Why Multi-Threading Should No Longer Be a DIY Job

Jannes Timm
TU Delft
Delft, Netherlands
J.N.Timm@student.tudelft.nl

Jan S. Rellermeyer
Leibniz University Hannover
Hannover, Germany
rellermeyer@vss.uni-hannover.de

ABSTRACT

Parallel and concurrent design has become paramount to achieving good performance in the era of multi-socket and multi-core architectures. Thread pools often play a crucial role in system software but finding the right number of threads is a tedious task, which often falls on the users to fine-tune this parameter. The best setting is not only different per application, but can also vary with the workload. We believe this decision should be made at the level of the operating system and offered as a service to benefit from metrics the kernel collects on the application performance. In this paper, we present a first step towards this vision in the form of a self-adaptive thread pool which uses OS-metrics to automatically control the optimal number of threads over time.

KEYWORDS

operating systems, multi-threading, thread pools

1 INTRODUCTION AND BACKGROUND

The end of Moore's law has also marked the end of the convenient delivery of additional of computational power through an increase of clock frequency by every new chip generation [9, 32]. Instead, we have gotten used to a new world in which we need to invest significant effort into structuring our software in ways that leverage the capabilities of new CPUs [16]. For instance, with entering the multi-core era, the clock frequency of CPUs actually temporarily decreased and the only way to benefit from the added resources was to effectively parallelize the programs and make them multi-threaded [30, 31], otherwise software could have experienced a slow-down [36]. The problem of thread-based parallelization was indeed not new and the discussions on how to best achieve this are as old as the first SMP machines [8, 26, 28]. However, while those were primarily reserved for the most demanding server-based applications, the advent of multi-core architectures for even mainstream CPUs (now all the way to mobile phones) has turned this into a universal problem.

FGBS '22, March 17–18, 2022, Hamburg, Germany

https://doi.org/10.18420/fgbs2022f-02

Several architectural patterns exist that provide blueprints for building scalable systems, some avoiding the use of threads (e.g., the fork-join pattern [23] or the single-threaded event-driven architecture [8]). Multi-threaded designs often closely resemble the *producer-consumer pattern* with producer(s) enqueuing work items and the worker threads *consuming* the work by performing the necessary operations to serve the request. Since using an unbounded number of threads on the consumer side has the potential for overwhelming the system, the pattern is typically implemented with either a fixed-sized or limited-sized thread pool. Re-using the same threads through a pool also eliminates the overhead in the operating system of repeatedly allocating new threads. Given the prominence and importance of thread pools in modern software systems, it is surprising that no standardized solution (e.g., as an extension of POSIX threads) and only few reusable building blocks [14, 21, 22, 24] for thread pools exist. Instead, most software packages implement their own thread pools and consequently make their own autonomous decisions on how to scale the pool sizes (e.g., web servers [10], NO-SQL DBs [15], or message queues [25]).

In most cases, the thread pool size and/or limit is set to match the number or a multiple of the physically available cores as a default and then configuration parameters are exposed to the user for overriding this setting. For best concurrency and overall efficiency, finding the right thread pool size to maximize application performance is critical but also a tedious effort [17]. In many cases, however, it is ultimately a futile effort because the workloads handled by the application are not static. They can exhibit different phases that are bottlenecked by different resources. For instance, in some cases it is more advantageous to decrease the number of threads when the workload becomes I/O-bound [13, 29] because adding more threads only leads to more contention. Some modern systems have identified this issue and use dedicated thread pools, e.g., for I/O in Redis [3], RabbitMQ [33], and MongoDB [5], or more fine-grained for different tasks in RocksDB [2], unfortunately with the net effect of even increasing the tuning effort.

