Introducing scheduling decisions in cooperative multitasking: an automated approach

Martin Takeya Weber
École Polytechnique Fédérale de Lausanne
martin.weber@epfl.ch

ABSTRACT

Userspace threads are execution contexts that are managed by a userspace runtime rather than the kernel. For any two userspace threads that run on top of the same kernel thread, a context switch between those two userspace threads does not require any transition to kernelspace, as all thread context data is managed and available in userspace. Context switching there is thus significantly more lightweight than between kernel threads. However, as there is no way to interrupt processes without the kernel, userspace threads cannot be preemptively scheduled purely from userspace; instead, they must run *cooperatively*, i. e. they must regularly yield control over the processor to allow other userspace threads to run as well.

Writing software for such an environment requires yields to be inserted in the right places: if some long-running execution path in the code was no considered and contains no yields, it will end up blocking other processes from running. We therefore propose a mechanism to relieve the programmer from having to handle yielding manually: we have written an LLVM [5] Pass that instruments the code at compile time and automatically inserts yields at specific positions. This allows code to be written for userspace threads as if it would run in a preemptively scheduled environment, leaving scheduling decisions to the compiler.

Preliminary benchmark results indicate that our method does not severely degrade throughput performance. Microbenchmarks run on single cores show that our instrumented variants of userspace threads have execution time overheads of 5 to 8 % for certain parameters compared to kernel threads. With some more refinement, we believe that this performance can be improved further.

1. INTRODUCTION

Writing a server application normally consists of reading data from a socket, processing it, and responding to the client. One way to implement such software is to create a separate thread per connection: each thread handles incoming data for a single connection, and maintains only that connection's state. This multithreaded approach is fairly intuitive; however, it does not scale particularly well with a growing number of connections, as the kernel may not be able to spawn tens of thousands of threads at once.

An alternative is to keep only a single thread that waits for events on all sockets (e. g. with `epoll_wait`), and processes all incoming data within the context of the corresponding connection. This scales a bit better than the aforementioned multithreaded approach, as we do not spawn any new threads; however, the experienced latency for individual connections may suffer: if one connection misbehaves and keeps our single thread busy for a significant amount of time, other connections are not handled within reasonable delays. This can be improved by using *thread pools*, i. e. a fixed number of worker threads, where the main thread can simply dispatch incoming data to the next available thread, without risking to become blocked on problematic input data itself. This ensures *fairness* between connections; however, this also makes matters more complex. Furthermore, as threads will handle data for various connections, connection states cannot be reflected in an intuitive way (simply by the progress within the code), but must instead be encoded in and loaded from some independent context data.

1.1 Fibers

The scalability issue stems from the fact that the kernel cannot reasonably create any arbitray number of threads. We can avoid this limitation by not using kernel threads, but *fibers* [3]: a fiber is an execution context that is created and maintained on top of a kernel thread, typically by a userspace runtime. A fiber can be considered a "userspace thread" insofar as it is associated an individual stack and can exist independently from the fiber that created it; just like a thread, a fiber can wait for and join with other fibers, or create new fibers on its own; and just like threads, fibers rely on a separate component that manages and schedules them.

As all the required metadata and stack memory is maintained in userspace, context switching between fibers is a matter of saving a few registers on the current fiber's stack, pointing the stack pointer to the new fiber's stack, then restoring the registers from there, and jumping to the new fiber's current execution position (also restored from the stack); this is less costly than having to switch to kernel space to perform a full thread context switch.

As the kernel has no knowledge of fibers running on top of its threads, it cannot preemptively interrupt and schedule fibers (it can only do so for entire threads). Context switching to another fiber therefore requires that the fibers cooperate: as long as a fiber does not yield on its own, the userspace scheduler can not schedule another fiber. This is a major appeal of cooperative multitasking: the lack of kernel preemption allows tighter control over the execution flow of a multitasked program. However, it requires the programmer to strategically place yields in their code to ensure that fibers do not prevent each other from running, which can be tricky: if any execution follows a path in the code that the programmer has not accounted for (and thus placed yields

insufficiently), this may result in a fiber blocking execution for other fibers.

