Journal of Parallel and Distributed Computing 133 (2019) 232–243

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

CTS: An operating system CPU scheduler to mitigate tail latency for latency-sensitive multi-threaded applications

Esmail Asyabi [a,b,*], Erfan Sharafzadeh [a], SeyedAlireza SanaeeKohroudi [a], Mohsen Sharifi [a]

[a] Distributed Systems Research Lab, School of Computer Engineering, Iran University of Science and Technology (IUST), Tehran, Iran
[b] Computer Science Department, Boston University, Boston, United States

HIGHLIGHTS

- It has been proven that FCFS scheduling of threads leads to lower tail latency.
- Experiments show that CFS policies lead to LCFS scheduling, aggravating tail latency.
- CTS policies ensure FCFS thread scheduling, which yields lower tail latency.
- CTS enforces our policies while maintaining the key features of the Linux scheduler.
- Experimental results show that CTS significantly outperforms the Linux scheduler.

ARTICLE INFO

Article history:
Received 30 September 2017
Accepted 3 April 2018
Available online 12 April 2018

Keywords:
CPU scheduling
Tail latency
Multi-threaded applications
Linux CFS scheduler

ABSTRACT

Large-scale interactive Web services break a user's request to many sub-requests and send them to a large number of independent servers so as to consult multi-terabyte datasets instantaneously. Service responsiveness hinges on the slowest server, making the tail of the latency distribution of individual servers a matter of great concern. A large number of latency-sensitive applications hosted on individual servers use thread-driven concurrency model wherein a thread is spawned for each user connection. Threaded applications rely on the operating system CPU scheduler for determining the order of thread execution. Our experiments show that the default Linux scheduler (CFS) idiosyncrasies result in LCFS (Last Come First Served) scheduling of threads belonging to the same application. On the other hand, studies have shown that FCFS (First Come First Served) scheduling yields the lowest response time variability and tail latency, making the default scheduler of Linux a source of long tail latency for multi-threaded applications. In this paper, we present CTS, an operating system CPU scheduler to trim the tail of the latency distribution for latency-sensitive multi-threaded applications while maintaining the key characteristics of the default Linux scheduler (e.g., fairness). By adding new data structures to the Linux kernel, CTS tracks threads belonging to an application in a timely manner and schedules them in FCFS manner, mitigating the tail latency. To keep the existing features of the default Linux scheduler intact, CTS keeps CFS responsible for system-wide load balancing and core level process scheduling; CTS merely schedules threads of the CFS chosen process in FCFS order, ensuring tail latency mitigation without sacrificing the default Linux scheduler properties. Experiments with a prototype implementation of CTS in the Linux kernel demonstrate that CTS significantly outperforms the Linux default scheduler. For example, CTS mitigates the tail latency of a Null RPC server by up to 96%, a Thrift server by up to 90% and an Apache Web server by up to 51% at 99.9th percentile.

1. Introduction

Large-scale Web applications (e.g., search engines and social networks) use parallelization to process large datasets instantaneously by breaking a user request into many sub-requests and distributing the sub-requests across a large number of individual servers. The user request, therefore, does not complete until the slowest of these sub-requests has fulfilled. In fact, the responsiveness of individual servers dominates the quality of delivered

* Corresponding author at: Computer Science Department, Boston University, Boston, United States.

E-mail addresses: easyabi@bu.edu (E. Asyabi), e_sharafzadeh@comp.iust.ac.ir (E. Sharafzadeh), sarsanaee@comp.iust.ac.ir (S. SanaeeKohroudi), msharifi@iust.ac.ir (M. Sharifi).

https://doi.org/10.1016/j.jpdc.2018.04.003
0743-7315/

services because not only the main request's responsiveness hinges on the slowest sub-operations but also the middleware may decide to drop replies arriving after a predefined deadline, which further degrades the quality of service. In such systems, focusing on the average latency is not sufficient. System designers concentrate on the tail of the latency distribution of individual servers that is the key driver of user perception [2,15,19]. For example, to answer a user request, consider a scenario wherein a user-facing system collects responses from 200 individual servers whose tail latency at 99th percentile is one *second*, meaning that one request out of 100 requests takes more than one *second* to complete on each server. It can be calculated[1] that 86.6% of the user requests will take more than one *second*. Therefore, under high degrees of parallelism, poor tail latency of individual servers impacts most of the user requests. This makes the tail of the latency a great challenge for developers of individual services and the main subject of intensive research that aims at eliminating the tail latency contributors by proposing novel application architectures, runtime environments, operating systems and hardware supports [8,24,30,32].

Latency-sensitive applications typically use a thread-driven or an event-driven approach. Thread-driven applications use blocking/synchronous I/O where a newly spawned thread handles the I/O requests for each new client connection. The number of threads running on the system is hence proportional to the number of active connections. Event-driven applications, on the other hand, use asynchronous/non-blocking I/O wherein several main threads (worker threads) handle I/O tasks by registering callbacks to be notified asynchronously. A dispatcher pulls out events (i.e., I/O tasks) from a buffer of ready file descriptors (FDs) and passes them to worker threads. The number of worker threads is fixed and typically is equal to the number of available CPU cores.

In thread-driven applications, the operating system process scheduler decides the execution order of threads. In event-driven applications, an application-level dispatcher determines the serving order [7,26,28,34]. Regardless of the concurrency model, the execution time of an I/O task is typically short. In fact, the execution times of I/O tasks are so short that the need for preemption is obviated. This leads us to conclude that non-preemptive queuing disciplines are suitable for modeling systems running latency-sensitive applications. Therefore, classical queuing disciplines such as First Come First Served (FCFS), Last Come First Served (LCFS) and Random Order of Service (ROS) are suitable policies for scheduling I/O tasks.

Many research works (such as [9,10]) have proven that even though FCFS, LCFS, and ROS policies lead to the same average response time, they result in different response time variability and tail latency. Studies (such as [6,23]) have shown that FCFS scheduling provides the lowest tail latency and variability compared to LCFS and ROS, making FCFS queuing the best performer in terms of the tail of the latency distribution. In event-driven applications, the application-level dispatcher can be tuned to serve I/O tasks in FCFS manner when the tail latency matters, whereas, in thread-driven applications, the operating system CPU scheduler determines the serving order.

CFS (Completely Fair Scheduler) is currently the default scheduler of Linux. The main objective of CFS is to share processor resources among running tasks fairly. To achieve this goal, CFS assigns each task an attribute called *vRuntime* to track the CPU consumption of running tasks. It also uses a red–black tree to keep the ready tasks sorted based on the values of their *vRuntime* attribute. At each scheduling decision, the scheduler chooses the task with the minimum *vRuntime* to be executed next, ensuring fairness. Since tasks belonging to IO-intensive applications spend less time on CPU compared to their CPU-intensive counterparts, they have a higher chance to be executed first, enhancing responsiveness. Moreover, CFS sets the *vRuntime* of newly awoken tasks to be the minimum of *vRuntime* values to increase the chance of immediate execution of newly woken tasks because it is likely that the task has been woken up due to an incoming I/O request [11,17,22,25]. CFS policy for prompt execution of newly woken tasks leads to LCFS scheduling of threads belonging to a threaded application. As mentioned before, compared to FCFS, LCFS service discipline exacerbates the tail of the latency distribution, making current Linux CPU scheduler unsupportive of thread-driven and latency-sensitive applications, regarding the tail of the latency distribution.

In this paper, we present CTS, an operating system CPU scheduler whose main goal is to trim the tail latency for thread-driven and latency-sensitive workloads. CTS is a 3-dimensional CPU scheduler. It leverages the default scheduler of Linux to perform the system-wide load balancing to distribute tasks among cores evenly (first dimension). At each core, CTS lets the default Linux scheduler to choose a process that one of its threads must be executed next (second dimension). Once the next process is selected, the CTS thread scheduler chooses a ready thread of the chosen process for execution (third dimension).

The main objective of the thread scheduler is to execute threads in FCFS order. To achieve this, we have added a new data structure called *Shadow Graph* to perform all operations needed for CTS, including tracking threads belonging to a process in $O(1)$ time complexity. At each scheduling decision, once the default Linux scheduler chooses the next process, CTS finds the first thread of the process to be executed next, ensuring sibling threads are served in FCFS order, alleviating the tail of the latency distribution for latency sensitive, threaded workloads. Note that, Linux refers to schedulable&executable entities as tasks. In this paper, we use the same terminology. However, we use threads to refer to tasks that do not have any child (forked) tasks and processes to indicate tasks that are the parents of one or more threads.

