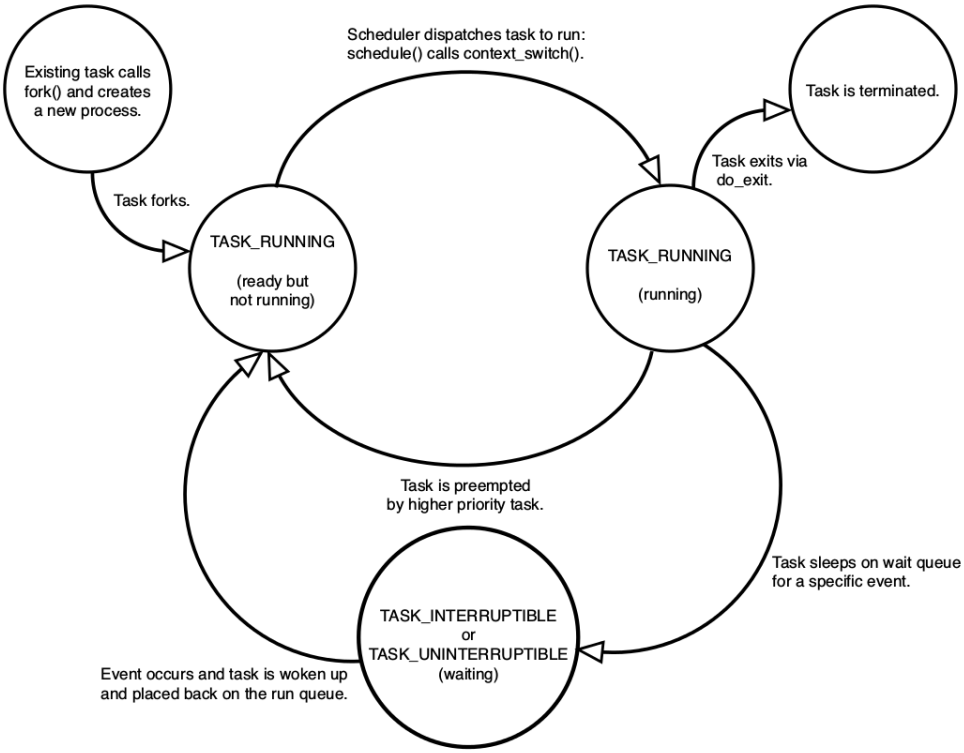

Figure 3.5: Linux Kernel Process State [31]

FLAG	Description	Value
TASK_RUNNING	In the runnable queue. It might be running or not.	0
TASK_INTERRUPTIBLE	Blocked but interruptible.	1
TASK_UNINTERRUPTIBLE	Blocked, no interruptible.	2
__TASK_STOPPED	Task has been stopped	4
__TASK_TRACED	Task is being traced	8

Table 3.1: Linux Kernel process states

Finally, preemption is enabled again skipping the schedule check. Otherwise, it could happen that after enabling preemption, the schedule function is called a second time at this point, forcing an undesired call to `kpm_pre()`.

When the process is scheduled again, it exits the schedule function and proceeds to write the unblock event in the `kpm_post()` function shown in listing 3.5. The process is similar to `kpm_pre()`, although some special care must be taken for migrated threads. Runtime workers are pinned threads, meaning that will not be migrated because of the OS will. However the runtime can instruct the kernel to pin a worker to another CPU anytime.

A worker can be migrated while it is running, blocked, ready or stopped because of the syscall `sched_setaffinity()`. However, for a processes to change its CPU it must cease running first by calling the schedule function. Then, its data structures are simply moved from the source CPU run queue to the target CPU run queue. To preempt the running process, the kernel wakes up a high priority real time kernel thread named "migration" that forces a preemption through the common scheduler mechanisms (real time threads preempt immediately all the other schedulers threads but deadline threads).

If a migration is requested on a blocked, stopped or a ready thread that has just awaken after being

```
static void kpm_pre(struct task_struct * prev) {
  preempt_disable();
  if (prev->se.um_active) {
    int cpu = smp_processor_id();
    int state = prev->state;

    trace_kpm_pre(state, prev->se.um_oldcpu);
    prev->se.um_prev_state = state;
    if (state) {
      const __u64 towrite = 1;
      __u64 written;
      struct eventfd_ctx *ctx;

      ctx = prev->group_leader->se.um_cpufds[cpu];
      /* non-blocking write, no need to enable preemption */
      written = eventfd_signal(ctx, towrite);
      BUG_ON(written != towrite);
    }
  }
  preempt_enable_no_resched();
}
```

Listing 3.4: Source code for the Linux kernel block EFD event

blocked or stopped, it means that the thread wrote the block event prior to leaving its last CPU and that it will write the unblocked event on the new CPU. As a consequence, for the old CPU the number of block and unblock events match and for the new CPU there is an extra unblock event. This is important because the user-space application will see that the number of ready threads is 0 for the old CPU and 1 for the new CPU (supposing that there is a single monitored thread).

If a migration is requested on a running thread, the "migration" kernel thread will preempt it, which means that when it calls the schedule function it will not write neither the block event on the old CPU nor the unblock event on the new CPU (because in both cases its state is TASK_RUNNING). However, this thread had, for sure, previously written an unblocked event on the old CPU, which means that the old CPU will lack a block event and the new CPU will lack an unblock event. Therefore, the EFD counters would be spoiled.

To fix the unmatching events problem in the case of a preemption, several options have been studied. Ideally, the `kpm_pre()` function should detect that a preemption caused by a migration is happening and force a write to the EFD (the same applies for `kpm_post()`). However, at the scheduler level, it is not possible to check that a migration is the cause. It could be checked if the next process to run is the migration thread but there is no guarantee that another user thread will preempt the process to be migrated before than the migration thread. Also, it is not cost free to simply set a flag at the `sched_setaffinity()` syscall because it would have to be done atomically in conjunction with the process' allowed CPU bitmask, which would require to protect a new code region with locks (the CPU bitmask indicates the allowed set of CPUs where a processes can run). The CPU bitmask could be checked in `kpm_pre()` to anticipate the scheduler decision, however if a call to `sched_setaffinity()` modifies the CPU bitmask in parallel just after the CPU bitmask check in `kpm_pre()`, the EFD would not be written and the scheduler would perform a migration.

A more conservative approach has been chosen. The `kpm_post()` code first checks if the current CPU is different from the last time `kpm_post()` was run by this process. If it is the case, then a

```
1  static void kpm_post(struct task_struct *prev) {
2    preempt_disable();
3
4      if (prev->se.um_active) {
5      int force = 0;
6      int oldcpu = prev->se.um_oldcpu;
7      int cpu = smp_processor_id();
8
9      trace_kpm_post(prev->state, prev->se.um_prev_state,
10                    oldcpu, cpu);
11
12     // if we are on a new cpu for the first time
13     if ( cpu != oldcpu ) {
14       // and the process was preempted last time was running
15       // in the old cpu, then do the extra write
16       if (prev->se.um_prev_state == 0) {
17         const __u64 towrite = 1;
18         __u64 written;
19         struct eventfd_ctx *ctx;
20
21         force = 1;
22         ctx = prev->group_leader->se.um_cpufds[oldcpu];
23         /* non-blocking write, no need to enable preemption */
24         written = eventfd_signal(ctx, towrite);
25        BUG_ON(written != towrite);
26       }
27       prev->se.um_oldcpu = cpu; // update cpu
28     }
29
30     if (prev->se.um_prev_state || force) {
31       const __u64 towrite = ((__u64) 1) << 32);
32       __u64 written;
33       struct eventfd_ctx *ctx;
34
35       ctx = prev->group_leader->se.um_cpufds[cpu];
36       /* non-blocking write, no need to enable preemption */
37       written = eventfd_signal(ctx, towrite);
38      BUG_ON(written != towrite);
39     }
40   }
41
42   preempt_enable_no_resched();
43 }
```

Listing 3.5: Source code for the Linux kernel unblock EFD event

migration has just happened. Then it is checked if a preemption occurred by inspecting if the last status of the processes when `kpm_pre()` was called is TASK_RUNNING. If so, the missed block event is done at line 24. Effectively, because the block event is done too late, this means that the EFDs will not reflect the correct status of the processes while it is migrating, however, because migrations should not be done often, this can be considered a corner case.

After the extra block event write, the analogous normal unblock write is done either if the process state before calling schedule (stored in `um_prev_state`) reveals that was not preempted or the force flag is set. This flag is set at line 21 if a migration with preemption has been detected.

It has been explained that both `kpm_pre()` and `kpm_post()` enable and disable preemption before operating. Ideally, this should be done inside the original scheduler function given that this function also disables preemption. The problem is that inside the original scheduler function, preemption is enabled at

assembler level just after doing the context switch, meaning that `kpm_post()` should also run at that level. For simplicity, pre and post function have been placed outside, although a redundant preemption enable and disable is done. In fact, it is possible for a timer interrupt to trigger between the exit of `kpm_pre()` and the call to the main scheduler which could lead to another call to scheduler. This is a problem because `kpm_pre()` would be called twice, however, this is unlikely to happen and easy to detect from user-space. For this reason it remains as future work.

3.3.3 User-Space Runtime Side

In order to validate the proposal, the *Nanos6* task-based runtime of the *OmpSs* [1] programming model has been adapted to exploit the new KPM feature of the Linux Kernel. Originally, the Nanos6 threading model is based on an explicit management of thread binding: as long as there are ready tasks to execute, the runtime keeps a single worker binded to each CPU. Workers continuously ask for more tasks to execute and only leave the CPU voluntarily when: no more tasks are left, an explicit *taskwait* prevents the task to continue until all their childs complete, or the next ready task to execute is already bound to another worker (in which case there is a swap of workers). The new Nanos6 threading model adds the possibility to have multiple workers bound to the same CPU although it is desired that only one of them is in the ready execution state and the others are blocked. A per CPU counter of workers in the ready state is used to detect when a CPU is idling. When one is found, an idle worker is awaken to be run there. Workers use the CPU ready counters to detect oversubscription and fix it by voluntarily stopping.

3.3.3.1 Leader Thread

Nanos6 initialization phase creates a single not bound Leader Thread and a per CPU bound worker thread. As part of the early Leader Thread initialization phase shown at listing 3.6, KPM mode is enabled. To do so, it calls the new `kpm_mode_enable()` syscall at line 7 using the `syscall()` system call to invoke a syscall using its numeric identifier[8]. Then, it registers the EFDs in an epoll object (created before and not shown here) at line 15. The Nanos6 CPU object is bound to the EFD in the `epoll_ctl()` call, hence, when an event is triggered, `epoll_wait()` returns the CPU object pointer. When workers start running and before starting to execute any task, they call the `ctlschedkpm()` syscall to instruct the kernel to monitor their actions.

A simplified version of the Leader Thread main loop is shown at listing 3.7. While the global variable `mustExit` is not set, the leader thread performs a blocking read over all EFDs using the `epollfd` *epoll* file descriptor at line 12. When one of the monitored threads produces an event, the Leader Thread gets unblocked and reads the EFDs by calling the `process_events()` function described later. Then, it proceeds to wake up additional workers on the idle CPUs by calling the `wakeUpIdleWorkersOnIdleCPUs()` function.