Another issue is what happens when a fiber makes a blocking system call: ideally another fiber could take over and continue running while the first one is waiting for the system call to return. However, the kernel has no knowledge of fibers; such a system call will simply block the entire kernel thread. Fibers are therefore often written in a way that they only make non-blocking system calls; however, this might impose a specific design choice on the program that may not always be suitable.

Some fiber implementations work around this issue and provide mechanisms to allow programmers to use blocking system calls with fibers. Anderson et al proposed so-called *scheduler activations* [2], i. e. an extension to the kernel such that it can distinguish regular threads from threads that run fibers and interact with userspace schedulers and "activate" them in specific events. Brian Watling's *libfiber* [8] on the other hand provides a drop-in replacement wrapper (*shim*) around blocking system calls; the shim calls the respective *non-blocking* variants, and then yields as long as the call returns `E_WOULDBLOCK`, and only returns once the system call returns a real result, giving the fiber the illusion that it made a blocking system call. A major advantage of this approach over scheduler activations is that this can be done entirely in userspace and does not require any special support in the kernel.

On the other hand, no fiber implementation has any mechanisms to guarantee *fairness* in cases where a fiber misbehaves and does not yield on its own. One approach would be to use interprocess communication mechanisms like interrupts or signals that trigger a routine in the userspace scheduler, which preemptively switches to another fiber. However, we would lose all benefits of having lightweight context switches between fibers if each context switch was preceded by comparatively costly IPC involving the kernel.

For the sake of performance, we prefer to have fibers yield on their own, but for the sake of robustness (and for practical reasons), we would like to relieve the programmer from having to insert yields manually. Our proposal is thus a mechanism that automatically inserts yields into the program at compile time.

1.2 LLVM

LLVM [5] is a compilation framework that handles various aspects of compilation, including optimisation and code generation. As input, it expects a specific LLVM-*bitcode* (rather than code in a regular programming language). The goal is to relieve language designers from having to write a full blown compiler; instead, they only need to write an LLVM *frontend* for their language (lexer and parser), transform it to LLVM bitcode, and have the LLVM *backend* handle the rest of the compilation procedure. A well-known LLVM frontend for the C language family is Clang [1].

This modular design is also applied to the internals of the backend: LLVM implements the various components involved in the compilation procedure—such as optimisation—as so-called *Passes* that each run on the code. Third-party developers can add Passes themselves and thereby add further, more specific functionality to LLVM if desired.

For our work, we intend to write an LLVM Pass that automatically injects yields into the input code.

1.3 Libfiber

Libfiber [8] is a userspace library that provides an API similar to POSIX threads for creating and joining fibers and creating mutexes, semaphores and spin locks, and provides a function `fiber_yield` that wraps around some fiber management operations and fiber context switches.

As it provides a familiar API, handles blocking I/O purely in userspace, and does not pose any specific restrictions on the number of fibers that can be created, we have decided to use libfiber as a basis for our work.

2. IMPLEMENTATION

Libfiber has been designed to work on multi-core systems (i. e. it can multiplex fibers on top of multiple kernel threads), and by default, it uses a work-stealing scheduler, implemented in `fiber_scheduler_wsd.c`. However, if limited to a single kernel thread, we have observed that this scheduler only switches back and forth between the same two fibers, with the remaining fibers never getting CPU time at all. We have thus enabled the other scheduler also provided in libfiber, implemented in `fiber_scheduler_dist.c`, which schedules fibers in a round-robin fashion.

For writing our Pass, we have used LLVM and Clang 7.0. We have derived our Pass from the predefined `ModulePass` class, as it poses the fewest restrictions on what our Pass can do (but also predefines fewer things compared to other, more specifically applicable Passes).