CTS scheduling strategy ensures that the main characteristics of the Linux scheduler, namely fairness and responsiveness, are not adversely affected. To achieve this, CTS leverages the default scheduler for process scheduling. At each scheduling decision, a process having a thread with the minimum *vRuntime* value is chosen as the next process. Hence, fairness is kept intact at process level, meaning that every process gets a fair share of processing resources as before. Given that under CTS, threads of the process that is chosen by CFS, run in FCFS order, and threads belonging to the same process typically perform the same job (stays on CPUs for the same amount of time that is bounded by the time slice), threads will get a fair share of processor resources. Similarly, CTS architecture ensures to keep the responsiveness of the default Linux scheduler intact at the process level, meaning that a process's CPU access latency is not impacted under CTS because the default Linux scheduler is still in charge of process scheduling. Furthermore, for threads belonging to a process, CTS enhances responsiveness by executing threads in FCFS order, resulting in lower response time variability and tail latency.

In summary, we make the following contributions:

- We present CTS, an operating system CPU scheduler whose objective is to trim the tail of the latency distribution for thread-driven workloads. To do so, using new data structures added to the Linux kernel, CTS tracks threads belonging to an application and guarantees to execute them in FCFS order using a thread scheduler, mitigating the tail latency for latency-sensitive threaded workloads.
- We use a 3-dimensional scheduling technique to maintain the main characteristics of the default Linux scheduler including fairness and responsiveness while mitigating tail

[1] $P(\text{response time} > 1\text{ s}) = P(\text{at least one subrequest takes more than } 1\text{ s}) = 1 - P(\text{none of the subrequests takes more than } 1\text{ s}) = 1 - 0.99^{200} = 0.866$.

latency. To this end, CTS leverages CFS to perform system-wide load balancing (first dimension) and core level process scheduling (second dimension). Finally, The CTS thread scheduler schedules threads of the CFS's chosen process in FCFS order. Having CFS in fully functional mode as the process scheduler guarantees maintaining the key properties of the default Linux scheduler.

- We have implemented a prototype of CTS in the Linux kernel and conducted extensive experiments using both micro-benchmarks and application-level benchmarks. The results demonstrate that CTS significantly outperforms the default Linux CPU scheduler. For example, it mitigates the tail of the latency distribution for a null RPC server by up to 96%, a Thrift server by up to 90% and the Apache Web server by up to 51% at 99.9th percentile.

The remainder of this paper is organized as follows. Section 2 presents a background on the Linux default scheduler and the impact of different queuing disciplines on tail latency followed by the design of CTS in Section 3. Section 4 reports the results of experiments on a prototype implementation of CTS in Linux. Section 5 presents related work and Section 6 concludes the paper.

2. Background

2.1. The default Linux scheduler

The completely fair scheduler (CFS) is the current default scheduler of Linux. Its main objective is to share processor time among running tasks fairly. To do this, it assigns each task an attribute called *vRuntime* to track the execution time of tasks. While a task executes on a processor, its *vRuntime* inflates. The speed of inflation depends on the task priority. The higher the priority, the lower the rate of inflation. To achieve fairness, CFS tries to keep the *vRuntime* of all existing tasks nearly equivalent. Therefore, tasks with the same priority receive an equal share of processor time, and since *vRuntime* values of tasks with higher priorities grow relatively slower, they get larger portions of processors time depending on their priorities [11,17,22].

CFS leverages a red–black tree to keep the ready tasks sorted based on their *vRuntime* values. Using the red–black tree, ready tasks are inserted or removed in $O(\log n)$ time. The leftmost task always has the minimum *vRuntime* value and is accessible in $O(1)$ time using a pointer to the leftmost node. At each scheduling decision, the scheduler chooses the task with the minimum *vRuntime* to be executed for a predefined amount of time (time-slice). The time-slice length varies, depending on the number of ready tasks on the red–black tree [17,25].

Fairness is a satisfactory feature for batch jobs that are desperate for spending time on CPUs. Interactive jobs, on the other hand, favor instant CPU occupation (responsiveness) over fairness. These jobs typically spend most of their lives in sleeping state, waiting for I/O devices. When an I/O event is triggered, the corresponding task is woken up. To achieve responsiveness, it is imperative to execute the newly awoken tasks as promptly as possible. Fortunately, CFS idiosyncrasies (e.g., *vRuntime*) help attain responsiveness. Since IO-intensive jobs typically do not spend much time on CPUs, their *vRuntime* values grow more slowly compared to batch jobs. Therefore, whenever they are woken up due to an I/O event, they have a higher chance to be the task with minimum *vRuntime* value and therefore be executed immediately. Nevertheless, CFS takes a conservative step to guarantee prompt execution of newly woken tasks: whenever a task is woken up, CFS sets its *vRuntime* to be the minimum *vRuntime*. Thus, the newly woken tasks having the highest chance to be executed next, enhancing responsiveness for IO-intensive tasks.

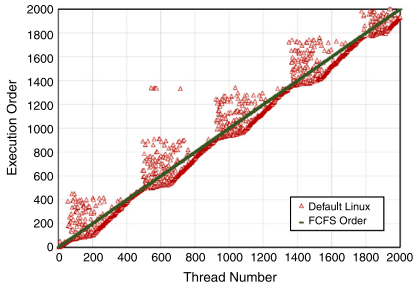


Fig. 1. Execution order of threads under the default Linux scheduler.

Prompt execution of newly awakened tasks by manipulating their *vRuntime* values to be minimum enhances responsiveness. However, as a side effect, it leads to a near LCFS service behavior. This means that in thread-driven applications, it is almost the case that the last thread receiving an I/O packet is the first to run. To demonstrate this phenomenon, we modified CFS to conduct an experiment to exhibit the behavior of CFS while serving a thread-driven application. We defined a global incremental variable that is incremented whenever a task is woken up. Each task is assigned a task number, which is set to the value of the global variable when the task is woken up. Whenever a task is executed on the processor, we simply print the task number. Using this information, we can demonstrate the order in which tasks are executed under CFS. A uniprocessor machine with a modified Linux kernel hosts a thread-driven application (Null RPC Server). We stress the RPC server using another machine to reach 70% CPU utilization in the server. Fig. 1 depicts the CFS serving order. The x-axis is the task number, and the Y-axis shows the order of execution. As it can be seen, the curve once a while deviates from $y=x$ (FCFS) and shapes harmonic spurs in the output spectrum. As we will describe in the next section, these spurs aggravate the long tail latency for thread-driven applications.

2.2. The service discipline impact on tail latency

Many research efforts have underscored that the queuing policy determines not only the average response time but also the variances and the tail of the latency distribution. Therefore, when it comes to latency-sensitive applications, employing a decent queuing strategy that leads to a reasonable average latency while preventing the variability of response times and occurrence of long tails is a matter of great concern.

When a hosted latency-sensitive application is thread-driven, the application threads typically perform similar jobs, and they usually spend less than the expected scheduling time-slice on CPUs to handle the incoming I/O request (less than 1 ms). This behavior leads us to use non-preemptive policies for modeling the systems that host latency-sensitive, thread-driven workloads whose tasks are so small that the need for preemption is obviated. Therefore, FCFS, LCFS and ROS are applicable policies in this context.