If `mustExit` is set, another final non-blocking call to `epoll_wait()` (last parameter is zero) is done to processes the remaining events before exiting. As a measure to detect errors in the debugging phase (which requires the DEBUG macro to be set), all CPU ready counters are checked to be zero when Nanos exits (i.e. there are 0 ready threads in all CPUs). If at some point Nanos or the kernel

[8]The standard way to call a syscall by its name is to add a libc wrapper for the syscall. Because it has not been done here, the generic `syscall()` system call is used

```
1  void LeaderThread::enableKPM() {
2    std::vector<CPU *> const &cpurefs = CPUManager::getCPUListReference();
3    int i, rc, ncpus, cpuid;
4    int cpufds[CPU_SETSIZE];
5    struct epoll_event ev;
6
7    ncpus = syscall(KPMMODEENABLE, cpufds); // enable KPM
8    /* process error on ncpus < 0 */
9
10   for (CPU *cpu : cpurefs) {
11     cpuid = cpu->_systemCPUId;
12     cpu->setefd(cpufds[cpuid]);
13     ev.events = EPOLLIN;
14     ev.data.ptr = cpu;
15     rc = epoll_ctl(epollfd, EPOLL_CTL_ADD, cpufds[cpuid], &ev);
16     /* process error on rc == -1 */
17   }
18 }
```

Listing 3.6: Source code for the Nanos6 KPM initialization

```
1  void LeaderThread::body() {
2    while (true) {
3      if (not mustExit) {
4        #ifdef DEBUG
5        nev = epoll_wait(epollfd, events, numCPUs, 0);
6        /* processes errors on nev == -1 */
7        if (nev == 0) break;
8        #else
9        break;
10       #endif
11     } else {
12       nev = epoll_wait(epollfd, events, numCPUs, delay_ms);
13     }
14     process_events(nev, events);
15     wakeUpIdleWorkersOnIdleCPUs();
16   }
17 }
```

Listing 3.7: Source code for the Nanos6 Leader Thread Body

implementation have misbehaved, Nanos is likely to exit with a CPU counter with a non zero value. The last check is done to ensure that all events have been processed.

The code at listing 3.8 shows how events are processed. The EFD counters are consumed through the standard `read()` system call, which resets the EFD counters to zero. The number of blocked and unblocked events are separated from the reading using masks. By subtracting the number of unblocked tasks to the number of blocked tasks, the number of ready workers of the associated CPU EFD is known. However, because each read operation erases the count, it is necessary to keep a user-space per CPU count of the ready threads by simply adding the result of the subtraction. Each Nanos6 CPU object maintains an atomic counter which is updated at line 13. The operation is repeated until there are no more events, this is done because a preemption while updating the counters could result in an outdated picture of ready workers.

Listing 3.9 shows the code that uses the CPU counters to determine whether a worker should be woken up on an idle CPU. In the code, for each CPU it is checked whether the CPU meets the conditions to wake

```
1  void LeaderThread::process_events(int nev, struct epoll_event *events) {
2    CPU *cpu;
3    uint64_t status;
4    unsigned int nblocks, nunblocks, ntotal;
5
6    while(nev) {
7      for (int i = 0; i < nev; i++) {
8        cpu = (CPU *) events[i].data.ptr;
9        status = cpu->read_status(); //non-blocking read
10       if (status) {
11         nblocks   = 0xFFFFFFFF & status;
12         nunblocks = status >> 32;
13         cpu->updateReadyWorkers(nunblocks - nblocks);
14       }
15     }
16     nev = epoll_wait(epollfd, events, numCPUs, 0); // non blocking
17     /* proccess errors on nev == -1 */
18   }
19 }
```

Listing 3.8: Source code for the Nanos6 Leader Thread process EFD events function

up a worker as seen in line 8: The CPU must be enabled, idle, and a worker must have not been awaken by the Leader thread in this CPU since the last unblock event on the CPU.

The first condition is necessary for the runtime to detect CPUs that should not be used for computation as part of the a CPU plug and play mechanism. The second condition actually checks the counter of ready workers. If 0, the cpu->isIdle() returns true. The last condition is checked to avoid the Leader thread to wake up multiple workers in the same CPU before the awakened worker had time to be scheduled. This is implemented with an atomic variable that is set at line 24 and unset when an unblock event on the CPU is found. The worker awakened by the leader thread might not be the one that unsets the atomic variable but, in any case, if an unblock event is generated it means that the CPU is no longer idle and the flag can be cleaned.

If all conditions are met, the Leader thread asks the Nanos6 task scheduler for a ready task at line 9. If the task does not have a worker assigned, the function `getIdleThread()` at line 15 is called to either get an idle thread from a pool or create a new one (second parameter needs to be false, otherwise a new thread would not be created if the idle pool is empty) if the maximum number of threads `CPU::maxWorkersPerCPU` is not exceeded. If no more threads can be created, the task is returned to the scheduler and the leader thread stops. At this point workers are not started yet, but added to the `workersToBeWoken` list at line 25. Waking up a worker requires to acquire a global lock, hence, to minimize the time the lock is held, workers to be woken are first collected and then awakened afterwards in the `wakeUpWorkers()` function at line 32.

The function `wakeUpWorkers()` shown at listing 3.10 does two actions: it updates the list of current idle CPUs and wakes up workers. In one hand, Nanos6 uses the KPM kernel feature to keep the count of ready workers, used to detect CPUs not running any worker, either because workers are blocked or because there are no workers. In the other hand, it keeps a list of idle CPUs that do not currently have any worker assigned. The idle CPU list is the original mechanism used by Nanos6 without UMT to distribute work among CPUs. When a new task is created, the list of idle CPUs is checked to quickly get a place to execute it. When a worker becomes idle it checks if it is the last running worker to place the current CPU into the list of idle CPUs. To perform this checks, an atomic per CPU counter of not idle

```
1  void LeaderThread::wakeUpIdleWorkersOnIdleCPUs() {
2    WorkerThread * replacementThread, * nextThread;
3    std::vector<CPU *> const &cpurefs = CPUManager::getCPUListReference();
4    Task * task;
5
6    for (CPU *cpu : cpurefs) {
7      if (CPUActivation::isEnabled(cpu) && cpu->isIdle()
8          && !cpu->checkScheduledWork()) {
9        task = Scheduler::getReadyTask(cpu);
10       if (task != nullptr) {
11         nextThread = task->getThread();
12         if (nextThread == nullptr) {
13           int totalWorkers = cpu->getTotalWorkers();
14           bool doNotCreate = (totalWorkers >= CPU::maxWorkersPerCPU);
15           nextThread = ThreadManager::getIdleThread(cpu, doNotCreate);
16           if (nextThread != nullptr) {
17             nextThread->setTask(task);
18           } else {
19             //return task to scheduler
20             Scheduler::returnTask(task, cpu);
21             break;
22           }
23         }
24         cpu->checkAndSetScheduledWork();
25         workersToBeWoken.push_back(std::make_pair(cpu, nextThread));
26       } else {
27         break;
28       }
29     }
30   }
31   if (workersToBeWoken.size() > 0) {
32     CPUManager::wakeUpWorkers(workersToBeWoken);
33     workersToBeWoken.clear();
34   }
35 }
```

Listing 3.9: Source code for the Nanos6 Leader Thread wake up workers on Idle CPUs function

workers is used. When a worker is to be woken on a CPU, the corresponding entry on the idle CPU list is set to false as seen in line 10. It is possible to completely replace the idle CPU list and exclusively rely on the ready workers counter, however this would require to force a counters update every time the Idle CPU list would be checked, which might slow too much the process of adding tasks, for instance. In any case, this option has not been deeply and remains has future work.

Workers are woken up at line 18 by unblocking a mutex variable in the `resume()` method of the worker object. This is the precise point in which the original Nanos6 threading model is broken, meaning that there can now be multiple ready thread per CPU at a time. In other words, if there was a binded thread at the CPU where the Leader thread has just moved another thread, it might happen that when this blocked thread resumes, it has to compete with the extra thread. However, this oversubscription problem only prevails for a limited amount of time.

Because the Leader thread runs for a small computational burst every time it is awaken, it is unlikely that another thread preempts it while using the Completely Fair Scheduler (the default). This is because most of the time the Leader thread is sleeping and hence it should be placed at the leftmost of the kernel rbtree (similar to an interactive processes) see the CFQ 2.2.2 section for more details. There are extreme cases in which the Leader thread could be overwhelmed of events to process and require a single CPU for

```
1  void CPUManager::wakeUpWorkers(std::vector<std::pair<CPU *, WorkerThread *> >
       workersToBeWoken) {
2    CPU * cpu;
3    WorkerThread *wk;
4    std::lock_guard<SpinLock> guard(_idleCPUsLock);
5    {
6      for (std::pair<CPU *, WorkerThread *> pr : workersToBeWoken) {
7        cpu = pr.first;
8        wk = pr.second;
9        if (_idleCPUs[cpu->Id]) {
10         _idleCPUs[cpu->Id] = false;
11       }
12     }
13   }
14
15   for (std::pair<CPU *, WorkerThread *> pr : workersToBeWoken) {
16     cpu = pr.first;
17     wk = pr.second;
18     wk->resume(cpu);
19   }
20 }
```

Listing 3.10: Source code for the Nanos6 Leader Thread wake up workers on Idle CPUs function

itself. However, the problem in this case might not be the Leader thread performance but an inappropriate use of the programming model such as using a too small task granularity and/or too many taskwait clauses.

3.3.3.2 Workers

Listing 3.11 shows a simplified version of the worker's idle loop. Workers ask the scheduler for tasks to execute. If there are no tasks, or the task received already has a worker assigned, the current worker adds itself to the idle list at lines 9 and 22 and blocks on a mutex. Otherwise, it executes the task at line 12. Nanos6 workers check for oversubscription just after finishing executing a task and before getting a new one. To do so, workers update the counters of ready workers at function `LeaderThread::updateCPUcounters()` as seen at line 13. This function essentially does a non-blocking read on the EFDs and then calls the same `process_events()` as seen at listing 3.8. Then, the counters are used to check whether the number of ready threads for the current CPU is greater than 1. If so, the current worker returns to the pool of idle threads to palliate oversubscription.

It could happen that just after a worker pretends to block itself to reduce oversubscription, it is preempted by another worker. This second worker could also do the oversubscription check and decide to block. This implies that both workers would sleep leading behind an idle CPU. However, if this happened, either the Leader Thread would schedule another worker in the idle CPU after detecting the situation, or a worker would use the idle CPU list to run there a new task. In any case, this issue is not a problems because in one hand it is unlikely to happen and in the other hand, if it happens the problem would be fixed naturally.

Having multiple workers competing for the same core resources might pollute its cache and drop performance. However, assuming that the application defines multiple fine-grained tasks, the noise should not last much. In the experimentation section, the oversubscription noise is carefully studied and commented on the results.

```
1  void WorkerThread::body() {
2    while (not mustExit) {
3      task = Scheduler::getReadyTask();
4      if (task != nullptr) {
5        WorkerThread *assignedThread = task->getThread();
6        // A task already assigned to another thread
7        if (assignedThread != nullptr) {
8          task = nullptr;
9          ThreadManager::addIdler(this);
10         switchTo(assignedThread);
11       } else {
12         task->handle();
13         LeaderThread::updateCPUcounters(_events);
14         int nready = _cpu->getReadyWorkers();
15         if (nready > 1) {
16           tracepoint(umfd, ust_idle, 1, nready);
17           ThreadManager::addIdler(this);
18           switchTo(nullptr);
19         }
20       }
21     } else {
22       ThreadManager::addIdler(this);
23       switchTo(nullptr);
24     }
25   }
26 }
```

Listing 3.11: Source code for the Nanos6 Worker idle loop

Chapter 4

Experimental Validation

The UMT implementation described before is now evaluated by executing two benchmarks: A synthetic benchmark that heavily uses the *mmap()* system call and the mock-up of an industry used application based on the Full-Waveform Inversion (FWI) algorithm.