The severity of the problem in practice lead to several efforts with the goal of effectively taking the decision of threading out of the hands of programmers or operators. Von Behren et al. were among the first to criticize the underlying virtual processor model of threading [34] and instead

presented Cappricio [35], a user-level threading library built around a central resource-aware scheduler which adapts the number of threads according to the global system utilization. Apple introduced a task-based abstraction for concurrent processing in OS X through the Grand Central Dispatch [27]. This centralized OS service requires developers to submit tasks to a single system-wide thread pool, thereby relieving the developer and operator from manual tuning and ensuring fairness across multiple applications.

In this paper, we argue that we should follow this trend and make the handling and tuning of threading a centralized service of the OS, instead of leaving it to the programmer of each application to make unilateral decisions. In practice, this leads to massive tuning efforts and friction when the workload (through inherent dynamics) or the environment (e.g., through containerization or co-locating of multiple workloads on one machine) changes. In view of the increasing heterogeneity of modern computers and the growing importance of off-CPU resources for efficient data flow, we believe that better abstractions over concurrent flows of execution can provide crucial leverage for future-proofing applications for new architectures.

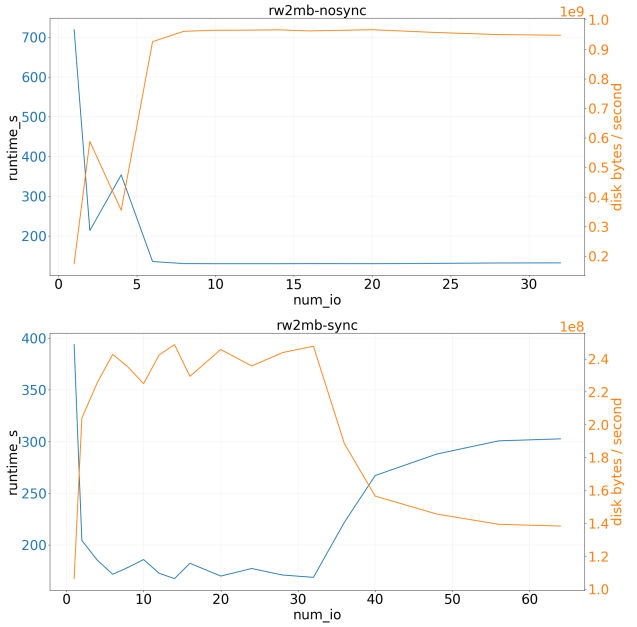

Figure 1: *rwchar* rate & runtime over pool size

2 TOWARDS A SELF-ADAPTING THREAD POOL SERVICE

Offering thread execution as a service requires the OS to provide interfaces for submitting tasks, implementing a central controller for the different thread pools, and employ mechanisms to monitor the progress of the individual application as well as the state of the computer system. This is required to decide on the scheduling of threads based on a global optimization strategy but also to adjust the total number of threads that is dedicated to each individual application.

We consider this to be an open research challenge; even Grand Central Dispatch as the closest existing and practically adopted concept is only used for very specific tasks like UI components and has not been able to substitute handmade thread pools in applications. However, we have gained significant experience in replacing thread pools in individual applications through a system-wide and portable abstraction that fits closely into the `pthread` model but automatically adapts the pool size.

For the common use case of thread pools executing a restricted set of disk-I/O [1] bound jobs (e.g asynchronous runtime libraries such as Tokio [20] and NodeJS [19] use a thread pool solely for disk-I/O related operations), we present a solution approach solely based on information provided by the OS. The adaptive thread pool we propose maximizes throughput (/minimizes total runtime) while minimizing the amount of worker threads.

Our approach integrates a controller that determines scaling actions based on a single target metric computed from OS-metrics into a thread pool to make it adaptive. The controller has to be periodically invoked in order to update internal state (such as target metric's value over the last interval and the current controller phase). It offers a single method for both updating itself and reporting the amount of worker threads that need to be terminated or newly spawned. This method may be invoked by the worker threads themselves before execution of jobs or an extra manager thread may be spawned that sleeps for regular intervals and calls the controller method in between.