The specification of a single server queuing system is usually demonstrated with the mean and variance of response time. It has been repeatedly reported (e.g., in [9,13,29]) that the above queuing disciplines lead to the same average response time; however, the variance is evidently affected by different queuing policies. Regarding the fact that the distribution of service time does not matter as long as it follows a light-tailed distribution, in an M/G/1 queue, the variance of response times for the mentioned policies, denoted by *Var[W]*, have been derived and compared as follows [10]:

$$Var[W]^{FCFS} < Var[W]^{ROS} < Var[W]^{LCFS} \quad (1)$$

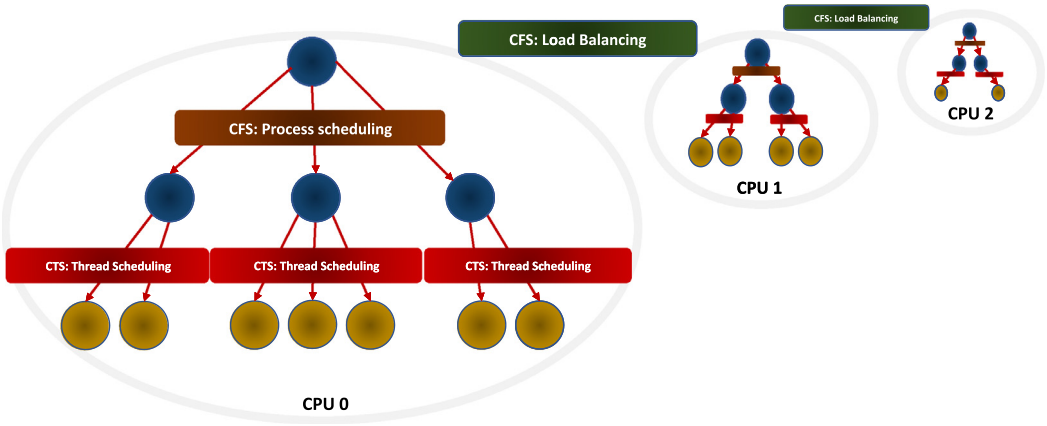


Fig. 2. CTS scheduling policy.

From Eq. (1), we can observe that FCFS will cause the least variability in response times compared to LCFS and ROS. Nguyen et al. [19] have addressed the mean and variability of response times as the key elements for predicting the tail latency behavior of queuing systems and have presented generalized equations for this aim. If the distribution of response time is assumed to be normal, a linear relation between percentiles and the standard deviation of the latency can be derived, which ultimately results in the occurrence of long tails, meaning that the higher the variance, the higher the tail of the latency distribution [18]. Dean and Barroso have a similar claim [2] while introducing the sources of tail latencies in data centers.

As another observation, Stolyar and Ramanan have introduced the *Largest Weighted Delay First* discipline, which is reduced to a simple FIFO scheme when the input entities are of the same weight. They have proved that their algorithm performs optimally by numerically deriving the exponential workload decay rate, ultimately resulting in tail latency mitigation [23]. Egorova [6] provides a deep comparison of FCFS, LCFS and processor sharing scheduling strategies with the same consideration, stating that FCFS will cause the least response tails in an M/D/1 queue. Finally, Li et al. [15] have examined the sources of tail latency behaviors at different levels of operating systems. Using simulation, they have shown that FCFS service discipline outperforms other policies regarding the tail latency. Taking into account the above findings, we can conclude that FCFS discipline will be the best performer concerning long response alleviation.

3. Design

In this section, we describe the CTS design including the CTS policy and its corresponding mechanism.

3.1. CTS policy

The Linux CPU scheduling policies lead to LCFS service discipline of threads, which is proven to aggravate the long tail latency for latency-sensitive multi-threaded applications. As discussed in Section 2.2, FCFS order of serving short-lived threads is the best discipline regarding the tail of the latency distribution. Therefore, the main objective of CTS is to serve threads belonging to the same process in FCFS manner, meaning that the sooner a task is inserted into the run queue (red–black tree), the sooner it gets scheduled compared to the other threads of the process. Another goal of CTS is to maintain the current characteristics of the default Linux scheduler (e.g., fairness) to ensure a fair share of processor resources for all running threads and responsiveness for latency-sensitive applications.

To achieve the mentioned goals, first, we leverage CFS for all imperative functionalities including run-time tracking, red–black tree operations, and group scheduling. We then propose a 3-dimensional scheduling policy. Fig. 2 depicts our scheduling strategy. As it can be seen, CFS is responsible for load balancing in order to distribute tasks among CPUs evenly to utilize the processing capacity (first dimension). At each CPU core, CFS operates in a fully functional mode to choose the next process that one of its threads should run next (second dimension). Finally, once the next process is chosen, a thread scheduler decides which thread of the process must be executed next (third dimension). The main objective of the thread scheduler is to schedule the process's threads in FCFS order.

Therefore, at each core, CFS operates at the inter-process level, picking the next process that one of its tasks should run next, and the CTS thread scheduler operates at the intra-process level, choosing a thread of the process to execute next. Keeping CFS in charge of load balancing and process scheduling ensures the fairness and responsiveness delivered by the default Linux scheduler and executing threads belong to an application in FCFS order by CTS thread scheduler results in lower tail latency for latency-sensitive multi-threaded applications. In the next subsection, we describe the CTS mechanism for realizing the mentioned policies in detail.

3.2. CTS mechanism

CTS thread scheduler operates at intra-process level deciding which thread of a multi-threaded process must be executed next. Efficient tracking of threads belonging to a process is the first requirement of the CTS thread scheduler. To this end, we design a new data structure called Shadow Graph in order to trace threads of a process or recognize a thread's corresponding parent process in a timely manner. Shadow graph is constructed from the red–black tree's nodes (ready tasks). Whenever a task is inserted to or removed from the red–black tree, it is inserted into or removed from the Shadow Graph accordingly. Each node (task) t of the Shadow Graph has a pointer to its parent (the task that forked t) and an ordered list of pointers that point to the task's children (tasks forked by t). The children list is ordered based on the insertion times of tasks, the sooner the insertion the closer to the head of the list. When a new task is inserted into the Shadow Graph, the parent pointer of the task points to its parent, and a new pointer is added to the tail of the children list of the task's parent, which points to the new task. Fig. 3 depicts an example of a Shadow Graph. Using Shadow Graph, CTS is able to access a task's parent, check if the parent is multi-threaded or single-threaded and access the first child of the parent all in $O(1)$ time complexity.

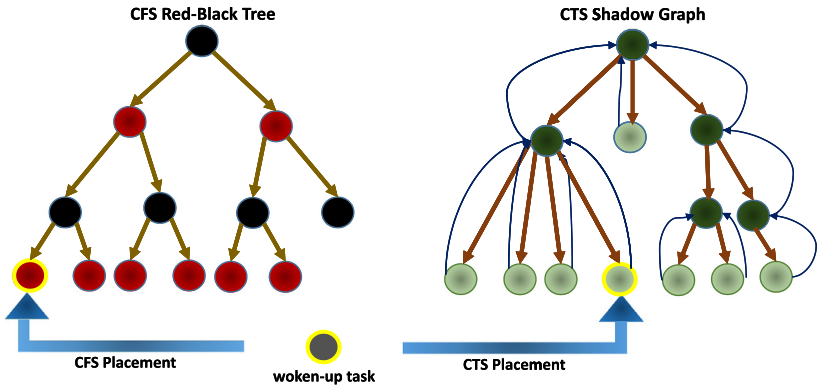


Fig. 3. An example of shadow graph, operating besides a red–black tree.

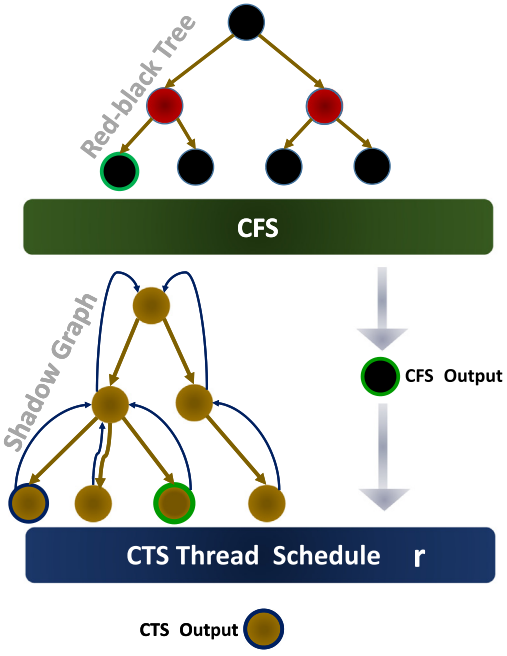


Fig. 4. CTS scheduling mechanism.

