Imperial College London

Individual Project Report

Imperial College London

Department of Computing

Tense - a CFS extension for virtual-time execution in Linux

Author:
Tencho Tenev

Supervisors:
Tony Field
David Wei

June 18, 2018

Abstract

The execution of programs in virtual time is a powerful method for supporting a continuous performance engineering development lifecycle. Firstly, it allows for the integration of timing models that run alongside a program, replacing parts of the application which have been specified but not yet implemented early in development. Secondly, it enables virtual time scaling in a multi-threaded application and between multiple processes such as a web server and a database. This is useful during the later stages of development when 'what if' experiments can help identify potential optimisation opportunities that might significantly impact the execution of the program or workload of interest. We present an extension for the Linux scheduler which provides an OS mechanism for virtual-time execution. It attempts to model the actual scheduler (CFS) by executing it with a modified notion of virtual time. The mechanism in particular supports moving tasks in time, dynamic per-task time-scaling, and accurate sleep duration in the presence of time-scaling of other tasks. We evaluate the accuracy and overhead of the solution in a series of experiments that were also used to drive the development of the framework. The measured time scaling overhead is in the order of a microsecond and we are able to accurately predict a 10% speed-up of a workload from the PARSEC benchmark under a speculative optimisation of a single function.

Contents

1 Introduction **4**
- 1.1 Objective . . . 4
- 1.2 Solution . . . 5
- 1.3 Contributions . . . 5

2 Background **6**
- 2.1 Software performance engineering . . . 6
  - 2.1.1 Software profiling . . . 6
  - 2.1.2 Performance models . . . 8
  - 2.1.3 Symbolic execution and static analysis . . . 11
- 2.2 Virtual-time execution . . . 12
  - 2.2.1 VEX . . . 12
  - 2.2.2 Progress of threads in virtual time . . . 13
  - 2.2.3 VEX in the Linux kernel . . . 14
  - 2.2.4 Network emulation . . . 16
  - 2.2.5 Causal profiling . . . 16
- 2.3 Linux . . . 16
  - 2.3.1 Linux kernel 101 . . . 17
  - 2.3.2 Process priorities . . . 17
  - 2.3.3 Clocks and timers . . . 18
  - 2.3.4 Fair process scheduling . . . 19
  - 2.3.5 Fair group scheduling . . . 20
  - 2.3.6 CFS and Multi-processing . . . 20

3 Tense **21**
- 3.1 Approach to virtual-time execution . . . 21
- 3.2 A note on alternative ideas . . . 22
  - 3.2.1 Tense as a scheduler class . . . 22
  - 3.2.2 Shared memory as a communication mechanism with the kernel 22
  - 3.2.3 Tense as a modification to the JVM . . . 22
- 3.3 Algorithm . . . 23
  - 3.3.1 Dynamic time-scaling . . . 23
  - 3.3.2 Jumping in virtual time . . . 25
  - 3.3.3 Sleeping in virtual time . . . 25
  - 3.3.4 Synchronisation of multiple timelines . . . 26
- 3.4 User library and utilities . . . 28

4 Evaluation **30**
- 4.1 Experimental setup . . . 30
- 4.2 Dynamic time-scaling . . . 30

4.2.1 Using process priorities 30
4.2.2 Scaling `vruntime` 31
4.2.3 Illustrating the difference between the two approaches 31
4.3 Accurate sleep duration in the presence of time-scaling 34
4.4 Dynamic time-scaling in a large application 37

5 Conclusion **38**
5.1 Summary 38
5.2 Future work 38
5.2.1 Avoid the need to run a custom kernel 38
5.2.2 Improve user-space features 38
5.2.3 Continue the development and evaluation of the SMP algorithm 39

Bibliography **40**

1 Introduction

The performance of every software system starts as a non-functional requirement when it is first specified. Meeting this requirement is often crucial, just like meeting security requirements. If the function of a system is to service an order, but it takes 10 minutes to do so or it leaks the credentials of the user, no one would use this system, even though it meets its functional requirements. The parallels between performance and security don't end here - in practice both suffer from the *fix-it-later* phenomenon and there is substantial research and growing adoption of model-based techniques that integrate these processes in the software development lifecycle. A survey [1] on the state of performance engineering in 2016 found that 70% of respondents agree the importance of performance engineering is growing due to the rise of complex composite applications. In the same survey, performance testing and tuning and the design of post-deployment performance testing and monitoring infrastructure were identified as the main responsibilities of a performance engineer. In comparison, performance modelling, influence on non-functional requirements in early stages of development, and developing design guidelines that relate to performance were selected by just over a half of the respondents. We believe providing and promoting tools for early-stage performance modelling can help drive these numbers up and result in many more systems meeting their performance requirements. This is important for businesses because problems caused by performance issues with large systems often lead to lost revenue either through service outages or poor UX that drives customers away.

1.1 Objective

The objective of this project is to build a virtual-time execution framework. Such a framework will enable us to explore the performance characteristics of software during its development as part of the software performance engineering process. Specifically, real code can be executed alongside performance models that predict the response times of software components (which may not yet exist) along a unified virtual timeline. In addition, the perceived time of processes or threads can be changed using a time dilation factor to create the illusion of functions running relatively faster or slower. This will allow hypothetical "what if" experiments to be undertaken to find potential bottlenecks and to predict the effects of optimising specific parts of the code.

Virtual time execution has previously been explored for the JVM in VEX ([2], [3]) and even though our project is a spiritual successor of that work we highlight some important differences in our approach aimed at dealing with some of VEX's limitations. Regardless of the approach, software performance depends on all layers of a system - from CPU architecture, through OS kernel and VM runtime, to user libraries and applications, as well as the components at each layer - OS scheduler, VM garbage collector, threading synchronisation primitives, to name a few. It is therefore easy to get lost down the rabbit hole without setting some assumptions, restrictions

and guiding principles. There isn't a strictly-defined class of software systems under consideration, but we target CPU-bound processes running on a single-core or SMP architecture, while keeping in mind how any decisions could be affected by I/O and other system resources.

1.2 Solution

The presented solution combines ideas from related work (VEX [3] and Timekeeper [4]). The objective is to support fine-grained per-thread time-scaling and synchronisation in virtual time while having precise control over time and kernel-level events. The former is more easily conceived in user-space either as an instrumentation framework or modification of a runtime environment. However, there are risks associated to the extensibility of such an approach beyond the scope of this project such as integration into the I/O subsystem and robust virtual time execution on SMP. This is the reason we choose a 'best of both worlds' path by building the mechanisms for virtual time execution in a loadable kernel module (and directly into the Linux kernel where necessary). The kernel module represents a virtual time device that is managed by standard operations on a Linux file (open, read, write, seek, etc.). This interface is sufficient to use the solution, but we present a C library and additional utilities that greatly enhance the user experience.

1.3 Contributions

- Modifications to the Linux kernel, a loadable kernel module, and a user-space C library combine ideas from related work into a framework that can take full control of the experimental environment both in user- and kernel-space. This is sufficient to address concerns about the extensibility of our work with features that require arbitrary hacks to various subsystems of the kernel.
- Evaluation of two different approaches to time-scaling in the Linux scheduler - overriding `nice` values vs. scaling `vruntime` of scheduling entities, and a review of other solutions (VEX [2], Coz [5], Timekeeper [4]).
- Evaluation of the accuracy and overhead of the presented solution with controlled experiments as well as a popular benchmark.

2 | Background

2.1 Software performance engineering

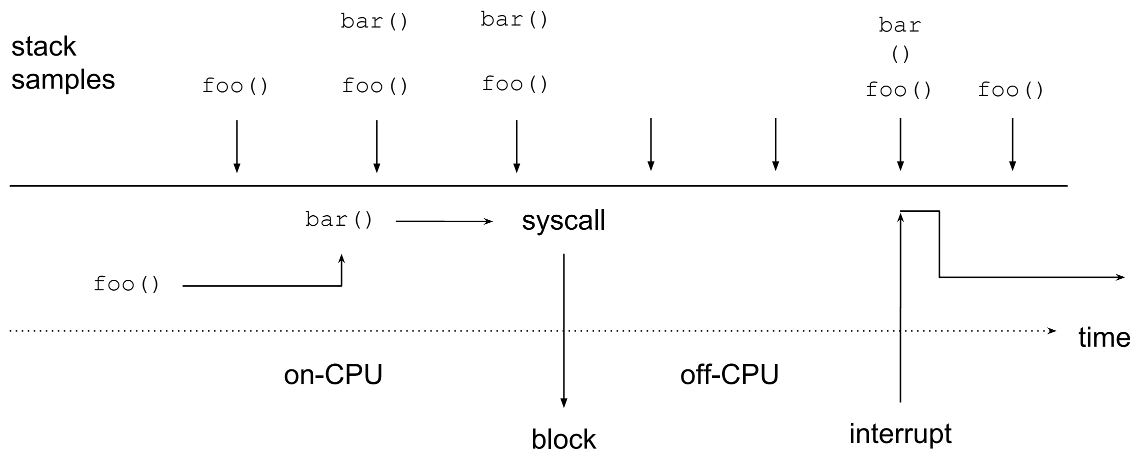
Software Performance Engineering (SPE) has an interesting history [6] as a field of computer science which has evolved over the years to meet the demands of new computing paradigms and development methodologies. In its foundation are quantitative techniques to predict the performance of software *early in design* and effectively address performance risks to meet non-functional requirements. Academic reviews [7] [8] of the field over the years identify the adoption of the *fix-it-later* mentality as a major cause of performance failures. This approach prevails when it becomes more expensive to model and evaluate software in design and development than to fix and tune it later. In the context of SPE, this project is an instance of the efforts to develop more efficient modelling and measurement approaches that outweigh the benefits of *fix-it-later*. There is substantial interest in the design of adaptive self-managed systems that are able to dynamically evaluate and adjust their performance. The ability to execute in virtual time and measure the effects of possible optimisations at a macroscopic level is one tool to implement adaptation strategies.

The performance of software is typically a matter of timeliness (responsiveness), efficiency, and scalability. Common measures of timeliness are response time and throughput. Efficiency can be measured by considering resource utilisation. Scalability can be determined by observing timeliness and efficiency as the service demand and number of resources increases. Common ways to assess the performance of software are the use of performance models, software profiling, and symbolic execution. Each of these techniques has its advantages and disadvantages and they can be often applied in a combination. I give a brief summary of each with an example application.

2.1.1 Software profiling

In practice software performance engineers use high-precision timers and sophisticated tools to analyse running systems. Profiling is a form of dynamic program analysis which relies on collecting information about a program as it executes. There are different types of profiling based on what the collected information is and what method is used to gather it. The job of a *sampling* profiler is to periodically look at which code is being executed (by examining the instruction pointer for example, Figure 2.1). As an alternative, a *tracing* (or *instrumenting*) profiler modifies the application code or the runtime by adding snippets of code. The main difference[1] is that tracing code is part of the normal execution of the profiled program while sampling acts as an independent observer. A popular tool is the `perf` Linux profiler [9] which is used to sample and trace events. The core mechanisms for tracing in Linux is supported by ftrace [10].

[1]See `https://danluu.com/perf-tracing` for a detailed discussion with great examples




```
# Sample CPU stack traces for the specified PID, at 99 Hertz, for 10 seconds:
> perf record -F 99 -p PID -g -- sleep 10
```

Figure 2.1: CPU profiling by sampling the stack - 60% of CPU time spent in `bar`, 100% in `foo`, and $\frac{2}{7}$ of total time off-CPU with example `perf` command.

There is a trade-off to how frequently a sampling profiler can sample and still be representative of reality, because high sampling frequency comes with increased overhead and interference with the observed code. Even worse, sampling only sees what is happening - sometimes it is more important what isn't, such as waiting on a software lock. Finally, sometimes (in Google's data centres) the tail of a distribution is what is actually important [11] and sampling only captures the average. This last point is a bit harder to understand so consider the following example. A query is answered by issuing thousands of remote procedure calls (RPCs) to various machines (think map-reduce). In order to prepare a response, all RPCs need to return so it is the slowest one that determines the response latency. However, each machine involved is simultaneously handling thousands of RPCs for various queries and a sampling profiler would report an average which is not indicative of the true performance for a query. With all these disadvantages in mind, sampling profilers still play a significant, if not dominant, role in performance engineering. The reason for this is their simplicity and the fact that the majority of performance problems can be analysed by looking at the approximate time spent on a line of code. Brendan Gregg, 'all-round performance guru', even recommends CPU profiling ('not technically "tracing" of events, as it is sampling') as the first thing to learn on his page about Linux tracing[2] before going into more depth.

