Brief announcement: 2D-stack - A scalable lock-free stack design that continuously relaxes semantics for better performance

Downloaded from: https://research.chalmers.se, 2022-08-24 18:09 UTC

Citation for the original published paper (version of record):

N.B. When citing this work, cite the original published paper.
ABSTRACT

We briefly describe an efficient lock-free concurrent stack design with tunable and tenable relaxed semantics to allow for better performance. The design is tunable and allow for a continuous monotonic trade of weaker semantics for better throughput performance. Concurrent stacks have an inherent scalability bottleneck due to their single access point for both their operations. Elimination and semantics relaxation have been proposed in the literature to address this problem. Semantics relaxation has the potential to reach monotonically very high throughput by continuously trading relaxation for throughput. Previous solutions could not fully leverage this potential. We suggest a new two dimensional design that can achieve this by exploiting disjoint access parallelism in one dimension and locality in the other within tight accuracy bounds. The behaviour of the algorithm is tightly bound. We compare experimentally to previous work, with respect to throughput and relaxed behaviour observed, on different relaxation and concurrency settings. The experimental evaluation shows that our algorithm significantly outperform all other algorithms in terms of performance, also maintain better accuracy in contrast to other designs with relaxed semantics.

KEYWORDS

Lock-free, Data-structures, Relaxation, Distributed, Concurrency, Parallel, NUMA, Shared-Memory

1 INTRODUCTION

To improve performance scalability of concurrent data structures, recent research has focused on expanding the set of legal behaviours, including; weakening consistency and semantic relaxation for providing trade-offs between scalability and linearizability guarantees. Relaxed semantics definitions including; k-Out-of-Order, k-Lateness and k-Stuttering have been proposed in the literature as interesting relaxation models to consider. Distributing parts and hence access of the data-structure [4, 7, 13], has come out as a frequent technique used to implement relaxation. A given data-structure is split into multiple sub-structures (horizontal) with independent access points to improve on disjoint access parallelism. Operations are distributed over the sub-structures using different scheduling techniques; thread binding [13], random access [7], load-balancing [4], round robin and a combination of others. Until now, most proposed relaxed data-structures are one dimension, horizontal or vertical. horizonal for disjoint parallelism, vertical for locality.

Concurrent stacks, are fundamental data structures that suffer from an inherent scalability bottleneck, due to their single access point for both of their operations. Because of that, semantic relaxation is a promising approach to be used for improving their performance. We propose a lock-free concurrent design for stacks (2D-Stack) that leverages semantic relaxation through exploiting both disjoint access parallelism and locality leading to a two dimensional design. To achieve this, we implement a light weight synchronization mechanism that also maintains tight accuracy bounds. Our design, compared to previous solutions, would not only increase the performance for a given configuration but also give to the application the capability to monotonically trade accuracy for better performance, which was not possible before. We compare our design with known scheduling techniques and other stacks from the literature. Among the scheduling techniques, we compare with; random (random), random choice of two (random-c2) [7] and round-robin (k-robin). From the literature, we compare with segmented (k-segment), elimination (elimination) and Treiber stack (treiber). 2D-stack significantly outperforms previous stack implementations as observed in the experimental evaluation Section 4.

2 RELATED WORK

Concurrent stacks suffer from their inherent single point access bottleneck. In the quest to improve performance scalability, disjoint access strategies have been proposed for designing concurrent stacks including; elimination trees [1, 9], combining funnels [10] and elimination back-off [3, 5]. Elimination back-off implements a collision array in which pop operations try to collide and cancel with concurrent push operations. Such operation pairs create disjoint collisions that are executed in parallel with operations accessing the main stack implementation. Elimination back-off mostly benefits symmetric workloads in which the numbers of push and pop operations are roughly equal, its performance deteriorates when workloads are asymmetric.