Target metric For disk-I/O workloads the disk throughput, i.e., bytes read & written per second, is presumably a good indicator for general throughput (jobs completed per second). We used a few workloads (description follows in section 3.1) that consist of reading and writing files to disk to confirm that disk throughput is indeed highly correlated with general throughput and thus also with total runtime of a workload. On Linux a logical disk throughput metric is exposed through through the `/proc` virtual filesystem, the `rchar` value for a given thread equals the total amount of bytes a thread has requested to read since creation, conversely the `wchar` value equals written bytes [12].

We define the *rwchar* rate over an interval $[t_1, t_2]$ to be: $R = (\sum_{w \in W}(rchar_{w,t_2} + wchar_{w,t_2}) - (rchar_{w,t_1} + wchar_{w,t_1}))/(t_2 -$

[1] Our experimentation included modern storage devices like SSDs and NVMes and we use the traditional term "disk-I/O" liberally for all modern forms of storage

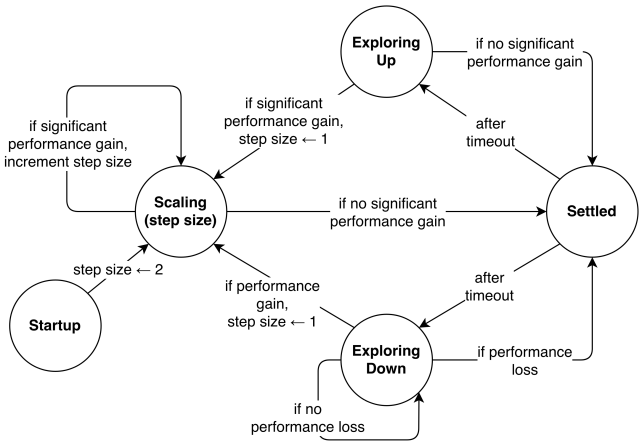

Figure 2: States of the controller

t_1), where W is the set of workers in the pool at t_2 and $rchar_{w,t}$ and $wchar_{w,t}$ are the reported `rchar` and `wchar` values respectively for the worker thread w at time t (if the worker thread w does not exists yet at t, then $rchar_{w,t} = wchar_{w,t} = 0$). Figure 1 shows the average *rwchar* rate in bytes/second over the whole execution (orange) and the total runtime in seconds (blue) for different pool sizes for two read-write workloads (both experiments were repeated 3 times, the mean values are reported here).

For both workloads the total runtime decreases with an increasing amount of worker threads, reaching an optimal size at 6 and 14 worker threads. The function average *rwchar* rate over pool size is highly inversely correlated with the runtime over pool size function, therefore maximizing the *rwchar* rate should be equivalent to maximizing general overall throughput (as in jobs completed per second). Even though total runtime does not monotonically decrease up to the optimal amount of threads, we show that a controller that uses a hillclimbing approach that maximizes the *rwchar* rate shows promising results.

Scale Controller We propose a feedback-based adaptive thread pool that uses a controller to adapt pool size based on observed *rwchar* rate. The controller switches between different states depending on the *rwchar* rate it observes, attempting to maximize the throughput while keeping the pool size low. Generally it will be either in a phase of *scaling* the thread pool down/up, *settled* on a fixed size, or *exploring* smaller/bigger pool sizes. The controller state is reevaluated at fixed intervals, according to the *rwchar* rate over the respective last interval. Figure 2 illustrates the whole state machine that defines states and state changes of the controller.

The algorithm is based on the assumption of increasing throughput as the pool size is increased up to some threshold where throughput stagnates or drops. So the controller will remain in the Scaling state, where the pool size is increased after every interval, until no significant increase in *rwchar* is observed anymore. Then the controller switches to the Settled state where pool size is not changed anymore. However, as we can see for the top workload shown in Figure 1 the function runtime over pool size is not monotonically decreasing up to the optimal pool size, so the pool size may not be optimal when entering the *Settled* state. Secondly, the optimal pool size most likely changes throughout the workload, the load may change, the jobs and their characteristics may change.