It is important to understand how time is used in performance engineering. By doing this one can reason about the choice of abstractions, in what cases the simulation is accurate and thus useful, and how it might fail to represent the modelled system. A program like `perf` can hook into thousands of events (see Figure 2.2 for a break-down of the main classes) and this is just one of many tools. In the context of virtual-time execution we are mostly interested in events generated by the scheduler such as context switches, wake-ups, migrations to another run-queue, blocking on I/O, etc. It is thus beyond the scope of this report to give a complete overview of what is commonly used by performance engineers. However, it is easy to come up with experiments that validate our expectations about virtual time execution. For example, if a piece of code

[2] http://www.brendangregg.com/blog/2015-07-08/choosing-a-linux-tracer.html - Choosing a Linux tracer

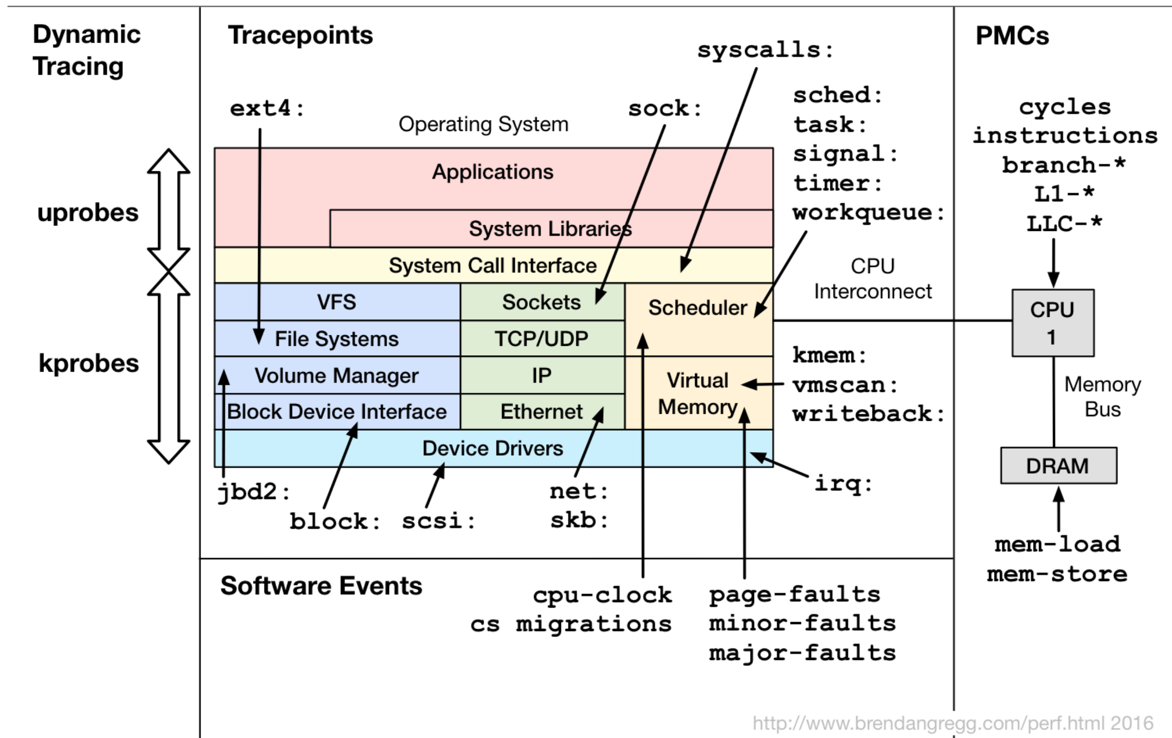


Figure 2.2: `perf` instruments various events - CPU performance monitoring counters, kernel counters (e.g. page faults), static trace-points at interesting places in the kernel, dynamic tracing to generate an event anywhere (figure by Brendan Gregg [12]).

is made to run twice as fast in virtual time, a sampling profiler should see this code two times less frequently than during a normal execution. In the same experiment, a trace should show an instrumented method called by the accelerated code was called the same number of times as normal unless the calls are driven by the progress of time. However, it is hard to have any reasonable expectation about the number of page faults or cache misses, or time spent holding a lock, especially in realistic scenarios. Very specific knowledge about the whole stack (application, libraries, runtime, OS) is required when it comes to interpreting a measurement and looking for performance bugs or optimisations.

Finally, profilers record vast amounts of data, mostly stack traces. It is challenging to do any kind of analysis by merely reading a stack trace. This is why visualisation plays a role in software profiling, too. A popular one, invented by Brendan Gregg, is a flame graph (Figure 2.3). The x-axis is an alphabetically sorted stack to make sure as many samples as possible are merged to form 'towers'. The y-axis simply shows the stack depth. Different colours are used to highlight kernel, JVM, and Java user code. The edge of the 'flame' shows which function is using the CPU, the context of a function call can be traced down the y-axis, and more samples lead to a wider tower.

2.1.2 Performance models

The aim of a performance model is to predict performance aspects of a modelled system. It consists of an abstract representation of the system's hardware and software and a workload model which describes the types of tasks handled by the system, the service demand for a task, and how system resources interact to complete it.

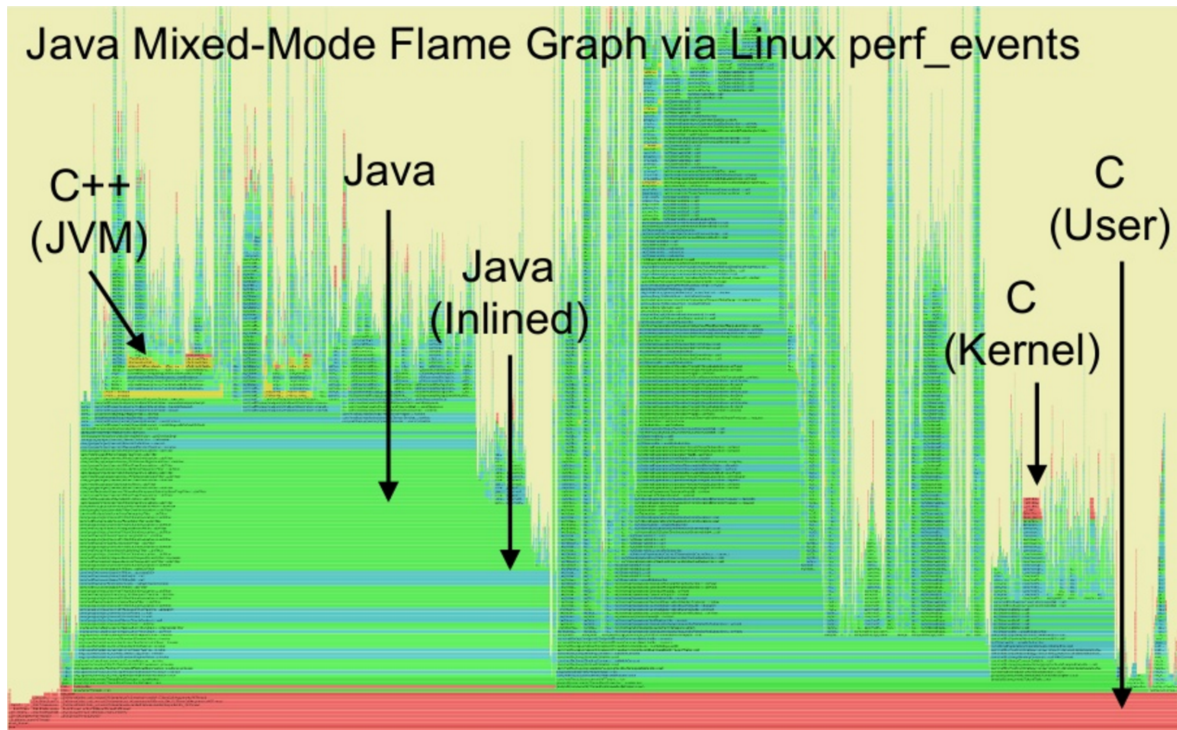


Figure 2.3: Example flame graph by Brendan Gregg [13]

For example, a performance model of a cache for a web application could contain abstractions for clients requesting various resources at different times and the state of the cache itself. The workload could be modelled by specifying the popularity of each resource - viral videos could be requested 100 times per second, while an old news article - 2 times a day. With this specification in place there are a range of techniques to build the actual performance model. These techniques can be broadly grouped in two categories - analytical models and simulation models.

Simulation models

The aim of this project is conceptually to build a simulation model - it tries to imitate the performance of a software system. The modelled software system consists of parts of the system itself, abstractions to imitate the slower or faster execution of certain parts of the system, and abstractions to execute other performance models and use their predictions to imitate interaction with non-existent parts of the system.

A simulation is a computer program that imitates how the state of the modelled system evolves as it is exposed to the workload model. Simulations can be stochastic (often using probability theory to introduce the element of randomness) or deterministic. A discrete event simulation (DES) processes a time-ordered stream of events that mutate the state of the system and cause new events to occur. The running example with the web cache can be modelled using a DES. As an example, Figure 2.4[3]shows the hit ratio from three runs of a simulated cache with a First-In-First-Out cache policy and cache size of 10. There are 100 resources with popularity following the harmonic series.

In contrast to DES, continuous dynamic simulations solve a set of equations (often with a changing solution over time) to model processes such as fluid dynamics in a flight

[3]The cache example is based on work I did in collaboration with Andreas Asprou in the Simulation and Modelling course

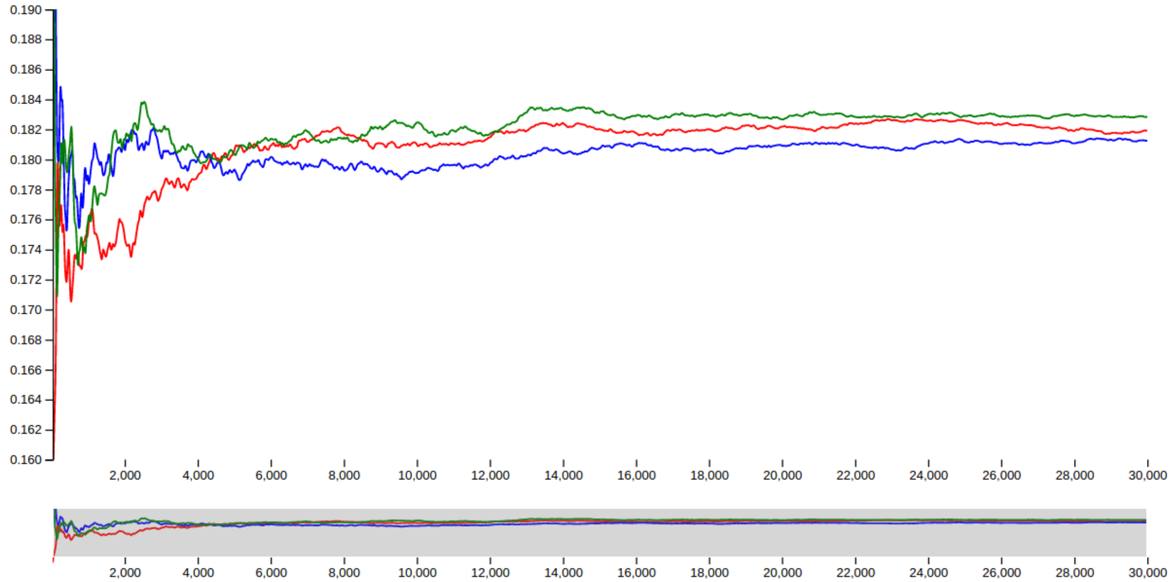


Figure 2.4: Hit ratio over time of a simulated cache (3 runs, FIFO, cache size = 10)

simulator or the fundamental laws of physics in a simulation of the universe. These models often combine aspects of simulation and analysis [14]. They are less common in performance engineering, but can be applied to dynamically adapting systems [15].

An important property of all simulation models is that the accuracy can be increased arbitrarily - a running system is a perfect simulation of itself. Usually the prediction of a (stochastic) simulation model comes with a confidence interval (CI) [16] which is a type of interval estimate based on observed data. In our example with the cache the observed data is $n = 10$ measurements for each row of Table 2.1. The confidence level associated with a CI is usually interpreted as the probability of finding the true value of the estimated parameter inside the produced range (90% in our example). The most common case is when the underlying distribution is normal and the estimated parameter is its mean. Also in most cases, the standard deviation is unknown, which makes the sample mean follow a *t-distribution*. The CI in such setting is given by the following formula:

$$\bar{x} \pm z \frac{s}{\sqrt{n}}$$

where $\bar{x}$ is the sample mean, s is the bias-corrected sample standard deviation, n is the number of observations, and z is the critical value of the *t-distribution* with $n - 1$ degrees of freedom at the chosen confidence level. Critical values are taken from statistical tables [17] that contain approximations for common confidence levels.