In both cases the performance is evaluated when each benchmark is executed on top of the original Nanos6 runtime, and on top of the UMT Nanos6 (modified version of Nanos6 running on top of the Linux kernel with the KPM extension). Several tools have been used to analyze the results, which include the Linux kernel perf tool, the Linux Trace Toolkit next generation (LTTng) [32], the visualization tool Trace Compass, as well as the babeltrace parser to debug, visually inspect, and automatically report metrics of the benchmark traces.

4.1 Environment, Tools and Metrics

All tests have been run on a single node of the BSC's "Cobi" Intel Scalable System Framework (SSF) cluster. The node features two Intel Xeon E5-2690v4 processors with a total of 28 real cores and 56 hardware threads at 2.60GHz, 125GiB of DRAM4 memory at 2400 MHZ and an Intel DC S3520 solid state drive with 222GiB (of which only 160 are available). The Linux kernel version used is the stable 4.10.5, configured with default options. The Linux distribution used over it is a minimal install of a Suse Linux Enterprise Server (SLES) 12.2-0.

The perf tool is a complete open source performance tool distributed with the Linux Kernel source code. It is capable of getting performance counters metrics, call graph of both user and kernel space, trace Linux Kernel tracepoints and monitor scheduling latencies among others.

The LTTng tool is an open source Linux Kernel offline tracer which uses the standard Linux kernel *TRACE_EVENT* [30] interface to define a set of static tracepoints. The Linux kernel features a set of default TRACE_EVENT tracepoints placed into relevant code locations that have been used to understand the execution flow of the benchmarks. The legend for all LTTng views is shown in Figure 4.1.

Trace Compass is an open source visualization tool based on Eclipse. It has been used to analyze individual LTTng traces and validate the correct behavior of the execution. Along this Section, several Trace Compass control flow and resources views are used to support the explanations.

Finally, the babeltrace Python API has been used to develop a custom LTTng trace parsing script to extract relevant metrics. Specifically, the script reports the total benchmark execution time, the average

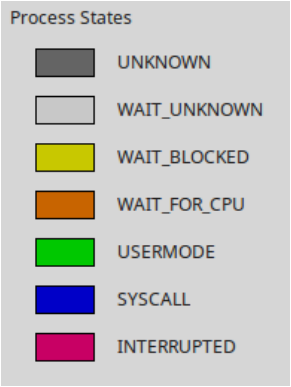

Figure 4.1: LTTng views legend

CPU usage and the following custom metrics:

- Ready Jam: Percentage of time in which multiple threads state bound to the same CPU is either running or ready. In other words, this metric is the average of the per CPU time percentage in which there is oversubscription.
- Number of in-benchmark context switch: Context switch in which the previous running thread is a preempted (not blocked) benchmark thread and the next thread to run is also a benchmark thread. Hence, this measures number of preemptions among the benchmark threads.
- Number of benchmark created threads: Number of Nanos6 worker threads, including all worker threads and the Leader Thread.

4.2 Synthetic Benchmark

In this Section, a synthetic benchmark based on the `mmap()` syscall is used to compare the performance obtained with an unmodified Nanos6 version and the modified Nanos6 with UMT, both running over the same modified kernel.

The custom mmap benchmark maps a randomly generated 100MiB file into memory and performs a set of read/update/write operations on the file. More specifically, the benchmark main task creates a specified number of independent Nanos6 tasks in which each one reads a random block of a certain size from the file, performs a certain number of floating point operations on the data and writes back the changes to disk through the `msync()` system call. The exact block size, number of blocks/tasks and number of computations are specified for each described test.

Next, it is analyzed a mmap execution in which 500 blocks of 2000 elements are read, updated and written without doing almost no computation on the update part.

Figure 4.2 shows the Trace Compass control flow view of the mmap without UMT benchmark LTTng trace. A total of 59 threads are created, of which 56 are worker threads. The view only shows 10 of them for clarity. For each Nanos6 thread, the view shows the process status (blocked, ready, running) and whether it is running in user-space or executing a system call in kernel space. It can be distinguished a pattern enforced by the disk data responses. When workers call the `msync()` system call, they block until the disk has finished writing the data. The disk serves multiple workers data requests at the same time, and when it finishes, unblocks all pending workers just as a barrier would do. Because the time

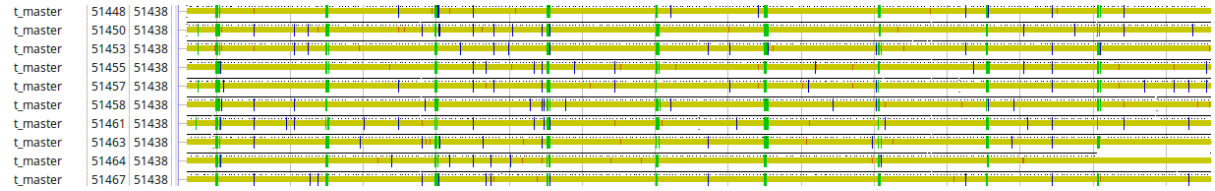

Figure 4.2: Trace Compass control flow view of a complete mmap benchmark without UMT (Only 10 threads of 59 are shown). The total trace length is 14.1s.

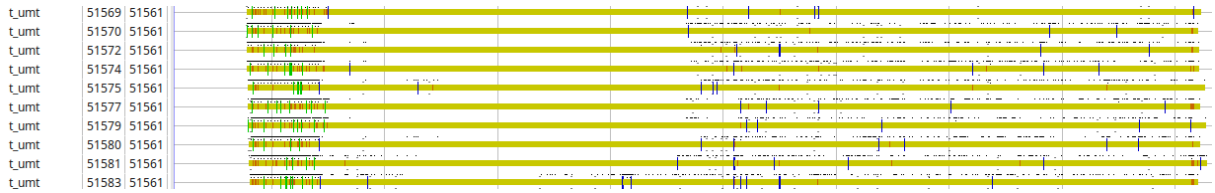

Figure 4.3: Trace Compass control flow view of a complete mmap benchmark with UMT (Only 10 threads of 507 are shown). The total trace length is 1.77s.

required by the disk to flush the changes is large, workers remain blocked most of the time and the CPU remains idle.

Figure 4.3 shows the trace of the same benchmark but compiled against Nanos6 with UMT support. A total of 507 threads are created in response to the blocking events, of which only ten are displayed. Whenever a blocking or unblocking event occurs, the Leader Thread is put in the ready queue of some CPU and wakes up or creates another worker thread on the idle CPU. Now more I/O operations are queued in parallel. However, the CPU usage still remains low.

The total execution time of the original and the UMT version are, respectively, 14.08 and 1.76 seconds (8x speedup). Although in both executions much of the CPU time is spent idling, the execution of the UMT version is faster because more I/O requests have been served in parallel. In the original version, only 56 workers could perform I/O operations at the same time because there are 56 logic CPU's and Nanos6 is not aware of when any of them blocks. Instead, the UMT version is able to schedule more workers and queue more I/O requests. However it is worth noting that the obtained speedup is highly dependent on the underlying hardware. In general, the slower the storage medium the higher is the speedup.

Because the amount of computation in the update part is small, the CPU usage is not improved much. The metrics scripts reports that the CPUs have spent a 95% of time idling on the master version and 88% on the UMT version. However, it means that threads do not suffer of oversubscription. The metrics scripts report a 0.002% of ready jam for the master version and 1.58% for the UMT version.

Follows another mmap example but this time, a more important workload is done on the update part for each task. The non UMT is shown at figure 4.4 and the UMT is shown at figure 4.5

Now, the CPU usage rate is improved from the non UMT version with a 77% of CPU idle time against a 37% on the UMT version. However, the oversubscription problem arises and limits the benefits of UMT. The metric extraction scripts report a 0.003% of ready jam and approximately 50 in-benchmark context switches for the non UMT version compared with 29% of ready jam and approximately 3000 in-benchmark context switches per CPU. Here, the speedup obtained is 3.35X and it gets worse as the amount of computation is increased.

Focusing again on I/O operations, figure 4.6 shows the speedup evolution of a modified version of

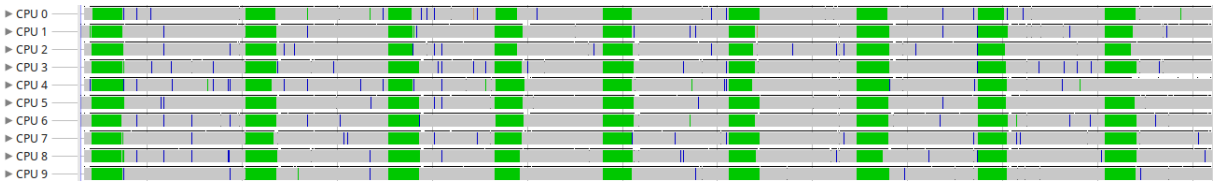

Figure 4.4: Trace Compass resource usage view of a complete mmap benchmark without UMT and with an incremented workload on the update part. Only 10 CPUs of 56 are shown for clarity. It can be appreciated the same pattern as in figure 4.2 with the difference that now CPUs spent less time idling. The total trace length is 12.4s.

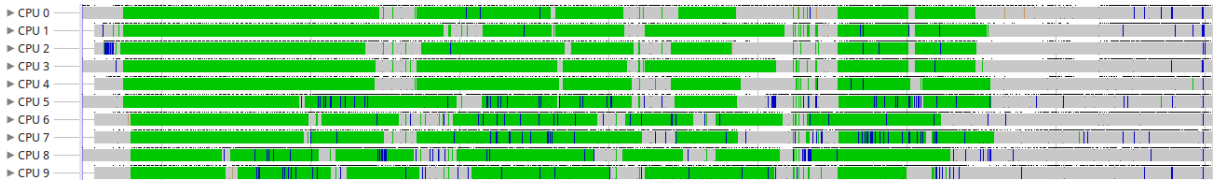

Figure 4.5: Trace Compass resource usage view of a complete mmap benchmark with UMT and with an incremented workload on the update part. Only 10 CPUs of 56 are shown for clarity. It is interesting to note that at the end of the execution CPUs are mostly idle while waiting for the storage device synchronization. After the synchronization is done, the application ends because all the computational work has already been done in parallel. The total trace length is 3.7s.

the mmap benchmark in which only writes are done (instead of read/update/write) against an increasing number of blocks. For each number of blocks tested, the figure shows the average speedup for three repetitions. It can be seen that for a size of 2000, a peak speedup of 10x is achieved. However, for bigger sizes the speedup drops.