To inject yields into the code, we assumed that every long running code will eventually make function calls and/or run in a loop of some sort. In our initial version, we therefore iterated through all functions and all loops in the input code, and inserted a call to `fiber_yield()` in each terminating basic block there; essentially, at the end of each function and each loop, the resulting code would yield.

However, as context switching itself adds some execution time overhead itself, we cannot yield at *every* such point (the program would spend a significant portion of its time just context switching between fibers and not doing any real work). We therefore decided to keep a counter that is incremented each time execution passes through such a point, and yield only when the counter reaches a certain threshold (yielding every single time would thus correspond to a threshold value of 1); our LLVM Pass thus inserts something like this:

```
++globalcount;
if (globalcount >= YIELD_THRESHOLD) {
        globalcount = 0;
        fiber_yield();
}
```

The code is available in our public git repository[1].

3. EVALUATION

The goal is to compare the cost of context switching between fibers with the cost of context switching between kernel threads, but also see how much of an overhead is added by maintaining an additional counter for determining when to yield.

In this section, unless otherwise denoted, we refer to both userspace and kernel threads as simply "threads".

[1] `https://gitlab.gnugen.ch/mtweber/cooperative-multitasking`

3.1 Setup

We have written two simple throughput benchmarks (we measure the time between the first thread starting and the last thread finishing):

- **Counting**: Let each thread increment a thread-specific variable from 0 to some given, relatively high value.
- **Pointer Chasing**: Generate a list of elements that each contain pointers to their respective predecessors and successors, and store the elements in an array in a random order; then let each thread walk through that list until it ends up at its starting position. To avoid that threads "profit" from each others' memory reads by avoiding cache misses, each thread starts in a different position on the chain and—given the randomised nature of the chain in memory—should follow different paths.

Both benchmarks are compiled in multiple ways:

- With libfiber, instrumented with our LLVM Pass and with yields inserted automatically;
- With libfiber, uninstrumented and with yields inserted manually;
- With libfiber, uninstrumented and without any yields;
- With POSIX (kernel) threads.

For the variants that contain yield calls, we create multiple executables, one for each threshold level among the following: 5, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8 and 10^9.

The executables are built without any compiler optimisations (`-O0`). The test system is an Intel Core i5-2500 CPU (second generation, "Sandy Bridge") with 6144 KiB cache, at 3.30 GHz, and 8 GiB of main memory, running Linux 4.20.3 on Arch Linux.

We also want to exclude effects on our measurements caused by true parallelism (i. e. threads running on multiple cores); we are purely interested in comparing context switch times on a single CPU core. For the variants that use fibers we therefore initialise libfiber with a single kernel thread, and for the benchmarks using POSIX threads we "pin" all kernel threads to a single core with `taskset`.

We run each variant and each threshold level (where applicable) with the following parameters:

- 2, 3, 4, 5, 10, 20, 50 and 100 threads;
- Counting from 0 to 500,000,000; and pointer chasing through 16,000,000 elements à 16 bytes each (256 MB).

We run each variant–parameter combination three times, and sum up the measured running times.

3.2 Expectation

We expect that maintaining a yield counter adds some overhead, making our fiber implementation slower than kernel threads in this regard; however, we also expect context switching between fibers to take less time than switching between kernel threads, making fibers faster.

Furthermore, since yielding itself adds an overhead to the execution time, we expect the programs compiled with a higher yield threshold (thus yielding less frequently) to perform better than programs with a lower threshold. Consequently, we expect the extreme case (uninstrumented, without any yields) to perform best of all, even better than kernel threads, as it does not perform any more actions than the kernel threads, but also never context-switches during the execution of a fiber.

We hope that the overhead added by maintaining a counter is outweighed by the benefits of faster context switching, and that for some of the higher (but still reasonably low) threshold values, the fibers perform better than kernel threads. If not, we hope that fibers do not run too much slower than kernel threads (not more than 5 or 10 percent added overhead).