Recently, semantic relaxation has been proposed for data-structures that provide trade-offs between scalability and linearizability guarantees. Relaxation introduce an acceptable error within the legal
strict semantics of a given data-structure, i.e. the pop operation of a
relaxed stack can return any of the \(k\) items of the stack. To quantify
this error, relaxed semantic definitions have been introduced \([6, 11]\).
Based on these definitions, a \(k\)-Out-of-Order stack has been pro-
posed in \([6]\), referred to as \(k\)-segment. It is composed of a linked list
of memory segments whose size is defined by \(k\) number of indexes.
The stack items can only be accessed through the topmost segment,
where an operation pushes or pops an item from any \(k\) indexes. A
Push operation adds a new segment if top segment is full whereas a
Pop removes a segment if it is empty and not the last segment.

Other relaxed data-structures proposed in the literature include,
priority queues \([2, 7, 13]\) and distributed queues \([4]\).

### 3 2D-STACK

Our stack is composed of multiple lock-free sub-stacks. An individ-
ual sub-stack is implemented using a linked list whose operations
follow the Treiber stack design \([12]\). Each sub-stack has a unique
descrriptor that keeps track of the sub-stack information including;
pointer to the topmost item and item-counter. A descriptor has a
dedicated memory location accessed through an array (stack-array).
Using a Compare and Exchange (CAE\(^1\)) instruction we can update the descriptor contents in
one atomic step to maintain correctness.

We introduce and implement an operational region (window) in
which an operation can occur. It is defined by two parameters; width and
depth. \(\text{width}\) defines the number of sub-stacks whereas depth
defines the maximum number of items acceptable for a single sub-
stack per window. We also implement a global counter (Global) that
defines the maximum number of items per sub-stack. The window
and Global together help us to tightly bound both relaxation and
execution time.

To perform an operation, a thread searches for a sub-stack based
on the Global. A thread selects a sub-stack, then, compares the
sub-stack item-count with the Global. The thread can then proceed
on the selected sub-stack only if the comparison evaluates to true.
Otherwise the thread has to search for another sub-stack. For each
operation, the thread starts from the previously known sub-stack on
which it succeeded. First the thread tries a given number of random
hops, then switches to round robin until a valid sub-stack is found,
or the thread updates the Global, after failing on all sub-stacks.
The Global is updated in relation to depth. If the thread detects
contention on a sub-stack, a random hop to another sub-stack is
performed. This is to reduce possible contention on consecutive
sub-stacks that might arise from round robin hops.

During the search, the thread validates each sub-stack item-count
against the Global. The item-count must be less than Global for
Push or greater than the difference between Global and depth for
Pop. If the item-count is zero, then the sub-stack is empty. If no
valid sub-stack is found, the Global is updated atomically. Push adds
whereas Pop subtracts a value (\(\text{shift}\)), \(\text{shift}\) must be less than or
equal to depth. Then the search is restarted with a fresh search
count. If a valid sub-stack is found, the thread tries to operate on the
it, on success the sub-stack descriptor is updated otherwise another
sub-stack is searched for, starting from a random index. A successful
Push increments whereas a Push decrements the item-counter by

\[^1\]Compare and Exchange (CAE) atomically compares 16 bytes of memory content and exchanges it with new content on success.

one. Also the topmost item pointer is updated. At this point, a
Push adds an item whereas a Pop returns an item for a non empty
sub-stack or NULL for empty. An empty sub-stack is represented
by a NULL item pointer within the descriptor. As an optimization
strategy, the thread keeps track of the Global for every hop during
the search process, restarting for every Global change detected.

2D-stack is correct with respect to \(k\)-Out-of-Order stack semantics.
The deterministic bound for the relaxation is tunable, con-
trolled by the parameters of our design, given by \(k\)-Out-of-Order
stack semantics, where \(k = (2^\text{shift} + \text{depth})(\text{width} - 1)\).

