Multithreading in Desktop Applications
A Characterization Study and Concurrency Bug Mitigation Strategy

Blake Jackson and Ben Ylvisaker

***Abstract*—We investigate thread-level concurrency in several common desktop applications. We find that the majority of active periods (periods of uninterrupted CPU activity for a single thread) are relatively short, while the few long active periods account for most of the active time. The shortest 90% of the active periods only account for roughly 12.75% of total active time. We speculate that this is generally true for most applications, and that there might be some way to take advantage of this fact in the scheduler. Due to the difficulties in catching, testing, and fixing concurrency bugs, we propose modifying the thread scheduler to reduce the risk of concurrency bugs where possible. Our simulations show that our modification may work well for certain applications depending on the level of CPU demand.**

***Keywords*—threads, concurrency, thread scheduler, parallelism, multithreading.**

I. Introduction

In the software industry, there is now more demand for highly interactive and responsive applications than ever before. There is also more multicore hardware in the hands of the typical consumer than ever before. One of the most popular tools for taking advantage of this multicore hardware and providing this interactivity is multithreading. This has made thread-level concurrency common in modern applications. Unfortunately, designing, implementing, and testing asynchronous concurrent applications is difficult [3]. This difficulty stems largely from the notorious non-determinism in the order of execution of multithreaded code which causes problems in detecting and reproducing concurrency bugs. Thus, it is challenging and onerous for programmers to isolate and fix concurrency related issues in their applications.

Much of the concurrency within today's software is intentional and hugely beneficial to performance. However, there also exists physical simultaneity between active periods that occurs incidentally without any real benefits in terms of speed, especially in applications that sre not particularly CPU intensive. We will term this type of unnecessary physical simultaneity "incidental concurrency". To mitigate the risk of concurrency bugs, we propose that any reduction in incidental concurrency would be beneficial to typical desktop applications, at least until tools for concurrency bug detection and remediation and concurrent programming language design have caught up to the demand for multithreaded software.

We begin with a characterization study of active periods for four common desktop applications. Our results show that, in general, the overwhelming majority of active periods for the typical application are relatively short, while the few long active periods account for most of the application's active time.

Furthermore, perhaps due to the aforementioned inherent difficulties that plague parallel programming, processor resources in modern multi-core computers and phones are seriously underutilized in many typical use cases [1], [2]. This result, along with the relatively short durations of most active periods, leads us to believe that there is room to reduce incidental concurrency with minimal detriment to application performance and user experience by spreading out simultaneously running active periods into what would otherwise be idle time. Therefore, we propose modifying the scheduler to reduce incidental concurrency while influencing application responsiveness and runtime as little as possible as shown in figure 1. Our simulations indicate that our methods for accomplishing this may be viable and merit further investigation.

This is desirable because physical simultaneity of threads that share memory increases the risk of concurrency bugs. For example, if two threads simultaneously read, increment by 1, and write the same integer in memory, then the integer will only reflect an increase of 1 when it should have been increased by 2. These types of issues are currently dealt with by having threads acquire locks, but, as shown by Lu et al, existing solutions are not adequate [3]. It is not a reasonable goal to eliminate all concurrency, nor is it what we are suggesting with this study. Concurrency is crucial to performance. We would like to reduce incidental overlaps between short active periods, ideally while leaving the relatively few long active periods and intentionally simultaneous computations unaltered.

II. Background

Threads represent the smallest sequence of processor instructions that can be managed independently by an operating system's scheduler. Multiple threads can exist for any given application, and they share resources such as memory. Multithreading is when multiple threads execute concurrently both through physical simultaneity on multicore hardware and by quickly switching between threads on single CPU cores. Such context switching generally happens often enough to give the appearance that the threads are executing in parallel. This multithreading allows reactivity and interactivity in applications since some threads can handle user input, while others handle disk reads and writes, while still others perform some computation, etc. On multi-core processing hardware, several threads (that may share memory resources) can truly run in parallel. This is beneficial in terms of speed, but can give rise to bugs and errors not possible in sequentially executing code (concurrency bugs). Note too that

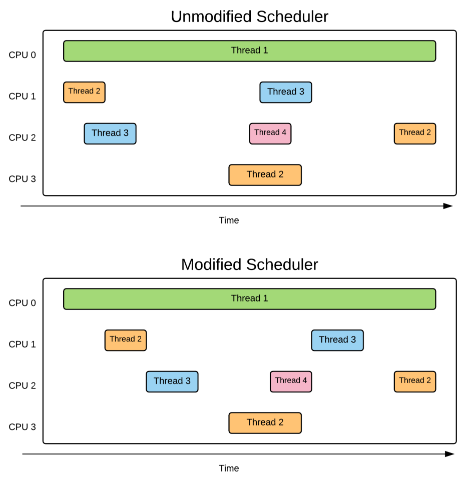


Fig. 1. Illustration of proposed scheduler modification

rapid context switching also occurs on multi-core systems, so there are non-deterministic complexities in the order of thread execution not easily foreseen by programmers. This non-determinism comes from factors beyond an application's control like disk read/write speed, network latency, CPU core temperatures, and user actions.

In any normal personal computing system, each processor core will have active periods, when it is actively performing computations for some thread, and idle periods, when it is waiting for a task from the scheduler. At the most basic level, the thread scheduler simply allocates jobs to the CPU cores to accomplish the necessary computations as quickly as possible. This is done with some sort of priority queue of threads waiting to run. The simplest way would be to use a first-in-first-out (FIFO) queue so that active periods would run in the order that they became ready to run. However, the reality is that scheduling is much more complicated in modern systems. Typically, it will involve a priority queue where each queuing thread has a priority. These priorities can dynamically change in response to changes in the environment while the threads are queuing to maximize interactivity. Once an active period is running, it will continue to run until it finishes, yields, becomes unable to continue, or a higher priority thread becomes available in the queue. Schedulers also consider CPU core temperature and power consumption in their decisions, so an active period may be stopped on one processor core only to immediately resume on another core. In order to attain the absolute best performance from the available hardware, modern schedulers have become extremely complex.

III. Motivation for Characterization Study

Recent years have seen a spike in the demand for highly interactive software. Common applications increasingly rely on network communication, and users simultaneously expect speedy responses to numerous sources of input. This has resulted in an industry-wide push for higher levels of concurrency. The increasing amount of processor cores in personal computing hardware provides further impetus to write heavily concurrent software. Today, threads are a very popular tool to take advantage of multi-processor hardware and provide interactivity through concurrency. Threads are prevalent in many applications, both desktop and mobile, across operating systems. Therefore, it is critically important to understand how they typically behave. This understanding could help application developers to better utilize threads, and help operating systems and hardware engineers to better support multi-threading in their products. Alternatively, a greater understanding of thread behaviour could, in time, show that threads are not really the best concurrency option, thus shifting the industry paradigm of parallel programming. To our knowledge, the information collected and analyzed in the characterization portion of this study is not published elsewhere, and we hope to add it to the academic knowledge base of computer science.

IV. Motivation for Simulating Scheduler Tweaks

Concurrent programs are becoming more and more prevalent both to take better advantage of now ubiquitous multi-core hardware and to create responsive interactive applications. As a result, writing good quality concurrent programs has become critically important. Unfortunately, this is quite challenging, and concurrency bugs are present (if not rampant), even in high profile applications (e.g. MySQL, Mozilla Firefox, OpenOffice). To make matters worse, the non-determinism in the order of execution of concurrent programs makes concurrency bugs notoriously difficult to detect and reproduce during testing [3]. Therefore, we propose that any reduction in concurrency at minimal cost to performance would be advantageous, at least until concurrent programming language design and tools for concurrency bug detection and remediation have caught up to multi-core hardware.

Previous research has shown that processing resources are generally severely underutilized in multi-core computers and phones. [1], [2]. Gao et al studied 20 popular mobile applications and found that all display some thread level parallelism, but that none took full advantage of all available processing resources. They found that ”mobile applications are utilizing less than 2 cores on average.” Blake et al found a similar result for common desktop applications with the notable exception of video authoring software which made better use of all processor cores. They found that ”2-3 cores are more than adequate for most applications.” This surplus of processing resources indicates that we may be able to reduce concurrency in thread execution without too much detriment to responsiveness or runtime. This would not be the case if applications displayed thread level parallelism that took full advantage of all cores at all times. Furthermore, interactive applications often create bursts of CPU activity punctuated by long periods of idle time. Forcing some activity into these idle periods instead of allowing it to run in parallel would reduce concurrency without hurting the overall runtime of the application. Thus, responsiveness becomes the main concern in that situation.

We propose modifying the thread scheduler such that, when an active period starts on one CPU core, it briefly blocks other active periods from the same application from starting on the other cores. This block would persist until either the blocking active period ends or yields, or it runs for longer than the maximum block time. Because most active periods tend to be short, we hope that many of them can finish before the maximum block time elapses, thus allowing the next queueing active period to start without the possibility of creating some concurrency issues with the first. Figure 1 provides a simple graphic explanation of this concept. We also propose the possibility of keeping track of threads that had long running active periods in the recent past, and not blocking when they become active. This would decrease the time cost of our scheduler modification, and blocking in those cases should not reduce concurrency anyway because the duration of the active periods of the historically long running threads would be expected to exceed the maximum block time. We evaluate the efficacy of these ideas in the ”Results” section.