Analytical models

An analytical model uses mathematical theories to model a system. The main challenge is to represent a real-world system architecture at the high-level of abstraction of a formalism. However, such models can be analysed with more precision and when understood well require less effort to build than a simulation. For example, queueing theory can be used instead of a DES to model the response times in our cache example. Once we've estimated the probability of a cache hit, this can be used to construct a network to model the interaction with a database to retrieve the requested resource (Figure 2.5). In a practical scenario this could be used to try out hypothetical cache

Cache (policy and size)	Point estimate of hit ratio	Confidence Interval (90%) 10 measurements in steady-state
FIFO 10	0.18238	(0.18212, 0.18263)
RAND 10	0.18363	(0.18306, 0.18421)
FIFO 50	0.40625	(0.40601, 0.40650)
RAND 50	0.40730	(0.40698, 0.40763)
FIFO 100	0.51854	(0.51804, 0.51904)
RAND 100	0.52227	(0.52187, 0.52267)

Table 2.1: Confidence intervals for hit ratio with different cache policy and size based on 10 runs of a DES

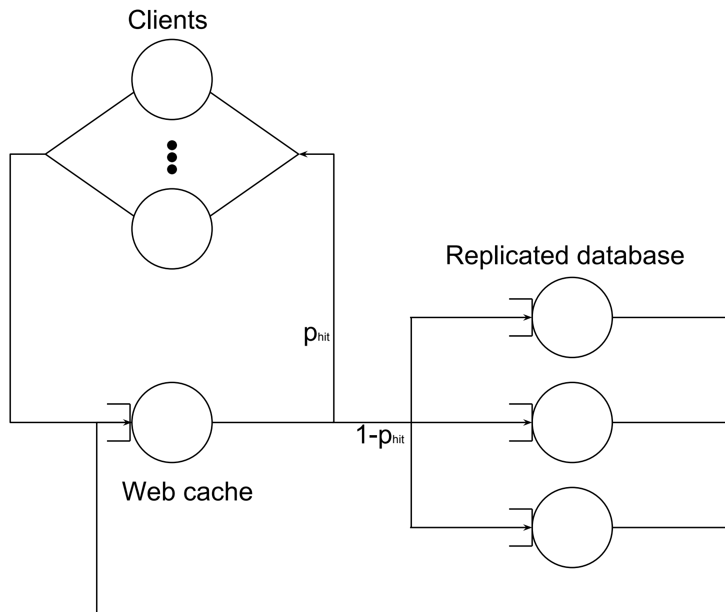


Figure 2.5: A closed queuing network with cache and database models

policies against a real database system to evaluate its performance at workloads caused by cache miss behaviour.

In addition, a continuous-time Markov chain could be used to estimate the probability of a cache hit instead of the DES. For this reason, analytical models are very useful when evaluating simulations. Figure 2.6 shows the CTMC for a small cache - each state shows which resources are currently cached, transitions show the exponentially distributed rate at which one state can move to another. The probability of being in each state can be calculated by solving a system of linear equations that models the system at equilibrium (steady-state). On the other hand, when the prediction of an analytical model doesn't meet what is observed in reality, a fine-grained simulation can be used to find details that were lost in abstraction.

2.1.3 Symbolic execution and static analysis

Symbolic execution is a program analysis technique commonly used in system verification to test whether software meets or violates a specification or property. It is often described as a middle ground between performance models and profiling. Performance models provide a very coarse-grained abstraction, while profiling can only capture data

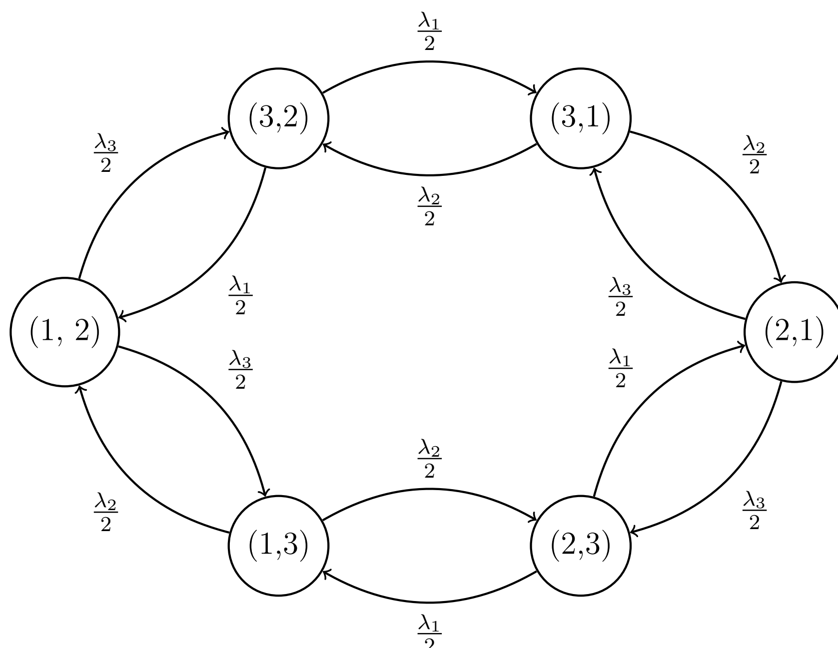


Figure 2.6: CTMC of the web cache with size of 2, 3 resources in total, and RAND replacement policy

from a finite set of possible executions. Probabilistic symbolic execution has been used to generate performance distributions [18].

2.2 Virtual-time execution

Virtual time was introduced by [19] as a paradigm for programming distributed systems that relies on optimistic simulation and state roll-backs to enforce causality constraints without explicit synchronisation. The described time-warp mechanism is nowadays used in large-scale parallel discrete event simulations [20]. However, we are interested in a slightly different notion of virtual time because of the direct execution aspect. A DES in the spirit of [19] that solves our problem would require detailed models of the hardware, host operating system, virtual machine, etc., running in a specialised simulator such as the one presented in [20]. Clearly, using a supercomputer to analyse the performance of a web application is not very practical and doesn't scale very well to being used by many developers on different systems.

2.2.1 VEX

This project is based on VEX [2], [3] for the JVM which describes a user-space scheduling profiler that is used to enforce a correct virtual-time execution sequence while recording timing events. A Java agent is used to instrument an application so that it communicates thread state changes and events from profiled methods to a scheduler which is controlled by VEX. For example a time-scaled method marked by a Java annotation is instrumented to trap into VEX so that its virtual time can be updated. Similarly, Java synchronisation primitives are modified to communicate with the scheduler. The thread that VEX wants to schedule is the only thread that is made runnable by the OS scheduler - VEX effectively forces the OS scheduler to respect its decisions. We proceed by answering some common questions related to schedulers with quotes

from VEX:

How is the next task to run chosen?

> The next thread to be resumed is the one that was suspended earliest on the virtual timeline; this is essentially the Borrowed Virtual Time (BVT) scheduling policy [21] with our method time scaling factors being equivalent to "weights" of BVT processes.

How is the timeslice chosen?

> The virtual time scheduler enforces a priority-free round-robin scheduling policy, with a default virtual timeslice of $100ms$ (which is the base time quantum for the default static priority in Linux).
>
> The scheduler allows an application thread to run for one timeslice by invoking a timed wait for the duration of the timeslice, suitably scaled by the Time Scaling Factor (TSF) of the thread's currently executed method. For example, if the default scheduler timeslice is 100ms and the current method has a TSF equal to 3 then the thread executing it gets a timeslice of $300ms$ in real time. The method's estimated virtual time is then its measured execution time divided by 3, thus adding $100ms$ of virtual time for $300ms$ of execution. This simulates the effect of the method executing 3 times faster within a single timeslice.

When VEX was designed the scheduling algorithm in Linux indeed had a fixed timeslice based on priority. However, CFS's distinctive feature are dynamic timeslices. This increases the implementation complexity for any solution that tries to keep the dynamic nature of CFS.

What happens if the scaling factor changes during the timeslice?

> In this case the scheduler is notified and updates the duration of the remaining timeslot accordingly. The scheduler is similarly notified if a method with scaling factor 1 calls a method with $TSF \neq 1$.

Trapping into the scheduler, regardless of whether it's in user-space or the kernel, deviates from the normal execution of the instrumented program. VEX tackles this by accounting for the overhead and adjusting accordingly. However, at a lower level of abstraction, side-effects such as flushing CPU caches could modify the execution in an unpredictable way. It is important to note that a significant part of VEX deals with time-scaling and prediction of I/O in virtual time.

2.2.2 Progress of threads in virtual time

More recent work in [22] extends VEX to run with multiple processes so that the performance of distributed programs can be analysed with the virtual time framework. Figure 2.7 shows a visualisation technique that was invented by O. Myerscough to clearly present complex execution sequences in both real and virtual time. Each thread of execution is shown in a different colour and pattern. Diagonal lines mean a thread is currently executing, horizontal lines mean a thread is not executing, vertical lines are instant jumps in virtual time. The staircase pattern in this figure is typical for uniprocessor systems where only one thread can progress.

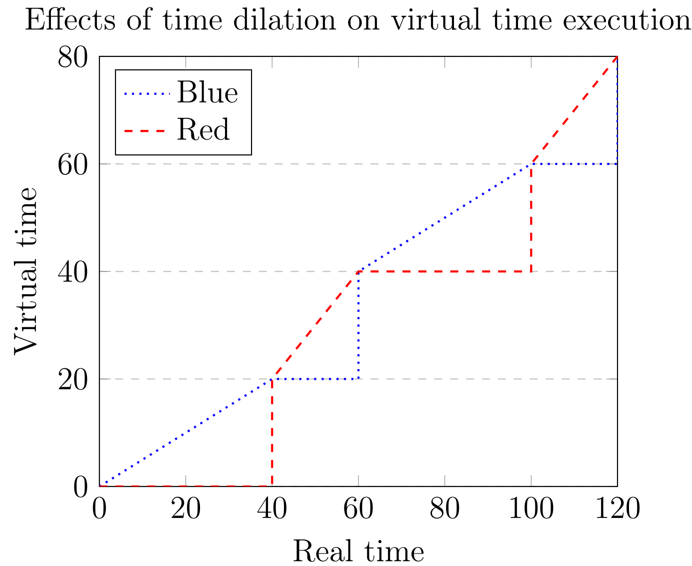


Figure 2.7: Progress in real and virtual time of two threads - blue appears to run two times faster because at virtual time 80 it has consumed $\frac{2}{3}$ of real time

In a multiprocessor simulation environment it is crucial to synchronise the virtual timelines of all cores to avoid causality errors. Consider the same example as above running on two cores without synchronisation (Figure 2.8). Imagine the red thread (producer) creates a piece of work for the blue one (consumer). Now suppose that we want to simulate the blue thread being two times faster than real. However, you can see that at real time 40 the blue thread has received two jobs already. We have managed to modify the timing behaviour by making the blue thread perceive real time 40 as 20. Unfortunately, the functional behaviour is the same as before and the simulation won't tell us anything.

In order to fix this we would need to make the red thread wait for its blue counterpart to catch up in virtual time before progressing further (Figure 2.9). Note that the diagram shows a particular synchronisation scheme that synchronises every 20 units of virtual time. There are many options, both conservative and optimistic, and choosing the right one is a matter of understanding what is being modelled. In reality the red thread wouldn't block simply because the blue one would execute two times faster. At granularity much lower than the minimum granularity of the scheduler this would introduce significant simulation inaccuracies. However, in a real system tasks get preempted all the time and events that happen at the granularity of preemption could be accurately simulated by integrating such synchronisation into the system scheduler.

2.2.3 VEX in the Linux kernel

A student project by James Greenhalgh from 2012 presents an implementation of virtual-time execution in the Linux kernel (K'VEX). Crucially, the author identifies a few limitations in his approach that my project tries to address. Firstly, the implementation is uniprocessor - this doesn't expose interesting thread schedules that would matter to the performance of a multi-threaded program running on a multi-core architecture. Secondly, the information passing mechanism to the kernel introduces a lot of overhead and adds noise to the simulated execution. The author identifies that he could optimise out a lot of the system calls and presents empirical measurement of the overhead.

Same as Figure 2.7 but on two cores without synchronisation

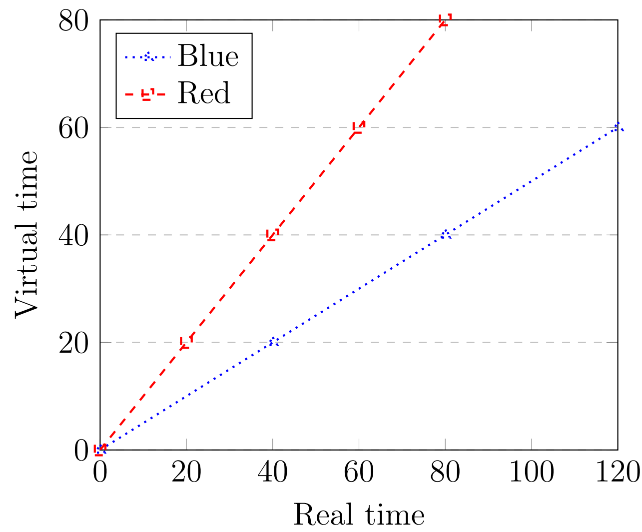


Figure 2.8: The marks show points that should be seen together in real time and how the diverge.

A correct schedule achieved through synchronisation

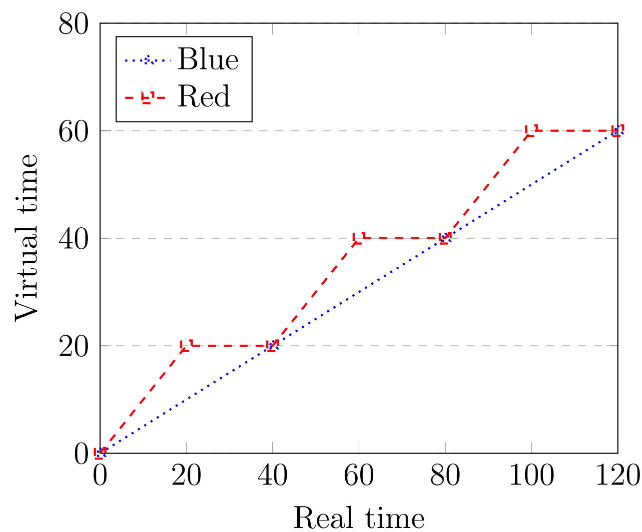


Figure 2.9: The plot suggests round-based execution with a synchronisation step of 20 time units.

