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

Downloaded from: https://research.chalmers.se, 2019-02-27 03:30 UTC

Citation for the original published paper (version of record):
Brief announcement: 2D-stack - A scalable lock-free stack design that continuously relaxes semantics for better performance
http://dx.doi.org/10.1145/3212734.3212794

N.B. When citing this work, cite the original published paper.
Brief Announcement: 2D-Stack – A Scalable Lock-Free Stack Design that Continuously Relaxes Semantics for Better Performance*

Adones Rukundo  
Chalmers University of Technology, Sweden  
adones@chalmers.se

Aras Atalar  
Chalmers University of Technology, Sweden  
aaras@chalmers.se

Philippines Tsigas  
Chalmers University of Technology, Sweden  
tsigas@chalmers.se

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 \cite{6, 11} as interesting relaxation models to consider. Distributing parts and hence relaxation introduce an acceptable error within the legal benefits symmetric workloads in which the numbers of push and pop operations are roughly equal, its performance deteriorates as observed in the experimental evaluation Section 4.

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; \textit{k-Out-of-Order}, \textit{k-Lateness} and \textit{k-Stuttering} have been proposed in the literature \cite{6, 11} as interesting relaxation models to consider. Distributing parts and hence access of the data-structure \cite{4, 7, 13}, has come out as a frequent technique used to implement relaxation. A given data-structure is split into multiple sub-structures (\textit{horizontal}) with independent access points to improve on disjoint access parallelism. Operations are distributed over the sub-structures using different scheduling techniques; \textit{thread} binding \cite{13}, random access \cite{7}, load-balancing \cite{4}, round robin and a combination of others. Until now, most proposed relaxed data-structures are one dimension, \textit{horizontal} or \textit{vertical}. \textit{horizontal} for disjoint parallelism, \textit{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 (\textit{2D-Stack}) that leverages semantics relaxation through exploiting both disjoint access parallelism and locality leading to a two dimensional design. To achieve this, we implement a lightweight 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 (\textit{random}), random choice of two (\textit{random-c2}) \cite{7} and round-robin (\textit{k-robin}). From the literature, we compare with segmented (\textit{k-segment}), elimination (\textit{elimination}) and Treiber stack (\textit{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 \cite{1, 9}, combining funnels \cite{10} and elimination back-off \cite{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 \( k\text{-Out-of-Order} \) stack has been proposed in \([6]\), referred to as \( k\text{-segment} \). 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, whereas a \textit{Push} removes a segment if it is empty and not the last segment. The \textit{Pop} operation adds a new segment if top segment is full whereas a \textit{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 individual sub-stack is implemented using a linked list whose operations follow the Treiber stack design \([12]\). Each sub-stack has a unique descriptor 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 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 \( depth \). \( width \) 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 (\( \text{Global} \)) that defines the maximum number of items per sub-stack. The window and \( \text{Global} \) 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 \( \text{Global} \). A thread selects a sub-stack, then, compares the sub-stack item-count with the \( \text{Global} \). 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 \( \text{Global} \), after failing on all sub-stacks. The \( \text{Global} \) 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 \( \text{Global} \). The item-count must be less than \( \text{Global} \) for \textit{Push} or greater than the difference between \( \text{Global} \) and \( depth \) for \textit{Pop}. If the item-count is zero, then the sub-stack is empty. If no valid sub-stack is found, the \( \text{Global} \) is updated atomically. \textit{Push} adds whereas \textit{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 \textit{Push} increments whereas a \textit{Push} decrements the item-counter by one. Also the topmost item pointer is updated. At this point, a \textit{Push} adds an item whereas a \textit{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 \( \text{Global} \) for every hop during the search process, restarting for every \( \text{Global} \) change detected.

2D-stack is correct with respect to \( k\text{-Out-of-Order} \) stack semantics. The deterministic bound for the relaxation is tunable, controlled by the parameters of our design, given by Theorem 1. Also, the step complexity analysis provide a tight bound for the algorithm \([8]\).

\textbf{Theorem 1.} 2D-stack is linearizable with respect to \( k\text{-Out-of-Order} \) stack semantics, where \( k = (\text{shift} + \text{depth})(\text{width} - 1) \).

4 EXPERIMENTAL EVALUATION

We evaluate the performance of our implementation together with other existing stack designs including; the \( k\text{-segment} \) relaxed stack \([6]\), Elimination-Stack (elimination) \([5]\) and Treiber-Stack (treiber) \([12]\). All experiments run on an Intel Xeon CPU E5-2687W v2 machine 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 \textit{Pop} and \textit{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 algorithms are initialized in this way to avoid NULL returns that might arise from empty sub-stacks. Throughput is measured in terms of operations per second, whereas accuracy (quality) is measured in terms of error distance from the LIFO semantics.

To measure the quality, we adopt a similar method used in \([2, 7]\). A sequential linked list is run alongside the stack, for each \textit{Push} or \textit{Pop} a simultaneous insert or delete is performed on the list respectively. Items on the stack are duplicated on the list and can be identified by their unique labels. Insert operations happen at the head of the list similar to the push whereas the delete operation searches for the given item deletes it and returns its distance from the head (error distance). We then calculate the expected error distance for a given experiment run for 5 seconds with 5 repeats. Scalability is evaluated on both increasing relaxation and concurrency, for different NUMA settings. Experiment results are then

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure1.png}
\caption{Throughput and observed accuracy as the \( k \) bound for relaxation increases. (\( k \) bounded algorithms).}
\end{figure}

\( ^1 \)CAE\( ^1 \)Compare and Exchange (CAE) atomically compares 16 bytes of memory content and exchanges it with new content on success.
While for the other algorithms the quality reduces almost linearly with the number of threads, even for the NUMA settings. As the number of threads increases, random, random-c2 and k-robin, 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.

5 CONCLUSION AND FUTURE WORK

The aim of this work is to design an efficient lock-free stack algorithm that can continuously relax 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.

REFERENCES