Note that there is some precedent in industry for modifying the scheduler to sacrifice a bit of speed or responsiveness to improve some other aspect of the system. For example, Apple’s OS X Mavericks uses timer coalescing when the computer is running on battery power to increase the idle time of the CPU. This decreases power usage thereby improving battery life as shown in figures 2 and 3. Also, their new scheduler attempts to use as few CPU cores as possible, leaving as many as possible totally idle as long as demand allows [4].

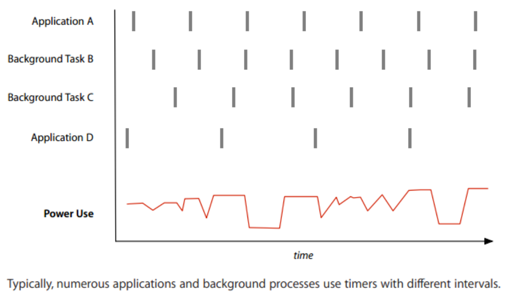

Application A
Background Task B
Background Task C
Application D
Power Use
time
Typically, numerous applications and background processes use timers with different intervals.

Fig. 2. Power use in OS X without timer coalescing

V. Methodology

We collected system wide traces of thread context switches using trace-cmd, which is a user-space front-end command-line tool for Linux’s ftrace. Ftrace is a tracing framework included with the Linux kernel. We used the Ubuntu operating system running on a virtual machine on a Windows 7 host. The virtual machine had access to all 4 cores of the host machine’s processor and 4 gigabytes of RAM. We collected data on Mozilla Firefox, LibreOffice Calc, LibreOffice Writer, and Blender. The details and versions of these aspects of the data collection are shown in Table I.

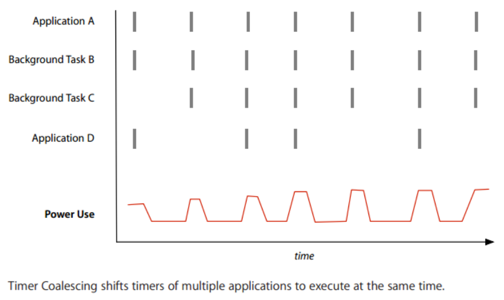

Application A
Background Task B
Background Task C
Application D
Power Use
time
Timer Coalescing shifts timers of multiple applications to execute at the same time.

Fig. 3. Power use in OS X with timer coalescing

Component	Version Details
trace-cmd	2.3.1
Ubuntu	14.04.3 LTS 64-bit
Kernel	3.13.0-76-generic
LXDE	0.5.0-4ubuntu4
Processor	Intel i7-2640M CPU @ 2.80GHz. Driver version 6.1.7600.16385
Virtual Box	5.0.2 r102096
Windows 7 (host)	Windows 7 Professional Service Pack 1
Firefox	43.0.4
LibreOffice	4.2.8.2 420m0(Build:2)
Blender	2.76b

TABLE I. Components and Version Details

We selected LibreOffice Calc, LibreOffice Writer, and Mozilla Firefox because of their large user bases and because the functionality that they provide is widely considered a necessity on modern personal computers. We chose to study Blender because we expected it to display a higher level of concurrency than the other applications. The testing actions for each of these applications last for about 60 seconds. We found that this is enough time to cover most typical functions for the LibreOffice applications. The tests on Firefox simply involved streaming a 1 minute long HD video from YouTube. The tests on Blender involved rendering an image of a three dimensional scene using the Cycles ray tracing render engine. All experiments were repeated 5 times, and we observe very little change between trials.

Determining which threads are doing work for a particular application is not straightforward. Oftentimes, applications will take advantage of preexisting operating system threads to perform some of their computations. Because this study is primarily concerned with threads belonging to the same application, we used the Lightweight X11 Desktop Environment (LXDE) to minimize the load on the CPU from sources other than the applications being studied. Nonetheless, some of the active periods in our data do come from other sources like the operating system and trace-cmd. Tables 2-9 below indicate that, despite the presence of some unrelated threads, the majority of the active periods and active time during the experiments come from the applications being tested. The tables were each generated from a single experiment with each application, but there is very little difference between trials in the top ten threads. There are many more threads present in each experiment that are too insignificant to appear on the tables.

Thread Name	Total Active Time (s)	Cumulative Percentage of Active Time
soffice.bin	7.36	62.03
Xorg	3.29	89.76
gdbus	0.30	92.32
lxpanel	0.18	93.87
trace-cmd	0.14	95.04
pcmanfm	0.09	95.76
ibus-daemon	0.08	96.44
openbox	0.07	97.04
kworker/u8:0	0.03	97.33
ibus-ui-gtk3	0.03	97.61

TABLE II. ACTIVE TIME BY THREAD IN LIBREOFFICE CALC

Thread Name	Total Active Periods	Cumulative Percentage of Active Periods
Xorg	11035	30.87
soffice.bin	10840	61.19
gdbus	3282	70.37
trace-cmd	2175	76.45
ibus-daemon	1624	81.00
openbox	1213	84.39
rcu_sched	1013	87.22
lxpanel	941	89.85
ibus-ui-gtk3	349	90.83
pcmanfm	330	91.75

TABLE III. ACTIVE PERIODS BY THREAD IN LIBREOFFICE CALC

Thread Name	Total Active Time (s)	Cumulative Percentage of Active Time
HGCM-NOTIFY	40.157501	79.74
soffice.bin	6.503134	92.65
Xorg	2.078598	96.78
trace-cmd	0.436562	97.65
gdbus	0.379625	98.40
lxpanel	0.160768	98.72
openbox	0.118533	98.95
ibus-daemon	0.102665	99.16
ibus-ui-gtk3	0.045302	99.25
xscreensaver	0.035916	99.32

TABLE IV. ACTIVE TIME BY THREAD IN LIBREOFFICE WRITER

Thread Name	Total Active Periods	Cumulative Percentage of Active Periods
HGCM-NOTIFY	184008	85.33
soffice.bin	9091	89.55
Xorg	8971	93.71
gdbus	3289	95.23
trace-cmd	1944	96.13
ibus-daemon	1610	96.88
openbox	1279	97.47
rcu_sched	1009	97.94
lxpanel	769	98.30
xscreensaver	296	98.43

TABLE V. ACTIVE PERIODS BY THREAD IN LIBREOFFICE WRITER

Note the appearance of the thread ”HGCM-NOTIFY” at the top of the tables for LibreOffice Writer data. This thread is associated with Virtual Box, which gives ample reason for skepticism regarding its pertinence to Writer. However, it is at the very top of the active time and active periods lists for all 5 Writer experiments, and it is not present for any other application studied. This indicates that it is indeed performing work specifically for writer. Furthermore, we repeated the Writer experiments on a machine running Linux Mint with 2 cores. On that machine, the thread ”Cinnamon” was consistently second on these lists. We suspect that ”Cinnamon” was doing the same work there that ”HGCM-NOTIFY” was doing in the virtual machine, and that the Virtual Box software does not drastically influence the behaviour of Writer.

Thread Name	Total Active Time (s)	Cumulative Percentage of Active Time
Xorg	55.097729	42.53
MediaPl∼back_#6	26.871962	63.28
Compositor	17.52024	76.80
firefox	10.696766	85.06
MediaPD∼oder_#2	2.308864	86.84
alsa-sink-Intel	2.282829	88.60
MediaPD∼oder_#3	2.269279	90.36
MediaPD∼oder_#1	2.196414	92.05
ImageBridgeChil	1.913534	93.53
Socket_Thread	1.677432	94.82

TABLE VI. ACTIVE TIME BY THREAD IN MOZILLA FIREFOX

Thread Name	Total Active Periods	Cumulative Percentage of Active Periods
Compositor	110439	29.50
Xorg	104233	57.35
MediaPl∼back_#6	18894	62.40
firefox	16090	66.70
ImageBridgeChil	9919	69.35
MediaPl∼back_#1	9309	71.83
MediaPl∼back_#4	9292	74.32
MediaPl∼back_#5	9249	76.79
MediaPl∼back_#2	9180	79.24
MediaPl∼back_#3	9160	81.69

TABLE VII. ACTIVE PERIODS BY THREAD IN MOZILLA FIREFOX

Thread Name	Total Active Time (s)	Cumulative Percentage of Active Time
blender	203.98057	97.08
Xorg	2.496559998	98.27
alsa-sink-Intel	2.216301998	99.33
pulseaudio	0.391725001	99.51
threaded-ml	0.274172	99.65
trace-cmd	0.22989	99.75
kworker/u8:2	0.143979	99.82
kswapd0	0.100234	99.87
lxpanel	0.073227	99.91
lxterminal	0.036065	99.92

TABLE VIII. ACTIVE TIME BY THREAD IN BLENDER

Thread Name	Total Active Periods	Cumulative Percentage of Active Periods
blender	153918	73.77
Xorg	28113	87.24
alsa-sink-Intel	7809	90.99
threaded-ml	7295	94.48
pulseaudio	6985	97.83
trace-cmd	2274	98.92
rcu_sched	404	99.11
lxpanel	289	99.25
gdbus	158	99.33
kworker/0:0	158	99.40

TABLE IX. Active Periods by Thread in Blender

As shown in the above tables, the most significant threads directly pertain to the applications. This is most obvious with Blender, where the threads "blender" and "Xorg" (the most popular Linux display server) account for 98.27% of the active time, but the concept holds true for the other applications as well. The tables for Firefox end with relatively low cumulative percentages (94.82% and 81.69%), but this is at least partially attributable to the presence of several more numbered "MediaPlb̃ack" and "MediaPDõder" threads further down the lists. We have good reason to believe that these threads all belong to Firefox because they do not appear for any of the other applications. In general, we posit that if a thread is highly significant across trials for some application then it is doing work for that application. Thus, one could filter out threads that are only significant for a portion of trials for each application and assume that they belong to background tasks. However, the general consistency that we see between trials indicates that this would not significantly impact our results.