2.2.4 Network emulation

Virtual-time execution is closely related to network emulation. Time-scaling VMs or containers is commonly used to analyse the effects of a faster network on an application or to test a network protocol against a simulated execution [23], [24]. A key challenge is placing events from emulation and simulation on a unified causally-consistent timeline [25].

Timekeeper, a solution that modifies the Linux kernel and uses time dilation in Linux containers (LXC), is presented in [4] and integrated with network emulators in [26]. We considered adapting Timekeeper to achieve similar effect for virtual-time execution. However, this has been shown to work only at coarse granularity - network delays dominate the overhead. In order to achieve method-level scaling and performance model execution, fine-grained control is required. The freeze-unfreeze mechanism in Timekeeper relies on signals which is not desirable in our case. A `SIGSTOP` signal acts on a thread group and would block all `pthreads` in a process instead of stopping only one of them until another one catches up. Another difference is that time-scaling factors in Timekeeper experiments are not meant to change dynamically at the function level.

In his work Baltas identifies this difference between time-scaling in network emulation and virtual-time execution as a key one:

> Grau et al. introduced the time-virtualised emulation environment [27], which dynamically changes the TDF according to some load metric to better balance the emulation's resource utilisation. Nevertheless, in both cases the passing of time is altered at the level of a virtual machine and for all its components. This results into poor granularity in the level of alteration and restricts these techniques from uses beyond network emulation like extending or accelerating the duration of an application method to identify its performance effect.

We note that there is soon-to-be-published work by the authors of Timekeeper which promises to deliver "much higher accuracy, scalability and repeatability than the older version and should be able to run in a distributed mode as well".

2.2.5 Causal profiling

Another approach to virtual-time execution which focuses on performance engineering is presented in [5]. Coz uses a technique called *virtual speedup* to measure the effect of speeding up a line of code by inserting forced slowdowns in other threads of execution. The Coz causal profiler implements virtual speedup by sampling the instruction pointer to find how often a section of code that falls within a virtual speedup section runs. A statistical approach is used to compute the average slowdown incurred by each thread. Coz instruments the `pthread` library calls for blocking and unblocking threads to correct the accumulated delays while a thread is blocked and another one is going through delays that both threads need to be credited for.

2.3 Linux

The target operating system for this project is Linux because of its open nature that would allow us to subject it to various hacks as well as its adoption among software developers who would benefit from using virtual-time execution for performance

engineering. What follows is an overview of some Linux fundamentals and detailed description of topics which are relevant to our implementation. The reason why we are interested in the kernel is because it is responsible for sharing hardware resources between processes. We have seen in the previous section that time dilation in virtual time is mostly concerned with changing the shares of a process in a given resource. For a more complete reference on the Linux kernel see [28], [29] and the source [30].

2.3.1 Linux kernel 101

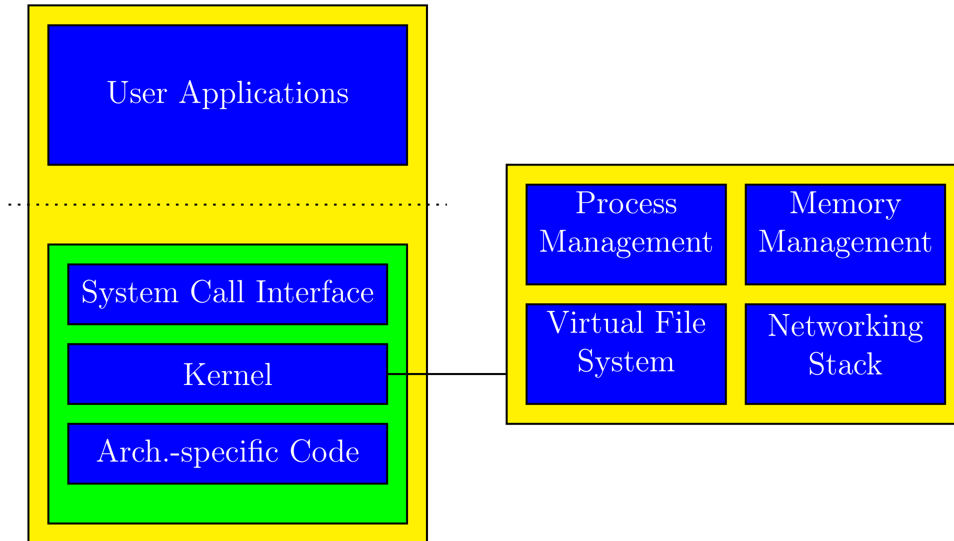
A Linux machine is in one of three states at any given time - executing user code, kernel code in process context (such as a system call), or kernel code in an interrupt handler (such as a timer tick). There are also kernel threads which are treated as lightweight processes that work on asynchronous jobs. The operating system defines core abstractions such as tasks (processes), files, virtual memory areas, etc., and subsystems that operate on them such as the scheduler, memory and file management. The main communication mechanism from user code into the operating system are system calls. New features can be implemented through the addition of system calls but this is discouraged. Instead, it's best to follow the device driver model, either exposing a file interface via `sysfs`, `ioctl`, a custom file system, `netlink`, etc. The Linux kernel is monolithic, but it supports dynamically importing and removing kernel code packaged in loadable kernel modules. Various kernel functions are "exported" for use in modules.

Apart from loadable modules, Linux has a few more unique features. Both processes and user-level threads are handled by the task abstraction, but threads are created as processes that share a set of resources (most notably the virtual address space) by passing the right arguments to the `clone` system call. The kernel supports symmetric multiprocessing (SMP) which introduces some revolutionary changes related to fine-grained synchronisation, per-CPU data, and how the OS works with hardware in general. It also significantly increases the complexity of the code-base. Another interesting trait is that Linux is a preemptible kernel [31]. In a nutshell, this means kernel code executing in process context is subject to context switches. This greatly improves the performance of some device drivers, but again at the cost of added complexity, because special considerations need to be made in regards to synchronising re-entrant functions. Note that, the implementation of spin-locks in Linux disables preemption.

2.3.2 Process priorities

As already mentioned, we are interested in understanding resource sharing in Linux and we focus on CPU-bound processes. In order to get a grasp on CPU sharing, we need to review process priorities. Linux, just as every major operating system, has a mechanism for specifying that a process is more important (has higher priority) than another process. When it comes to priorities, Linux separates processes in two main classes - real-time and other. Real-time processes are handled by the `FIFO` and `Round-Robin` schedulers which run as soon as there is a runnable real-time process. In this project we are interested in the other class which comprises of most processes running on your machine right now. Their priorities are called `nice` levels. They specify how "nice" a process is to others - the higher the nice level of a process, the lower its priority. Historically, every scheduler implementation has treated `nice` levels differently. In the current one only relative differences in `nice` values matter. The rule of thumb is that there is a 10% difference in weight between each level (See Table 2.2).

Figure 2.10: High-level architecture of Linux - user- and kernel-space and major subsystems in the kernel.



Nice	Weight	Nice	Weigh	Nice	Weigh	Nice	Weight
-20	88761	-10	9548	0	1024	10	110
-19	71755	-9	7620	1	820	11	87
-18	56483	-8	6100	2	655	12	70
-17	46273	-7	4904	3	526	13	56
-16	36291	-6	3906	4	423	14	45
-15	29154	-5	3121	5	335	15	36
-14	23254	-4	2501	6	272	16	29
-13	18705	-3	1991	7	215	17	23
-12	14949	-2	1586	8	172	18	18
-11	11916	-1	1277	9	137	19	15

Table 2.2: Nice levels to weights in Linux

Weights are the actual values that correspond to CPU shares linearly and you can see that the mapping to `nice` levels is exponential.

$$weight \approx \frac{1024}{1.25^{nice}}$$

For example a difference of 5 gives approximately three times more CPU time and it doesn't matter if the increase is from 5 to 10 or from 10 to 15.

2.3.3 Clocks and timers

Linux has a sophisticated and well-organised system for keeping track of time. The key abstractions over hardware are clock sources and events [32]. Clock sources are responsible for providing a monotonic timeline with high resolution and stable frequency. Clock events trigger interrupts at specified points on the timeline. There is an inherent trade-off between overhead and precision in timekeeping and the kernel provides various options to cover all use cases. For example the `sched_clock()` function used by the scheduler is very fast at the cost of some loss of precision. Another example are `hrtimers` [33] which accept a `slack` argument that allows for events with overlapping ranges to be grouped in a single interrupt. We note that the `hrtimer` API is by far the

cleanest, most accessible, well-organised and documented part of the kernel we have seen.

2.3.4 Fair process scheduling

The scheduler of an operating system is responsible for sharing the most important resource in a computer - the CPU, while respecting various constraints such as process priorities and affinity. The default scheduling algorithm in Linux - the Completely Fair Scheduler[4] [28], already uses the concept of virtual time. From the CFS design document:

> On real hardware, we can run only a single task at once, so we have to introduce the concept of "virtual runtime". The virtual runtime of a task specifies when its next timeslice would start execution on the ideal multi-tasking CPU described above. In practice, the virtual runtime of a task is its actual runtime normalized to the total number of running tasks.

However, the simple idea of perfect multitasking hides the complexity of implementation.

The Completely Fair Scheduler was designed to improve the performance of interactive processes which are the most common type of workload on a desktop system. An interactive process is best serviced by making sure it gets scheduled as soon as it becomes runnable. Normally, schedulers achieve this by allocating a larger timeslice and giving higher priority to interactive processes than to purely CPU-bound workloads. Consider an interactive text editor versus a CPU-heavy video encoder. The text editor is given a larger timeslice not because it needs it all the time, but because the system wants to make sure it has enough time when it is really needed. It is also given a higher priority so that it preempts the video encoder as soon as it becomes runnable. In the following paragraph we describe CFS's unorthodox interpretation of priority and timeslices in order to achieve interactivity.

CFS does something completely different than the example above. Assuming both processes have the same `nice` values they are allocated the same 'timeslice'. Timeslice is in quotes here because CFS doesn't really have the concept of a timeslice. In CFS processes are not assigned a timeslice, they are assigned a proportion of the processor. When a process is about to be scheduled the time it is given is calculated dynamically. Based on compile-time options an `hrtimer` could be set to drive pre-emption (a.k.a dynamic ticks) or the timing information is updated by a conventional scheduler tick at a fixed frequency (250 Hz on most systems, but also configurable). This is contrary to previous solutions with fixed timeslices based on process priorities which have problems with fairness [28].

The two main parameters for tuning CFS are *target latency* and *minimum granularity*. The target latency is a period of time in which all runnable tasks are guaranteed to run. It is fairly divided between the currently runnable tasks according to their weight[5]. However, as the number of runnable tasks grows, the interactivity provided by low target latency will be outweighed by high switching costs. The other parameter, minimum granularity, defines the shortest possible timeslice that a task should receive. It is also used to solve the above problem by stretching the scheduling period to the minimum granularity times the number of runnable tasks if that would give a longer

[4] `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`

[5] I'll come back to the notion of weight in the next section

period. CFS doesn't use the scheduling period as an epoch or round in which actually all tasks would get to run for their fair share. Instead the period is computed just before a task is about to start running so that this task's share (dynamic timeslice) can be determined. This is what allows interactive processes to possibly run multiple times during what would be considered a single scheduling period in an epoch-based view of the scheduler.

By using the `vruntime` value CFS keeps track of how much time each process has spent running on the CPU out of the time that it was allocated. Clearly, the interactive process spends only a fraction of its allocated time actually executing. When the text editor wakes up, CFS compares the time each process has spent using the CPU and lets it preempt the video encoder until it blocks again waiting for user input or the virtual runtimes of both processes become equal.

An interesting architectural pattern in the Linux scheduler are *scheduler classes*. The core of the scheduler encapsulates logic which is policy-agnostic and exposes hooks for implementing various scheduling policies. For example an algorithm can decide what happens when a task becomes runnable and when it leaves the queue, or how to check if the current task should be preempted by a given task. A list of scheduler classes can be defined such that the core of the scheduler chooses a task from the first class that has a runnable one. CFS is implemented as one of the scheduler classes in this list. The others are the real-time schedulers and the idle scheduler.

2.3.5 Fair group scheduling

The CFS algorithm described above operates on individual tasks, but remember that in Linux threads are tasks, too. An application could potentially starve others or have a significant impact on fairness by spawning many threads. By default runtime statistics about all threads in a process are grouped in a parent entity which is compared against other such entities. System administrators are free to create various other groups based on users, types of applications, time of day, etc. This gives rise to a hierarchy of scheduling entities (`struct sched_entitiy`) that CFS operates on instead of tasks.

2.3.6 CFS and Multi-processing

Everything described so far about process scheduling is true for a uniprocessor system. There is a lot of added complexity as the OS tries to make use of multi-core architectures. As alredy mentioned, modern versions of Linux support symmetric multiprocessing (SMP). Even though in this paradigm a single OS is shared by all processors, for scheduling every processor has its own *runqueue* to avoid synchronising access to a shared data structure on every context switch. This optimisation prevents CFS from making global scheduling decisions as the best task to run might be in the runqueue of another CPU.