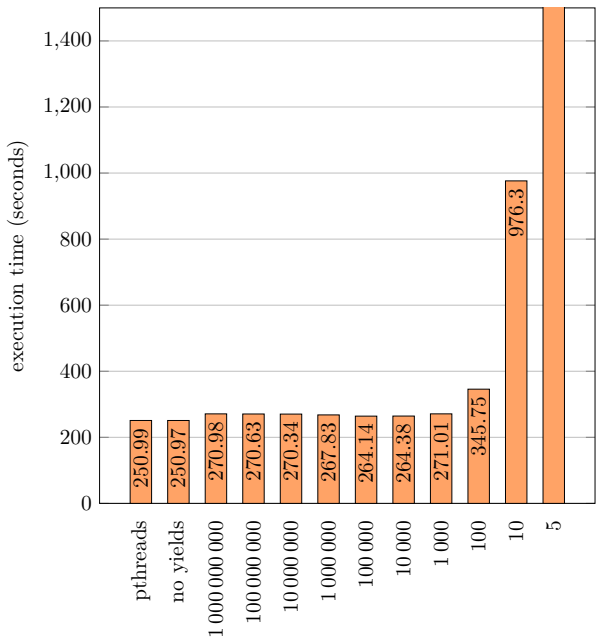

Figure 1: 100 threads counting, with automatically inserted yields

3.3 Results

3.3.1 Counting

The execution times for 100 threads counting from 0 to 500,000,000 can be seen in figure 1. As expected, the execution time appears to be inversely proportional to the yield threshold, although this relation remains largely unnoticeable for thresholds from 10^9 to 10^4 (all with overheads of around 5.2 to 8 % in comparison to kernel threads); execution times only start rising noticeably when we get as low as 100 (37.8 % overhead), and rather extremely with 10 (289 %) and 5 (580.2 %). We get very similar results for any other number of threads.

Unlike expected, the uninstrumented fiber variant (that neither counts nor yields) does not run significantly faster than kernel threads. And more confusingly, if we write the benchmark in a way that we manually count and yield (instead of instrumenting the code with the LLVM Pass), the execution becomes faster than both kernel threads and non-yielding fibers (figure 2).

