Intra-process Caching and Reuse of Threads

Dave Dice
Oracle Labs
first.last@oracle.com

Alex Kogan
Oracle Labs
first.last@oracle.com

Abstract
Creating and destroying threads on modern Linux systems incurs high latency, absent concurrency, and fails to scale as we increase concurrency. To address this concern we introduce a process-local cache of idle threads. Specifically, instead of destroying a thread when it terminates, we cache and then recycle that thread in the context of subsequent thread creation requests. This approach shows significant promise in various applications and benchmarks that create and destroy threads rapidly and illustrates the need for and potential benefits of improved concurrency infrastructure. With caching, the cost of creating a new thread drops by almost an order of magnitude. As our experiments demonstrate, this results in significant performance improvements for multiple applications that aggressively create and destroy numerous threads.

CCS Concepts → Software and its engineering → Multithreading; Concurrency control;

Keywords Threads, Concurrency Control

1 Introduction
To mitigate the costs of thread creation and destruction, we implement a user-mode process-local cache of idle threads. These are threads that have logically terminated, but which we capture and retain for subsequent reuse. Threads that would normally exit to the kernel are now retained and reused, sparing the cost of creating new threads in the future. Using various benchmarks, we show that our approach can yield improved performance.

2 Implementation
We implemented our proof-of-concept cache as an LD_PRELOAD interposition module which intercepts POSIX [1, 3] pthread_create, pthread_exit, pthread_detach and pthread_join calls, reimplementing those operators as necessary to provide a thread cache. Using LD_PRELOAD interposition allows us to enable or disable the cache by simply setting an environment variable, and allows us to use unmodified application executables. Logically terminated threads are retained on a local Idle list. When creating new threads, our implementation first attempts to allocate a thread from the idle list. Failing that, it falls back to the underlying pthread_create API. The idle list is maintained in a LIFO fashion to better leverage residual cache residency and scheduling affinity.

3 Performance Evaluation
All experiments were run on an Oracle X5-2. The system has 2 sockets, each populated with an Intel Xeon E5-2699 v3 CPU running at 2.30GHz. Each socket has 18 cores, and each core is 2-way hyperthreaded, yielding 72 logical CPUs in total. The system was running Ubuntu 20.04 with a stock Linux version 5.4 kernel, and all software was compiled using the provided GCC version 9.3 toolchain at optimization level -O3. 64-bit C or C++ code was used for all experiments. Factory-provided system defaults were used in all cases, and Turbo mode was left enabled. In all cases default free-range unbound threads were used.

In Table-1 we compare the performance of the default system against the same system with our cache activated. The Spawn benchmark creates 32 concurrent threads. We use C++ std::thread where the GNU/Linux implementation maps each such thread 1:1 to an underlying native POSIX pthread. Each of those threads loops, creating an additional thread and then waiting for that thread to join. The threads created in the loop exit immediately. Each of the 32 “creator” threads is independent and there is no communication or synchronisation within this set of threads. At the end of a 10 second measurement interval the benchmark reports the number of threads spawned per second. We report the median of 7 runs. (We use the median of 7 independent runs for all results shown in this table). The stdasync benchmark is similar but uses the newer C++ std::async construct instead of explicit threads. GNU/Linux implements each std::async instance as a native POSIX thread.

HuggingFace is a machine learning BERT language model [2] inference benchmark provided with the huggingface transformers package [1] running on Pytorch version 1.5. As used by PyTorch, the underlying GNU OpenMP

[1] https://huggingface.co/transformers/benchmarks.html
[2] The command line was as follows: python3 examples/benchmarks.py --torch --models bert-base-cased --no_memory --batch_sizes 1 --slice_sizes 64
(libgomp) spawns and terminates threads rapidly, impacting performance of the inference benchmark.