In general, the more I/O petitions in flight, the higher is the probability of multiple petitions to coincide in the same page. When doing a sync operation, the threads stall until all data is written to disk. On mmap without UMT this implies that the maximum number of on-going sync operations is limited by the number of CPUs. On the UMT version there can be much more operations in-flight, which leads to more blocks overlapping and then, less writes that made into the storage device (several writes are merged into the page cache). For example, if the benchmark issues 10 writes to the page cache and 8 fall in the same page, then, only 3 pages need to be written at some point. This is why high speedup is achieved. In this case, the block size is 2048 (which means that 2048 bytes are written per block), the system page size is 4KiB and the number of pages of the 100MiB mapped file is $100 * 1024/4 = 20480$ pages. Subsequently, for each processed byte in a block, there is a $2048/20480 = 1/10$ chance of being written in the same page as another byte.

However, when the block size is big enough, it is likely that all pages of the 100MiB file are touched. Hence, increasing the size of the block from this point will not add extra work for the storage device but for the fast Linux page cache. This is combined with the fact that increasing the size of blocks without increasing the number of blocks does not adds more synchronization points. The synchronization points are like barriers for the non UMT version that delay its pace. As a consequence, if no more barriers are added and the workload of the writing stage is not bigger because the page cache is withstanding it, the difference between the two mmap versions is reduced and the speedup decreases.

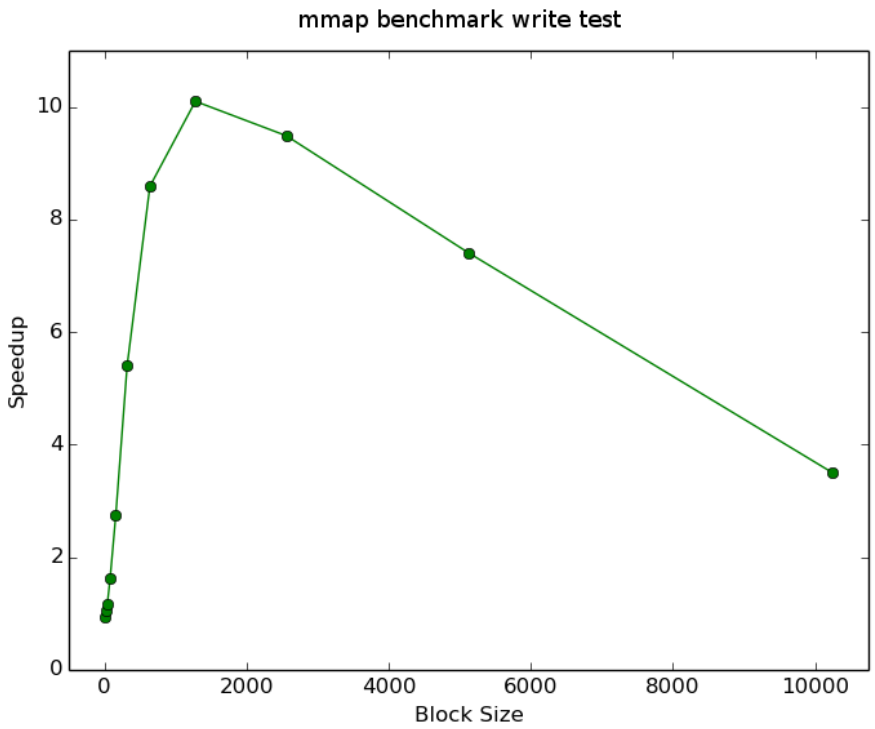

Figure 4.6: mmap with UMT vs mmap without UMT for an incremental number of blocks/tasks in which only write operations are performed.

4.3 Full Waveform Inversion Mock-up (FWI)

4.3.1 Introduction

The acoustic Full Waveform Inversion (FWI) [33] method aims to generate high resolution subsoil velocity models from acquired seismic data through an iterative process. The time-dependent seismic wave equation is solved forward and backward in time in order to estimate the velocity model. From the differences between acquired data and the computed velocity model, a gradient volume is calculated and used to update the velocity model on the next iteration.

The inverse problem is nonlinear and ill-conditioned. This makes it difficult solving the problem at high frequencies. Instead, the initial stimulus is decomposed into a spectrum of frequencies. Then, low frequencies are solved first on a coarse grid providing a good guess for higher frequencies.

Conceptually, FWI can be divided into three main steps. A pre-processing step estimates the computational resources needed to solve the problem according to the number of shots, wavelet frequency and domain dimensions. Then, the wave propagator solves the time domain formulation of the wave equation forward and backward in time. Finally, a post-processing step gathers the information from the computation of all different frequencies into a single final velocity model. The workflow is shown in Figure 4.7.

All three stages of the FWI require intensive I/O operations. While pre and post-processing steps perform sequential read and write operations on large shared velocity model files, the wave propagator mostly performs local I/O. From the computational point of view, the pre and post-processing stages do not represent a major issue on the performance of the FWI.

The wave propagator is used twice to propagate the wave forward and backwards in time. These two

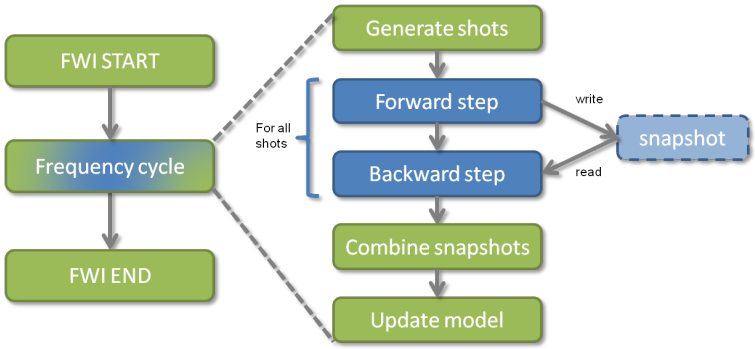

Figure 4.7: Full Waveform Inversion method (FWI) overview.

steps are commonly referred to as "Forward step" and "Backward step". An 8th order stencil is needed in order to ensure the stability of the numerical method. Given the length of the stencil, a good balance between high bandwidth memory access and computing power is needed in order to get good performance from the code. However, the wave propagator requires writing relatively large files at almost each time step. This results in inefficient computation of the propagator on most accelerators.

4.3.2 FWI Analysis

This section analyzes the implementation of the FWI application, details how it was ported to the OmpSs programming model and studies its expected behaviour when run under the UMT feature.

4.3.2.1 FWI OmpSs Porting

FWI works over a model for a sequence of time steps. As explained before, there are two phases: the forward propagation and the backward propagation. During the forward propagation phase, the model is updated at each time step and an snapshot is saved to disk every certain timesteps. Next, in the backward propagation phase all timesteps are processed again but in inverse order. During the backward propagation, both the velocity and stress models are updated in the same way as in the forward propagation phase, however, snapshots are read from disk instead of being written.

The FWI major data structures are two three-dimensional volumes of velocities and stresses. At each time step the velocity volume is first written/read to/from disk, then it is updated with the stress model, next the stress model is updated with the new velocity volume and finally, the source is inserted into the two middle slices of the stress volume.

The velocity and stress data structures are implemented as an array (in this case a three-dimensional matrix) of structs. Each element of the array is named cell and it contains 13 components for the velocity volume and 45 for the stress volume. By default, the components are implemented as 4 byte floats. Only 9 of the 13 velocity components of each cell is written to disk. Because the 13 components are stored sequentially in memory, the 9 components of each cell that is to be stored to disk, are first copied to an auxiliary buffer which is later sent to the OS.

Listing 4.1 shows the simplified timestep loop that implements both forward and backward propagations. `v` is the velocity volume and `s` is the stress volume. The `y0` and `yf` variables hold the indexes of the Y planes that are to be processed in each function call, in particular, y0 contains the first plane index and yf contains the last plane index to compute. The constant `stacki` variable is used to configure

```
1  for(int t=0; t < timesteps; t++) {
2      if ( t%stacki == 0 && direction == FORWARD)  write_snapshot(t, v);
3      if ( t%stacki == 0 && direction == BACKWARD) read_snapshot(ntbwd-t, v);
4
5      // Velocity volume update
6      velocity_propagator(v, s, dt, dy, dx, dy, y0, yf);
7
8      // Stress volume update
9      stress_propagator(v, s, dt, dy, dx, dz, y0, yf);
10
11     // Source insertion into stress volume
12     source_insertion(s, dt, wlv[t], dy, dx, dz);
13 }
```

Listing 4.1: Source code for the FWI timestep loop

every how many number of timesteps the velocity model is written or read to disk. As the name suggest, `write_snapshot` and `read_snapshot()` functions perform the velocity volume disk write/read operation as indicated by the *direction* of the propagation (*FORWARD* or *BACKWARD*). The rest of variables and parameters are of no interest for this section.

The propagate function has been taskified for parallelization at the volume slice level, as seen in listing 4.2. The volumes are now processed in terms of Y-planes (slice), the constant `BS` variable is used to determine the number of slices processed per each task. The slices execution flow is fixed by the OmpSs dependencies expressed in each task clause as can be seen in lines 10,18 and 23.

Assuming $BS = 1$, the `velocity_propagator()` function updates a single velocity volume slice of index `y0` given the actual velocity slice `y0` and 8 slices of stress surrounding `y0` as required by the algorithm. This is expressed as follows: the updated velocity slice is placed into the pragma `inout(v[y0])`, indicating that is both read and written. The index range for the 8 stress tasks is placed into the in clause `in(s[y0-4:yf+3-1])` (the last element in the range is included) indicating that it is only read (the difference between `y0` and `yf` is BS). Similarly, the same applies for the `stress_propagator()` function although here a a stress slice is updated given 8 input velocity slices. Next, a single task is created for the source insertion function `source_insertion()` at line 23. This tasks requires the two stress slices `y` and `y-1` which are updated to propagate the wave.

It is important to notice that I/O is still being done sequentially as in the classic parallelization approaches. This means that at the end of the timestep, it is necessary to wait for all tasks to complete before starting the next iteration. Otherwise, the I/O operations at the beginning of the next timestep might not have all the necessary data to proceed correctly. As a consequence, the task wait enforces a barrier that does not allow tasks of different iterations to be interleaved, effectively breaking the dataflow. This code is used later in the experimentation section to compare it with the fully parallelized FWI version.

The full code for the next FWI implementation iteration is shown at listing 4.3. The I/O function is now taskified and the taskwait at the end of each timestep removed. Instead of waiting, the execution flow of tasks is now fully determined by their dependencies. In other words, when the FWI execution starts, a single thread loops on the entire loop of timesteps and creates all defined tasks, without executing them. Once the loop is finished, it waits for all created tasks to be completed at the taskwait placed afterwards. Meanwhile, as tasks are created, Nanos workers wake up and execute them.

The I/O functions to read and write snapshots are now substituted by multiple calls to the `pwrite_snapshot()` and `pread_snapshot()`. Each call writes the specified number of slices

```
const integer BS = 1;
for(int t=0; t < timesteps; t++) {
    if ( t%stacki == 0 && direction == FORWARD)  write_snapshot(t, v);
    if ( t%stacki == 0 && direction == BACKWARD) read_snapshot(ntbwd-t, v);

    // Velocity volume update
    for(integer y0=HALO; y0 < (NYF - HALO); y0+=BS) {
        integer yf = min( y0 + BS, NYF - HALO);

        #pragma oss task inout(v[y0]) in(s[y0-4:yf+3-1]) label(vel_task)
        velocity_propagator(v, s, dt, dy, dx, dy, y0, yf);
    }

    // Stress volume update
    for(integer y0=HALO; y0 < (NYF - HALO); y0+=BS) {
        integer yf = min( y0 + BS, NYF - HALO);

        #pragma oss task inout(s[y0]) in(v[y0-4:yf+3-1]) label(stress_task)
        stress_propagator ( v, s, dt, dy, dx, dz, y0, yf);
    }

    // Source insertion into stress volume
    #pragma oss task inout(s[y-1:y]) label(source_task)
    source_insertion( s, dt, wlv[t], dy, dx, dz, y);

    #pragma oss taskwait
}
```

Listing 4.2: Source code for the partially parallelized FWI timestep loop

in the range specified between its third (`y0`) and forth argument (`y0+BS`). Hence, a write or read task only depends on the affected input velocity slices as specified with the clause `in(v[y0;BS])` (which defines a range starting at index `y0` of length `BS`).