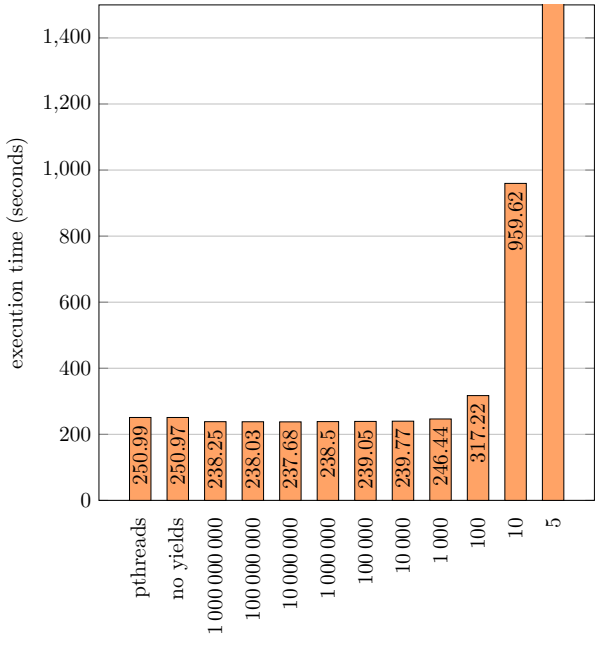

Figure 2: 100 threads counting, with manually inserted yields

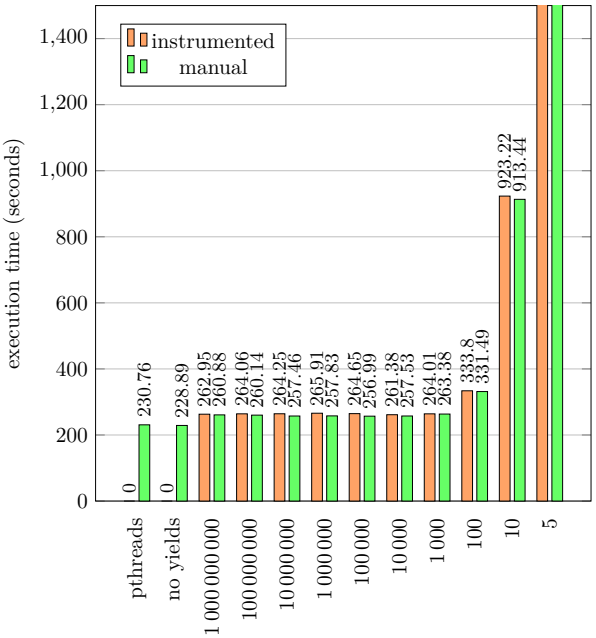

Figure 3: 100 threads counting, compiled with optimisations

As the behaviour is otherwise rather similar to the instrumented variants (more yields equals slower overall execution), this would indicate that our LLVM Pass simply produces less optimised code. To see if compiler optimisation would even out these differences, we built and ran the benchmark with `-O3` (figure 3), and it turns out that compiler optimisations do cause the instrumented and manual variants to give more similar results.

There is an awkward detail, however: if we compare the absolute running times for the manual variants, we observe that the optimised versions actually run *slower* than the non-optimised versions.

3.3.2 *Pointer chasing*

The execution times for 100 threads chasing pointers over 16,000,000 elements can be seen in figure 4. The results are *wildly* unexpected: for one, we get the lowest execution times with threshold values around 100 (which completes in 70 % of the time taken by kernel threads), whereas larger yield thresholds (10^9 to 10^6) all result in roughly the same execution times as the kernel threads. As expected, for very low yield threshold values, the execution times go up again.

We have observed that neither changing the size of the chain nor changing the number of threads changes this behaviour, with one exception: if run with only 2 threads, the results are as shown in figure 5. Compiling with optimisations does not change this pattern.

Using the performance data obtained with linux-perf, we can compare the data for the different variants (table 1 shows a comparison between thresholds 5 and 10^8, for illustration). We observe that low-threshold variants generally appear to have fewer L1 cache misses, which appears to result in a significantly higher number of non-stalled instructions per cycle. The lower cache miss rate for lower yield threshold values appears to indicate that the chain is not sufficiently