We found that it is not feasible to consider only threads created by the applications in question, even though trace-cmd can provide this information, because applications commonly utilize preexisting threads to perform their computations. This was perhaps most evident during preliminary data collection on the LibreOffice applications, where filtering out preexisting threads caused the data to reflect a single active period that took up almost all of the runtime (roughly one minute). This was not actually the case, but all context switches after that thread began involved some preexisting thread and were, therefore, filtered out during data collection. This raises some difficult questions about what exactly it means for a thread to "belong" to an application. For purposes of this paper, we assume, based on minimizing irrelevant CPU demand and on consistency across trials, that most of the data collected is directly attributable to the applications being studied.

VI. Results

A. Data Characterization

Here, we show that the majority of active periods for the applications studied are relatively short in duration. Figure 4 shows the cumulative fraction of active periods and the cumulative fraction of active time as functions of the active period duration for all of the data for all applications together. The next 4 figures show the same metrics for each application separately. To see these metrics for each individual trial (5 per application) see Appendix B.

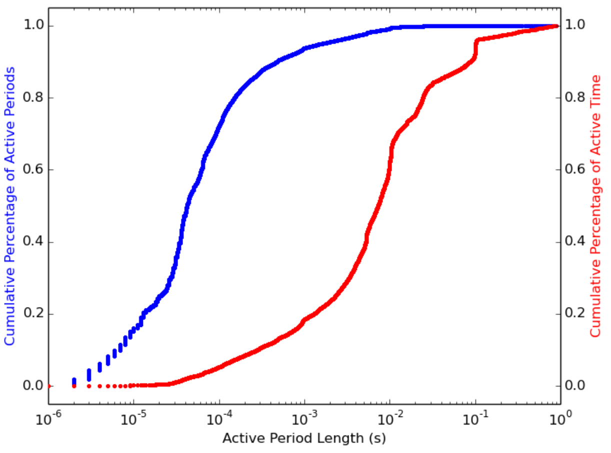


Fig. 4. All data: Active periods and active time vs. active period length

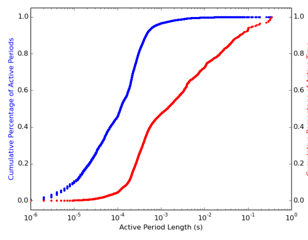


Fig. 5. Calc active periods

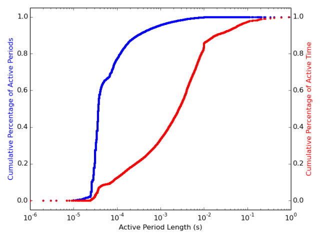


Fig. 6. Writer active periods

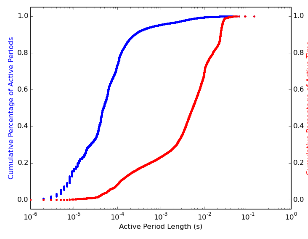


Fig. 7. Firefox active periods

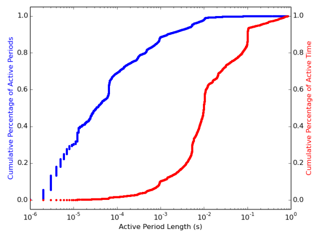


Fig. 8. Blender active periods

These plots show that, in general, the majority of active periods for these applications are relatively short, while the few long active periods account for most of each application's active time. Looking at the data for all four applications together, 90% of the active periods are less than or equal to 475 microseconds in length, but all of these together only account for roughly 12.75% of total active time.

We speculate that this general pattern holds for the vast majority of widely used software. The two LibreOffice test protocols involved quite a bit of user input throughout the testing. In contrast, streaming a video on Firefox and rendering in Blender required no user input. Thus, we have two interactive and two non-interactive applications displaying roughly the same pattern. Furthermore, one of our reasons for including Blender in this study is that we suspected that its specialized nature and relatively small user base might lead to different behaviour here. As is evident from Figure 8, this was not the case. Blender followed the same pattern as the other applications. Future testing with more applications could confirm this hypothesis.

It worth noting the presence of some active periods here that do not belong to our applications. As discussed above, these account for a small minority of the data collected, but our data suggest that they are disproportionately on the shorter end of the duration spectrum.

B. *Scheduler Modification Simulation*

Here, we evaluate the thread scheduler modifications. We begin with the modification where, when an active period starts on one CPU core, the scheduler briefly blocks all other active periods from starting on the other cores. This block persists until either the blocking active period ends or yields, or it runs for longer than the maximum block time. We call this the ”blocking scheduler”. We have seen that most active periods tend to be short (90% shorter than 475 microseconds), so we hope that many of them can finish before the maximum block time elapses. This allows the next queueing active period to start without the possibility of creating concurrency issues with the first. We also evaluate the possibility of not blocking for threads that recently had long running active periods. We call this the ”selective blocking scheduler”, and the figures that correspond to this method are in Appendix B. We evaluate these concepts with block times from 0 to 1500 microseconds. Our categories for evaluation are reduction of concurrency, impact on total application runtime, and impact on application responsiveness.

We begin with LibreOffice Calc. Figure 10 shows that the increase in total runtime is negligible for the full range of block times. Figure 11 shows that the effect on the responsiveness is a more serious issue. The maximum delay time for an active period with block times greater than 1100 microseconds could certainly be visually perceptible to humans. However, the median delay time would barely effect performance. Figure 9 shows that this scheduler generally decreases all measures of concurrency, but that the marginal benefit with increasing block times is clearly diminishing. Any block time over roughly 410 microseconds eliminates all concurrency not involving ”long” active periods. For purposes of this study, we define ”long” active periods as those lasting more than 500 microseconds. Note that fewer than 10% of active periods are long. Eliminating concurrency involving only short active periods is desirable because it would allow application developers to focus their debugging efforts on only the code expected to generate long running active periods.

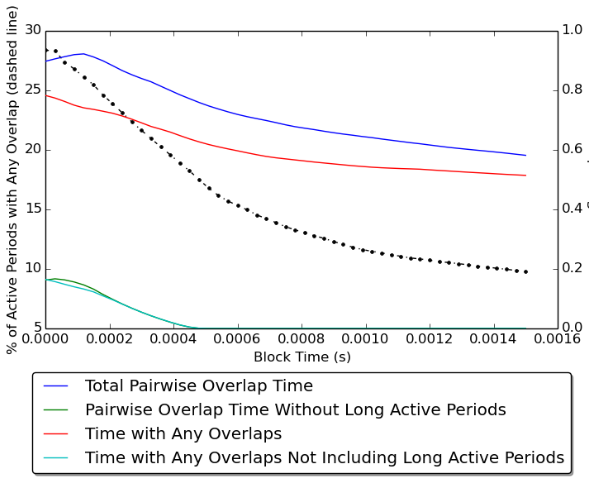


Fig. 9. Reduction in concurrency with the blocking scheduler in Calc

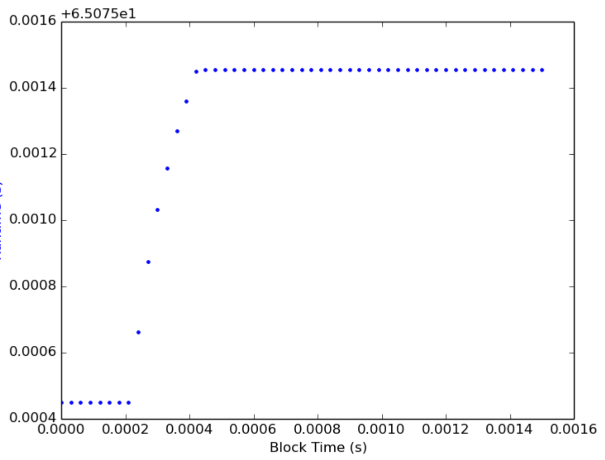


Fig. 10. Impact of the blocking scheduler on runtime in Calc

Figure 22, figure 23, and figure 24 show very similar results for the selective blocking scheduler. The impact to total runtime remains inconsequential, there is slightly less detriment to responsiveness, and there is slightly less benefit in terms of concurrency reduction. Notably, the selective blocking scheduler does not completely eliminate concurrency involving only short active periods.

All in all, these schedulers seems to work fairly well with Calc. It is important to keep in mind, however, that the original Calc data contained less than 0.9 seconds of overlap time for all measures of overlap time, and that there were also only about 12 seconds of total active time split over all

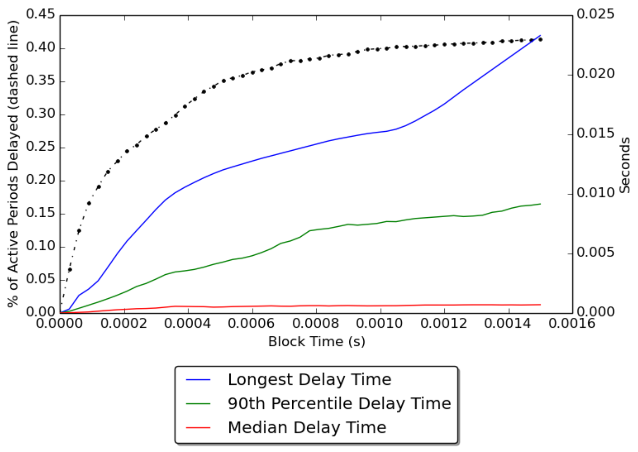


Fig. 11. Impact of the blocking scheduler on responsiveness in Calc

four processors for just over a minute of total runtime. In other words, Calc is not at all CPU intensive.