Each task writes or reads a non-overlapping part of a file in parallel with other threads. To deal with parallelism at the OS level, the system calls `pwrite()` and `pread()` are used instead of the traditional `write()` and `read()` to explicitly state an starting offset position per syscall instead of having to set the global file position with `lseek()` (which would lead to data races in a parallel enviroment).

Figure 4.8 shows a timestep task decomposition for a forward propagation. The backward propagation task decomposition is analogous to the forward propagation and it is not shown. Notice that the figure shows the source insertion task, which is unique per timestep. It can be seen that stress tasks can be computed in parallel with write tasks. However, velocity tasks depend on the write tasks because the velocity model data must be written to disk before it is updated with the next timestep iteration.

4.3.2.2 FWI and UMT Analysis

During the forward propagation step, several timesteps are computed and saved to disk. As data is generated, it is not reused until it is read from disk in the backward propagation step.

With a frequency of 60Hz, 712 timesteps are computed (356 for write and 356 for read). The complete velocity volume of components that is to be written to disk per iteration is 90MiB large. There are a total of 160 slices of 576KiB or 144 pages each (assuming 4KiB/page). Hence, for each iteration 160 I/O requests to write or read 144 pages are issued to the OS. At the end of the execution, 32040MiB (32GiB) are written and read to/from disk.

```
const integer BS = 1;
for(int t=0; t < timesteps; t++) {
    if( t % 10 == 0 ) print_info("Computing %d-th timestep", t);

    if ( t%stacki == 0 && direction == FORWARD) {
        for(integer y0=0; y0 < NYF; y0+=BS) {
            #pragma oss task in(v[y0;BS]) label(write_task)
            pwrite_snapshot(t, v, y0, y0+BS);
        }
    }

    if ( t%stacki == 0 && direction == BACKWARD) {
        for(integer y0=0; y0 < NYF; y0+=BS) {
            #pragma oss task out(v[y0;BS]) label(read_task)
            pread_snapshot(t, v, y0, y0+BS);
        }
    }

    // Velocity volume update
    for(integer y0=HALO; y0 < (NYF - HALO); y0+=BS) {
        integer yf = min_int( y0 + BS, NYF - HALO);

        #pragma oss task inout(v[y0]) in(s[y0-4:yf+3-1]) label(vel_task)
        velocity_propagator(v, s, dt, dy, dx, dy, y0, yf);
    }

    // Stress volume update
    for(integer y0=HALO; y0 < (NYF - HALO); y0+=BS) {
        integer yf = min_int( y0 + BS, NYF - HALO);

        #pragma oss task inout(s[y0]) in(v[y0-4:yf+3-1]) label(stress_task)
        stress_propagator ( v, s, dt, dy, dx, dz, y0, yf);
    }

    // Source insertion into stress volume
    #pragma oss task inout(s[y-1:y]) label(source_task)
    source_insertion( s, dt, wlv[t], dy, dx, dz, y);
}
#pragma oss taskwait
```

Listing 4.3: Source code for the fully parallelized FWI timestep loop

When each slice is sent to the OS to be written, it is first cached in the Linux page cache. The OS defers the disk write to a later point in time as explained in section 2.2.3. If the problem input size is not big enough, the entire problem (all volumes of all iterations) might fit into the Linux page cache and writes might be done completely asynchronous i.e. they are non-blocking. If threads do not block, the UMT feature does not has a chance to act. When the page cache cannot hold all data, writes became blocking operations because the page cache must free some space (which might require to flush pages to disk) before writes can continue.

When the input problem is greater than 85Hz, the entire problem size is approximately 122GiB which almost reaches the node's 125GiB of main memory. However, when the entire problem size exceeds the main memory capacity, it quickly reaches the 160GiB SSD storage limit, meaning that the execution cannot continue. This is not the most common situation. Usually, clusters have much more disk capacity than the main memory and consequently, UMT has more chances to work.

This issue has been solved by virtualizing a common environment by limiting the amount of main

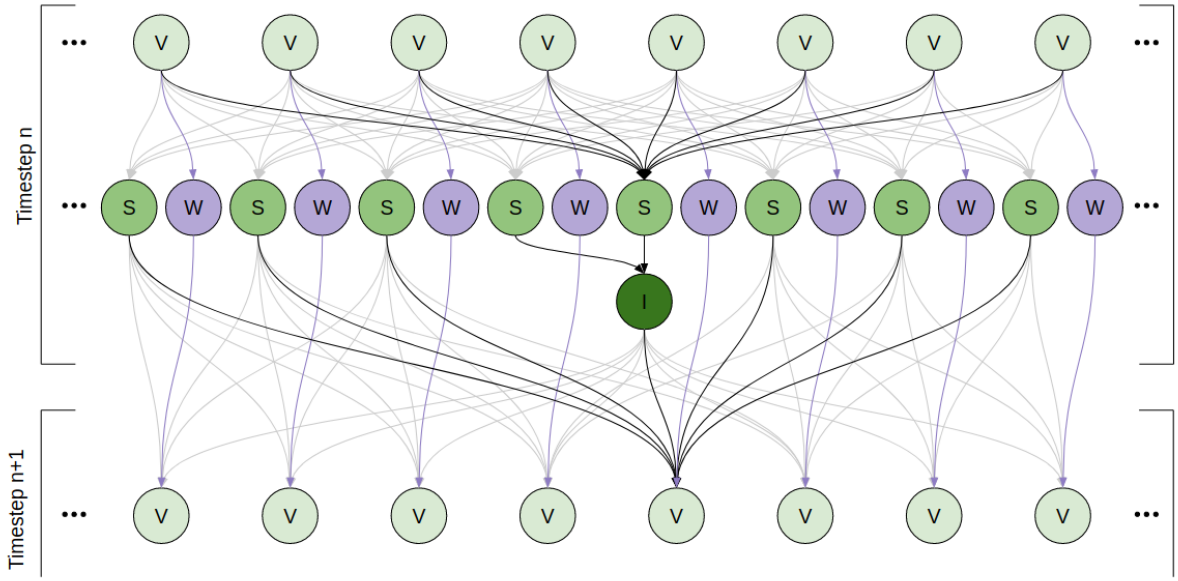

Figure 4.8: FWI task decomposition. This figure shows the complete stencil used in the Full-Waveform Inversion algorithm. Each circle is a task. `V` stands for `velocity_propagator` task, "S" for `stress_propagator` task, "I" for `source_insertion` task and `W` for write task. Dependencies are shown as arrows. The dependencies for a single stencil step are colored in black for clarity, grey dependencies belong to other stencil steps. Purple arrows describe I/O dependencies.

memory (including page cache memory) that an application can use. This has been achieved by using the *cgroupsv2*[1] Linux Kernel feature. All FWI instances have been run in a cgroup environment in which the main memory was limited to 2GiB. Because FWI requires approximately 850MiB of main memory to hold essential data structures (mostly the initial model, the complete velocity volume and the stress volume) the page cache is left approximately 1GiB to buffer I/O operations.

However, UMT might not perform well in case of memory demand pressure. When limiting the size of the page cache with cgroups, FWI fills the page cache fast. Hence, most of the time the memory is flushed because no free space left. This implies that flusher threads work at almost maximum capacity because they have a lot of dirty pages to write. UMT start operating when the kernel's */proc/sys/vm/dirty_ratio* is exceeded (when worker thread's writes start blocking). At this point, new threads are spawned (or woken up) with possibly the objective of writing new pages. Because at this point the memory threshold of free memory has been exceeded, these workers might get blocked as well until the flusher gang ends and before they can write anything. Even if the workers can write, they will just add another set of new pages to the page cache, not to disk. Moreover, because flusher threads are already working at almost full capacity, having more threads queuing work might not affect the result. Also, because this written pages will be the newest ones, they will be flushed to disk the last.

The only solutions that lasts is to explicitly force a flush of the page cache or not use a page cache at all. In the first case, it is possible to use an explicit sync after each slice is written. The worker doing the

[1]cgroups is a Linux Kernel feature that allows to limiting system resources to processes. This is mostly used but not limited to virtualization. The kernel provides a pseudo filesystem to configure groups of processes in a hierarchical basis whose accesses to system resources have been limited. cgroupsv2 is a reimplementation of cgroups which tries to fix several inconsistencies in the original model that limited the scalability of it. Currently the original cgroups version is far more extended than the second version, but eventually the second should became the new standard.

sync operation will block and UMT will wake another thread to continue issuing writes. Because the page cache should not yet be full, this new workers should not block and succeed writing new pages into the page cache. This leads to the multiple possible FWI adaptations defined in the next section.

4.3.2.3 FWI Versions

With the objective of exploiting the UMT feature as much as possible, four FWI adaptations are presented based on the idea of explicitly flushing the Linux Kernel page cache or not use it at all.

The first version considered is the **fsync/fdatasync** approach. The easiest option to explicitly flush the cache is to add a `fsync()` or `fdatasync()` (to do not flush modified metadata) system call just after writing a slice (thereby, one `fsync()` per write task). The problem of this approach is the `fsync()` granularity. It is not possible to flush just part of the file with this system call. When a `fsync()` is issued, all files' pages are synced. Moreover, if there are multiple ongoing `fsync()` operations on the same file, in-kernel locking contention might be a problem.

More precisely, when the `fsync()` system call is invoked, the kernel creates a list of the files' dirty pages that must be flushed and it does not returns until all of them are safe to disk. This list can contain pages of any slice because `fsync()` works file wide. Before flushing a page, it first checks whether the page has an I/O operation in flight. If so, the process blocks until the page is flushed. Then it flushes it again if it is still dirty. Also, manipulating a page requires to acquire a per-page lock. Hence, multiple `fsync()` calls on the same file might constantly stall each other.

Next is the **multifile** approach. Because `fsync()` only applies to a specific file, the easiest fix is to keep each slice in a separate file. Hence, after each slice is written, `fsync()` is called on the slice's file. This means that now there is no kernel lock contention and the writing process is smother. The main drawback is that the output of the application is modified, which might affect external applications that depend on this application output format. An auxiliary tool (that has been written to validate the implementation) would be needed to merge the individual files again. In the case of FWI, this affects how the timesteps are later visualized on a graphical application display. Another small drawback is that too many files might be created and the OS limit of open files might be exceeded. Using the `ulimit` command, the soft and/or hard limits can be increased. However, increasing the hard limit of open files requires root privileges and depends on the distribution used.

