RiSE: Relaxed Systems Engineering?

Christoph Kirsch
University of Salzburg
Application:
>10k #threads, producer/consumer, blocking

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k threads, producer/consumer, blocking

allocate memory

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k #threads, producer/consumer, blocking

allocate memory

access memory

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k #threads, producer/consumer, blocking

allocate memory
access memory
share memory

Hardware:
CPUs, cores, MMUs, memory, caches
Hardware:
CPUs, cores, MMUs, memory, caches

Application:
>10k #threads, producer/consumer, blocking

allocate memory
access memory
share memory
deallocate memory
Application:
>10k threads, producer/consumer, blocking

allocate memory
access memory
share memory
deallocate memory

throughput

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k #threads, producer/consumer, blocking

allocate 
memory

access 
memory

share 
memory

deallocate 
memory

throughput

scalability

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k #threads, producer/consumer, blocking

allocate memory
access memory
share memory
deallocate memory

throughput
scalability
latency

Hardware:
CPUs, cores, MMUs, memory, caches
Application:
>10k #threads, producer/consumer, blocking

allocate memory
access memory
share memory
deallocate memory

throughput
scalability
latency
memory consumption

Hardware:
CPUs, cores, MMUs, memory, caches
free lists!
free lists!

thread-local
free lists!

core-local

thread-local
free lists!

core-local  thread-local  CPU-local
global

free lists!

core-local  thread-local  CPU-local
lock-based  global  lock-free

core-local  thread-local  CPU-local

free lists!
is it a stack?

lock-based  global  lock-free

free lists!

core-local  thread-local  CPU-local
lock-based  

is it a stack?

global  

free lists!  

is it a queue?

lock-free  

core-local  

thread-local  

CPU-local
“Relaxed” Semantics vs. “Operational” Performance vs. “Denotational” Performance
“Relaxed” Semantics

[PaCT13]
[CF13]
[POPL13]

vs.

“Operational” Performance

vs.

“Denotational” Performance
“Relaxed” Semantics

[PaCT13]  [CF13]  [POPL13]

vs.

“Operational” Performance

vs.

“Denotational” Performance

[RACES12]
The only other allocator with competitive performance is jemalloc. Both allocators are significantly slower in allocation and deallocation performance compared to ptmalloc2. Here, scalloc shows a slightly larger memory consumption than jemalloc, but it is still competitive with llalloc and outperforms the other allocators by two orders of magnitude. Scalloc-eager, however, suffers from a higher degree of locking, which reduces performance more significantly and are about three orders of magnitude faster than ptmalloc2.

In order to evaluate the performance and scalability of the block-pool ACDC, we configured ACDC, similar to the previous experiment, to perform allocations and deallocations of objects with large size range from 16 bytes to 1MB where small objects are more likely to be allocated than large objects.

In order to confirm that the access to the span-pool and, in this experiment, to the load on it increases for an increasing number of threads. Among all threads such that the probability of a remote free is not just thread-locally but also by sharing objects. Differences, yet not just thread-locally but also by sharing objects. Among all threads such that the probability of a remote free is not just thread-locally but also by sharing objects.

Figure 7: ACDC for increasing object sizes

Figure 8: ACDC for an increasing number of threads allocating thread-local objects from a large size range

Figure 9: ACDC for an increasing number of threads allocating shared objects from a large size range
Scalable Concurrent Data Structures:
scal.cs.uni-salzburg.at
github.com/cksystemsgroup/scal

Scalable Concurrent Memory Allocator:
github.com/cksystemsgroup/scalloc

Allocator Benchmarking:
acdc.cs.uni-salzburg.at
github.com/cksystemsgroup/acdc