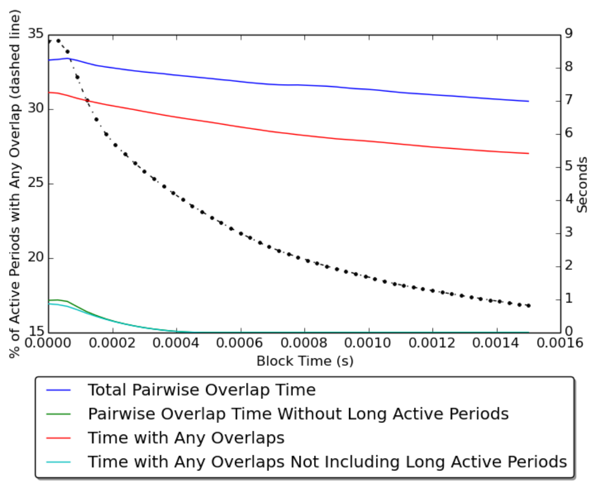


Fig. 12. Reduction in concurrency with the blocking scheduler in Writer

Now, we perform a similar analysis for LibreOffice Writer. Figure 13 shows that the increase in total runtime, though greater than with Calc, is still negligible for the full range of block times with a maximum less than 460 microseconds. However, figure 14 shows that the effect on responsiveness, which was the main concern in Calc, is even greater here. The maximum delay time increases sharply to levels that would noticeably impact performance for block times greater than 700 microseconds. Even with shorter block times, the maximum active period delay is reason for concern, but, as with Calc, the median delay time remains wholly negligible across all block times. The 90th percentile delay time for Writer is consistently close to 10 times the corresponding value for Calc. Figure 12 shows that this scheduler has the potential to greatly decrease the number of active periods with any overlap. Furthermore, as with Calc, the scheduler completely eliminates overlaps involving only short active periods at a block time of roughly 400 microseconds. With Writer, however, the benefits as quantified by pairwise overlap time and time with any overlaps are not as pronounced. With this information in mind, the analysis for the selective blocking scheduler on Writer is completely analogous to that same analysis for Calc.

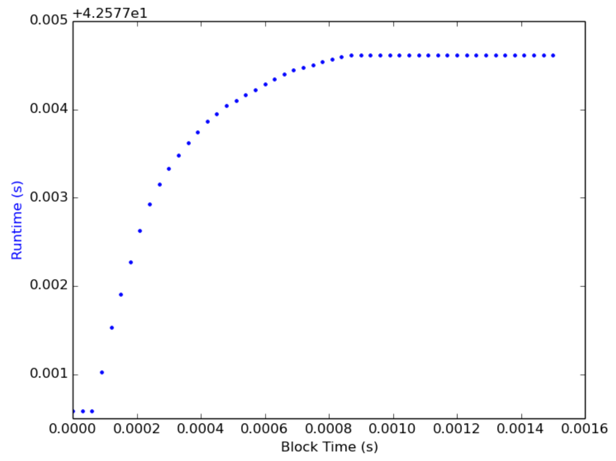


Fig. 13. Impact of the blocking scheduler on runtime in Writer

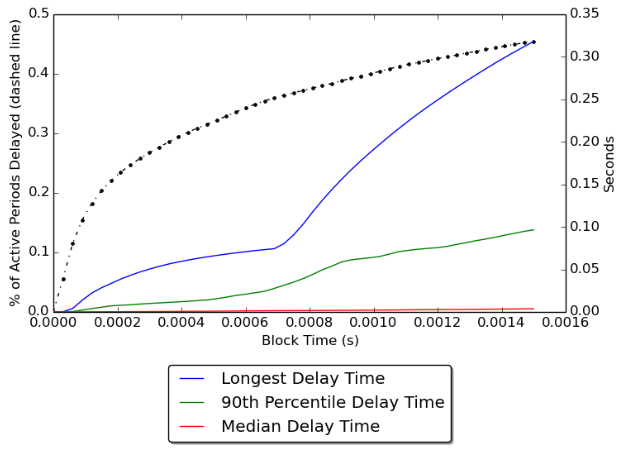


Fig. 14. Impact of the blocking scheduler on responsiveness in Writer

The original data for this particular experimental trial had about 40 seconds of runtime with 45 seconds of active time across the 4 cores. This means that Writer is much more CPU intensive than Calc, which is the reason for the greater detriment to responsiveness with these schedulers.

The blocking scheduler's effect on runtime with Firefox

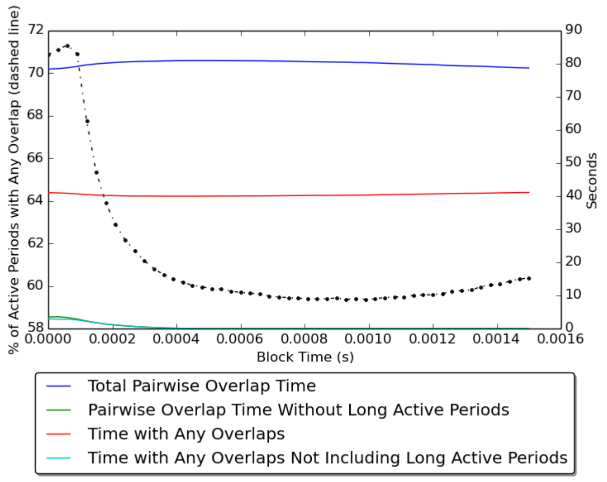


Fig. 15. Reduction in concurrency with the blocking scheduler in Firefox

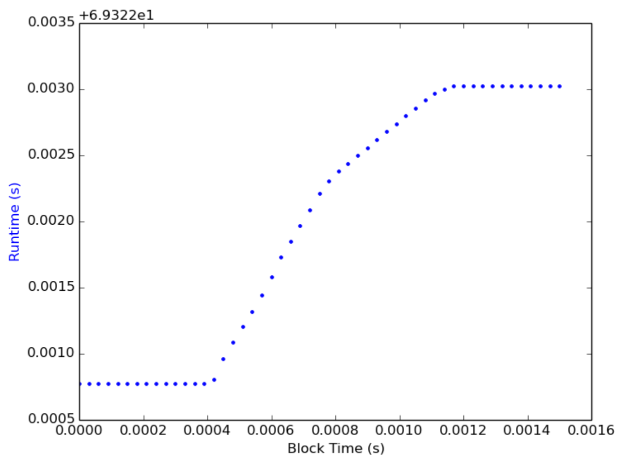


Fig. 16. Impact of the blocking scheduler on runtime in Firefox

is similar to what we have seen above. That is to say that runtime is not the main concern. The main concern with Firefox is that the maximum and 90th percentile delay times can be several tenths of a second for block times greater than 400 microseconds. This would certainly cause obvious damage to responsiveness. Furthermore, though it can reduce the proportion of active periods that overlap by over 10%, the blocking scheduler does not appear to reduce pairwise overlap time or the time with any overlaps at all, although it does bring these metrics to zero when long active periods are disregarded. We see essentially the same (though slightly tempered) reductions in concurrency with the selective blocking scheduler, and the detriment to responsiveness, though not quite as steep, remains in the same ballpark.

Firefox is more CPU intensive than the two LibreOffice applicaitons, with about 126 seconds of active CPU time

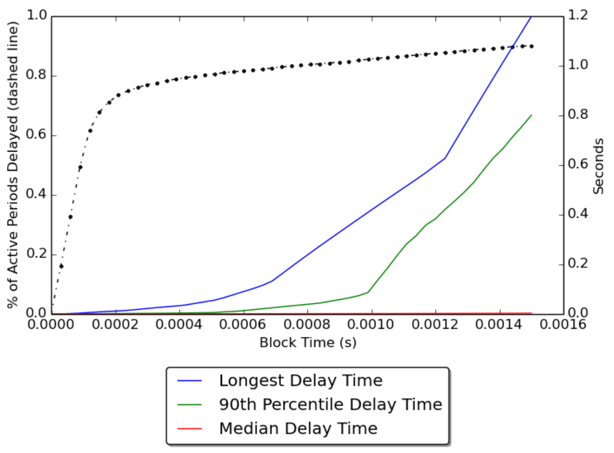


Fig. 17. Impact of the blocking scheduler on responsiveness in Firefox

during 70 seconds of data collection. It also has relatively high original levels of concurrency as shown in figure 15 with a block time of 0 seconds. This helps to explain why the scheduler modifications hurt responsiveness more than with the LibreOffice applications.