### 4 EXPERIMENTAL EVALUATION

We evaluate the performance of our implementation together with
other existing stack designs including; the \(k\)-segment relaxed stack
\([6]\), Elimination-Stack (elimination) \([5]\) and Treiber-Stack (treiber)
\([12]\). All experiments run on an Intel Xeon CPU E5-2687W v2 ma-
chine with two 8-core Intel Xeon processors (2 threads per core).
We pin one thread per core, filling one processor at a time up to 16
threads before we switch to hyper-threading. Two NUMA settings
are tested; intra-socket (1 to 8 threads) and inter-socket (9 to 16
threads). Threads select operations uniformly at random (i.e. with
probability 1/2) from \(\text{Pop}\) and \(\text{Push}\) operations. To simulate high
contention, we put no computational load between operations. For
each experiment, the stack is initialized with 32,768 items, run for
five seconds obtaining an average of five repeats. The stack algo-

\[\text{Theorem 1.} \quad \text{2D-stack is linearizable with respect to } \text{k-Out-of-Order stack semantics, where } k = (2^{\text{shift} + \text{depth}})(\text{width} - 1)\]
plotted using logarithmic scales, throughput (solid lines) and error distance (dotted lines) sharing the x-axis. Based on experimental observations and analysis presented in the full paper [8], we select 4P (P stands for number of threads and width = 4P) as the optimal performance configuration for 2D-stack width. In Figure 1, we evaluate the performance of all algorithms, that are linearizable with respect to k-Out-of-Order stack (k-robin, 2D-stack and k-segment), at different relaxation levels. We observe that 2D-stack consistently outperforms the other algorithms. On low degree of relaxation, 2D-stack avoids contention by hopping to another sub-stack on a failed CAE. This highly improves performance compared to k-robin that keeps retrying on the same sub-stack. As the relaxation increases, 2D-stack combines contention avoidance with locality exploitation, a parameter exclusive to the 2D-stack design as explained in [8]. While for the other algorithms the quality reduces almost linearly with the increase in relaxation, 2D-stack maintains good quality with width > 4P (k > 200 for P = 8 and k > 600 for P = 16). At this point, the algorithm switches from horizontal to vertical by increasing the depth. This change has a smaller negative impact on the quality, compared to the other algorithms. 2D-stack continuously trades off quality for throughput by switching between relaxation dimensions for different relaxation levels. k-segment is mostly affected by the high cost of maintaining segments coupled with increased number of hops as relaxation increases.

We now configure the algorithms to obtain high throughput performance for both intra and inter-socket settings, Figure 2. Two “non-relaxed” algorithms elimination and treiber are also included in the experiment to compare the power of relaxation to improve performance compared to other strict semantics efficiency improvement techniques. We generally observe that, 2D-stack is able to maintain the increase in throughput also while increasing the number of threads, even for the NUMA settings. As the number of threads increases, random, random-c2 and k-segment maintain almost constant quality due to the fixed number of sub-stacks. k-robin and 2D-stack vary the number of sub-stacks as the number of threads change. k-robin reduces number of sub-stacks with the increase in number of threads to keep the quality bound, this improves quality but hurts throughput due to the increased contention. Overall, 2D-stack shows a full control to leverage the semantics relaxation to reach very high throughput in a continuous way. A property that was missing from previous solutions.

5 CONCLUSION AND FUTURE WORK

The aim of this work is to design an efficient lock-free stack algorithm that can continuously relax k-Out-of-Order semantics to improve throughput through exploiting disjoint access parallelism and locality. We have achieved this through our two dimension relaxation technique that exploits disjoint access parallelism in one dimension and locality in the other. Our algorithm, 2D-stack, uses also an efficient widows based synchronization that manages to keep the relaxation low without receding significantly performance achieved by disjoint access parallelism and locality. 2D-stack significantly outperformed all the other stack implementations due to its capability to monotonically trade accuracy for better performance. In addition to 2D-stack, we have implemented and tested a set of other possible relaxed stack designs: random, random-c2 and k-robin. The full version of this paper further elaborates on a number of topics treated only briefly here, including complexity analysis, correctness, optimization but also Lock freedom and other experiments that due to space constraints have not been treated at all here [8].

As future work, we are working towards generalizing our design to work for other concurrent data structures.

REFERENCES