Having known threads belonging to a process, the CTS scheduling mechanism is quite straightforward. Fig. 4 depicts the CTS scheduling mechanism at each core. CTS lets the CFS perform in a fully functional mode to choose a task based on the CFS's criteria such as task *vRuntimes*. The output of CFS (the selected task) is passed to the CTS thread scheduler. The thread scheduler first finds the parent process of the chosen task using Shadow Graph and then chooses the first thread of the parent's process to execute next, which could be the CFS's chosen task; ensuring tasks are served in FCFS order. More precisely, at each scheduling decision, CFS finds the task with minimum *vRuntime* value using the red–black tree and passes it to the CTS thread scheduler. Referring the task returned by CFS as *T*, CTS, in turn, acts as follows:

- If *T* belongs to a single-threaded process, CTS simply returns the *T* (chosen task) as the final task that must be executed in the next scheduling period.
- If *T* belongs to a multi-threaded process and also is the first task of the process, CTS simply returns it as the final output.
- If *T* is not the first thread of its corresponding multi-threaded process, CTS substitutes the CFS's returned task with the first task of the process, which is accessible in $O(1)$ time using the Shadow Graph.

Under CTS's scheduling mechanism, threads of multi-threaded processes are served in FCFS order. As mentioned, CTS alters the decision of CFS when the CFS's chosen task is not the first task of its corresponding parent and substitutes the CFS's chosen task with its first sibling. CTS mechanism for substitution is to manipulate tasks *vRuntime* values. When CTS substitutes the CFS's chosen task (*tm*) with its parent's first task (*t1*), it performs the substitution by manipulating the *vRuntime* value of *t1* to be the minimum *vRuntime*. Given that the CFS's chosen task (*tm*) has the minimum *vRuntime*, CTS sets the *vRuntime* value of *t1* to the *vRuntime* value of *tm*.

$$vRuntime_{t1} = vRuntime_{tm} \tag{2}$$

Therefore, CTS's chosen task now has the minimum vRuntime and can be selected to execute next in the next scheduling round. This approach, however, violates fairness by blindly manipulating the vRuntime of process's first tasks. This result in tasks' vRuntime values, do not represent their actual CPU consumptions, misleading CFS that judges tasks based on their vRuntime values, and therefore violating fairness. To enforce fairness, we loan an amount of vRuntime to a task that needs to have the minimum vRuntime due to substitution. The loan is calculated using Eq. (3):

$$Loan = vRuntime_{t1} - vRuntime_{tm} \tag{3}$$

In the next step, we set the vRuntime of *t1* to the minimum vRuntime using Eq. (2). After the execution of *t1*, we take its loan back by updating its vRuntime using Eq. (4):

$$vRuntime_{t1} = vRuntime_{t1} + Loan \tag{4}$$

Therefore, by adopting this approach, we can make sure that the vRuntime values of tasks in the red–black tree always represent their actual processor time consumption.

Algorithm 1 outlines the main steps of CTS scheduling mechanism. At each scheduling decision, the CTS' thread scheduler gets CFS chosen task. It then finds the task's parent in $O(1)$ time using the Shadow Graph. It then checks if the parent is a single-threaded process. If so, it simply returns the CFS's output as the final output to be executed next. If the parent is a multi-threaded process, and the CFS's output is the first thread of the parent, CTS returns the CFS's output as the final decision. Finally, if the parent is a multi-threaded process and the chosen task is not the first task of the parent, the first sibling of the task is chosen to be run next by manipulating its vRuntime value using Eqs. (2)–(4). The time complexity of CTS's thread scheduler is $O(1)$. Therefore, the time complexity of CFS scheduler is still dominant; hence, CTS does not adversely impact the time complexity.

Algorithm 1 CTS scheduling algorithm

Input run-queue information of the current CPU
Output scheduling decision

```
Current_Task ← Current_Core.Current_Task
Current_Task.vRuntime ← Current_Task.vRuntime + Current_Task.Loan ▷ /* Reimburse the vRuntime that was taken out in previous scheduling decision */
Insert Current_Task to red black tree
CFS_Next_Task ← CFS_pick_next_task(Ready_Tasks)
Parent ← SHADOW_GRAPH(CFS_Next_Task)
if IsSingleThreaded(Parent) then
    return CFS_Next_Task
else if IsMultiThreaded(Parent) then
    CTS_Next_Task ← Parent'sFirstChild
end if
if CTS_Next_Task is CFS_Next_Task then
    return CFS_Next_Task
else
    loan ← CTS_Next_Task.vRuntime − CFS_Next_Task.vRuntime
    CTS_Next_Task.vRuntime ← CFS_Next_Task.vRuntime
    return CTS_Next_Task
end if
```

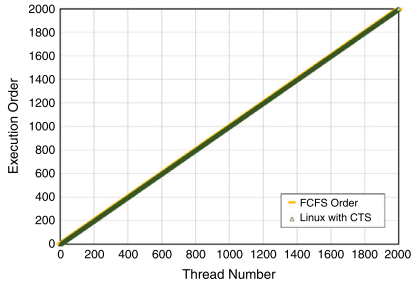


Fig. 5. Execution order of threads under CTS.

Finally, we repeated the experiment presented in Section 2.1 to examine how CTS serves threads of a threaded application. We hosted a thread-driven application (Null RPC server) on a single-processor physical machine whose operating system leveraged CTS as the CPU scheduler. We modified the kernel to show the order of thread execution as described in Section 2.1. Using a client machine, we stressed the Null RPC server by 46 clients. Fig. 5 depicts the order of execution. The x-axis shows the thread number and the y-axis indicates the order of execution. As shown, the output forms a linear graph, indicating threads have completely been served in FCFS manner, resulting in the mitigation of the tail of the latency distribution for latency-sensitive threaded workloads.

3.3. CTS impact on fairness

CFS's main characteristic, namely fairness, is guaranteed by the scheduling of ready tasks in a way that the vRuntime values of all existing tasks in red–black trees remain the same. The effect of CTS on fairness can be investigated at two levels: inter-process and intra-process levels. At inter-process level, the question is whether CTS violates the fairness between two processes, and at the intra-process level, the question is if CTS violates fairness between two threads of a process.

As mentioned before, under CTS, CFS is still responsible for inter-process scheduling, meaning that it determines the next process whose thread must run next. CTS is responsible for intra-process scheduling, meaning that it merely schedules threads of a process that has been chosen by CFS. Therefore, CTS does not violate fairness at inter-process level because CTS does not alter the CFS's chosen task with another task from a different process. In fact, CTS may only alter the CFS's chosen task with another task from the same process. Therefore, CTS does not adversely affect the fairness at inter-process level.

CTS alters the CFS's chosen task with another task in a multi-threaded application when the CFS's chosen task is not the first task of the corresponding process. The question is if CTS policy adversely affects the fairness at intra-process levels, meaning that some threads of a process get a larger portion of processor time in the long term. CTS schedules a process's threads in FCFS manner, which implies that each task of the parent process gets an equal turn of running on CPUs. Since tasks belonging to the same process typically spend an equal time on CPU (e.g., RPC server threads) and this time that is allocated to each thread at each turn is bounded by the time-slice length, it can be said that each task will get an equal share of process time in long-term. Therefore, CTS does not violate fairness at intra-process level.

3.4. CTS effect on responsiveness

Immediate running of newly woken tasks guarantees responsiveness for latency-sensitive tasks that give preference to immediate occupation of a CPU to fairness. The CTS effect on responsiveness can be also investigated at two levels: intra-process and inter-process levels. As mentioned before, CFS still performs inter-process scheduling and CTS never alters the CFS's chosen task with another task from another process. For example, it does not change the keyboard task with a *Memcached* task. Therefore, it does not adversely affect responsiveness at inter-process level.

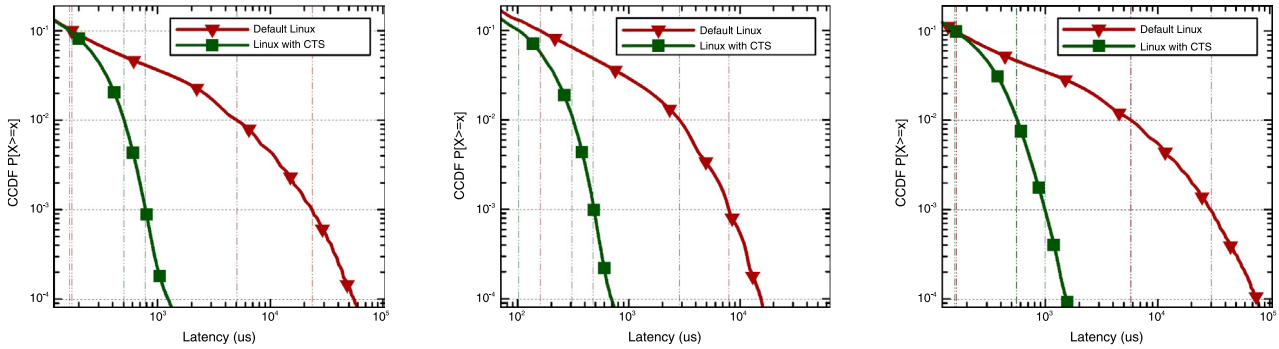
As shown in Section 3.2, at intra-process level, CTS serves tasks in FCFS order. This policy has the same average response time with LCFS and Random serving disciplines for latency-sensitive tasks. However, FCFS leads to less response variability and lower tail latency compared to other approaches, as mentioned in Section 2.1. Therefore, CTS does not adversely affect the responsiveness at intra-process level.