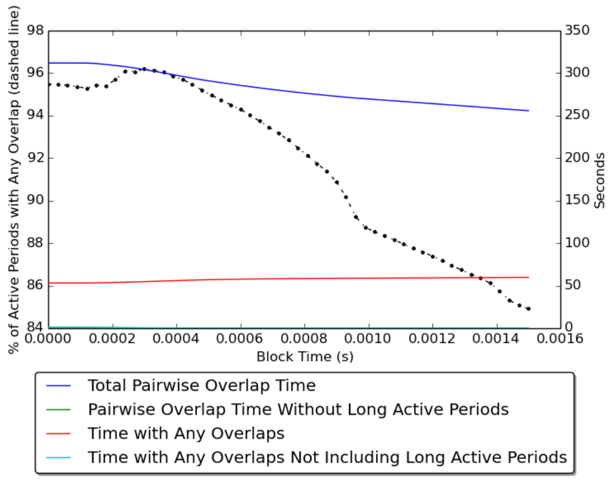


Fig. 18. Reduction in concurrency with the blocking scheduler in Blender

Blender is by far the most CPU intensive of the four applications with approximately 211 seconds of active CPU time in just under a minute. It also has intrinsically high levels of concurrency as shown in figure 18. Therefore, it is not surprising that the responsiveness of blender shows the greatest response to the scheduling changes. Even the median active period delay time, which is always very low for the other 3 applications, is several seconds for most of the range of block times with Blender (see figure 20). Also, almost all active periods are delayed. Figure 19 shows that the overall runtime of the render increases drastically once

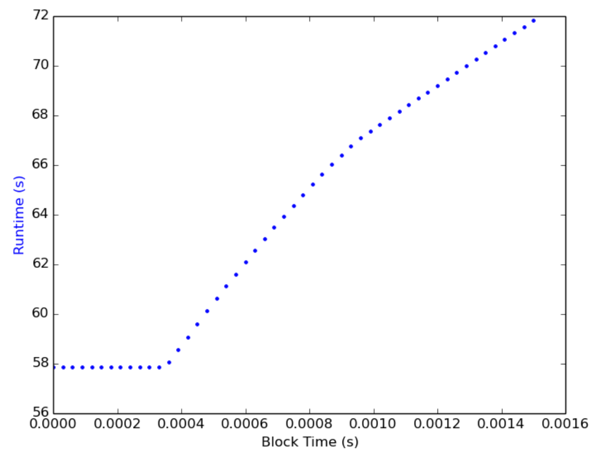


Fig. 19. Impact of the blocking scheduler on runtime in Blender

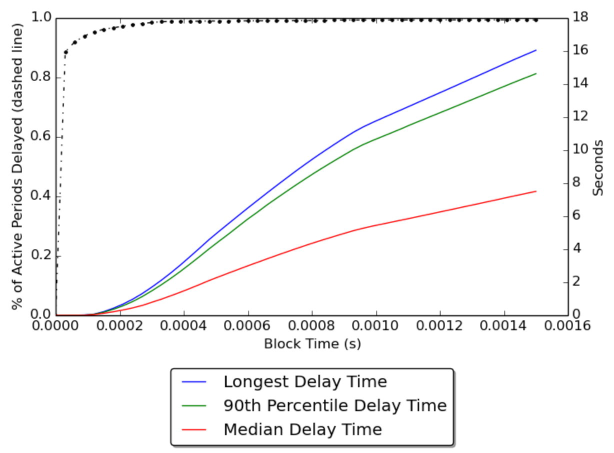


Fig. 20. Impact of the blocking scheduler on responsiveness in Blender

the block time is greater than 400 microseconds. From a performance standpoint, this is unacceptable. Furthermore, in figure 18, though the fraction of active periods with any concurrency does decrease with block time, the other measures of concurrency show relatively little change. All of this holds generally true for the selective blocking scheduler as well. Clearly these scheduling tweaks are much more costly with Blender than with the other applications, without any significant improvement to the benefits.

From this analysis it is clear that these particular scheduler modifications are beneficial and viable only for applications that have low CPU needs (like Calc). The costs simply outweigh the benefits for more CPU intensive applications. Ideally, the scheduler could identify CPU intensive applications or CPU intensive circumstances (if multiple applications are taxing the CPU) and either reduce blocking or cease blocking altogether in these circumstances.

Furthermore, performance would be better if the scheduler could more accurately predict whether a queuing active period would be long or short, and only block for short active periods. We suspect that event based threads tend to have shorter active periods than computation threads, so only blocking for event threads could be one way of accomplishing this. Also, allowing computation threads to run their active periods unblocked would help to mitigate performance costs with CPU intensive applications. The challenge is then to determine which threads are event based and which are computation focused. Because of the suspected difference in active period duration between event and computation threads, we initially suspected that thread ID might be a good predictor of active period duration. Table X shows, in descending order, the coefficients of variation of active period duration for the four applications. Only threads with 1000 or more active periods in one trial are shown.

Calc	Writer	Firefox	Blender
7.48	5.81	5.60	4.92
2.99	5.22	5.29	4.13
2.87	3.83	4.62	4.08
1.90	3.82	4.01	3.77
1.87	3.12	2.84	3.70
1.68	2.66	2.80	3.52
1.19	2.25	2.69	3.44
0.39	1.38	2.65	2.33
	0.97	2.45	2.32
	0.86	2.14	1.82
		2.10	1.45
		2.10	0.89
		2.09	0.61
		2.06	0.59
		2.03	
		1.92	
		1.78	
		1.56	
		1.41	
		1.31	
		1.28	
		1.27	
		1.27	
		1.01	
		0.84	
		0.64	
		0.57	

TABLE X. COEFFICIENTS OF VARIATION FOR ACTIVE PERIOD DURATION OF THREADS WITH AT LEAST 1000 ACTIVE PERIODS

Table X indicates that the duration of recent active periods for any given thread is not always an accurate predictor of duration of subsequent active periods. This information helps to explain why there was not a greater performance difference between the selective and non-selective blocking schedulers. Figure 21 shows that, despite some high coefficients of variation, the mean durations for the threads in Firefox have some spread and perhaps display some binning. The other applications produce similar results, but with fewer reliable data points because they all had fewer threads with over 1000 active periods than Firefox. We posit that the gap in mean duration may be the divide between event based and computation based threads, but further study is necessary to verify this. Also, the data suggest that looking at historic mean duration for a thread may provide a useful part of a more complex system for predicting the length of active periods.

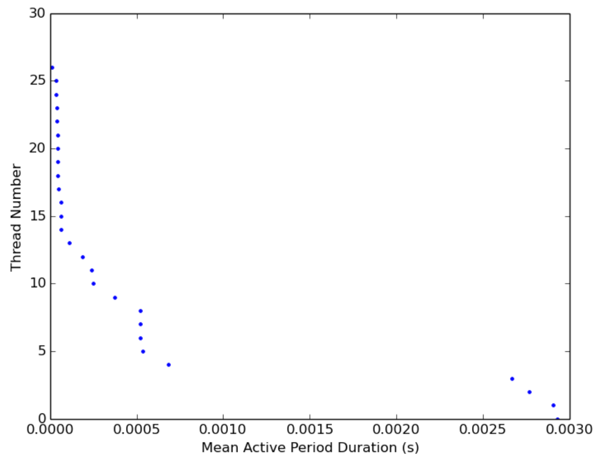


Fig. 21. Mean active period duration for the most active threads in Firefox

VII. Conclusion

In the characterization portion of this study, we found that the majority of active periods are relatively short, while the minority of long active periods account for most of each application's active time. For the applications that we studied, the shortest 90% of active periods only account for roughly 12.75% of total active time. Future work can investigate more operating systems (both desktop and mobile), more applications, and more hardware environments to figure out in what circumstances this result holds true.

We have also shown that our proposed blocking thread scheduling techniques are good for applications that have low CPU needs, but are simply too costly in terms of performance for CPU intensive applications. In the future, we would like to investigate dynamically adjusting the block time in response to CPU workload, develop a better method for predicting active period duration, and ensure that we only block threads that share memory space with one another.

References

[1] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner, ”Evolution of Thread-Level Parallelism in Desktop Applications,” in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[2] C. Gao, A. Gutierrez, R. G. Dreslinski, T. Mudge, K. Flautner, and G. Blake, ”A Study Of Thread Level Parallelism on Mobile Devices,” in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[3] S. Lu, S. Park, E. Seo, and Y. Zhou, ”Learning From Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics,” in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[4] N. Anderson, ”How OS X Mavericks works its power-saving magic: More details on App Nap, Timer Coalescing, and Compressed Memory,” in *ars technica*, 2013.