Periodically, the kernel executes a load-balancing algorithm to make sure CFS has the preconditions to provide fairness and keep all cores utilised. The load-balancing algorithm plays a crucial role when it comes to the perceived performance of multi-threaded applications [34]. This is why its behaviour will need to be considered when evaluating my implementation. The load tracking metric is a combination of the weight, average CPU utilisation, and the hierarchy of control groups. It is important to make sure any modifications described in the section above don't influence load-balancing decisions severely such as forcing a task to constantly migrate between two processors.

3 | Tense[1]

3.1 Approach to virtual-time execution

As already discussed, both the *weight* of each process and *virtual runtime* are important for scheduling decisions. This gives two opportunities to drop in our implementation of virtual time. Imagine a process enters a block of code that should be simulated at two times the normal speed. This is equivalent to letting this code run two times longer *relative* to the rest of the program.

The first idea is to artificially scale the weight of the simulated process by the time dilation factor that we want to achieve. The second idea is to "lie" to CFS about the the virtual runtime by scaling the increment which happens when a scheduler or dynamic tick updates the runtime statistics for a task. Because CFS tries to always keep virtual runtimes close to each other, a process that appears to spend less time will be given more time to catch up. In this case we would need to divide the increment by the time dilation factor. One of our contributions is to explore both of these ideas and produce experimental results to evaluate them.

As discussed in the introduction, the primary use case for model-based performance engineering and for this project is in complex systems. It is almost certain that any such system will be developed to take advantage of multiprocessing. So why do we start with a uniprocessor implementation? Firstly, separation of concerns - problems such as designing the API exposed to user programs and creating experimental setups to evaluate the implementation are mostly shared between uni- and multiprocessing implementations. A multiprocessing implementation introduces more complexity to the virtual time scheduling algorithm. Dealing with all these problems at once is not desirable. Secondly, a uniprocessor implementation could be used to get some (although limited) insight about multiprocessing behaviour through multitasking. Finally, the learning curve for myself (someone who is fairly new to kernel development) wouldn't be as steep with a uniprocessor implementation. On the down side, translating a uniprocessing to a multiprocessing one poses some risks - some assumptions might not have a natural extension leading to a complete re-design of the system. Other projects, such as the Linux kernel itself, successfully transitioned to SMP. VEX also started with a uniprocessor implementation and added support for multiprocessing later by introducing two synchronisation schemes. This will be a good starting and reference point for our implementation.

[1] Tense stands for Tencho's scheduler extension, staying true to the Computer Science tradition of putting the author's name in the name of an algorithm or project.

3.2 A note on alternative ideas

In this section we briefly discuss some ideas that we spent some time pondering with but decided to drop in favour of the presented solution.

3.2.1 Tense as a scheduler class

Linux kernel scheduler classes present an implementation opportunity for our case. Imagine that a new scheduler class is defined for virtual time execution and placed as the highest priority class to be considered by the core scheduler. It mostly shares the behaviour of CFS, so that the simulation is accurate, but applies the changes needed to simulate a correct virtual time execution sequence. We then make sure that the instrumented tasks (threads) are placed to be scheduled by this class on creation. The first benefit is the implementation is completely decoupled from CFS which would make it easier to maintain and configure. Secondly, it guarantees a high degree of scheduling isolation between the process under analysis and other processes that might be running on the system. This is not what we ended up doing in the implementation that is presented in the following section. However, if the scheduling isolation provided by the current solution is not enough in practice, we think this would be a good addition.

3.2.2 Shared memory as a communication mechanism with the kernel

An interesting idea is to eliminate the need for new system calls, by storing the timing information in shared memory [35] or thread local storage (TLS[2]) and processing it when the kernel runs normally. It is inspired by modern hypervisors [36] that face the same problem when hosting an OS that requires timer interrupts at a given frequency. A hypervisor, as well as the guest operating system, can't rely on running all the time, so it needs to catch up on time when it gets a turn. Our prototype using a memory-mapped page per-process achieved a lower overhead than the presented solution using file operation system calls. However, it doesn't play well with sleeping and virtual time synchronisation in general because it gives too much room for error between when a critical change in time dilation happens and when our implementation can action on it. This would lead to reduced guarantees about the accuracy of our solution, but might be a preferable approach in some cases.

3.2.3 Tense as a modification to the JVM

A suggested approach at the start of the project is to introduce virtual CPUs, virtual time and thread scheduling in the JVM, making it more akin to a hypervisor. Normally, Java threads are mapped $1:1$ directly to OS kernel threads and the JVM cedes all thread scheduling decisions to the OS kernel. Instead, we want to achieve a hybrid $m:n$ mapping where m Java threads are scheduled by the JVM to run on n OS threads under the JVM's control. These Java threads are lightweight user-space threads, similar to the concept of "fibers", "green threads" or "actors". We would introduce n OS kernel threads as JVM system threads that get scheduled together by the OS, where $n \leq$ number of logical CPUs and can be considered to be "virtual CPUs". The idea is to add a virtual clock to the JVM running independently of the system clock, only

[2] https://www.akkadia.org/drepper/tls.pdf

progressing when the JVM is running. Finally, we add a thread scheduler to the JVM to schedule the m Java threads on top of n "virtual CPUs" according to some scheduling policy.

In VEX [2], [3], one of the main issues has been the limited access to native thread states. This problem would still exist in the above approach unless additional information is sent up to user space from the OS kernel (an upcall). There is a lot of complexity hidden in the OS kernel, but the user-space solution also suffers from it due to hard synchronisation issues that always appear in architectures with layered schedulers.

Note that a solution for the JVM is unlikely to be platform-independent by design, because, based on our inspection of the HotSpot sources, the changes would need to be made in architecture-specific parts of the implementation.

3.3 Algorithm

The core algorithm consist of a set of procedures which are invoked either by hooks that we have inserted into the Linux scheduler (`update_curr`, `after_task_tick`) or by a user process interacting with the custom file operations that we define (`add_current`, `remove_current`, `sleep_current`, etc.). This is our way of separating the implementation from the exposed interface. We give a high-level description of the most important procedures in the list of algorithms below. Throughout `task` refers to the `struct tense_task` data structure associated to the current Linux task. It has the following fields:

- `task_struct` - a handle to the task which owns this data
- `wakeup_time` - the virtual time when the process should wake up
- `wakeup_timer` - an `hrtimer` used for waking up this process when it goes to sleep
- `faster` - how many times faster this process is than real time
- `slower` - how many times slower this process is than real time
- `list` - a list head for the list of all tense tasks

We store a pointer to each `tense_task` in the corresponding `task_struct` so that we can access the current one by using the macro `current→tense_task`. Apart from this modification to the `task_struct`, the insertion of a few calls to our hooks in core scheduler code, and a header file that defines those hooks, the rest of the implementation lives in a loadable kernel module. This allows us to quickly change, re-compile and re-insert the module without needing to rebuild the kernel and restart the machine every time a small change is required[3].

3.3.1 Dynamic time-scaling

In order to support dynamic time-scaling we need a way to keep track of tasks that are executing in virtual time, set their time dilation factor, and cause changes to the scheduling sequence.

[3] In practice, we use a struct of function pointers holding the definitions of our hooks. Kernel code initialises this structure with do-nothing functions. When the module is inserted, the pointers are changed to point to the implementation defined in the module. This requires us to forcefully remove the module (`rmmod -f` instead of just removing it normally) because the system thinks it is busy at all times.

Function 1 `add_current`

The currently executing task joins a Tense experiment by initialising the required data structure and adding it to the list of tense tasks. This function is called by the `open` file operation.

```
task→task_struct := current
task→faster := 1
task→slower := 1
task→wakeup_time := 0
list_add task, tense_tasks
current→tense_task := task
```

The complementary function to `add_current` (described in Function 1) is `remove_current`. It is called by the `release` file operation of our custom file interface. We spare the whole description as it is almost as boring as its counterpart. However, if the removed task is the last one in an experiment, making the list of Tense tasks empty, the virtual time will be reset to 0 marking the end of this experiment. Both of these functions work with dynamic memory allocation from kernel pages. It is important to carefully manage this memory to avoid common bugs and leaks from our module.

Moving on to some more interesting functions, the `set_tdf` function changes the time dilation factor of the current process. It is invoked by writing the new value to the Tense file. This function makes sure that the time spent executing before the call is accounted for with the previous scaling factor by forcing CFS to update the time statistics for the process before setting the new value.

The `update_curr` function is responsible for advancing virtual time by applying the correct scaling factor to an execution slice accounted by CFS. We insert a call to this function into the corresponding `update_curr` function of CFS. This is a change to the kernel sources and requires running a custom-compiled kernel which might be a concern in some use-cases. See 5.2 for a discussion of more user-friendly approaches. CFS calls its `update_curr` at various points such as when a task is enqueued or dequeued, and during static and dynamic scheduler ticks.

Because CFS mostly works with scheduling entities and not tasks, extra care is needed when updating our virtual time. For example, when a scheduler tick hits a thread, `update_curr` will be called for the scheduling entity of this thread and for its parents (the process, a.k.a thread group) all the way up the hierarchy of entities. We need to update virtual time only when the entity is a task, while `vruntime` needs to be scaled for all entities so that time-scaling can be achieved between multiple processes each with multiple threads (a web server and a database) and any other arbitrary scheduling hierarchy.

Another implementation detail is that when a task is about to go to sleep we calculate its virtual wake up time, but there is an additional `update_curr` that happens during the context switch as the task is being deactivated. This update moves our virtual time to what is should have been when the sleep calculation was made.

It is important to note that `update_curr` lies on a path of the scheduler that needs to be extremely fast - it gets called very frequently in contexts that hold many locks. Therefore, we need to be careful not to introduce any significant overhead. An optimisation opportunity is to represent the time dilation factor as a single value and a shift offset to replace the integer division when scaling `delta_exec` with a bit shift.

Function 2 `update_curr (delta_exec, is_task)`
A direct hook into the corresponding function of CFS. It is used to scale a slice of execution before updating the `vruntime` of a scheduling entity.

---

if current is not a tense task **then**
  return delta_exec
end if
delta_exec := delta_exec * task→slower / task→faster
if is_task **then**
  tense_time + = delta_exec
end if
if task sleeping **then**
  task→wakeup_time + = delta_exec
end if
return delta_exec

3.3.2 Jumping in virtual time

Simulated execution based on time predictions from a performance model is in fact fairly simple in the current setting. We just need to add the predicted time to the `vruntime` of the task (appropriately scaled by its weight) and force a scheduling decision in case there is another task with much lower virtual runtime after the jump of the current one.

This functionality is exposed through the `lseek` file operation[4]. When the `whence` argument is set to `SEEK_CUR` the given offset is added to the current virtual runtime. If the `whence` is `SEEK_SET` instead the virtual runtime is directly set to the given offset. If you like having fun with time travel, you will be delighted to learn that negative offsets are allowed and will in fact decrease the `vruntime` in the `SEEK_CUR` mode. However, the behaviour for `SEEK_END` is currently undefined and will result in an error. Suggestions on how it should behave are welcome. `SEEK_HOLE` is used for sleeping as presented later, while `SEEK_DATA` is reserved for predicted I/O operations.

3.3.3 Sleeping in virtual time

The functions presented so far are sufficient to implement time dilation of CPU-bound processes executing on a single core. The next objective is to support accurate sleep times in the presence of time dilation. As an example, consider a process goes to sleep for $10ms$ while another one is executing a workload. To simulate a faster execution of the workload, the sleep has to take longer. Otherwise, upon waking up the sleeping process would be able to observe a state that suggests the workload is not actually executing faster. In a nutshell, the sleep of $10ms$ needs to be transformed into a sleep of arbitrary duration that takes $10ms$ of *virtual time*. Assuming static time dilation and only two processes, the problem is easily solved by simply scaling the sleep duration by the time dilation of the other process. However, our setting is made more complicated because time dilation factors could change at any time and we have an arbitrary number of processes.

An initial approach is to add the sleep time to the `vruntime` of each process before it wakes up. This forces processes to wait longer on the run-queue after they wake

[4]Some might think this is an abuse of the file interface, but I think it is definitely better than inventing custom system calls. In fact, solving problems by creatively using existing tools and abstractions in the kernel is welcomed by the community.

up, if there is an executing process that is behind in virtual time. We are able to simulate a relative speed-up. However, this solution has two severe flaws. Firstly, it doesn't work at all for simulated slow-downs when the sleeping process should actually wake up earlier than set. Secondly, it doesn't match real execution when time dilation is switched off because the added sleep time leads to increased queueing latency for sleepers.

The presented approach solves the problems above by keeping track of the virtual wake-up time and managing its own `hrtimers` for sleeping instead of calling into the `nanosleep` API. A crucial observation is that virtual wake-ups need to be very accurate, because the normal mechanism using `hrtimers` is as accurate as it gets and CFS usually runs a process as soon as it wakes up to boost interactivity. On the other hand, execution of CPU-bound workloads can be a bit jittery because scheduling is non-deterministic from a high-level point of view. To achieve the required accuracy, `hrtimers` are used to trigger the wake-ups. However, a timer is only set when virtual time has advanced to a certain point. This is when the virtual wake-up time of a task falls between the current virtual time and a bound which represents the longest time that might pass without us receiving a virtual time update while a process is actually executing in virtual time (usually this is the full duration of a tick). At this point our complex problem is reduced to the two-process static time dilation scenario described above, so a timer that will trigger the wake-up is set. The only events that could spoil the party is a change of time dilation either of the current process or a context-switch to a process running with a different time dilation. In this case we need to restart all started timers. This seemingly 'bad' case doesn't really influence the complexity or performance of the implementation because we initially start the timers in response to a context switch (or scheduler tick), and restarting a timer is the same as starting it.