4. Evaluation

In this section, we present our evaluation of CTS prototype implemented in the Linux Kernel for different types of latency-sensitive multi-threaded applications. We evaluate the effectiveness of CTS using both micro-benchmarks and application-level benchmarks. All benchmark applications are thread-driven. We use an RPC socket server as a micro-benchmark and multi-threaded *Thrift* server and *Apache2* Web server as application-level benchmarks. *Apache Thrift* is a thread-driven, scalable and cross-language RPC framework initially developed by Facebook and now employed by other companies such as Google, Twitter, and Pinterest. *Apache2* is an open source platform-independent Web server that is one of the most frequently deployed Web server. It supports different concurrency models including process-driven, thread-driven, and hybrid architectures. Using these widespread adopted applications, we can ascertain that our benchmarks provide a realistic measure of tail latency performance under our proposed CPU scheduler.

4.1. Experimental setup

Our experimental testbed consists of servers with quad-core 3.2 GHz Intel Xeon CPUs and 16 GB of memory. Servers are connected via a Gigabit Ethernet network. All servers use Linux kernel version 4.4 as the operating system. For experiments with CTS, the server machine runs a modified Linux kernel that uses CTS as the CPU scheduler, and for default Linux experiments, server machine runs the vanilla Linux kernel that uses CFS (the default Linux scheduler) as the CPU scheduler.



(a) 600 client connections. (b) 900 client connections. (c) 1200 client connections.

Fig. 6. CCDFs for response times of RPC calls under both CTS and the default Linux scheduler.

4.2. Micro-Benchmarks

To evaluate the effectiveness of CTS for an RPC server, we host an RPC server on a server machine. Using several client machines, we stress the RPC server by RPC requests and measure their Round-Trip-Times (RTTs). To measure the RTTs more precisely, we alter the RPC request and response packets to measure NIC-to-NIC delay. To achieve this, we modified the network interface drivers, appending timestamps into the data segment of RPC packets arriving at the network driver (*T1*). Outgoing RPC packets also get their proper timestamp inserted into their data segment (*T2*). Eventually, we can measure the NIC-to-NIC delay by subtracting *T1* from *T2*. This mechanism excludes all irrelevant delays that might be imposed by the underlying infrastructure like networking delays and client queuing delays, resulting in measurements that are more accurate.

Fig. 6 represents the CCDF (Complementary Cumulative Distribution Function) for the last percentiles of the latency distribution when the RPC server is stressed with 600, 900 and 1200 clients, each sending 500 calls/second forming a Poisson process under both CTS and the default Linux scheduler. According to the results, CTS significantly mitigates the tail latency. For example, it mitigates the latency by up to 96% at 99.9th percentile. The primary reason is that the default Linux scheduling policy leads to LCFS serving manner of RPC threads, meaning the last thread that is woken up due to an incoming RPC request has the highest chance to be executed next. As described in Section 2.2, it has theoretically been proved [15] that LCFS policy exacerbates the tail latency. On the other hand, CTS scheduling policy leads to FCFS service discipline, which has been proved [23] to be the best policy for serving threads in terms of tail latency. In this experiment, CFS chooses the next process to be executed next. If CFS chooses the Null RPC server process, CTS makes sure that RPC server threads are scheduled in FCFS order, leading to lower tail latency as shown in Fig. 6.

4.3. Application-level benchmark

4.3.1. Thrift experiments

To assess the effectiveness of CTS performance for Apache Thrift RPC servers, a server machine hosts a threaded Thrift server application. Using several client machines, we generate RPC workload that forms a Poisson process. At each call, the client machine sends 50 bytes of data and the server responds with 100 bytes of random data. To precisely measure the completion time of each RPC call, we modified thrift request packets to capture NIC-to-NIC delay, following the same procedure as we did for RPC server. We add timestamps to incoming and outgoing packets as they arrive at network interface card. Timestamps can be read by the clients to measure the NIC-to-NIC RTTs.

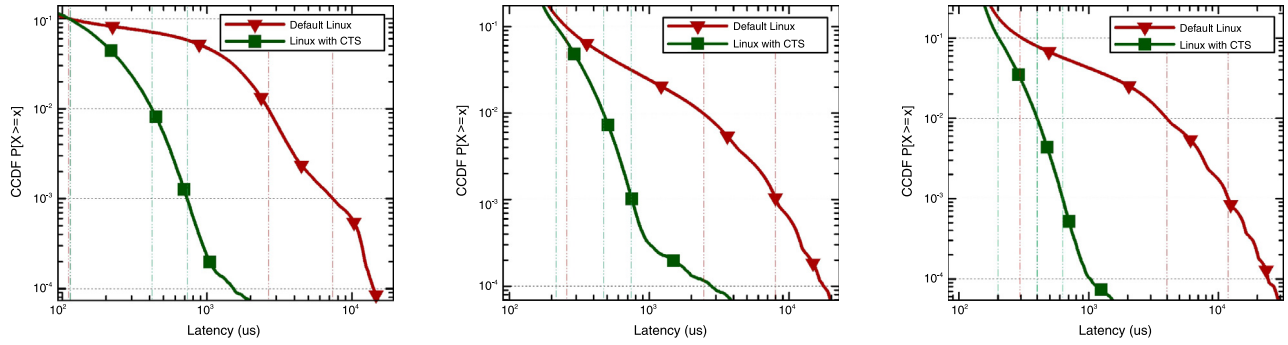
Fig. 7 presents the CCDF of the last percentile of the latency distributions of observed RTTs when the Thrift server is stressed by 500, 600, 700 clients under both CTS and the Linux default scheduler. According to the results, CTS mitigates the tail latency at 99.9th percentile by up to 90%. As mentioned, this is caused by CTS scheduling policy which results is serving Thrift server threads in FCFS discipline while the Linux scheduler policy leads to LCFS serving discipline, lengthening the tail latency.

4.3.2. Apache2 web server experiments

In this experiment, we evaluate the effectiveness of CTS for an Apache Web server as an I/O bound application. A server machine runs Apache Web server and another machine generates requests for a 400B Web page using the well-known *Apache Benchmark* tool, *AB*. We modified the *AB* source to include higher percentiles of the latency distribution in its reports. The CDF (Cumulative Distribution Function) of the last percentile of the latency distribution when the Apache Web server is stressed by 50, 100 and 150 clients are shown in Fig. 8. The results show that CTS consistently reduces the tail latency. For example, at 99.9th percentile, CTS reduces the tail latency from 33 ms to 16 ms, a 51% reduction. The main reason is that CTS serves Apache Server threads in FCFS order, leading to significant mitigation of the tail latency. In fact, as shown in Fig. 8a, CTS trims the tail latency.