Appendix A
Performance Evaluation Figures for the Selective Blocking Scheduler

LibreOffice Calc:

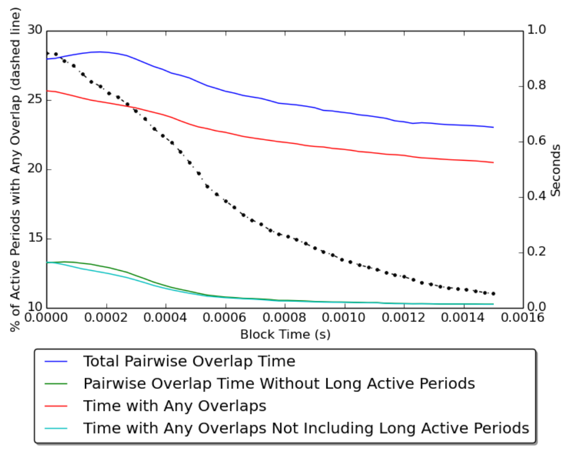


Fig. 22. Reduction in concurrency with the selective blocking scheduler in Calc

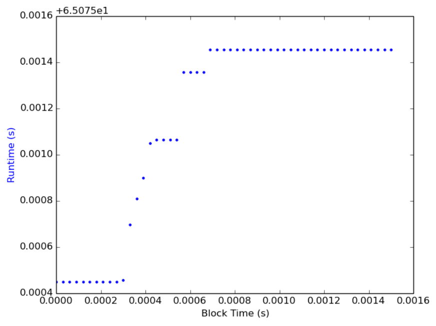


Fig. 23. Impact of the selective blocking scheduler on runtime in Calc

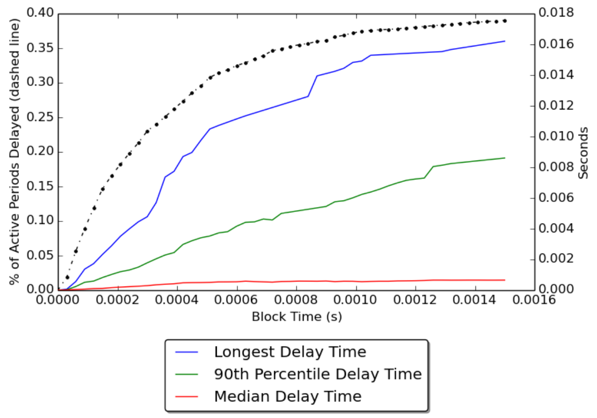


Fig. 24. Impact of the selective blocking scheduler on responsiveness in Calc

LibreOffice Writer:

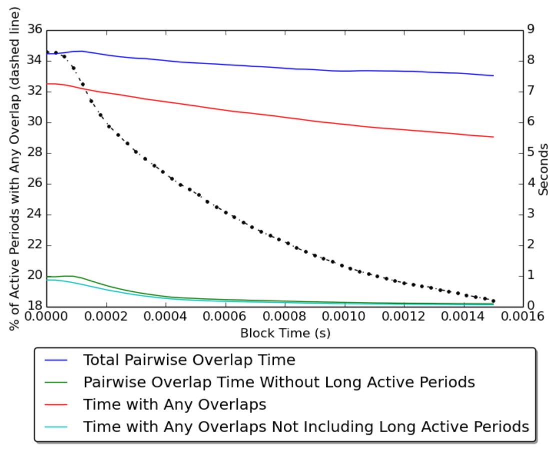


Fig. 25. Reduction in concurrency with the selective blocking scheduler in Writer

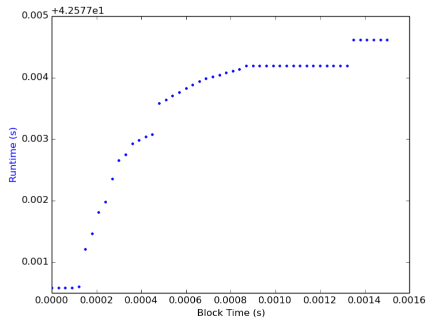


Fig. 26. Impact of the selective blocking scheduler on runtime in Writer

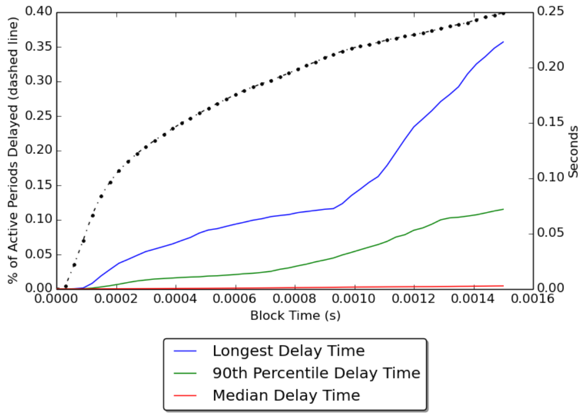


Fig. 27. Impact of the selective blocking scheduler on responsiveness in Writer

Mozilla Firefox video streaming:

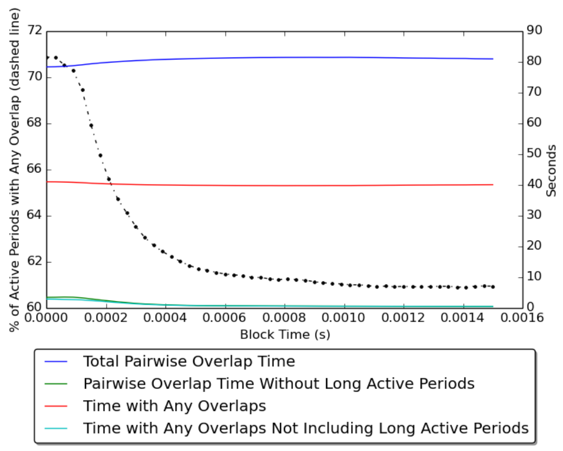


Fig. 28. Reduction in concurrency with the selective blocking scheduler in Firefox

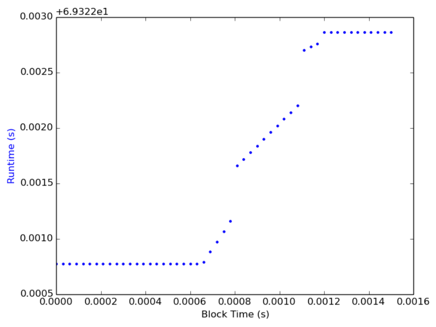


Fig. 29. Impact of the selective blocking scheduler on runtime in Firefox

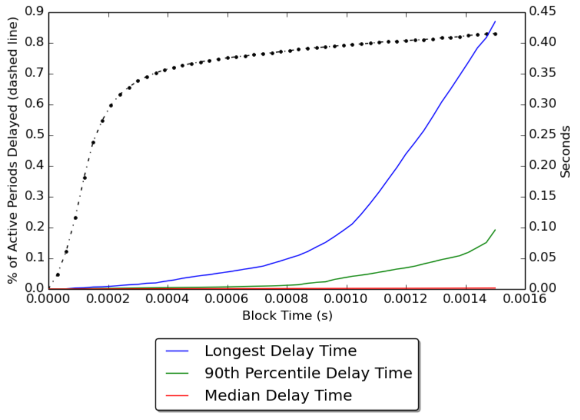


Fig. 30. Impact of the selective blocking scheduler on responsiveness in Firefox

Blender Cycles rendering:

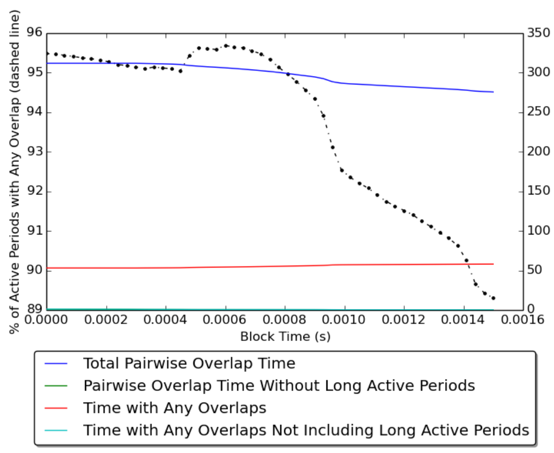


Fig. 31. Reduction in concurrency with the selective blocking scheduler in Blender

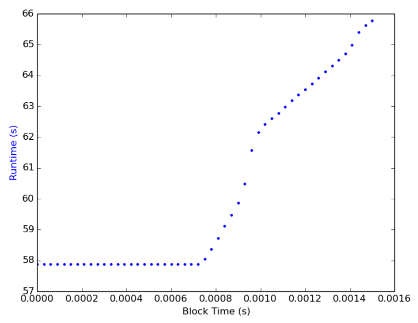


Fig. 32. Impact of the selective blocking scheduler on runtime in Blender

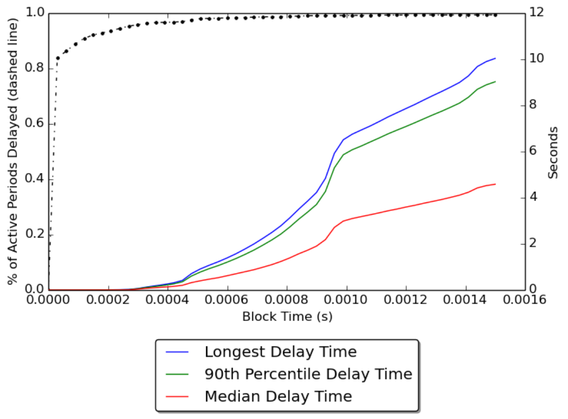
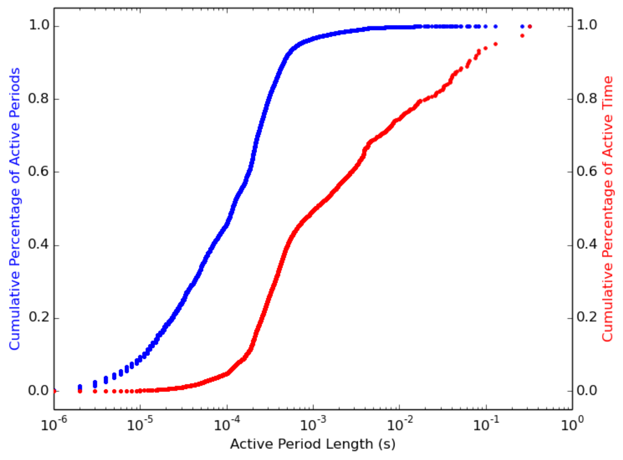
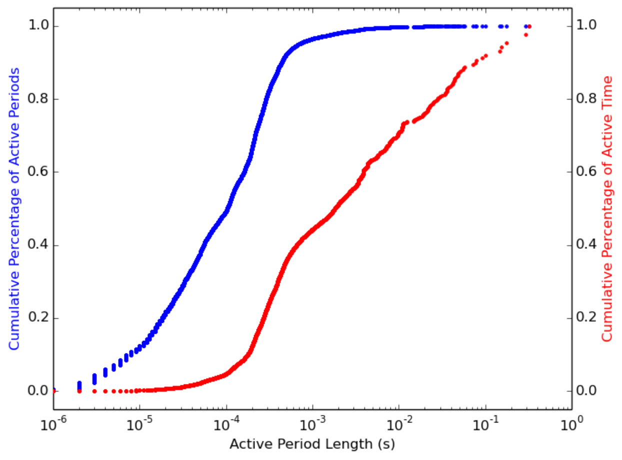