In order to address this, the controller enters the *Exploring* state after a timeout or if the *rwchar* rate changed significantly compared to the previous interval. With the addition of the explore mechanism the controller continuously adjusts the pool size according to the current logical disk throughput.

The algorithm has two core parameters that have a big impact on the scaling behavior; the interval length over which the *rwchar* is measured and which also determines the frequency of scaling actions, and the stability factor which determines the sensitivity to changes in *rwchar* rate, i.e., the relative increase/decrease considered to be significant. For the experiments in the following section an interval length of 1500ms and a stability factor of 0.9 (10% increase/decrease of target metric are considered significant) were used, as this combination delivered good performance over all workloads.

3 PRELIMINARY RESULTS

3.1 Workloads & Setup

We use three workloads to illustrate the correlation of *rwchar* rate with runtime and to experimentally evaluate the proposed adaptive thread pool. All workloads consist of a set of homogeneous jobs that are all submitted at once to the thread pool at the beginning of execution. They read a single 2Mb file into memory, then write the whole file to disk. The second workload then uses fsync to sync the file to disk, we call it the sync workload (and the first workload the nosync workload). The third workload consists of two phases, the first batch of jobs submitted are nosync jobs, the second batch are sync jobs, so the pool first executes all nosync jobs to completion and then continues with all sync jobs. In the final experiment, we run a write-heavy workload on RocksDB with our scale controller integrated into its flush pool responsible for flushing files to disk (by use of fsync).

Every experiment was run 3 times, the graphs show the mean over the 3 executions, error bars show the standard deviation. All experiments were run on a machine with an

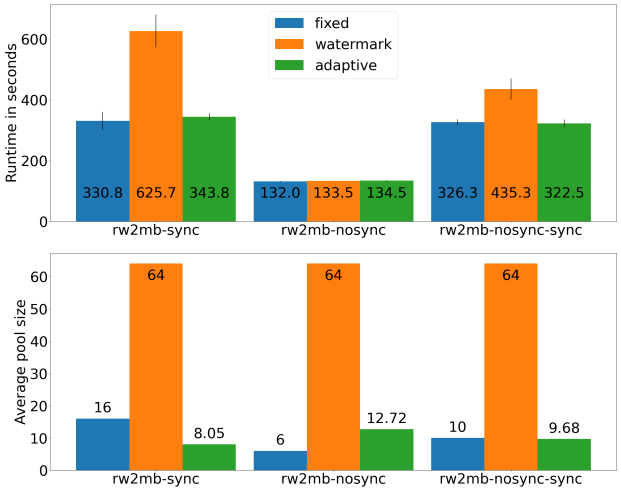

Figure 3: Avg runtime & pool size for fixed / watermark / adaptive

Intel i5-6500 CPU with 4 cores at 3.2GHz clock rate and a 6MB cache, 16GB DDR4 memory and an SSD.

3.2 Experimental results

We compare the adaptive pool to the optimal fixed-size pool and a *watermark scheme* pool with a thread limit of 64. The watermark scheme is a commonly implemented adaptive scheme that keeps the pool size between a minimum and maximum number of worker threads, scaling up when more jobs are available and terminating workers after some period of being idle (e.g Java's ThreadPoolExecutor and MariaDB's thread pool). The optimal pool size for the fixed pool is determined experimentally, the smallest size with maximum 3% longer runtime than the minimum is chosen as optimal as it results in smaller pool size with near-minimum runtime.

The controller of the adaptive pool is initialized with an interval length of 1500ms and a stability factor of 0.9, which was overall a good configuration for all the workloads tested. Figure 3 shows the average runtime (top graph) and average pool size (bottom graph) for the optimal fixed-size (blue), watermark (orange), and adaptive pool (green). The left barplots are the results of the sync workload (20000 jobs), the middle for the nosync (30000 jobs) one, the right for the 2-phase nosync-sync (30000+10000 jobs).