The third option is the **mmap** approach. To avoid changing the output format and creating too many files but still explicitly synchronize a single file partially, the `mmap()` and the `msync()` system calls can be used. With `mmap()`, all files to be written (all velocity volumes of all timesteps) can be memory mapped into main memory. With `msync()`, pages can be flushed individually. Memory mapping a file allocates a portion of the calling process address space to directly access the file as if it were loaded into main memory. However, the file is not read from disk yet, this is done when the file is accessed for the first time (unless the default behaviour is changed with `madvice()` syscall). Because mapping a file only consumes addresses, all files can be memory mapped. Memory mapped file pages are stored in the Linux page cache and are flushed to disk following the same policy as other pages in the page cache.

Using `msync()` only entire pages can be flushed to disk. Because `mmap()` only allows to memory map file chunks aligned to the page boundary and in a page size granularity, this implies that slices not multiple of the page size have to be flushed in bigger chunks than needed. In consequence, flushing a slice might also flush part of the preceding and/or following slice. Another option could be to store the slices

aligned to a page boundary in the file, adding padding when needed. However, this would again modify the output format which is what it is tried to be avoided. In any case, this is not really a concern with big problem sizes. For example, for 90Hz a slice is 310.64 pages large, meaning that every 310 pages two pages might be flushed twice. For 60Hz, slices are exactly 144 pages large and this issue is not present.

It is not possible to memory map a file that does not exists to create it. Hence, the snapshot files are first created with the `fallocate()` system call which creates files of the specified size without actually writing to disk. The created files are mapped to the zero page (a page that only contains zeros) until it is explicitly written.

And finally the last option studied is the **no cache** approach. In the case when the entire iteration space does not fit into main memory (which is the usual case), the Linux page cache is not really exploited. This is because data written is neither rewritten nor accessed before being flushed to disk. Hence, the page cache is not buffering useful I/O data but for the period between the last forward steps and the first backward steps (snapshots are read in the opposite order that are created).

The Linux Kernel page cache is skipped by opening all files with the O_DIRECT and O_SYNC flags. O_DIRECT is used to do not buffer I/O data and O_SYNC is used to perform synchronous I/O operations (writes become an always blocking operation and only return when data is stored into disk). For this approach, it is not needed any kind of explicit synchronization because all writes are by itself synchronous. However, the I/O requests need to be both memory and disk aligned to the disk block size, which is usually 512 bytes. If slices are not block aligned in the file, then is necessary to add padding, altering the output format. From the I/O point of view, it means that useless data is read and written, however, if the problem size is big enough, the padding is not relevant. For instance, for 90Hz, the slice size is 1272384 bytes large which means that $512 - 1272384 \bmod 512 = 448$ padding bytes are added per slice. For the total of 236 slices and 533 timesteps, $236 \cdot 533 \cdot 448 = 56353024$ bytes (53MiB) of useless data is added to the 149GiB of useful data.

4.3.2.4 FWI and the Block Layer

The default Linux kernel single queue block layer implementation might present problems on the tested platform. A single queue is shared among all CPUs of the system and the lock that protects the queue needs to be moved between the CPUs' caches, including moving between the two sockets of the node (more details are shown in section 2.2.3.2).

The test system, Cobi, has 56 CPUs split in two sockets connected by an Intel QPI interface. Because the FWI write and read operations are parallelized, this means that 56 threads will be issuing I/O requests to the block layer when running on an unmodified Nanos6 version and up to 160 (the number of slices for 60Hz) when running Nanos6 with UMT. In consequence, is worth considering that the single queue contention might be a problem when using the UMT feature. For this reason, FWI is tested in both the single and multi-queue Linux Kernel block layer.

4.3.3 Results

In this section all presented FWI versions and system configurations are analyzed. First it is shown the performance obtained when running FWI using the classic sequential IO approach. Then, the experiments are repeated with the parallel I/O versions for both Linux Kernel block layers. Next, the results are discussed and individual FWI versions are carefully analyzed to understand the observed behaviour.

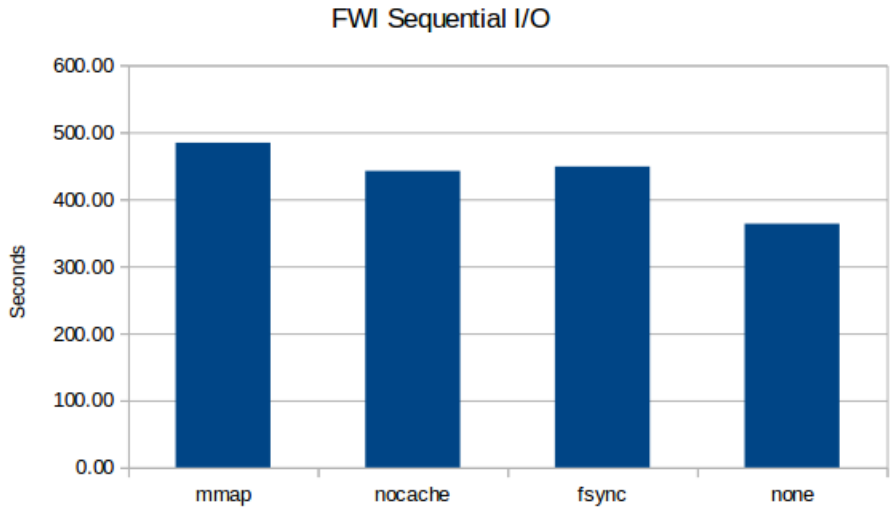

Figure 4.9: FWI sequential I/O results for all presented versions but multifile.

All experiments have been run on a single Cobi node, using the modified Linux Kernel with KPM, although not all tests enable the feature. All results are reported as the mean of 10 identical repetitions (unless otherwise stated). For all series of repetitions, the standard deviation has been calculated and manually inspected to ensure that no outlier has biased the results. Before each repetition, the data generated by the previous repetition is erased, the Linux page cache is cleared by issuing the command `echo 3 > /proc/sys/vm/drop_caches` and the SSD is trimmed to avoid consecutive executions suffer from the SSD garbage collector.

Figure 4.9 shows the results obtained after running the sequential I/O FWI versions as presented in listing 4.2 to evaluate the performance of the classic approach. The *multifile* version is not present because this solution only applies for parallel I/O (it uses one file per task). The *none* FWI version does not perform any kind of explicit synchronization, is simply writes and reads slices per task using pwrite/pread just as presented in the 4.3 code.

It can be seen that the *none* version is fastest. It could be thought that because no explicit synchronization is done, the complete time for writing the modified pages is not accounted in the total execution time of the application. Meaning that when the application exits, there are still pages to be written. This would be the general case, however this does not apply to FWI. During the forward propagation, pages are written to the page cache and are backed to disk when no more free memory is left in the page cache. At the end of the forward propagation, at most the maximum number of pages that the page cache can hold might not be written to disk yet (1GiB of pages). However, at the backward propagation step, all data has to be read again (32GiB), which means that the page replacement algorithm will eventually flush the remaining dirty pages to disk to allocate the more recent pages (see figure 4.10). Even if more main memory would be used, dirty pages would be saved to disk because of pages expiration time limit (*/proc/sys/vm/dirty_expire_centisecs*), which for the current system is 30s by default.

The reason for such outstanding timing is the excellent performance of the page cache to hide write and read latencies. All other FWI versions enforce an explicit synchronization of the data, which means that worker threads block until the data is saved. Because the runtime is not aware of when workers block, other threads cannot be scheduled on the idle CPUs and CPU time is lost until workers are waken up again.

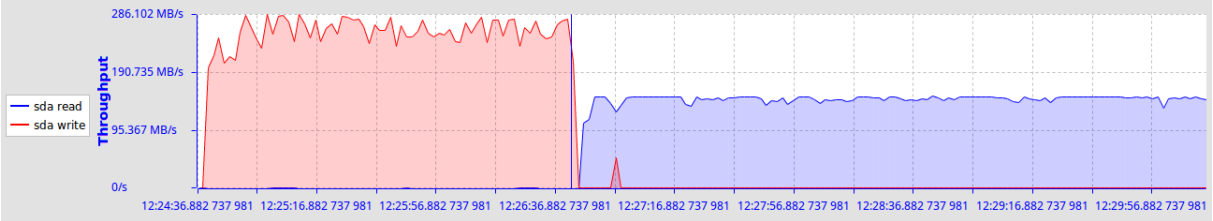

Figure 4.10: Complete Trace Compass view of a sequential I/O FWI execution without explicit data synchronization (*none* case). The vertical line separates the forward propagation (left) step from the backward propagation step (right). It can be seen that during the forward propagation data is written and in the backwards propagation step data is mostly read. However, a small peak of data being written is shown at the backward propagation step. This data is the remaining forward propagation dirty pages not written during the forward step that are now being replaced for the incoming new data of the backward phase.

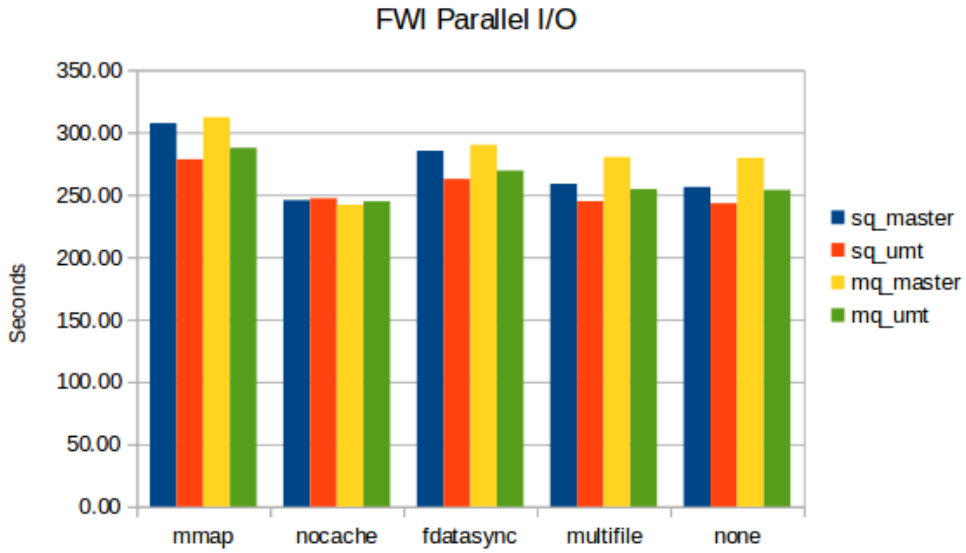

Figure 4.11: FWI Parallel I/O results for all FWI versions. Each FWI version has been run against an unmodified Nanos library (master) and a Nanos with UMT support (umt). Also, each of this versions has been run under the single queue (sq) and the multi queue block layer (mq).

Figure 4.11 shows the execution results for all Parallel FWI versions run under the single queue block layer (sq_ prefix) and the multi queue block layer (mq_ prefix). At first glance, it can be seen that any of the parallel I/O versions is faster than the sequential between a minimum of 1.16X and a maximum of 2X.

The figure shows that, in general, all tests run on the multi queue block layer are slower than their counterpart run on the single layer. From this it can be deduced that the block layer request queue is not a problem regardless of the 56 CPUs of the Cobi node and the number of I/O requests. The slower performance obtained on mq might be a consequence of I/O petitions not being reordered nor merged in the request queue because the absence of an I/O scheduler (not available in the Linux Kernel 4.10.5).