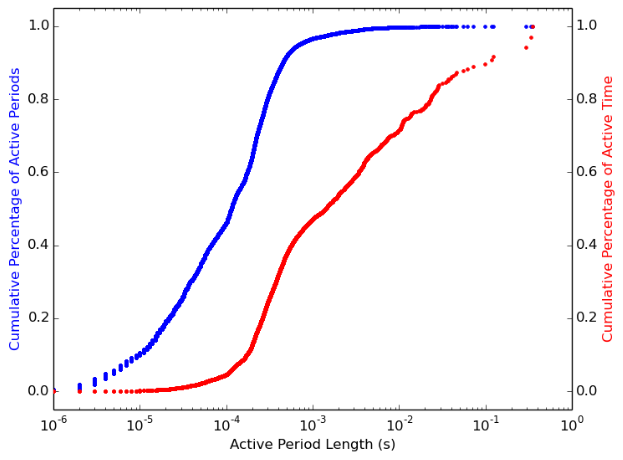
Fig. 33. Impact of the selective blocking scheduler on responsiveness in Blender

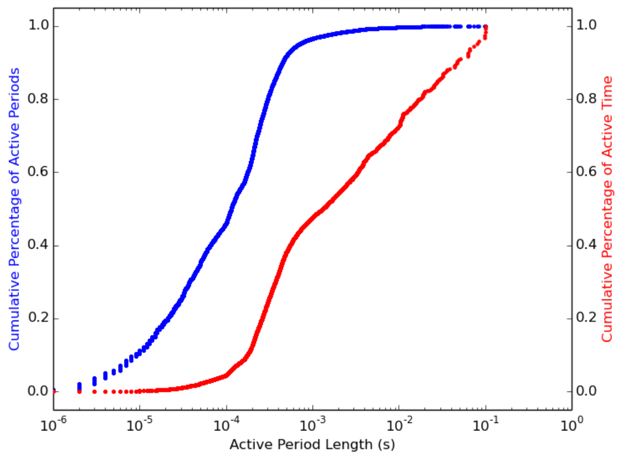
Appendix B
Active Period Duration Graphs for each Experimental Trial

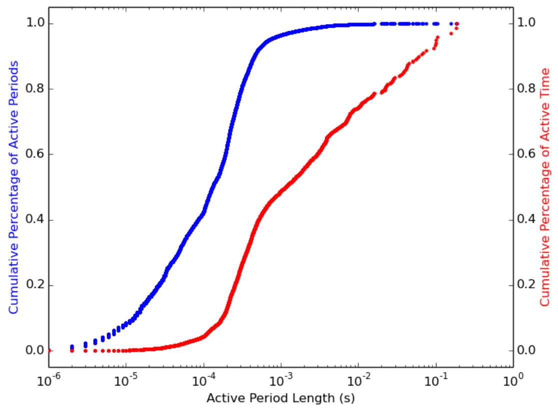
LibreOffice Calc:



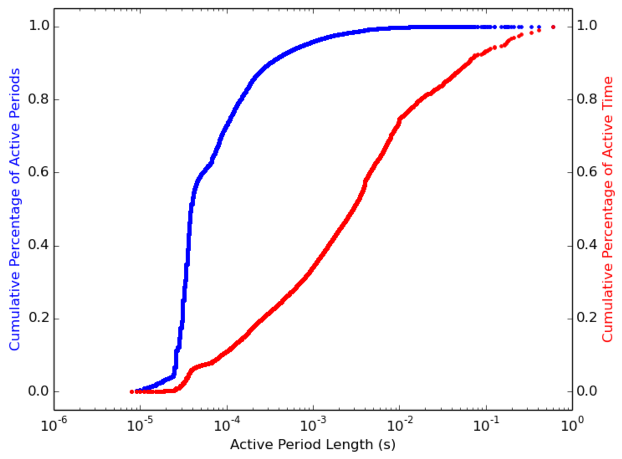


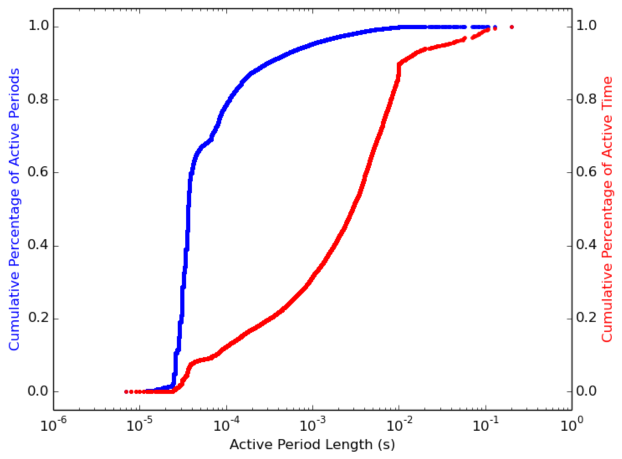


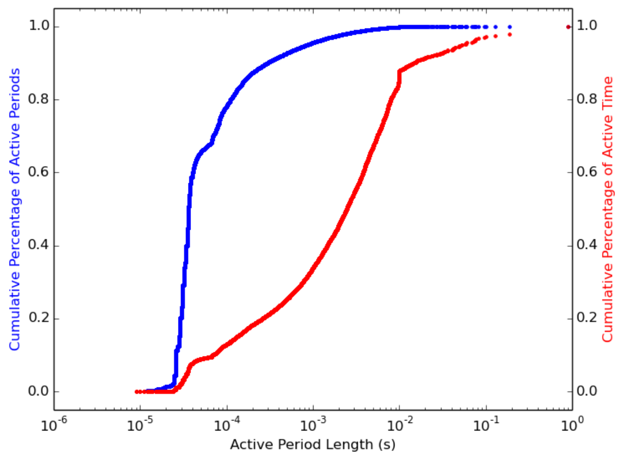


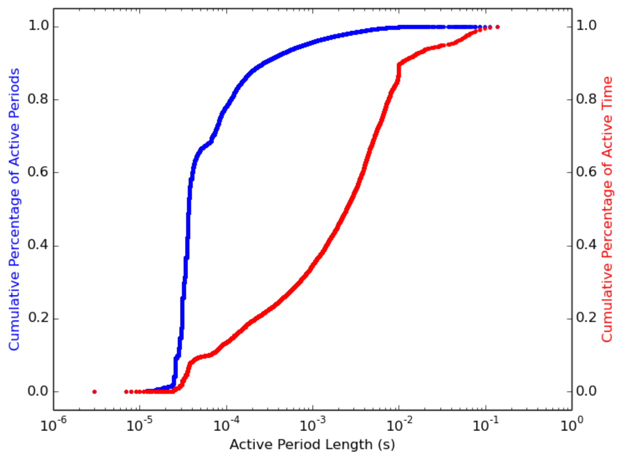


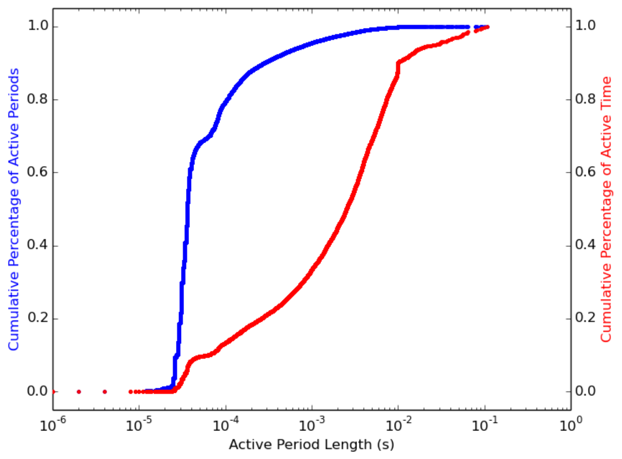
LibreOffice Writer:



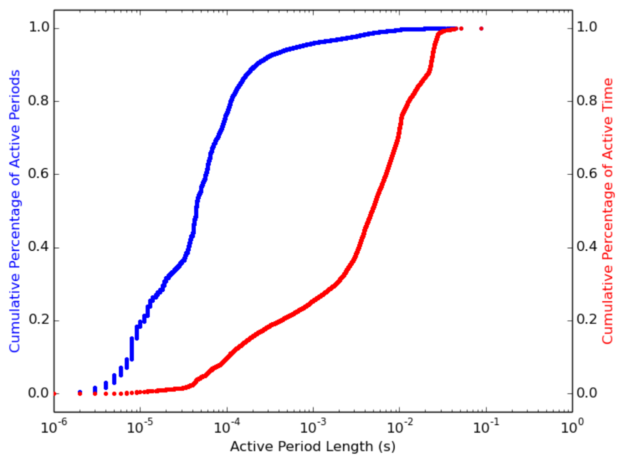


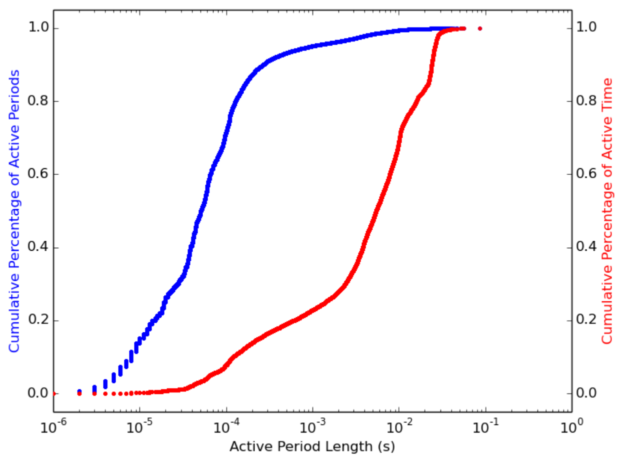


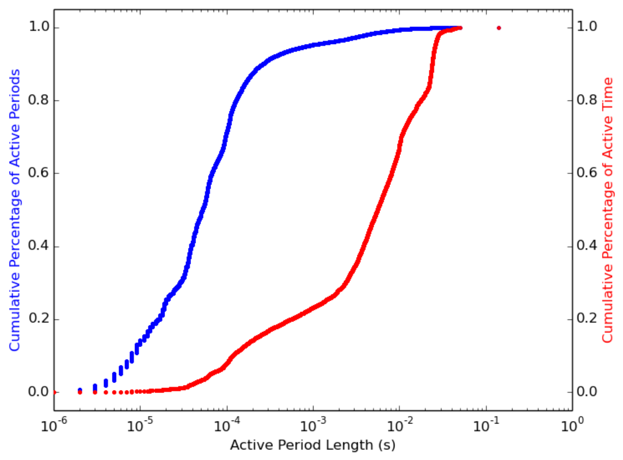


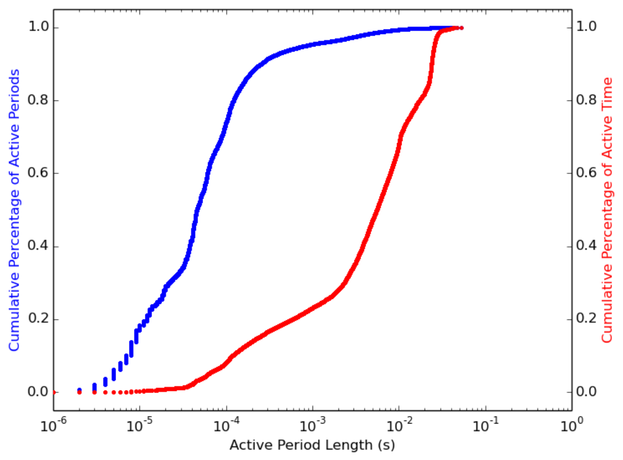


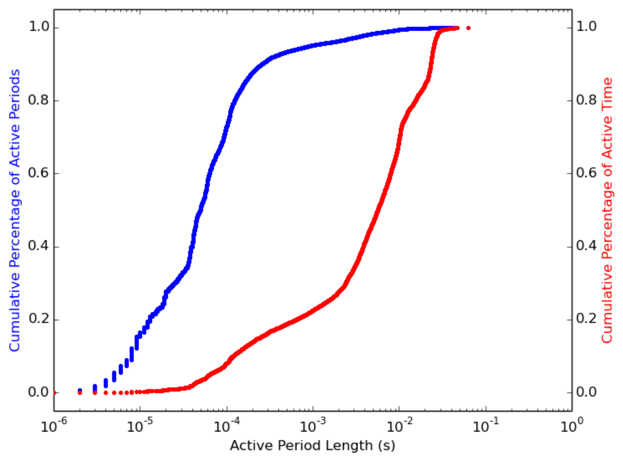
Mozilla Firefox video streaming:



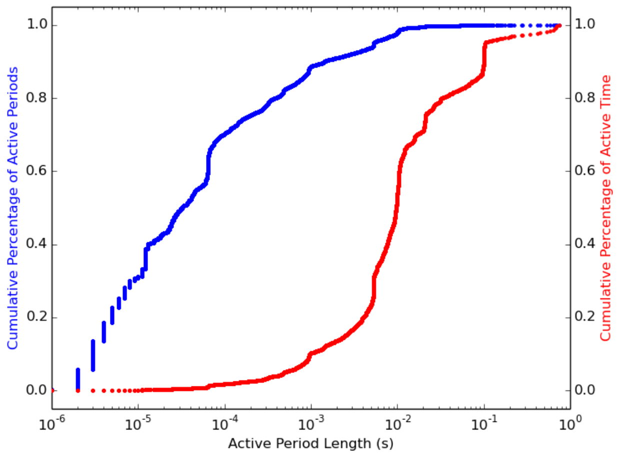


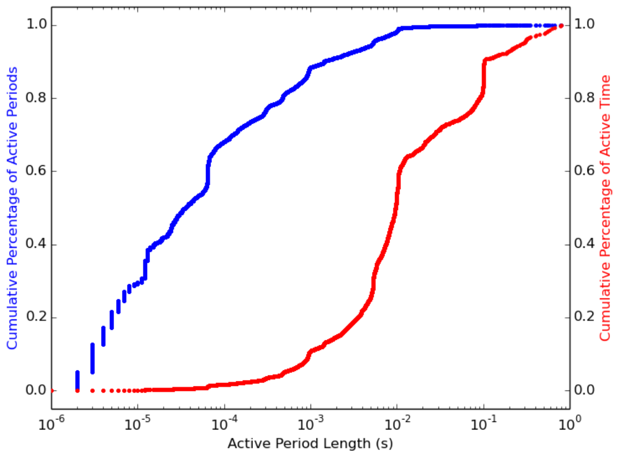


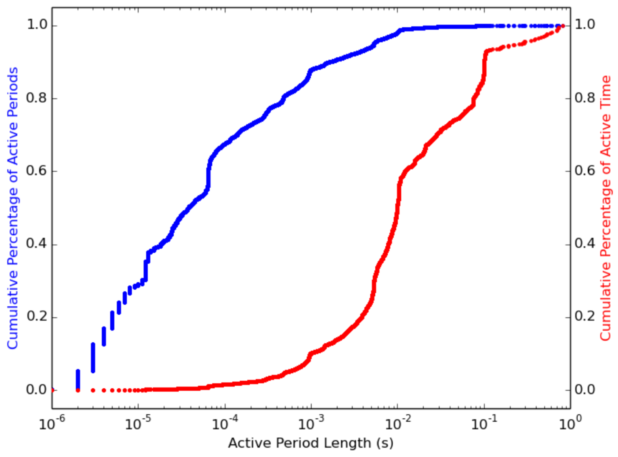


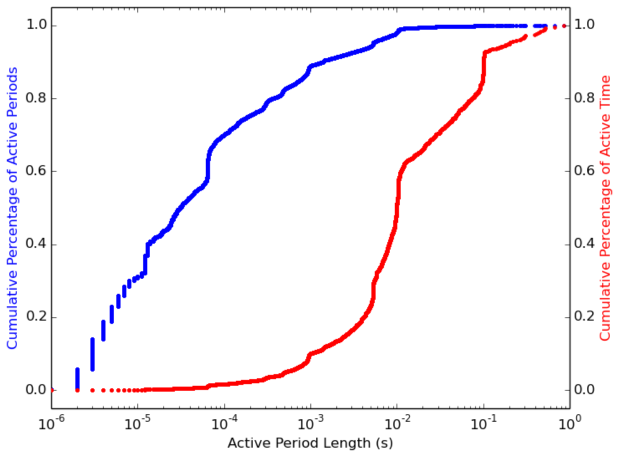


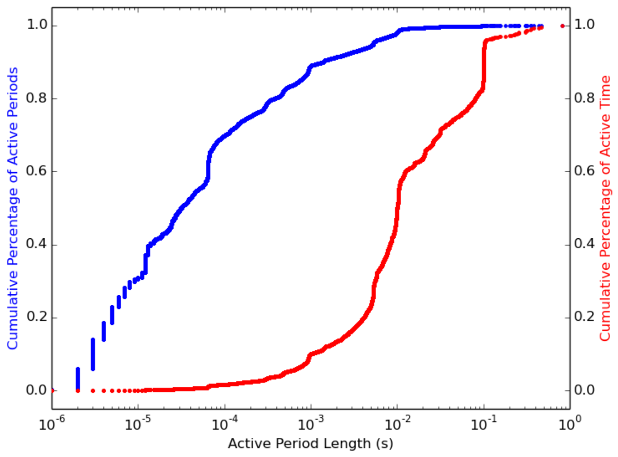
Blender Cycles rendering:


Cumulative Percentage of Active Periods
Cumulative Percentage of Active Time
Active Period Length (s)


Cumulative Percentage of Active Periods
Cumulative Percentage of Active Time
Active Period Length (s)


Cumulative Percentage of Active Periods
Cumulative Percentage of Active Time
Active Period Length (s)


Cumulative Percentage of Active Periods
Cumulative Percentage of Active Time
Active Period Length (s)


Cumulative Percentage of Active Periods
Cumulative Percentage of Active Time
Active Period Length (s)