Sync workload The adaptive pool has slightly lower average runtime than the fixed-size pool, whereas the watermark pool performs much worse. Thus, this workload serves to highlight the problem with the watermark scheme when used for disk I/O only jobs, it not only creates excessive amount of worker threads, but may result in significant throughput loss due to the unnecessary extra disk contention.

Nosync workload The watermark pool's runtime is just less than 1% higher than the fixed-size pool's, the adaptive pool's runtime in turn is less than 1% higher than the watermark pool's. While the differences in runtime are insignificant, the watermark's average pool size is more than 10 times higher than the optimal pool size, the adaptive pool size averages at around twice the optimal pool size.

Nosync-Sync workload The adaptive pool slightly outperforms the fixed-size pool here and the watermark pool again performs much worse, due to the negative effect on runtime of overscaling during the phase of sync read-write jobs. The average size of the adaptive pool of 9.7 is almost equal to the optimal fixed size of 10. Especially for multi-phase workloads, where optimal pool size may vary drastically between different phases, a fixed-size scheme (even when tuned perfectly) is most likely not able to reach the best throughput.

Figure 4 shows the scaling behavior of the adaptive pool during execution of the 2-phase workload. The *rwchar* rate over time is depicted in red, the pool size over time is depicted in blue. During the first phase the controller quickly scales up the pool size initially and then slowly reduces pool size, due to *rwchar* not decreasing after scaling down. The *rwchar* rate stays roughly constant around 1Gb/sec until the start of the second phase, where it drops sharply and then stabilizes around 150Mb/sec for a short while. While mostly stable with small fluctuations in the first phase, the *rwchar* rate shows a lot of variation with only short periods of stability in the second phase. During the period of the most fluctuations from around 230-270 seconds the controller scales down the pool to sizes ranging from 1-7. The high rate of manual fsync operations presumably leads to excessive disk contention and periodic drops and peaks in logical disk throughput.

Write-heavy workload - RocksDB We compare the runtime for a sequential key insertion workload that is part of RocksDB's benchmarking tools (fillseq) for the recommended[2] flush pool size (2, in blue), the experimentally determined optimal size (12, in red) and the adaptive version making use of our scale controller (in green for the same parameter values as previously and olive for a shortened interval length of 1000ms). Figure 5 shows these results, the adaptive pool with the previous controller configuration beats the default but is outperformed by the optimally sized

[2]by the official wiki at https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning

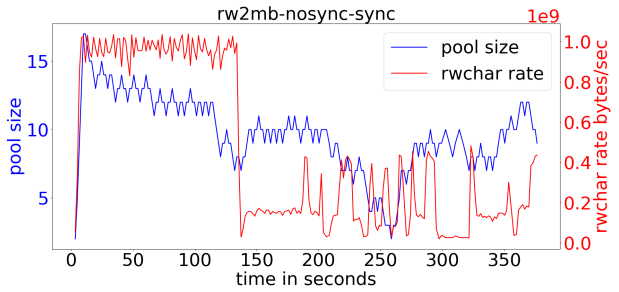

Figure 4: Adaptive pool - pool size & *rwchar* rate during execution

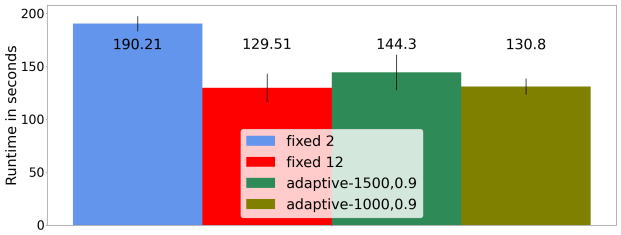

Figure 5: Avg. runtimes for fillseq workload, fixed/adaptive

pool, whereas the adaptive pool with a shorter controller scale interval of 1000ms approaches optimal performance.