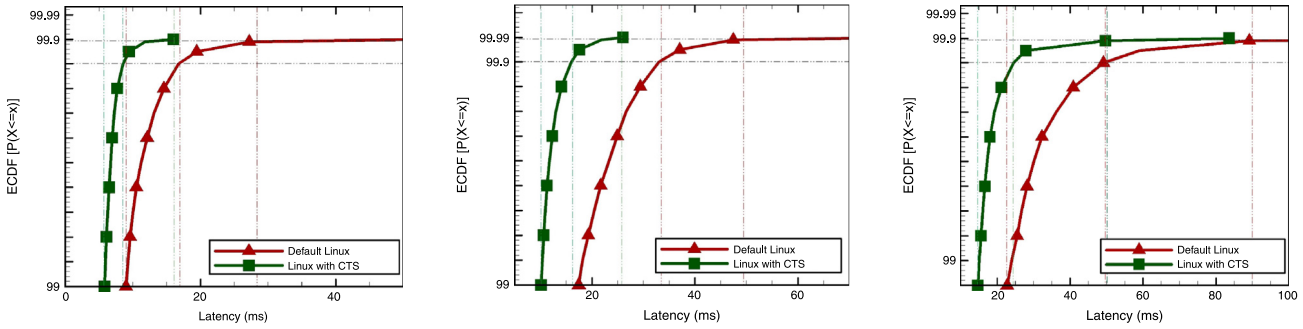
From Fig. 8, we observe that as the number of client connections increases, tail latency lengthens under both schedulers; however, compared to the default Linux scheduler, under CTS, the tail latency is less aggravated when the number of client threads increases. To investigate how increasing the number of threads affects the tail latency, we repeat the Apache Web server experiment. In this experiment, a server hosts an Apache Web server and a client server is used as workload generator. Fig. 9 shows the tail latency at 99th, 99.9th, and 99.99th percentiles when the number of client threads sending requests is increased from 1 to 200 under both schedulers. As the results show, CTS outperforms the default Linux scheduler for each number of client threads at each percentile. Since experimental points seem to follow a line, using the method of least squares, we approximate the line to go through the experimental points for each graph presented in Fig. 9. Table 1 shows the resulting least square lines for each percentile under both CTS and the default Linux scheduler.

As the results reported in the table imply, the slopes of approximated lines under CTS are notably less than the default Linux scheduler. For example, at 99.9th percentile, the slope of the approximated line for default Linux scheduler is 0.38 while under CTS it is 0.16, a 58% reduction. This proves that the performance of CTS is notably less affected when the number of threads sending



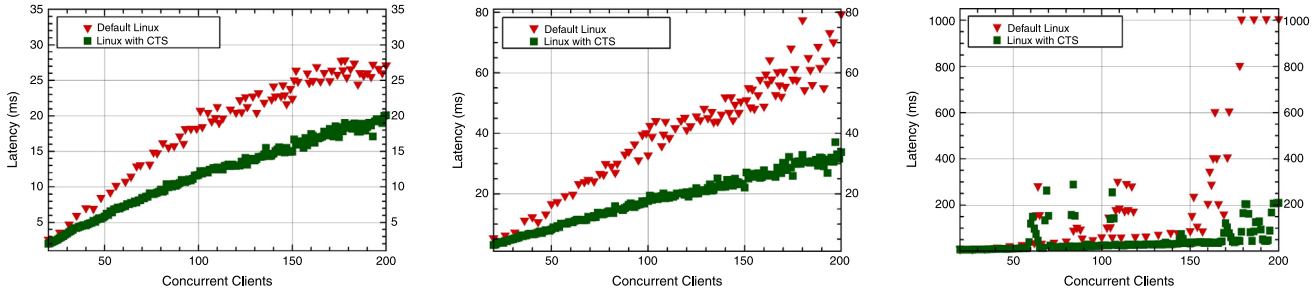
(a) 500 client connections. (b) 600 client connections. (c) 700 client connections.

Fig. 7. CCDFs for response times of thrift requests under both CTS and the default Linux scheduler.



(a) 50 client connections. (b) 100 client connections. (c) 150 client connections.

Fig. 8. CDFs for response times of HTTP requests under both CTS and the default Linux scheduler.



(a) 99th percentile. (b) 99.9th percentile. (c) 99.99th percentile.

Fig. 9. The effect of increases in the number of client connections on tail latency.

requests increases, indicating that CTS is more scalable when the tail latency is a stumbling block to scalability, meaning that under CTS much more threads can be served in the system, which in turn results in higher utilization of physical resources and also higher throughput.

4.4. CTS impact on fairness

As described in Section 3.3, CTS does not adversely affect the system fairness at both inter-process and intra-process levels, meaning that no tasks will get an unfair share of processor resources. Since the default Linux scheduler is still responsible for inter-process scheduling, fairness is not adversely impacted at this level. At intra-process level, CTS is responsible for scheduling, and it is almost the case that CTS alters the CFS chosen task in multi-threaded applications. To examine the impact of CTS on fairness for thread-driven applications, we conduct some experiments. In the first experiment, a server hosts a Null RPC server as a thread-driven application, and three server machines are used to generate workload. We stress the Null RPC server using 200 clients and quantify the time each thread executes on CPUs during the experiment. Fig. 10 depicts the amount of time each thread of the RPC server spends on CPUs. To measure fairness, we calculate the standard deviation of CPU consumption of the running threads. A low standard deviation indicates that the CPU consumption of threads tend to be close to the expected value, resulting in higher fairness. In this experiment, the standard deviation under CTS is 32 and under CFS is 45.

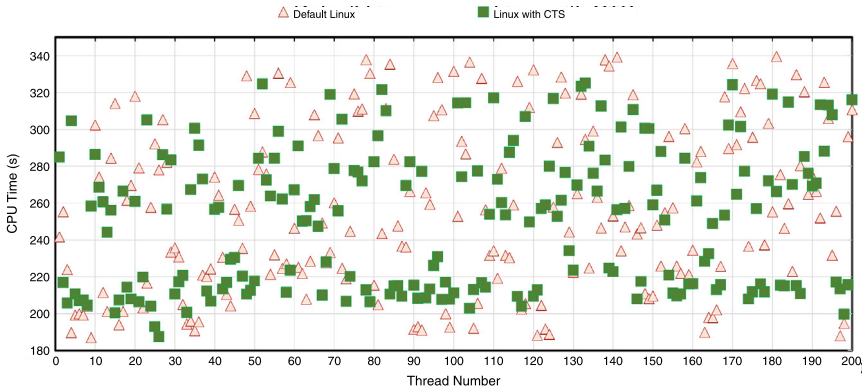


Fig. 10. CPU-time consumption of each thread of a Null RPC server.

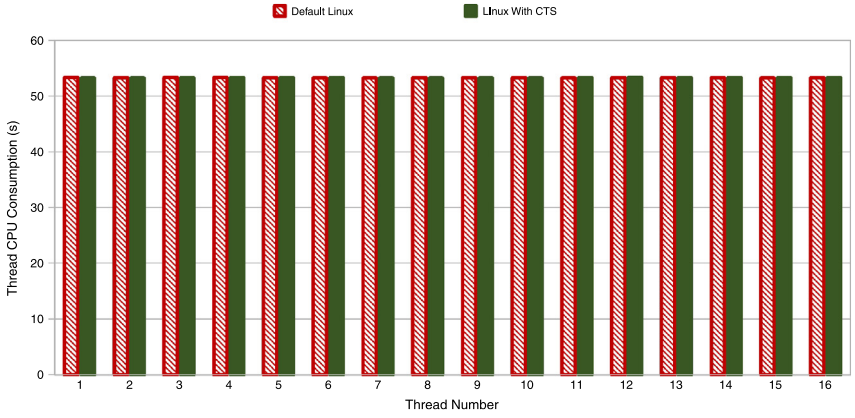


Fig. 11. CPU-time consumption of each thread of a *Sysbench* Benchmark.

Table 1
The regression results for the *Apache2* experiment.

	CTS			Improvement
	Y = aX + B	DoF	Slope	
99	Y = 1.37 + 0.09*X	0.9839	0.09	31%
99.9	Y = 0.64 + 0.16*X	0.9843	0.16	58%
99.99	Y = −0.30 + 0.44*X	0.1678	0.44	90%
	Default Linux scheduler			
	Y = aX + B	DoF	Slope	
99	Y = 2.55 + 0.13*X	0.9432	0.13	
99.9	Y = −3.71 + 0.38*X	0.7658	0.38	
99.99	Y = −278.3 + 4.51*X	0.5313	4.51	

Finally, we assess fairness for thread-driven CPU intensive applications. Unlike latency-sensitive threads, which stay on CPUs shortly and voluntarily relinquish CPUs, CPU intensive threads tend to spend as much time as possible on CPUs to perform their computations until they are finally preempted due to time slice expiration. In this experiment, a server hosts *Sysbench* as a CPU intensive benchmark, which finds the prime numbers between two predefined numbers. We configure *Sysbench* to run with 16 threads. Each thread finds prime numbers between 1 and 100000. Fig. 11 shows the time each thread spends on CPUs to perform its computation during the experiment. CTS results in a standard deviation of 45 and the default scheduler results in standard deviation of 53.