This theory is supported by the perf call graph of an FWI execution shown in figure 4.12. The lock that protects the request queue is defined inside the `struct request_queue` in the Linux kernel source file *include/linux/blkdev.h* as the member `spinlock_t queue_lock` and it is held every time the queue needs to be modified. The innermost kernel function to spin on a lock is named

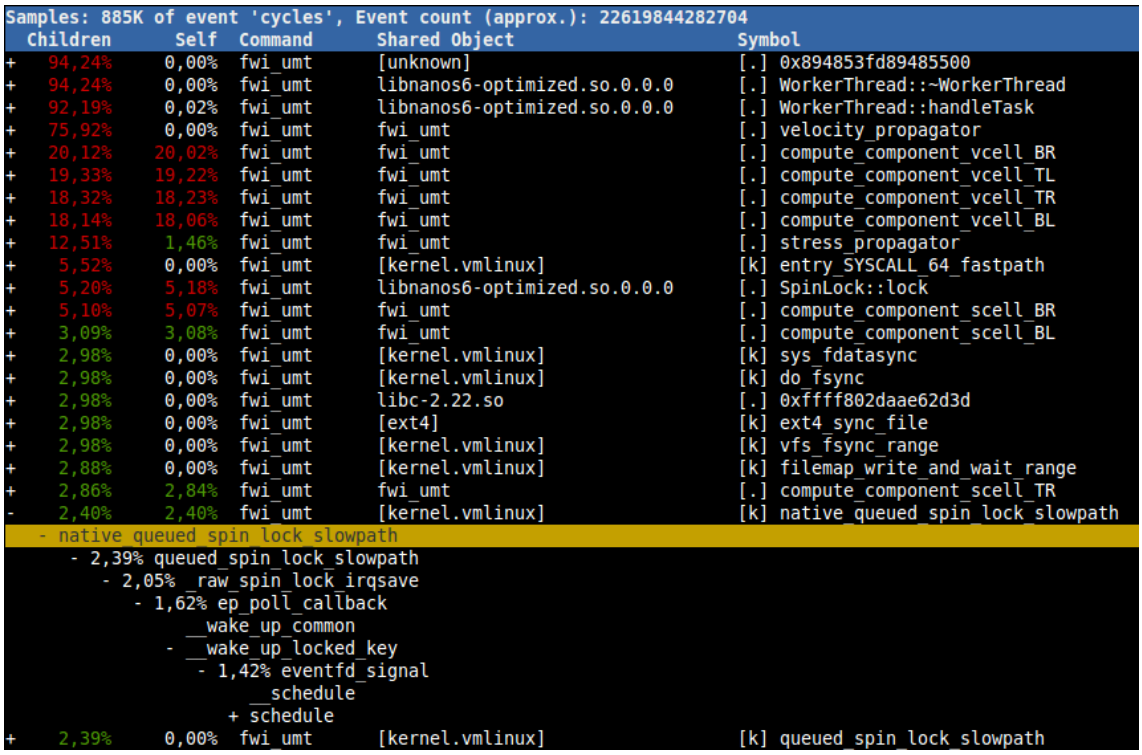

Figure 4.12: Perf call graph with a sampling frequency of 99Hz for an FWI nocache trace running on top of the single queue Linux Kernel block layer. The highlighted line shows the Linux Kernel innermost spinlock function. The entry is unfolded revealing the bottom to top call graph that leads to the function. It can be seen that there is no lock contention due to the kernel request queue lock.

`native_queued_spin_lock_slowpath`. On the perf trace it can be seen that most of the samples are collected on non-lock contention kernel functions. If there were lock contention, the spinlock functions protecting the request queue should have a big share of samples on the perf trace. However, it can be seen that `native_queue_spin_lock_slowpath` only has 2,40% of samples which a bottom to top call analysis shows that all of them came from the KPM eventfds. In conclusion, this FWI analysis now focus on the single queue block layer for the remaining of this section.

For the *mmap*, *fdatasync*, *multifile* and *none* FWI versions it can be seen that the UMT feature improves performance over the corresponding non UMT (master) version. The obtained speedup ranges between 10% for *mmap* and 6% for *multifile*.

However, the fastest FWI implementation is the *nocache* version, which performs quite stable regardless of KPM being enabled or the block layer used. The *multifile-umt* and *none-umt* versions also achieve similar results, however, it is no possible to conclude that the UMT feature presented in this thesis improves performance on the FWI application if an FWI version not using KPM such as *sq-nocache-master* achieves close results. Hereinafter, the goal of this section is to understand why performance is not obtained when using the UMT feature for the *nocache* version by comparing it with the versions that do perform better.

Figure 4.13 shows the Paraver view of an Extrae trace obtained from executing the *nocache* FWI master version to visually verify that the parallelization is done correctly. It can be seen that write, velocity and stress tasks are interlaced as they are run by multiple workers in parallel on the 56 CPUs. Nevertheless, it is worth pointing out that Extrae is not aware of OS preemptions.

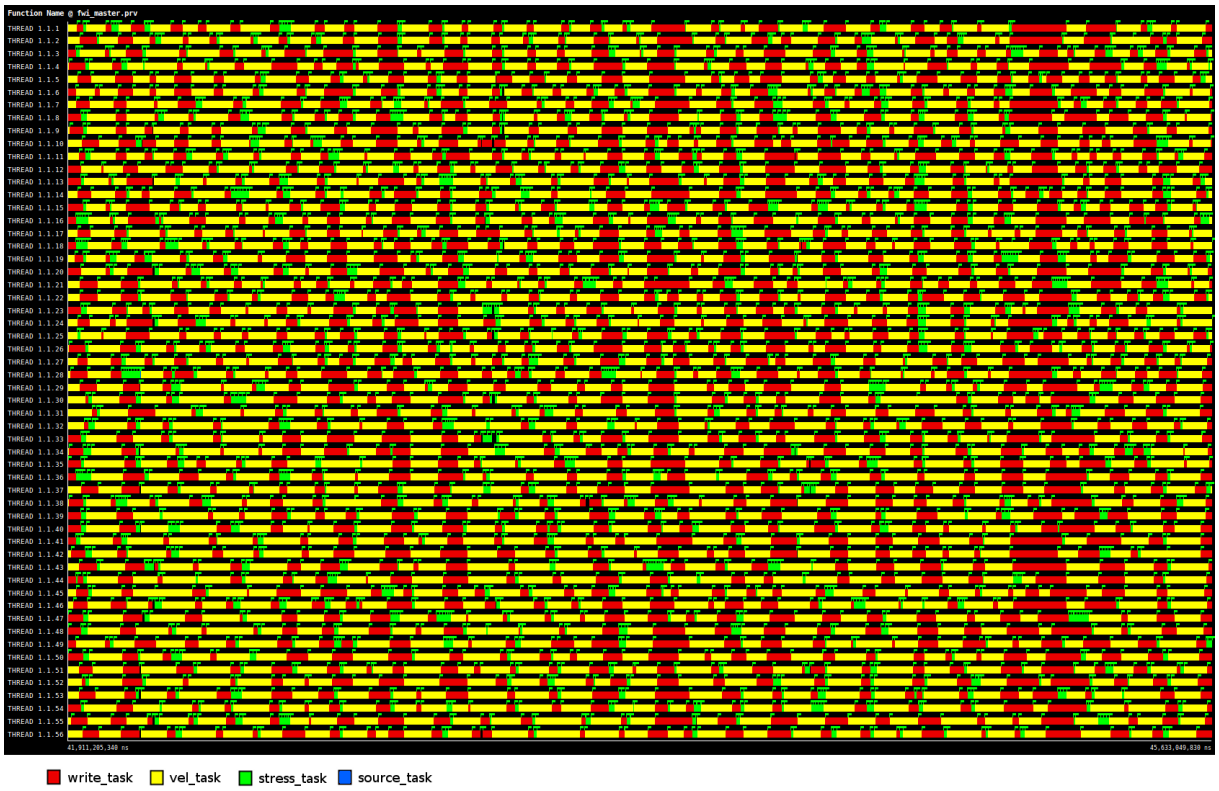

Figure 4.13: Paraver zoomed view of an FWI *nocache* Extrae trace. Each color segment displays the execution of a type of task as seen in the bottom legend. Four seconds of the forward propagation step are shown to illustrate the parallelization achieved after distributing tasks among workers. The rest of the trace follows a similar pattern and it is omitted.

Table 4.1 shows the custom metrics for all FWI executions. The *fdatasync-umt* version is not shown because the contention created by the parallel *fdatasync* system calls caused some events to be lost (too many events are generated). All UMT versions reduce the total time the CPUs remain idle because more work is done in parallel. However, the reduction of idle time is quite small compared to the synthetic mmap benchmark studied before. The Ready jam is also incremented in all of them, although the small task granularity keeps it small. As it is expected, the UMT versions create much more threads which leads to a higher number of context switches per CPU. The elevated number of context switches on the master versions is due to the migrations of workers performed by Nanos, not because of workers competing for the same CPU as the small ready jam and the Trace Compass count of system calls (not shown here) reveals. The most relevant information in this table is it that the *nocache* variants do not show a different behaviour as the other versions.

The UMT overhead might explain the lack of improvement in the nocache version. The perf call graph explained before in figure 4.12 also showed that most of the computational time is spent in the FWI inner functions `compute_component_vcell_` which performs the velocity update and the `compute_component_scell_` that performs the stress update. Table 4.2 shows the average of the perf samples grouped by their dynamic shared object (with `perf report -s dso`) of three nocache traces with sampling frequency 99, 999 and 10000 Hz. The table shows that the Nanos UMT overhead increase is notable compared to the Nanos without UMT version, approximately a 3% increment.

Not having enough parallelism could also explain why the improvements are limited. The tasks of the

FWI Version	Idle (%)	Ready Jam	Number of Threads	Context Switches/CPU
nocache-master	52.4	0.018	60	799
nocache-umt	43.4	2.3	2955	3507
mmap-master	57	0.007	60	1057
mmap-umt	45.9	6.2	3002	7553
fdatasync-master	56.8	0.0006	60	971.8
fdatasync-umt	-	-	-	-
multifile-master	54	0.008	60	419.9
multifile-umt	45.1	3.3	2952	5252
none-master	48.7	0.014	61	279
none-umt	35.9	2.4	2621	2350

Table 4.1: FWI versions metrics.

DSO Component	99Hz(%)	999Hz(%)	10000Hz(%)	Average
fwi (master)	89.47	89.49	89.08	89.35
fwi (umt)	88.03	87.46	88.42	87.97
nanos (master)	0.13	0.12	0.12	0.12
nanos (umt)	3.36	3.79	3.03	3.39
kernel (master)	10.05	10.05	10.39	10.16
kernel (umt)	8.34	8.47	8.25	8.35

Table 4.2: Perf samples distributions over fwi, nanos and kernel for three samples at three different sampling frequencies.

FWI version	Opportunity (%)	task (%)	thread (%)
nocache-umt	2.01	18.12	99.99
multifile-umt	1.80	15.91	100

Table 4.3: Leader Thread statistics for three repetitions of the nocache and multifile FWI versions.

FWI timesteps can be interleaved thanks to the task decomposition. However, the I/O buffer (purple arrow in figure 4.8) limits the parallelism between timesteps to avoid allocating extra memory for auxiliary I/O buffers. The UMT Nanos6 Leader thread has been modified to report usage statistics of tasks. Table 4.3 shows the average results reported after the repetitions of FWI nocache and multifile. The opportunity ratio is the percentage of CPUs that satisfied the conditions for the Leader thread to wake up a worker there. The task ratio shows, for all occasions in which the CPUs accepted a worker the the percentage of times that there was an available ready task. In the same way, the thread ratio shows, for all times that a ready task was available, the number of times that it was possible to create a worker (or get one form the pool of idle workers). Essentially, the statistics report that the actual number of available tasks is small in both versions.