4 RESEARCH CHALLENGES AND DISCUSSION

The most challenging and crucial aspect of an OS-feedback based pool size adaption scheme is the OS-metric or the OS-metrics that are used as performance indicators. For workloads restricted to specific jobs, such as the disk I/O-bound workloads we investigated, it is possible to use a single metric that is highly correlated with overall throughput. However, a highly correlated metric such as the *rwchar* rate may still be fluctuating heavily, as observed in the second phase of the workload execution seen in Figure 4. This is problematic for a controller-based approach, which is intrinsically sensitive to fluctuations in the target metric.

An even more effective target metric would be predictive of expected throughput, so the controller can reduce pool size preemptively to avoid throughput drops as a result of disk contention. Alternatively a second metric that captures contention may be used to detect the onset thereof, triggering the controller to scale down once a specific threshold is reached.

In order for the controller-based approach to be applicable for general workloads a global kernel-side implementation with jobs at the granularity of single system calls may be the most effective approach, as scheduling of jobs and adjustment of concurrency levels can be performed according to resource usage of a job and current resource utilization. The controller would have the most complete knowledge over system-wide outstanding jobs and system resources to make these decisions. The recently introduced `io_uring` interface to Linux offers a mechanism for submission of asynchronous system calls and collection of results, so there is already infrastructure in place for a kernel-side implementation that is exposed to the user at a system call granularity [1, 6]. `io_uring` is already used in concurrency libraries such as Glommio to replace dedicated I/O thread pools, shifting the responsibility of concurrent execution to the kernel, where an adaptive scheme to determine concurrency levels would benefit all user applications [7].

Scaling the single-application dynamic thread pool to an OS-wide service requires a deep integration into the resource management and the scheduler. Doing so, however, would provide much better control over QOS requirements and fairness between multiple concurrent application. We expect positive synergies (but also additional research challenges) with container frameworks but also with unikernel architectures which both through their own mechanisms would extend the scope from isolated processes to multi-process applications and services.

5 CONCLUSIONS

In this paper, we have argued why multi-threading is one of the most important methods for achieving performance on modern multi-core computers but also a burden for programmers that not only requires plenty of deliberation when designing the application but also a great amount of tuning to reach the promised performance in practice. Thread pools are an important part of many systems and turning them into self-adaptive, reusable components could provide a critical improvement over the current *DIY approach*. Ultimately, however, we believe that the decision on threading should be fully centralized in the operating system and offered as a service to benefit from the existing metrics that the kernel already collects on the application performance and at the same time enable global optimization.

We decided to not challenge the thread as the vehicle for concurrent execution. While alternative abstractions exist (e.g., Futures [4], Promises [18], Coroutines [11], etc.), they typically impose structural limitations on programs (e.g., a more asynchronous structure) and often internally use threads to implement these abstractions. Therefore, we believe that threads will always play a role in the design of performance-critical scalable applications, either explicitly or hidden behind libraries.

REFERENCES

[1] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. [Online; accessed 3. Feb. 2021].

[2] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 209–223.

[3] Josiah L Carlson. 2013. *Redis in action*. Manning.

[4] Arunodaya Chatterjee. 1989. Futures: a mechanism for concurrency among objects. In *Supercomputing'89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. IEEE, 562–567.

[5] Kristina Chodorow. 2013. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.".

[6] Jonathon Corbet. 2020. The rapid growth of io_uring [LWN.net]. https://lwn.net/Articles/810414. [Online; accessed 3. Feb. 2021].

[7] Glauber Costa. 2020. glommio. https://github.com/DataDog/glommio. [Online; accessed 9. Dec. 2020].

[8] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. 2002. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 186–189.

[9] Magnus Ekman, Fredrik Warg, and Jim Nilsson. 2005. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News* 33, 1 (2005), 144–147.