Function 3 `sleep_current (duration)`
Puts the current task into an interruptible sleep for the specified duration in virtual time. This function is called by seeking the Tense file in a `SEEK_HOLE` mode with the offset being used as the sleep duration. Negative offsets are converted to unsigned values. It returns the virtual time when the process woke up so that if a signal was the reason, the restarting behaviour of `nanosleep` can be emulated.

---

task→wakeup_time := tense_time + duration
repeat
  current→state := TASK_INTERRUPTIBLE
  schedule
until task→wakeup_time == NO_SLEEP or **signal_pending**(current)
current→state := TASK_RUNNING
return tense_time

The timer callback for waking up a process is not very interesting. It just calls the `wake_up_process` function exported by the scheduler and cleans up the `wakeup_time` field of the waking Tense task.

3.3.4 Synchronisation of multiple timelines

The *tense_time* value works well when all tasks in an experiment are running on the same CPU. However, the transition to an SMP algorithm introduces a timeline for each CPU. The key challenge is keeping those timelines synchronised so that an accelerated process running on one CPU actually gets more time than a normal one on another.

Function 4 `prepare_wake_up`
Sets `hrtimers` for virtual wake-up as described above. This function is called after a scheduler tick and during a context switch. We omit some checks that make sure the task is indeed sleeping and an optimisation that doesn't restart timers if the difference is too small.

```
cur := current→tense_task
wakeup_range := tick * cur→slower / cur→faster
for all tasks in tense_tasks do
   if tense_time < task→wakeup_time < tense_time + wakeup_range then
      real_sleep := (task→wakeup_time - tense_time) * cur→faster / cur→slower
      hrtimer_set(task→wakeup_timer, real_sleep)
   end if
end for
```

The time of the accelerated CPU would quickly fall behind that of the normal one by virtue of the updates being scaled down. We already discussed that CFS has a load balancing algorithm, but in this case it wouldn't do anything about the imbalance because both CPUs are actually utilised, and in fact even if CFS did something it would be undesirable. CFS would migrate the normal process onto the accelerated CPU which appears to be under-utilised - clearly not what we aim to achieve.

There are various approaches to solving this problem originating in parallel discrete event simulation. The SMP algorithm of Tense is a take on conservative synchronisation with a configurable upper bound between the timelines of CPUs. As soon as the time for a given CPU goes beyond that bound compared to the minimum time of all CPUs, we stop executing tense tasks on it until the rest can catch up.

There are a lot of tricky things to keep in mind, though. For example, we need to be careful not to account the time that a CPU spent "blocked" as actual execution time for a task which would get fed back in through the `update_curr` hook and cause even more waiting for synchronisation. To solve this issue we carefully deactivate Tense tasks instead of busy-looping while waiting for synchronisation. One needs to be very cautious about the context in which tasks are activated and deactivated as this might lead to hard to debug soft lock-ups.

Another issue is a possible deadlock if tasks on a "blocked" CPU hold locks that are required by tasks on the other CPUs in order to make progress. To solve this we keep track of the number of runnable Tense tasks on each CPU. Only the times of CPUs with at least one runnable task are considered when calculating the minimum.

There are also issues related to process migration which we are still working on at the time of writing. As an example, consider a CPU that is not part of a currently running experiment and thus has its `tense_time` at zero. This is fine because we just said a CPU with no runnable Tense tasks does not influence synchronisation. Now imagine that one of our Tense tasks is migrated to this CPU for whatever reason (load balancing, processor affinity, etc.). As soon as this task gains some execution time on its new CPU, the `tense_time` of this CPU starts playing a role in the synchronisation algorithm. However, it is highly likely that the times of other CPUs are far ahead as the experiment has been running for a while before the discussed migration happens. This causes the whole experiment to unnecessarily synchronise against the newly joined CPU. In the presented case we could just detect there has been no activity on this CPU and set its `tense_time` to the current minimum. However, we are looking for a solution which can solve the migration problem in general.

3.4 User library and utilities

Thinking about the project from a user's point of view early in development was very beneficial. Unfortunately, there is a large gap between providing a user-friendly interface and iterating fast while looking for a reasonable solution to the core problem that the project solves. In this section we present the vision and steps towards the user experience for this project.

The C dynamic library `libtense` is made up of two components. Firstly, it wraps the Linux file interface described in the previous section and exposes a simple and slightly opinionated C API.

```
// libtense/tense.h

int tense_init(void);

int tense_destroy(void);

int tense_health_check(void);

int tense_scale_percent(int percent);

int tense_scale_pop(void);

int tense_scale_clear(void);

int tense_move(const struct timespec *delta);

int tense_time(struct timespec *time);

int tense_sleep(const struct timespec *duration);
```

Most functions should be self-explanatory, but we provide a brief description of each. The return value always indicates whether the call was successful or if an error occurred.

- `tense_init` Must be called by each process or thread to add it to a Tense experiment or start a new one if there isn't one already running. This functions simply opens the Tense file. The behaviour of other functions (apart from health-check) is not defined unless `tense_init` has been called successfully.

- `tense_destroy` Must be called by each process or thread that successfully called `tense_init` in order to clean up when that process or thread is exiting an experiment. This is normally done right before a process exits or a thread returns, but this is not a requirement. This function closes the file and cleans up resources used by the library in user-space.

- `tense_health_check` This function checks that the system supports Tense by making sure the kernel module is inserted, the Tense file is mounted at the expected path, and various other checks.

- `tense_scale_percent` Sets the time dilation factor in percent.

- `tense_scale_pop` Resets the time dilation factor to the previous value or one if there is no previous value. This function can be used together with `tense_scale_percent` to implement nested time dilation without needing to keep a reference to the time dilation of the parent.

- `tense_scale_clear` Resets the time dilation factor to one and clears any previous values.
- `tense_move` Adds `delta` to the virtual runtime of the caller.
- `tense_time` Writes the current virtual time to the `struct timespec` pointed to by `time`.
- `tense_sleep` Puts the process to a virtual-time aware sleep for the specified duration.

The second component of the library is an instrumentation layer which wraps some common functions to avoid the need for adding boilerplate code and changing calls to standard library routines in a user's project. We refer to it as the pre-load library because it injects a function into the `init_array` section of the binary. If the path to `libtense` is specified in the `LD_PRELOAD` environment variable when running an executable, this function will check if Tense should be enabled for this executable and enable the instrumentation even if that binary was not linked with the library.

The instrumentation applies a wrapper to `pthread_create` which replaces the starting routine with one that initialises Tense before it, reports the time when it returns, cleans up, and returns the original value. Some of the other instrumented functions are `clock_gettime` and `nanosleep`. We replace calls to read the monotonic clock with calls that return our virtual time as this is probably the desired behaviour in 90% of the cases. The sleep calls are replaced so that we can trigger the wake up at the correct moment in virtual time as already discussed.

This machinery allows us to run and time applications in virtual time without any modification to their source code or the need to recompile them. However, if they want to apply time scaling or jumps in virtual time to certain parts of the code, a user would currently need to insert calls to the corresponding `libtense` functions in their project and recompile it. We realise this is not ideal and that perhaps the best solution is to use an ELF binary instrumentation library (`libelfsh`) to statically re-write the executable around a set of specified functions for time-scaling. Using a dynamic instrumentation solution such as Pin is not sensible because that would have some noticeable impact on the behaviour of the application.

4 | Evaluation

4.1 Experimental setup

Our patches are based on the 4.16 Linux kernel. In some of the early experiments we run an Ubuntu image in the Qemu hypervisor with the default KVM configuration and options that enable `virtio`. Later we run all experiments on bare-metal with the default configuration of Ubuntu 16.04. The machine used is a Lenovo y510p personal laptop, during experimental runs there is usually a text editor and a few browser tabs open, which we believe is representative of the environment of a normal user.

4.2 Dynamic time-scaling

In this section we evaluate the time-scaling behaviour of processes called "spinners" that count executions of a busy loop. The aim is to run a set of processes in parallel for a fixed amount of time. The output of each process, `cycles`, is the number of busy loop iterations it managed to complete in the specified amount of time. Instead of time scaling ratio we use an equivalent notion - `vtime_spd` which denotes the "speed of time" for each process. It's the inverse of a time scaling factor - you can't tell if time is going fast or the process is just slow. We expect that the `cycles` to `vtime_spd` ratio for each process is the same.

4.2.1 Using process priorities

We emulate the time dilation effect by changing the priority of each processes - a process with higher priority gets a larger share of the CPU making it seem faster than others. As already mentioned, Linux priorities translate to weights (Table 2.2) for scheduling entities in CFS.

Figure 4.1 shows the results from 40 runs of the following experiment. 4 spinners with nice levels $0, 3, 5, 6$ are executed concurrently. There are 10 runs for each of the selected durations $30ms, 100ms, 1s, 10s$. We compute the reference `cycles` to `weight` ratio for `nice` level 0 and compute the relative change for the same ratios at different `nice` levels. For example, for `nice` level 3 where c_0 and c_3 are the number of cycles completed by spinners 0 and 3 respectively the error is:

$$e_3 = \frac{\frac{c_0}{1024} - \frac{c_3}{526}}{\frac{c_0}{1024}}$$

The ratios converge nicely when given enough time ($10s$) and the error is about 10% in the $100ms$ range which matches the 10% prediction error and timeslice of $100ms$ by VEX. A caveat is that the experiment doesn't take the starting delay into account - the 4 spinners can't start executing at exactly the same time. They are given the

same amount of time to execute, but there is no guarantee the intervals will precisely overlap. This certainly has some negative effect when execution durations are short.

4.2.2 Scaling `vruntime`

Even though changing `nice` levels seems promising, we need something to compare against. It seems there isn't a significant enough difference to conclude one approach is better than the other (Figure 4.2). The bias towards the negative side of the error confirms our expectation that the $30ms$-duration experiment is severely influenced by starting delay.

4.2.3 Illustrating the difference between the two approaches

This is a reproduction of a similar experiment described in VEX. A program computing the first 5000 digits of π is used for the experiment. The idea is to run copies of this program with different scaling factors and make sure they finish in order - fastest to slowest. I present scheduling sequences obtained using `ftrace` and `kernelshark`. The correct behaviour is achieved by both priority scaling (Figure 4.3) and `vruntime` scaling (Figure 4.4), but the two approaches lead to very different scheduling sequences.

Priority scaling results in timeslices of varying duration - longer for faster processes and shorter for slower ones. On the other hand, with `vruntime` scaling all timeslices are of equal duration, but faster processes get a timeslice more frequently than slower ones. Another way to think about it is that the transformation we apply to timeslices is shifted from the time domain (priority scaling) into the frequency domain (`vruntime` scaling). We conclude that both approaches are viable, but the latter has a more elegant implementation and does not clash with the `nice` interface. Consequently, the experiments to follow use `vruntime` scaling unless stated otherwise.