The results demonstrate that CTS does not adversely affect fairness. This is because the FCFS serving discipline enforced by CTS results in equal execution turn of each thread of an application on CPUs. On the other hand, threads belonging to an application typically perform the same job, meaning that they stay on CPUs for the same amount of time. Therefore, all threads belonging to the same application eventually get an equal share of processor resources regardless of the (CPU-intensive or I/O-intensive) type of applications.

It should be noted that fairness in its traditional meaning; an equal share of processor resources; is not always the main concern for latency-sensitive, thread-driven applications whose threads typically handle each I/O request in a very short time (less than 1 ms) and then relinquish the CPU to sleep for the next incoming I/O request. The primary concern of these applications is to get CPUs as promptly as possible whenever an interrupt is raised due to an incoming I/O request because their CPU access delays determine their responsiveness. Therefore, to treat latency-sensitive tasks fairly, the scheduler should guarantee the same CPU access latency for each latency-sensitive thread. Therefore, based on the requirements of the applications and their service level objectives, the meaning of fairness can vary. In other words, fairness could be defined relative to the goals of a system. As an example for CPU intensive workloads, getting an equal fraction of CPU times establishes fairness, and for latency-sensitive applications, ensuring an equal CPU access latency establishes fairness. Fig. 12 shows the average CPU access latency (waiting time) of each running thread during the RPC server experiment. From Fig. 12, we observe that under the default Linux scheduler, the waiting times of RPC threads severely fluctuate while under CTS, the waiting times of threads tend to be close to each other. Therefore, it can be said that CTS is a fairer scheduler because it ensures that running threads get not only the same share of CPU time but also the same waiting delay to get CPU access.

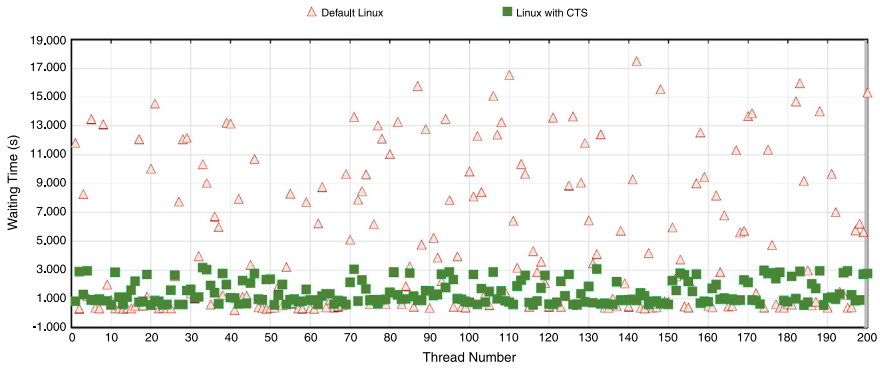


Fig. 12. Waiting time of each thread of a Null RPC server.

5. Related work

The choice between thread or event concurrency models for latency-sensitive applications have been investigated in the recent years [34]. The event-driven concurrency model offers high scalability, low resource utilization and better synchronization support [7,28]. On the other hand, Thread-driven systems provide an easier programming and debugging experience as they delegate scheduling and resource management to the underlying system. Von Behren et al. [26] believe that the inefficiencies related to thread-driven applications are mostly due to poor implementations rather than the threading paradigm, and both concurrency models offer the same performance. When it comes to tail latency, event-driven applications usually leverage an application-level dispatcher that can be tuned to serve events in FCFS order to mitigate tail latency while thread-driven applications rely on the CPU scheduler for ordering thread executions. The default Linux scheduling policies lead to LCFS order of execution of threads, lengthening the tail latency. The CPU scheduler of CTS ensures that threads of a latency-sensitive application will be served in FCFS order, leading to a lower tail latency and making operating systems' CPU schedulers more supportive of thread-driven latency-sensitive applications.

A large body of work is devoted to identifying and masking the long responses in the middleware solutions. Some researchers [8,12,21,24,27] have proposed request replication and middleware-level scheduling techniques to reduce the impact of long tails. A similar group of works targets energy-awareness and efficiency in warehouse-scale infrastructures while ensuring the tail latency defined in SLAs. Tarcil [4] and Eagle [3] introduce job-scheduling mechanisms to meet the predefined latency targets while keeping the resource utilization high. Middleware solutions have concentrated on dealing with tail latency sources without modifying the underlying system software. CTS proposes a workaround that scopes to individual server operating systems to trim tail latency for thread-driven, latency-sensitive applications. CTS deployment diminishes the need for middleware-level solutions. However, leveraging higher-level techniques in combination with CTS kernels will further improve the quality of delivered service regarding the tail latency.

Masking the tail latency in higher levels can be costly and inefficient; therefore, it will be beneficial to deal with the sources directly within the operating system Li et al. [15] have identified some sources of long responses and have proposed a few suggestions mostly by reconciling the existing configuration. They have identified background processes, interrupt processing mechanisms, NUMA unaware software designs, power saving mechanisms and tail unaware scheduling policies as the main contributors of long tail latency in individual servers. CTS makes the CPU scheduler aware of tail latency by serving sibling threads in FIFO order, making operating systems more supportive threaded applications.

Leverich et al. [4] have investigated the possibility of co-locating latency-critical workloads with other workloads to enhance the utilization of physical resources while guaranteeing QoS for latency-critical jobs. To this end, they find the default Linux scheduler unable to provide good latency guarantees and suggest replacing CFS with the BVT scheduler.

There are some other process schedulers like [14]. They lack the generality properties and do not target the tail latency problem [5]. We believe that the default scheduler that is proven to provide fairness and responsiveness simultaneously cannot be shelved easily by a brand new scheduling mechanism. Therefore, instead of replacing the main CPU scheduler of the Linux kernel, CTS tries to complement CFS by enforcing FIFO serving order for the sibling threads. The combination of the Linux default scheduler and CTS ensures a lower tail latency for thread-driven, latency-sensitive applications, while maintaining the Linux's scheduler characteristics. Some approaches [20,31] separate management and scheduling functions of the kernel from network processing so as to reduce latency. Using this approach, Applications have direct access to virtualized I/O devices, allowing most I/O operations to skip the kernel. IX [1], for example, achieves desired latency performance by eliminating the overheads of Linux system calls. However, the accomplishment of such an endeavor will be very costly due to radical changes and the building of a new operating system architecture and APIs [16]. CTS enhances the performance of the system regarding tail latency, and this improvement does not come at the cost of losing any generality.

6. Conclusion

Responsiveness is one of the main concerns of large-scale interactive Web applications. Amazon has found that every 100 ms of latency costs them 1% in sales. Google also found that an extra 0.5 s in search page generation time drops the traffic by 20% [33]. Large-scale interactive services leverage parallelization to fan out the sub-requests across a large number of individual servers. The main request does not complete until the slowest sub-request is fulfilled. Therefore, the tail latency of individual server dominates the responsiveness of the service. Individual servers typically host either thread-driven or event-driven latency-sensitive applications. Studies have shown that serving I/O tasks in FCFS manner leads to the least tail latency and response time variability. Event-driven applications can be tuned to serve tasks in FCFS order at the application level; whereas, thread-driven applications rely on operating system for ordering the thread execution. The experiments have demonstrated that the default CPU scheduler of Linux serves threads in LCFS order, making current scheduler unsupportive of

multi-threaded latency-sensitive applications regarding the tail latency.

CTS is an operating system CPU scheduler that trims the tail of the latency distribution for multi-threaded latency-sensitive applications by running applications' threads in FCFS manner, leading to significant tail latency mitigation. CTS leverages the default Linux scheduler for load balancing at the system level and process scheduling at core level. A thread scheduler finally selects a thread of the chosen process to be executed next, while making sure that threads are served in FCFS manner. The key character of the Linux scheduler, fairness, is guaranteed by letting the CFS operate in fully functional mode to schedule processes based on the CPU consumption of their threads.

Our evaluation of the CTS prototype implemented in the Linux kernel shows that it vastly outperforms the Linux default scheduler. For example, it mitigates the tail latency of Thrift server by up to 90% at 99.9th percentile. This significantly enhances the responsiveness of interactive Web services that run Thrift in individual servers. Furthermore, our experiments show that increasing the number of threads exacerbate the tail latency under both schedulers. However, the tail performance is notably less affected when the number of threads increases. This indicates that CTS is more scalable when tail latency matters, meaning that more threads can be served under CTS, which in turn results in a higher utilization of servers in data centers.