It has been seen that UMT effectively decreases the amount of time CPUs are idle. Although some of this increase is due to the analyzed UMT overhead (3%), the rest of work being done on the CPUs of blocked workers increases the application performance on most scenarios up to a 10%. Because of the FWI I/O patter, the Linux Kernel cache is of no use and the best timings are archived when it is bypassed (this is just the opposite to the synthetic mmap benchmark in which an intensive use of the page cache is done). However, the same time marks are also achieved with the UMT version of *multifile* and *none*. Possibly, UMT is not the best application to be used with UMT but in any case, its study has been of great

interest to understand all the OS and user-space mechanism involved.

4.4 KPM Overhead

The Linux kernel KPM extension overhead when the extension is not being used has been tested. To do so, a small application has been developed which simply launches two threads per CPU that perform each of them 500M integer additions. Because all CPUs are stressed and there are more threads than cores, the kernel is being forced to schedule the threads and execute the scheduler wrapper. Because the `kpm_mode_enable()` function has not been called, the wrapper does nothing else than calling the genuine schedule function. The results of several executions in both the modified and the unmodified kernel show that there is not a clear difference at the nanosecond scale. This is because the overhead of executing two extra if branches and enable and disable kernel preemption (which requires to write an atomic variable) before and after the genuine schedule is negligible.

Chapter 5

Conclusions and Future Work

5.1 Future Work

Regarding the UMT user-space Nanos implementation:

- Because the CPU counters can be updated from multiple threads concurrently, there can be temporal inconsistencies. For instance, if worker A reads an EFD of CPU 0, but gets preempted before it can actually update the shared Nanos6 atomic CPU counter, another worker B could try to update again the same EFD and use its value to determine whether to become idle or not. As a consequence, worker B will take a decision based on incorrect data. This could be solved by protecting this critical region with a lock, but this option has been declined because the situation is unlikely to happen, and in the worst case either the period of oversubscription will last longer or the CPU will remain idle until the leader thread wakes up some worker. However, further solutions could be studied.
- It could be interesting to create a Leader thread per CPU EFD instead of having a single Leader thread that monitors all EFD; this would reduce cache pollution. However, this would require much more Leader threads context switches. For instance, if four CPUs have generated events, a single context switch on the first idle CPU of the model with a single Leader thread will serve all of them. Instead, in the model with multiple Leader threads, we would need four context switches that might preempt four busy workers on these CPU's. Hence, it is not clear whether this would improve performances or not.
- In any case, the problem of excessive context switches would be fixed by implementing a *leader-follower* approach [34]: when a Leader thread is notified and there is an idle CPU, it would create or designate another thread to become the Leader thread of the current CPU. The current Leader thread would become a worker thread and start executing tasks. The recently nominated Leader thread would then try to read that CPU EFD and repeat the cycle again [1].
- It is interesting to further study how Nanos6 behaves when removing its CPU idle list. Without the list, Nanos would have to completely rely on the CPU counters of ready workers to find idle CPUs, which might slow down the processes of adding new tasks. However, without the list, it will suffer less locking contention and the code will be simplified.

Regarding the KPM kernel implementation:

[1] This is quite similar to how SA proposes to respond to kernel events

- The proposed design notifies the user-space whenever a worker blocks or unblocks. However, Nanos6 workers only respond to the event when the counter reaches the zero value. Subsequently, it would be interesting to adapt the Linux kernel to notify user-space only when there are no idle worker. This would require to change how the EFD is used because it is no longer interesting to keep both the number of blocked and unblocked workers. Instead, the EFD could simply be used to notify when the CPU becomes idle by writing a 1 when the last monitored worker blocks. This would require to keep a counter of read monitored workers in-kernel. Also, an alternative mechanism is needed to invalidate a "CPU becomes idle" event when the CPU stops being idle before the runtime had time to read the event. A non-invasive option could be to read the EFD from kernel space to erase the counter.
- It might be interesting to continue studying the KSM approach, particularly, the version which minimizes context switches. The implementation of it was stopped partially because new ideas appeared for KAM and partially because of the technical difficulties to code it. However, moving the minimum parts of the runtime into the kernel just to satisfy the worker fairness and the selective wake up property still seems an option to avoid the spurious wake ups of the KAM and KPM.

Regarding the experimentation with real applications:

- Evaluate UMT with other real applications such as a web server or a database.
- Elaborate a more in-depth study on the *nocache* FWI version to demonstrate the reason behind the lack of improvement.

5.2 Conclusion

In this work, it has been presented the proof-of-concept of a new mechanism called User Monitored Threads (UMT) to monitor the blocked and unblocked state of the system threads based on a simplified pipeline called eventfd. The proposal requires to extend the Linux kernel scheduler with a small and simple set of changes that add a minimum overhead over the usual system operation. This extensions has been implemented in the Linux kernel 4.10.5, and the Nanos6 runtime developed by the Barcelona Supercomputing Center team has been adapted to make use of this kernel feature. The Nanos6 runtime uses the presented mechanism to keep track of the number of ready workers bound to each system's CPU. When a CPU becomes idle because all workers are blocked while performing, for example, I/O operations, the runtime schedules more workers on them. Multiple workers bound to the same CPU lead to an oversubscription problem, but the runtime minimizes the effect by stopping workers when it detects that there are multiple ready threads one the same core.

Both a synthetic mmap based benchmark and the FWI mock-up application have been used to test both the Linux Kernel and Nanos6 coupling. The LTTng, Trace Compass and Babeltrace python API are used to compare benchmark executions with both the modified and unmodified versions of Nanos6. Finally, it has been concluded that UMT has two main effects: on the one hand, it provides a mechanism to queue more I/O operations which approaches the real I/O rate to the one specified by the manufacturer of the storage device. On the other hand, blocked processes no longer obstruct the core and useful computations can be done while I/O petitions are being served.

Although the oversubscription problem limits the performance, a 10X speedup has been reported on a synthetic benchmark when enough I/O requests are done in parallel to the page cache and the storage device.

The sequential I/O FWI version has been greatly improved by parallelizing the I/O part with OmpSs tasks. Then, it has been seen that UMT improved performance in most of the scenarios presented. However, it has not been able to improve the fastest FWI version in which the Linux page cache is bypassed with the O_DIRECT feature. The different FWI versions have been evaluated and the overall UMT overhead and oversubscription jam analyzed. The study suggest that increasing the parallelism by decreasing the task granularity might give the Leader thread more chances to wake up more worker in idle CPUs to achieve still better resource usage. Because of the FWI I/O pattern in which the Linux cache is almost not used (unlike the mmap synthetic benchmark), this might not be the best application to be used with UMT.

As the next steps, it is planned to implement a leader-follower approach in the next iteration of Nanos6 to minimize the number of unnecessary context switches. Also, the KPM implementation will be adapted to notify user-space only when the counter of ready workers is zero instead of notifying each individual event. Then, other production software such as a web or database server will be adapted to test the UMT mechanism.

Bibliography

[1] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[2] Thomas E Anderson, Brian N Bershad, Edward D Lazowska, and Henry M Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10(1):53–79, 1992.

[3] Nathan J Williams. An implementation of scheduler activations on the netbsd operating system. In *USENIX Annual Technical Conference, FREENIX Track*, pages 99–108, 2002.

[4] Christopher A Small and Margo I Seltzer. Scheduler activations on bsd: Sharing thread management between kernel and application, 1995.

[5] Jason Evans and Julian Elischer. Kernel-scheduled entities for freebsd, 2000.

[6] Vincent Danjean, Raymond Namyst, and Robert D Russell. Linux kernel activations to support multithreading. In *In Proc. 18th IASTED International Conference on Applied Informatics (AI 2000*. Citeseer, 2000.

[7] Vincent Danjean, Raymond Namyst, and Robert D Russell. Integrating kernel activations in a multithreaded runtime system on top of linux. In *International Parallel and Distributed Processing Symposium*, pages 1160–1167. Springer, 2000.

[8] Vincent Danjean and Raymond Namyst. Controlling kernel scheduling from user space: An approach to enhancing applications' reactivity to i/o events. In *International Conference on High-Performance Computing*, pages 490–499. Springer, 2003.

[9] Bill Huey Ingo Molnar. Linux kernel mailing list (lkml) discussion, 2002. [Online; accessed 12-September-2017].

[10] Mindaugas Rasiukevicius. Thread scheduling and related interfaces in netbsd 5.0, 2009.

[11] Microsoft. User-mode scheduling, 2017. [Online; accessed 12-September-2017].

[12] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W Wisniewski, and Jimi Xenidis. Scheduling in k42. *White Paper, Aug*, 2002.

[13] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[14] Alan R Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

[15] Alejandro Duran, Josep Perez, Eduard Ayguadé, Rosa Badia, and Jesus Labarta. Extending the openmp tasking model to allow dependent tasks. *OpenMP in a New Era of Parallelism*, pages 111–122, 2008.

[16] Eduard Ayguade, Rosa M Badia, Daniel Cabrera, Alejandro Duran, Marc Gonzalez, Francisco D Igual, Daniel Jiménez-González, Jesus Labarta, Xavier Martorell, Rafael Mayo, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. *IWOMP*, 9:154–167, 2009.

[17] Wikipedia. Linux range of use — wikipedia, the free encyclopedia, 2016. [Online; accessed 12-September-2017].

[18] Linux Kernel Community. How the (linux kernel) development process works, 2017. [Online; accessed 29-September-2017].

[19] Carlos Boneti, Roberto Gioiosa, Francisco J Cazorla, and Mateo Valero. A dynamic scheduler for balancing hpc applications. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 41. IEEE Press, 2008.

[20] Linux Community. Real-time linux wiki, 2016. [Online; accessed 13-October-2017].

[21] John T. Bell. Cpu scheduling - operating systems course notes, 2013. [Online; accessed 8-October-2017].

[22] Jonathan Corbet. The cpuidle subsystem, 2010. [Online; accessed 12-September-2017].

[23] Jeff Moyer. Ensuring data reaches disk, 2011. [Online; accessed 19-September-2017].

[24] Thomas Krenn. Linux storage stack diagram, 2017. [Online; accessed 4-October-2017].

[25] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, page 22. ACM, 2013.

[26] Jonathan Corbet. The multiqueue block layer, 2013. [Online; accessed 19-September-2017].

[27] Jonathan Corbet. Two new block i/o schedulers for 4.12, 2017. [Online; accessed 29-September-2017].

[28] Kevin Kaichuan He. Why and how to use netlink socket, 2005. [Online; accessed 21-September-2017].

[29] Linux Kernel Community. Adding a new system call. [Online; accessed 21-September-2017].

[30] Steven Rostedt. Using the trace_event() macro (part 1), 2010. [Online; accessed 29-September-2017].

[31] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[32] Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.

[33] J. Morillo A. Zitz. Final report on application experience in deliverable 6.3, deep-er project, 2017.

[34] Douglas C Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, et al. Leader/followers-a design pattern for efficient multi-threaded event demultiplexing and dispatching. In *University of Washington. http://www. cs. wustl. edu/˜ schmidt/PDF/lf. pdf*. Citeseer, 2000.