Figure 4.1: Relative error of “nice spinners” with different priorities

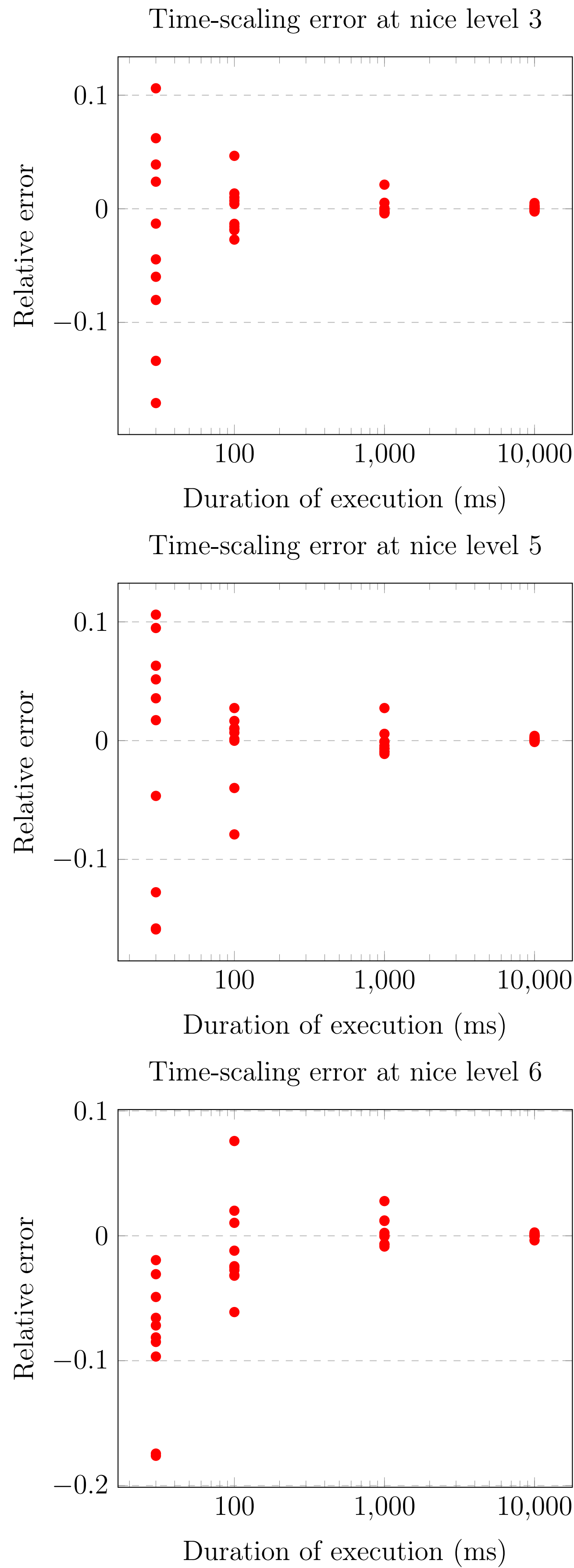
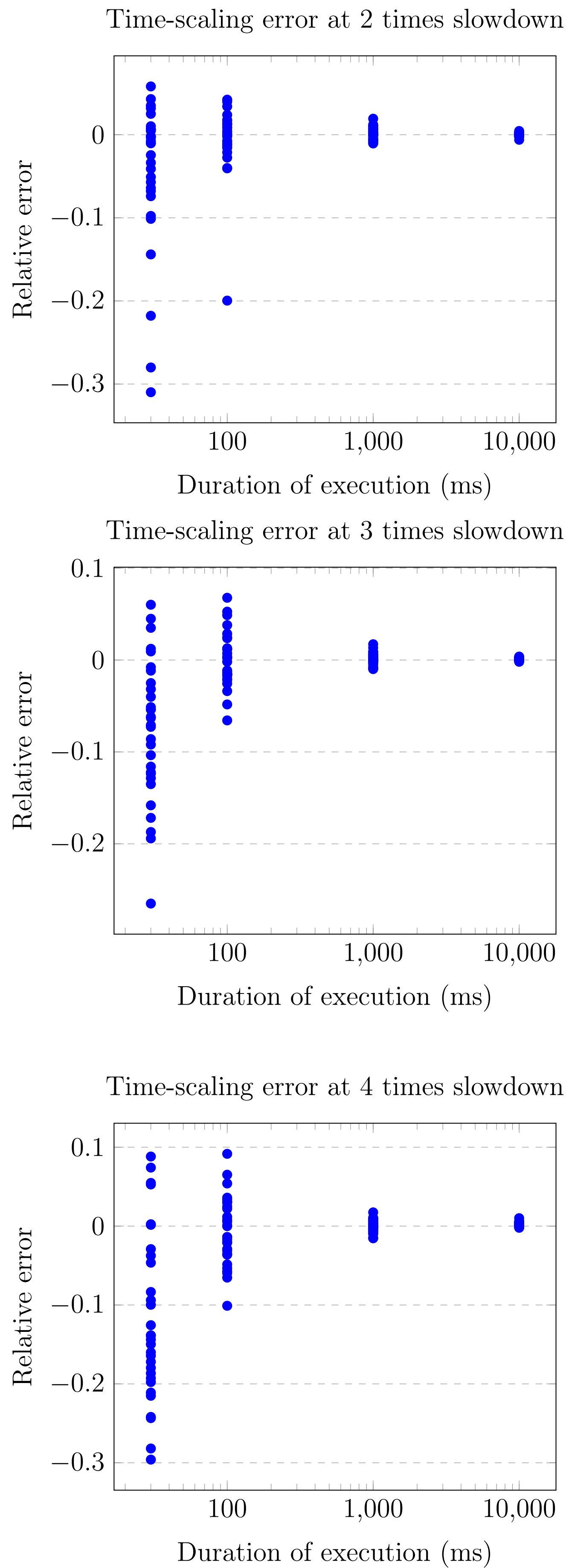


Figure 4.2: Relative error of “vruntime spinners” with different scaling factors



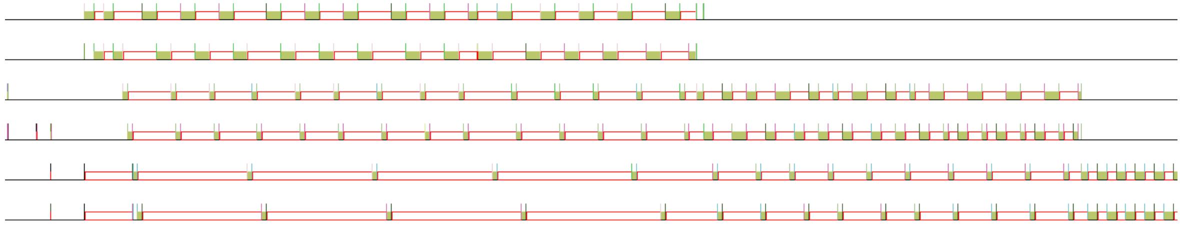


Figure 4.3: Scheduling sequence of 6 π-computers - 2 slow, 2 normal, and 2 fast; Using nice levels

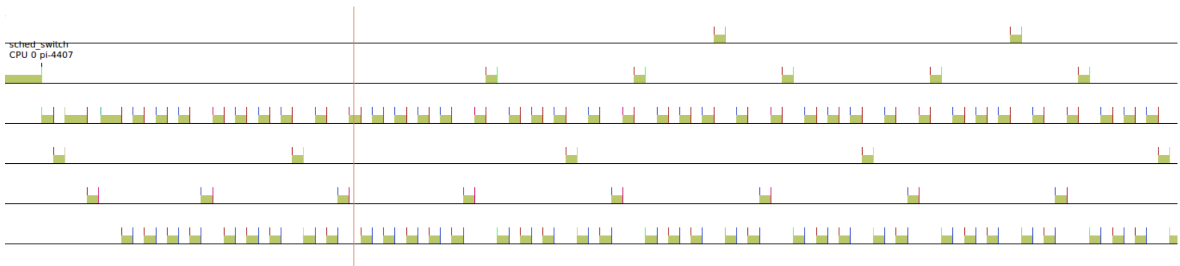


Figure 4.4: Scheduling sequence of 6 π-computers - 2 slow, 2 normal, and 2 fast; Using vruntime scaling

4.3 Accurate sleep duration in the presence of time-scaling

Moving on to less trivial processes, the next experiment consists of a process that spawns two `pthreads` - a producer and a consumer, communicating through a shared queue. The producer places a job in the queue and spends exponentially distributed periods of time sleeping, modelling a Poisson process. The consumer busy-waits for jobs while the queue is empty, records the time a job spent in the queue and executes a deterministic workload to simulate service time. We vary the service time by changing the number of iterations of a loop in the workload function. Again, we are interested in time-scaling the consumer to perform a what-if experiment. The main difference from previous experiments is that now there is an off-CPU process involved in the experiment. Remember that CFS's `vruntime` is only based on execution time. It also does a good job at boosting sleepers when they try to wake up. Because the producer spends very little time actually executing code, its `vruntime` will be less than that of the consumer at any point during the execution of the program. To simulate a speed-up of the consumer we would normally have its `vruntime` progress two times more slowly.

In the spirit of test-driven development we present this normal behaviour as red data points labelled `no sleep`. After modifying tense to correctly account for time spent sleeping as described in the previous chapter, the corresponding green data points are closer to the expectation. Note that in the plots both the waiting time and the utilisation are based on empirical measurements so we see noise in both “dimensions”.

We also present a measurement with all internal tense logging switched on. One can observe how the overhead of writing detailed logs from the kernel module translates into higher utilisation and waiting times.

Finally, we present a measurement that switches the time dilation factor on and off every time the server starts processing a new job. This is indicative of the significant overhead of the framework in this case which is especially visible in Figure 4.6. In that particular experiment the workload takes a very long time to run so the overhead

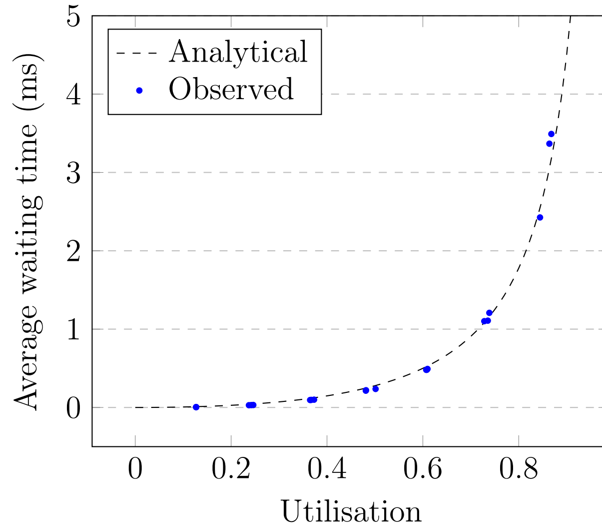
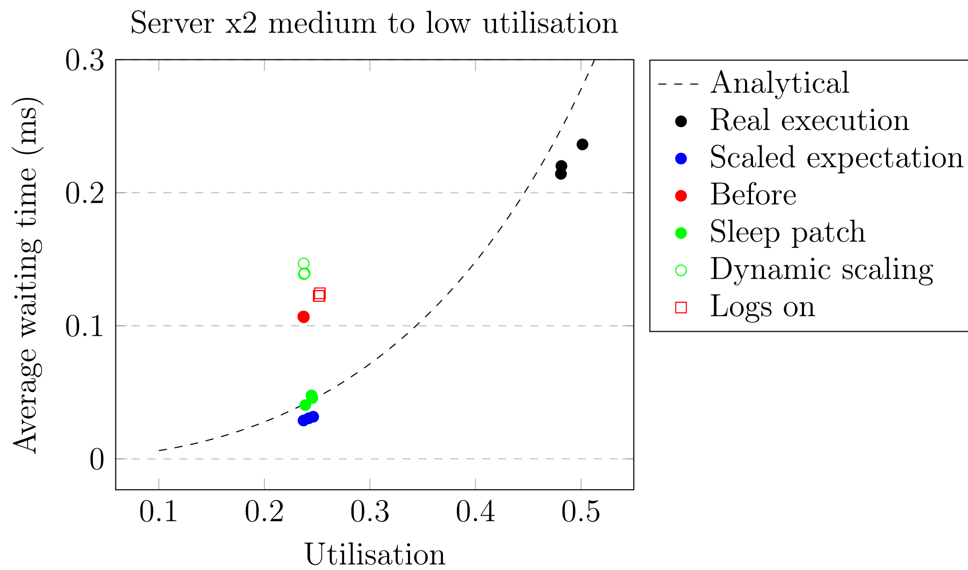


Figure 4.5: The test program is executed without any instrumentation to set expectations at 7 utilisation levels. There are 3 data points for each utilisation level, with all being clustered apart from the highest utilisation which causes some jitter.



builds up as time goes on.

[1] The inter-arrival times are actually sampled with rate 1 req/ms but the perceived arrival times are more spread out because placing a task on the queue is not instant. The 0.91 figure is a best fit.