References

[1] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, E. Bugnion, The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane, ACM Trans. Comput. Syst. 34 (4) (2016) 11:1–11:39.

[2] J. Dean, L.A. Barroso, The tail at scale, Commun. ACM 56 (2) (2013) 74–80.

[3] P. Delgado, D. Didona, F. Dinu, W. Zwaenepoel, Job-aware scheduling in eagle: Divide and stick to your probes, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, ACM, 2016, pp. 497–509.

[4] C. Delimitrou, D. Sanchez, C. Kozyrakis, Tarcil: reconciling scheduling speed and quality in large shared clusters, in: Proceedings of the Sixth ACM Symposium on Cloud Computing, ACM, 2015, pp. 97–110.

[5] A.B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, P.F. Sweeney, Why you should care about quantile regression, in: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems, vol. 48, ACM, 2013, pp. 207–218.

[6] R. Egorova, Sojourn time tails in processor-sharing systems (Masters thesis), Technische Universiteit Eindhoven, Eindhoven, 2009.

[7] K. Elmeleegy, A. Chanda, A.L. Cox, W. Zwaenepoel, Lazy asynchronous I/O for event-driven servers, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, USENIX Association, 2004, p. 21.

[8] M.E. Haque, Y.H. Eom, Y. He, S. Elnikety, R. Bianchini, K.S. McKinley, Few-to-many: Incremental parallelism for reducing tail latency in interactive services, SIGPLAN Not. 50 (4) (2015) 161–175.

[9] M. Harchol-Balter, J.J. Cochran, L.A. Cox, P. Keskinocak, J.P. Kharoufeh, J.C. Smith, Queueing Disciplines, Wiley Encyclopedia of Operations Research and Management Science, 2010.

[10] R. Jain, The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling, John Wiley & Sons, 1990.

[11] M. Kim, S. Noh, J. Hyeon, S. Hong, Fair-share scheduling in single-ISA asymmetric multicore architecture via scaled virtual runtime and load redistribution, J. Parallel Distrib. Comput. (2017).

[12] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, S. Choi, Delayed-dynamic-selective (DDS) prediction for reducing extreme tail latency in web search, in: Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, WSDM '15, ACM, New York, NY, USA, 2015, pp. 7–16.

[13] L. Kleinrock, Queueing Systems, Volume II: Computer Applications, Wiley Interscience, 1976.

[14] C. Kolivas, BFS Linux Process Scheduler FAQ. BFS Linux Process Scheduler. http://ck.kolivas.org/patches/bfs/bfs-faq.txt.

[15] J. Li, N.K. Sharma, D.R.K. Ports, S.D. Gribble, Tales of the tail: Hardware, OS, and application-level sources of tail latency, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, ACM, New York, NY, USA, 2014, pp. 9:1–9:14.

[16] F.P. Lilkaer, Enhancing Quality of Service metrics for high fan-in Node.js applications by optimising the network stack : Leveraging IX: The Dataplane Operating System (Masters thesis), EPFL, 2015.

[17] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, A. Fedorova, The Linux scheduler: a decade of wasted cores, in: Proceedings of the Eleventh European Conference on Computer Systems, ACM, 2016, p. 1.

[18] J. Martin, Systems Analysis for Data Transmission, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1978.

[19] M. Nguyen, Z. Li, F. Duan, H. Che, H. Jiang, The Tail at Scale: How to Predict it? HotCloud, 2016, usenix.org.

[20] S. Peter, J. Li, I. Zhang, D.R.K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, T. Roscoe, Arrakis: The operating system is the control plane, ACM Trans. Comput. Syst. 33 (4) (2015) 11:1–11:30.

[21] W. Reda, L. Suresh, M. Canini, S. Braithwaite, BRB: BetteR batch scheduling to reduce tail latencies in cloud data stores, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, vol. 45, ACM, 2015, pp. 607–608.

[22] J.C. Saez, A. Pousa, F. Castro, D. Chaver, M. Prieto-Matias, Towards completely fair scheduling on asymmetric single-ISA multicore processors, J. Parallel Distrib. Comput. 102 (C) (2017) 115–131.

[23] A.L. Stolyar, K. Ramanan, Largest weighted delay first scheduling: large deviations and optimality, Ann. Appl. Probab. 11 (1) (2001) 1–48.

[24] P.L. Suresh, M. Canini, S. Schmid, A. Feldmann, C3: Cutting tail latency in cloud data stores via adaptive replica selection, NSDI, 2015, pp. 513–527, usenix.org.

[25] P. Turner, B.B. Rao, N. Rao, CPU bandwidth control for CFS, in: Linux Symposium, vol. 10, 2010, 245–254.

[26] R. von Behren, J. Condit, E. Brewer, Why events are a bad idea (for high-concurrency servers), in: Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9, HOTOS'03, USENIX Association, Berkeley, CA, USA, 2003, p. 4.

[27] A. Vulimiri, O. Michel, P.B. Godfrey, S. Shenker, More is less: Reducing latency via redundancy, in: Proceedings of the 11th ACM Workshop on Hot Topics in Networks, HotNets-XI, ACM, New York, NY, USA, 2012, pp. 13–18.

[28] M. Welsh, The Staged Event-Driven Architecture for Highly-Concurrent Server Applications, 2000.

[29] A. Wierman, Scheduling for Today's Computer Systems: Bridging Theory and Practice, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.

[30] Y. Xu, Z. Musgrave, B. Noble, M. Bailey, Bobtail: Avoiding Long Tails in the Cloud, in: NSDI, vol. 13, 2013, pp. 329–342.

[31] K. Yasukata, M. Honda, D. Santry, L. Eggert, StackMap: Low-latency networking with the OS stack and dedicated nics, in: USENIX Annual Technical Conference, 2016, pp. 43–56, usenix.org.

[32] Y. Zhang, D. Meisner, J. Mars, et al., Treadmill: Attributing the source of tail latency through precise load testing and statistical inference, Architecture (ISCA) (2016) 2016.

[33] T. Zhu, J. Li, J. Kimball, J. Park, C.A. Lai, C. Pu, Q. Wang, Limitations of load balancing mechanisms for N-Tier systems in the presence of millibottlenecks, in: 2017 IEEE 37th International Conference on Distributed Computing Systems, ICDCS, 2017, pp. 1367–1377.

[34] Y. Zhu, D. Richins, M. Halpern, V.J. Reddi, Microarchitectural implications of event-driven server-side web applications, in: Proceedings of the 48th International Symposium on Microarchitecture, MICRO-48, ACM, New York, NY, USA, 2015, pp. 762–774.

Esmail Asyabi is currently a researcher in Computer Science Department at Boston University. His research focuses on designing and building system software (e.g., OS kernels, hypervisors, device drivers) to (1) mitigate energy consumption of computer systems, ranging from mobile devices to data center servers, (2) enhance the performance of computer systems for specific applications or workload types (e.g., IO intensive workloads) and (3) enable efficient system support for emerging computing models (e.g., cloud computing). Currently, his research focuses on providing efficient system support for virtualized clouds. He is mainly interested in distributed systems, operating systems and cloud computing. He is also a Ph.D. student in software engineering in the School of Computer Engineering of Iran University of Science and Technology, where he is a research assistant at distributed systems research laboratory.

Erfan Sharafzadeh received his honors B.S. degree from Iran University of Science and Technology at 2016 and is currently pursuing his Master's degree in computer software engineering at the same institution. His major research interests include Operating Systems, Performance Evaluation and Queuing Systems.

SeyedAlireza SanaeeKohroudi received his B.S. degree in computer science from (IUST) Iran University of Science in 2016 and he is studying M.Sc. of computer science at IUST. He is currently involved in Smart Grid project at École polytechnique fédérale de Lausanne. The research area he is following centers around computer systems and networked systems.

Mohsen Sharifi is a professor of software engineering in the School of Computer Engineering of Iran University of Science and Technology. He directs a distributed systems research group and laboratory. He is mainly interested in the engineering of distributed systems, solutions, and applications, particularly for use in various fields of science. The development of a true distributed operating system is on top of his wish list. He received his B.Sc., M.Sc., and Ph.D. in computer science from Victoria University, Manchester, UK, in 1982, 1986, and 1990, respectively. His website is http://webpages.iust.ac.ir/msharifi.