All the remaining benchmarks are from the Inncabs[4] benchmark suite, which is designed to measure the performance of the std::async construct. We obtained the source code from https://github.com/PeterTh/inncabs 3. The implementation of the C++ std::async construct defers creating an underlying thread until the returned promise is evaluated. All Inncabs benchmarks report elapsed time in milliseconds.

In Figure-1 we compare the scalability of thread creation using the spawn benchmark, above, varying the number of threads on the X-axis and showing aggregate thread creation rates on the Y-axis. For clarity and to convey the maximum amount of information to allow a comparison the algorithms, the Y-axis is logarithmic. Default reflects default thread creation while cache reflects performance when the thread cache is enabled. As we can see, the cache decreases latency and also improves scalability. At one thread, the benchmark serves as simple measure of unloaded latency for creating and destroying threads. The cache improves performance by 5.9 times at 1 thread.

| Benchmark      | Default | With Cache | Units            |
|----------------|---------|------------|------------------|
| Spawn          | 158940  | 1399817    | threads/second   |
| std::async     | 132993  | 1354640    | threads/second   |
| HuggingFace    | 94      | 41         | milliseconds     |
| Alignment 100  | 872     | 314        | milliseconds     |
| SparsaLRU      | 620     | 362        | milliseconds     |
| Sort 100000    | 770     | 41         | milliseconds     |
| Health small   | 60669   | 2713       | milliseconds     |
| Floorplan      | 3431    | 974        | milliseconds     |
| FFT 100000     | 3860    | 484        | milliseconds     |

Table 1. Performance Comparison

4 Future Directions: Cache Retention Policies

Our preferred implementation simply caches all terminated threads. If the application has a maximum of N concurrently live threads, then the worst case retention in the idle cache is N − 1 threads. Other more refined policies are possible, such as (a) clamping the number of threads in the cache; (b) aging out and culling threads idle threads that have not run in some tunable period, or (c) bounding the thread · seconds integral of the list and culling until the target is limit is reached. To reduce the memory footprint of cached threads, the implementation might also use the `advise(MAP_DONTNEED)` advisory service to inform the operating system that the dirty pages on an idle stack may be reclaimed and replaced lazily by demand-zero-fill pages.

All the policies above are process-local, but more sophisticated system-wide polices may be more effective, where processes cooperate, possibly informed by a measure of system memory pressure, to decide to trim their caches.

References

[1] David R. Butenhof. 1997. Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc.
[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. CoRR abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805
[3] IEEE and The Open Group. [n. d.]. The Open Group Base Specifications Issue 7, 2018 edition; IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008).
[4] P. Thoman, P. Gschwandtner, and T. Fahringer. 2015. On the Quality of Implementation of the C++11 Thread Support Library. In 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. https://doi.org/10.1109/PDP.2015.33

Figure 1. Thread Creation Scalability

3A number of the Inncabs benchmarks failed in default mode and were thus excluded.
Appendix : Additional Remarks

Various strategies to accelerate thread creation are possible. As noted above an implementation might cache threads proper. Caching of thread stacks is also a viable approach and has been used in the Solaris Operating Environment. An implementation might also avoid recycling threads, but instead keep an anticipatory cache of ready-to-run standby threads available for immediate dispatch.

Caching threads via an LD_PRELOAD module is not strictly sound, as, for instance, thread-local storage elements are not reset when a physical thread recycles. While we have not observed any errors related to this concern, we note that an implementation that caches threads is best implemented in the threading library proper which has access to the thread local storage elements and is able to reset those values in the expected fashion. Recycling threads also allows pthread_self thread identifiers to recycle rapidly, possibly causing latent application errors (related to holding potentially stale thread identifiers) to manifest more frequently.

We currently implement the idle list is a stack protected by a lock. Using per-NUMA node idle lists appears to be a valid implementation optimization. In addition, since the list can be subject to high traffic, a lock-free stack may be more appropriate, or a “half lock-free” approach where push operations are implemented via an atomic compare-and-swap (CAS) loop and pop are also implemented via atomics but protected by a lock to avoid the A-B-A pathology.