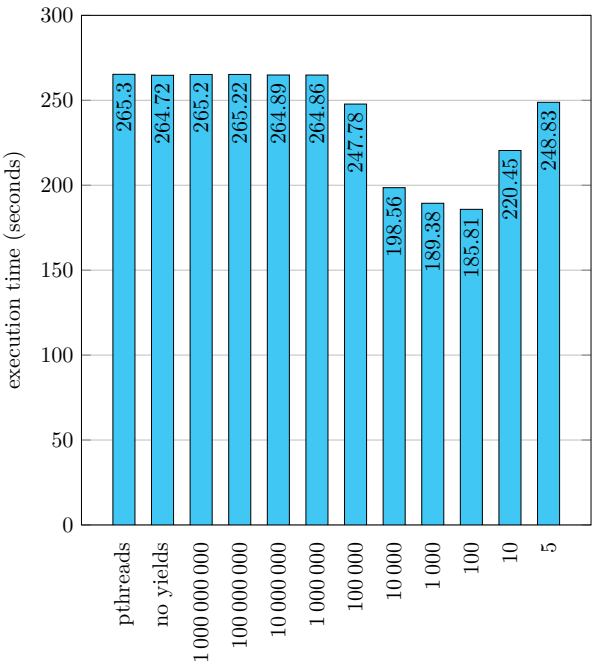

Figure 4: 100 threads chasing pointers

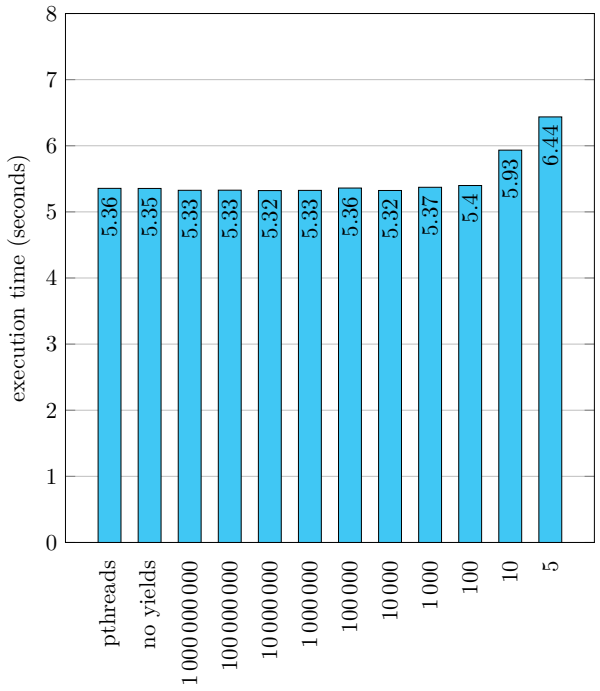

Figure 5: 2 threads chasing pointers

randomised and fibers thus follow too similar paths, and we therefore get an effect of fibers profiting from each others memory reads; however, the results do not change even after running on chains of different sizes (both longer and shorter), nor if shuffled with various different random seeds.

Unfortunately, linux-perf was unable to get the number of last-level cache (LLC) misses from the CPU.

4. DISCUSSION

The counting microbenchmark mostly gave us results as expected. However, the resulting binaries should be inspected more closely to understand why compiling the benchmark with compiler optimisations results in (slightly) slower code. The pointer chasing microbenchmark, on the other hand, behaved rather unexpectedly. The data gathered by linux-perf shows that caching has some impact on the results, which indicates that the microbenchmark might not be written in a way that adequately reflects throughput performance there.

We currently use a very simple approach for deciding when to inject (potential) yields. If we use a heuristic method that chooses insertion points more carefully, the overhead added by the instrumentation might be reduced.

Furthermore, given that we only count loops and functions, our approach is not very balanced; it favours fibers that run long functions or loops over fibers that only run very short functions or loops. There is also the problem that we currently don't reset the yield counter if there is a context switch by some other means (e. g. a blocking system call), which may lead to a fiber only receiving very little execution time, because it happens to be scheduled between a blocking system call and the yield counter reaching the yield threshold very soon after that.

Most importantly, our heuristic fails to address the case

	thresh=5	**thresh=10^8**
cycles	349,047,740,311 4,208,791.920 GHz	372,017,103,625 4,207,196.051 GHz
... stalled (frontend)	304,005,243,805 87.10 %	364,678,743,152 98.03 %
... stalled (backend)	251,376,381,280 72.02 %	335,560,362,901 90.20 %
instructions stalled	152,960,803,893 0.44 IPC 1.99 CPI	32,000,778,333 0.09 IPC 11.40 CPI
branches	32,000,182,047 385,855,836.000 M/s	3,200,177,626 36,191,278.680 M/s
... missed	3,357,845 0.01 %	8,813 0.00 %
L1-dcache (loads)	67,204,009,583 810,340,993.127 M/s	12,804,314,318 144,805,870.782 M/s
L1-dcache (misses)	5,333,447,255 7.94 %	2,365,264,383 18.47 %
LL cache (loads)	3,510,962,527 42,334,927.315 M/s	2,146,423,707 24,274,220.879 M/s
LL cache (misses)	—	—

Table 1: Comparison of linux-perf data for two variants of "100 threads chasing pointers"

where a fiber would make a long-running library call, where the library was not instrumented by our LLVM Pass. As there are no yields inserted in the library, fairness is not guaranteed: yields would only happen if the library made a blocking system call. A possible approach to mitigate this is to have the LLVM Pass detect calls to external functions, and insert code that makes the application fall back to using IPC mechanisms for interrupting the fiber for the duration of the library call. Depending on the density of library calls used in the application code, this might however cause a major overhead (as every single library function would add additional work before and after the call), not to mention the fact that the mere act of switching IPC mechanisms on and off is already relatively heavyweight on its own.