[10] James C Hu, Irfan Pyarali, and Douglas C Schmidt. 1997. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. In *GLOBECOM 97. IEEE Global Telecommunications Conference. Conference Record*, Vol. 3. IEEE, 1924–1931.

[11] Gilles Kahn and David MacQueen. 1976. Coroutines and networks of parallel processes. (1976).

[12] Michael Kerrisk. 2020. proc(5) - Linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html. [Online; accessed 4. Jan. 2021].

[13] Sobhan Omranian Khorasani, Jan S Rellermeyer, and Dick Epema. 2019. Self-adaptive Executors for Big Data Processing. In *Proceedings of the 20th International Middleware Conference*. 176–188.

[14] Ronald Kriemann. 2004. Implementation and Usage of a Thread Pool based on POSIX Threads. *Max-Planck-Institue for Mathematics in the Sciences, Inselstr* (2004), 22–26.

[15] CU Kumarasinghe, KLDU Liyanage, WAT Madushanka, and RACL Mendis. 2016. Performance comparison of NoSQL Databases in pseudo distributed mode: Cassandra, MongoDB & Redis.

[16] James Larus. 2009. Spending Moore's dividend. *Commun. ACM* 52, 5 (2009), 62–69.

[17] Yibei Ling, Tracy Mullen, and Xiaola Lin. 2000. Analysis of optimal thread pool size. *ACM SIGOPS Operating Systems Review* 34, 2 (2000), 42–55.

[18] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM Sigplan Notices* 23, 7 (1988), 260–267.

[19] The Node.js Maintainers. 2021. Node. https://github.com/nodejs/node. [Online; accessed 4. Jan. 2021].

[20] The Tokio Maintainers. 2020. Tokio. https://github.com/tokio-rs/tokio. [Online; accessed 7. Dec. 2020].

[21] Anton Malakhov. 2016. Composable multi-threading for Python libraries. In *Proc. 15th Python Sci. Conf.* 15–19.

[22] Kevin T Manley. 1998. General-purpose threads with I/O completion ports. *C/C++ Users Journal* 16, 4 (1998), 75–83.

[23] Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. 1999. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th international conference on Supercomputing*. 294–301.

[24] Scott Oaks and Henry Wong. 2004. *Java Threads: Understanding and Mastering Concurrent Programming*. " O'Reilly Media, Inc.".

[25] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C Schmidt. 2001. Evaluating and optimizing thread pool strategies for real-time CORBA. *ACM SIGPLAN Notices* 36, 8 (2001), 214–222.

[26] Dennis M Ritchie and Ken Thompson. 1978. The UNIX time-sharing system. *Bell System Technical Journal* 57, 6 (1978), 1905–1929.

[27] Kazuki Sakamoto and Tomohiko Furumoto. 2012. Grand central dispatch. In *Pro Multithreading and Memory Management for iOS and OS X*. Springer, 139–145.

[28] Jerome Howard Saltzer. 1966. *Traffic control in a multiplexed computer system*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[29] Charles E. Skinner and Jonathan R. Asher. 1969. Effects of storage contention on system performance. *IBM Systems Journal* 8, 4 (1969), 319–333.

[30] Angela C Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. 2010. Parallelism via multithreaded and multicore CPUs. *Computer* 43, 3 (2010), 24–32.

[31] Herb Sutter. 2005. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal* 30, 3 (2005), 202–210.

[32] Thomas N Theis and H-S Philip Wong. 2017. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering* 19, 2 (2017), 41–50.

[33] Alvaro Videla and Jason JW Williams. 2012. *RabbitMQ in action: distributed messaging for everyone*. Manning.

[34] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. 2003. Why Events Are a Bad Idea (for High-Concurrency Servers).. In *HotOS*. 19–24.

[35] Rob Von Behren, Jeremy Condit, Feng Zhou, George C Necula, and Eric Brewer. 2003. Capriccio: scalable threads for internet services. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 268–281.

[36] Dan Woods. 2009. Multi-Core Slow Down. *Forbes* (2009).