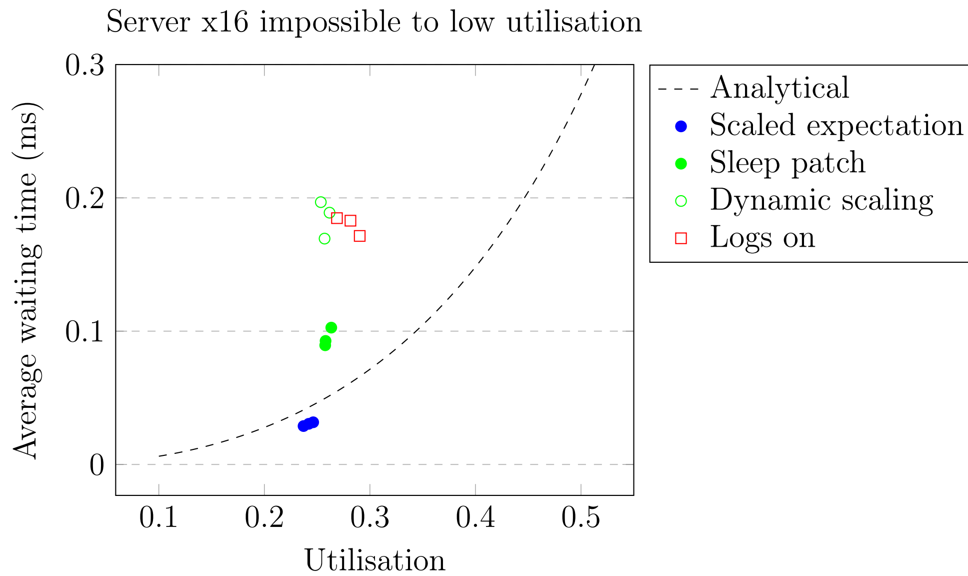
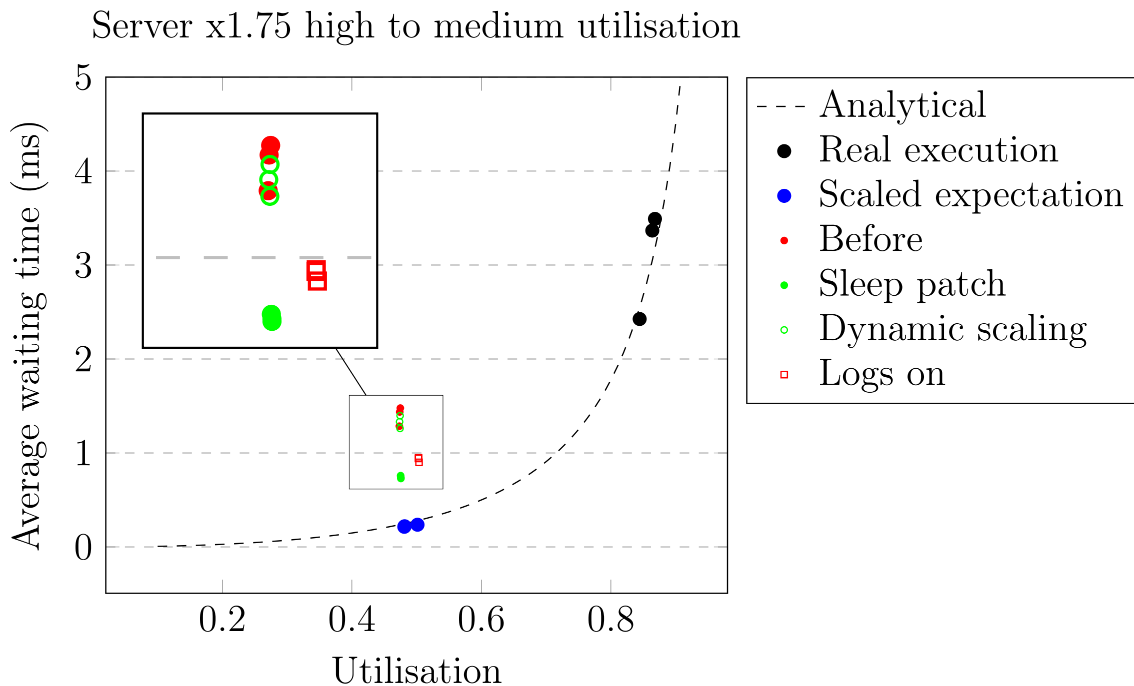


Figure 4.6: The data points for the real execution case are off the scale because this example is designed to cause the ring buffer that represents the queue to overflow because of how slow the server is. The prediction with the naive approach is also off-the scale.



4.4 Dynamic time-scaling in a large application

The next objective is to use the developed tools on a larger application which was not designed specifically for testing them. PARSEC [37] is a benchmark suite composed of multi-threaded programs. It is designed for evaluating the performance of multiprocessors, but we can restrict all threads to a single CPU to evaluate the uniprocessor implementation. Dedup is one of the compute-intensive applications in PARSEC. The workload is passed through a pipeline of stages. The first and last stage are serial (single thread) for input and output. The 3 parallel stages in between consist of thread pools whose size can be configured. Each pair of consecutive stages is connected by a queue. We are interested in observing the effects of speeding up a certain function in one of the parallel stages. The compression stage uses zlib. We recompile the library with options for faster compression to produce an expectation and then execute the slower version with a relative speed-up in virtual time. Each time we call dedup with the following options. This uses the native input size and spawns more than 20 pthreads:

```
dedup -c -p -v -t 8 -i FC-6-x86_64-disc1.iso -o output.dat.ddp -w gzip
```

The function of interest, `compress`, is called 168354 times when executing with the specified input. The average time it takes to execute a single call with the default `zlib` build is $99000ns$. With the fastest build this is reduced by $\approx 20\%$ to $79000ns$. The overall execution of the benchmark takes 18.9 and 16.9 seconds (mean of 5 runs) of user time in each case respectively - this is $\approx 10\%$ overall speed-up. Execution in virtual time without any speed-up vs. a speed-up of 20% of the `compress` function predicts an overall improvement of $\approx 11\%$ (also based on 5 runs). However, the predicted absolute times are about 2 seconds greater than reality in each of the two cases - with an without time-scaling. This indicates the overhead of the framework is $\approx 10\%$. Microbenchmarks of the dynamic-time scaling calls where times were measured using `strace` predict an overhead of about 1% so there is more to investigate on this front in the future.

5 Conclusion

5.1 Summary

We have presented an investigation into virtual-time execution and our solution - Tense. Even though there is a significant amount of work to be done before Tense becomes the best it can possibly be, this project lays a solid foundation. The key features of virtual time execution - moving threads in virtual time and dynamic time-scaling, are supported accurately by our loadable kernel module.

5.2 Future work

5.2.1 Avoid the need to run a custom kernel

As discussed earlier, some of the core functionality of Tense relies on hooks that were inserted into CFS and the core Linux scheduler code. This is not great in terms of usability because it means any developer wanting to install Tense would need to run a modified Linux kernel. A potential solution is to use the KProbes [38] package. KProbes use standard exception handling and breakpoint mechanisms to insert probes in the kernel. They are used for building debugging and tracing tools such as SystemTap [39]. In theory, KProbes look exactly like what we need, but there were some concerns that prevented us from implementing our current solution with them and will require further investigation. The cited article states that there are some performance issues with KProbes the most worrying of which is that the execution of KProbes is serialized across all CPUs on an SMP machine.

5.2.2 Improve user-space features

The development of Tense so far has been informed, but not driven by, some use cases around bottleneck detection and integration testing in virtual time on the JVM. Now that the project is moving out of the investigation phase our plan is to put a lot more effort into supporting these cases. In the bottleneck detection case this will require some cooperation with other tools that use tracing to spot potential bottlenecks in code. Tense would need to take the reported stack traces and perform experiments to simulate the effects of speeding up the region of interest. This would allow us to quantify the impact of optimising a potential bottleneck. In the other case, we would at least need to develop a user library or bindings to the existing C library for the JVM. There will probably be some additional issues related to handling internal JVM threads and safepoints. It is also interesting to explore the usefulness of profiles captured by traditional tools such as `perf`, but on programs executing in virtual time.

5.2.3 Continue the development and evaluation of the SMP algorithm

There is a lot more that we would like to know about the behaviour of the presented solution on multi-core architectures. More specifically, it is non-trivial to assess how the overhead of our (or any) synchronisation mechanism for virtual time affects the functional behaviour of CFS. There are subtle cases such as process migration and load balancing that we have safely ignored in the current evaluation, but are very common in real applications and need to be considered carefully.

In practice, there are commonly used concurrency patterns that could be identified in code via static analysis (or annotated by the programmer). Exploiting relaxations that can be applied in the presence of such high-level patterns could improve the speed of virtual time simulation without sacrificing accuracy.

Bibliography

[1] T. Decapua, "New roles, responsibilities redefining performance engineering in the enterprise." [Online]. Available: https://techbeacon.com/sites/default/files/gated_asset/state-of-performance-engineering-2015-16_final2.pdf

[2] N. Baltas and T. Field, "Software performance prediction with a time scaling scheduling profiler," in *IEEE International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems - Proceedings*, 2011, pp. 107–116.

[3] ——, "Continuous performance testing in virtual time," in *Proceedings - 2012 9th International Conference on Quantitative Evaluation of Systems, QEST 2012*, 2012, pp. 13–22.

[4] J. Lamps, D. M. Nicol, and M. Caesar, "TimeKeeper," in *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation - SIGSIM-PADS '14*. New York, New York, USA: ACM Press, 2014, pp. 179–186. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2601381.2601395

[5] C. Curtsinger and E. D. Berger, "Coz," in *Proceedings of the 25th Symposium on Operating Systems Principles - SOSP '15*. New York, New York, USA: ACM Press, 2015, pp. 184–197. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2815400.2815409

[6] C. U. Smith and C. U., *Performance engineering of software systems.* Addison-Wesley Pub. Co, 1990. [Online]. Available: https://dl.acm.org/citation.cfm?id=533597

[7] ——, "Software Performance Engineering Then and Now," in *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development - WOSP '15*. New York, New York, USA: ACM Press, 2015, pp. 1–3. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2693561.2693567

[8] M. Woodside, G. Franks, and D. C. Petriu, "The Future of Software Performance Engineering," in *Future of Software Engineering (FOSE '07)*. IEEE, 5 2007, pp. 171–187. [Online]. Available: http://ieeexplore.ieee.org/document/4221619/

[9] "Perf Linux profiler." [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[10] S. Rostedt, "ftrace - Function Tracer." [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt

[11] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, p. 74, 2 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2408776.2408794

[12] "Linux perf_events Event Sources." [Online]. Available: http://www.brendangregg.com/perf_events/perf_events_map.png

[13] "Flame Graph via Linux perf_events." [Online]. Available: http://www.brendangregg.com/flamegraphs.html

[14] L. G. Birta and G. Arbez, "Modelling of Continuous Time Dynamic Systems," 2013, pp. 269–289. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-2783-3_8

[15] "SolidThinking Embed 2017: Visual Environment for Model Based Development of Embedded Systems." [Online]. Available: https://solidthinking.com/embed_land.html

[16] J. Neyman, "Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability," pp. 333–380. [Online]. Available: https://www.jstor.org/stable/91337

[17] "t-distribution - Table of selected values." [Online]. Available: https://en.wikipedia.org/wiki/Student%27s_t-distribution#Table_of_selected_values

[18] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 49–60. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884794

[19] D. R. Jefferson and D. R., "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 7 1985. [Online]. Available: http://portal.acm.org/citation.cfm?doid=3916.3988

[20] P. D. Barnes, C. D. Carothers, D. R. Jefferson, and J. M. Lapre, "Warp Speed: Executing Time Warp on 1,966,080 Cores." [Online]. Available: https://gdo149.ucllnl.org/attachments/20776356/24674624.pdf

[21] K. J. Duda, D. R. Cheriton, K. J. Duda, and D. R. Cheriton, "Borrowed-virtual-time (BVT) scheduling," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 261–276, 12 1999. [Online]. Available: http://portal.acm.org/citation.cfm?doid=319344.319169

[22] O. Myerscough, "Distributed Virtual Time Execution of Programs," Tech. Rep., 2015. [Online]. Available: https://www.doc.ic.ac.uk/teaching/distinguished-projects/2015/o.myerscough.pdf

[23] D. Gupta, K. Yocum, M. Mcnett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To Infinity and Beyond: Time-Warped Network Emulation." [Online]. Available: https://www.usenix.org/legacy/event/nsdi06/tech/full_papers/gupta/gupta.pdf

[24] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "DieCast," *ACM Transactions on Computer Systems*, vol. 29, no. 2, pp. 1–48, 5 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1963559.1963560

[25] D. Jin, Y. Zheng, H. Zhu, D. M. Nicol, and L. Winterrowd, "Virtual Time Integration of Emulation and Parallel Simulation," in *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 7 2012, pp. 201–210. [Online]. Available: http://ieeexplore.ieee.org/document/6305913/

[26] J. Lamps, V. Babu, D. M. Nicol, V. Adam, and R. Kumar, "Temporal Integration of Emulation and Network Simulators on Linux Multiprocessors," *ACM Transactions on Modeling and Computer Simulation*, vol. 28, no. 1, pp. 1–25, 1 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3174299.3154386

[27] A. Grau, S. Maier, K. Herrmann, and K. Rothermel, "Time Jails: A Hybrid Approach to Scalable Network Emulation," in *2008 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE, 6 2008, pp. 7–14. [Online]. Available: http://ieeexplore.ieee.org/document/4545320/

[28] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley Professional, 2010.

[29] D. P. D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. O'Reilly, 2006. [Online]. Available: https://dl.acm.org/citation.cfm?id=1077084

[30] "Linux Kernel." [Online]. Available: http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git

[31] R. Love, "Lowering Latency in Linux: Introducing a Preemptible Kernel | Linux Journal," *Linux Journal*. [Online]. Available: https://www.linuxjournal.com/article/5600

[32] "Clock sources, Clock events, sched_clock() and delay timers." [Online]. Available: https://www.kernel.org/doc/Documentation/timers/timekeeping.txt

[33] T. Gleixner, T. Gleixner, and D. Niehaus, "Hrtimers and Beyond: Transforming the Linux Time Subsystems," *Proceedings of the Ottawa Linux Symposium*, 2006. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.576.453

[34] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler," in *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*. New York, New York, USA: ACM Press, 2016, pp. 1–16. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2901318.2901326

[35] T. Motylewski, "Sharing memory between kernel and user space in Linux." [Online]. Available: ftp://164.41.45.4/pub/os/rtlinux/papers/rtos-ws/p-c01_motylewski.pdf

[36] VMware and Inc, "Timekeeping in VMware Virtual Machines." [Online]. Available: https://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf

[37] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*. New York, New York, USA: ACM Press, 2008, p. 72. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1454115.1454128

[38] S. Goswami, "An introduction to KProbes." [Online]. Available: https://lwn.net/Articles/132196/

[39] V. Prasad IBM, W. Cohen, F. Ch Eigler, M. Hunt, J. Keniston IBM, and B. Chen, "Locating System Problems Using Dynamic Instrumentation." [Online]. Available: https://sourceware.org/systemtap/systemtap-ols.pdf