Another import aspect that we have not tested is the impact on latency: setting a high yield threshold will result in faster total execution times, but also result in higher latencies, as fibers yield less frequently. A latency benchmark should be performed to test that aspect.

Providing a compatibility API with POSIX threads would also allow us to build real-world applications with fibers, to see how their performance changes with our method.

5. RELATED WORK

Arachne [7] is a userspace runtime that manages fibers with a focus on distributing them to multiple cores and isolated from other kernel threads. The latter is achieved by the *Core Arbiter*, a subcomponent that splits the CPU cores into multiple `cpusets`, and assigns threads running fibers to dedicated cores, while preventing the kernel from preemp-

tively scheduling other threads on that core and degrading the performance of the fibers.

Although we consider separating fibered threads from regular threads to be rather elegant, Arachne does not specially handle blocking system calls; it assumes and encourages that fibers only make non-blocking system calls. Furthermore, the internal structure of the fiber scheduler limits the number of simultaneously existing fibers in a thread/core to only 56, which does not fit our usecase and the initially mentioned scaling problem we are trying to solve.

Boost.Fiber [4] is another fiber implementation written in (and intended for use with) C++. Like libfiber, it can handle an arbitrary number of fibers. It also documents how to make system calls in a pseudo-blocking fashion; however, it appears to be more involved[2] than the mechanism provided by libfiber. Nevertheless, it may be worth testing how our benchmark performance changes if used with Boost fibers.

6. CONCLUSION

Based on the results obtained from our microbenchmarks, it is unfortunately not possible to draw any final conclusions about the usability of our approach. Yet, despite the inconsistencies, most results indicate that performance (at least the throughput) is not substantially worse than with kernel threads, and our method could thus be used to write scalable server applications with fibers, in a both intuitive and convenient way.

Information gained from latency benchmarks might also tell us if this approach can be used to remove interrupts in ZygOS [6].

7. ACKNOWLEDGEMENT

First and foremost, I would like to thank Marios Kogias for supervising my work, introducing me into the world of cooperative multitasking and userspace threads, always supplying me with all the necessary resources on that topic, and giving me feedbacks and ideas for how to approach this project (and also these vastly irritating results). Furthermore, I thank Prof. Edouard Bugnion for letting me do this project in the EPFL Datacenter Systems Laboratory. Many thanks also go to Adrien Ghosn for his input and repeatedly taking the time to discuss the project and giving me various valuable tips for the benchmarks; and the DCSL team overall for participating in the project presentation and discussion, and raising questions about the benchmarks (and their results).

8. REFERENCES

[1] Clang. `https://clang.llvm.org/`.

[2] Anderson, T. E., Bershad, B. N., Lazowkska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for User-level Management of Parallelism. *ACM Transactions on Computer Systems 10*, 1 (Feb 1992), 53–79.

[3] Goodspeed, N., and Kowalke, O. Distinguishing coroutines and fibers. *ISO/IEC JTC1/SC22 – Programming languages and operating systems – C++ (WG21)* (May 2014).

[4] Kowalke, O. Boost.Fiber. `https://www.boost.org/doc/libs/1_69_0/libs/fiber/doc/html/index.html`, 2013.

[5] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Mar 2004).

[6] Prekas, G., Kogias, M., and Bugnion, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. *SOSP '17* (Oct 2017).

[7] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. Arachne: Core-Aware Thread Management. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* (Oct 2018), 145–160.

[8] Watling, B. libfiber. `https://github.com/brianwatling/libfiber`.

[2] `https://www.boost.org/doc/libs/1_69_0/libs/fiber/doc/html/fiber/nonblocking.html`