Design and Implementation of Highly Scalable Quantifiable Data Structures in C++

CHRISTINA PETERSON, VICTOR COOK, ZACHARY PAINTER

Overview

Motivation
- Correctness (Safety) Conditions
- Motivating Examples, Drivers for Change

Quantifiability
- Definition

Vector Space
- System Model

Entropy Measurement
- Correctness and Performance
- Engineering Case Study: k-FIFO Queue

Design and Implementation of Quantifiable Stack/Queue

Live Demonstration

Motivation

Correctness Conditions

A *correctness condition* defines correct behavior for a multiprocessor program.

- Serializability
- Sequential Consistency
- Quiescent Consistency
- Linearizability

All require reduction to a sequential history

64 threads with only one method call each, yields $64! = 10^{89}$
FYI: number of atoms in the universe is 10^{82}

Linearizability is the Standard

Linearizability is a correctness condition such that

1. A concurrent history of method calls is equivalent to a sequential history, and
2. Each method call appears to take effect at some instant between its invocation and response.

If a method call takes effect between its invocation and response, the method call takes effect in *real-time order*.

Practical Concerns about Linearizability

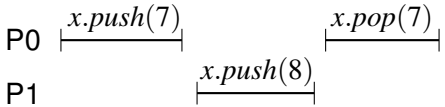
- Method calls not conserved
- Timing is critical to correctness yet can only be estimated by applications
- Intermediate system state is opaque

  (i.e. denotational, not small step operational)
- Correctness is binary, lacking any metric for engineering trade-offs
- Forced to make assumptions about abstract data type and object semantics

  (i.e. a *pop* called on an empty stack)
- Limits parallel computation to an outdated serial model

Correctness Depends upon Timing

Timing is critical to correctness yet opaque to the application. What about 4 wheels in an electric car!

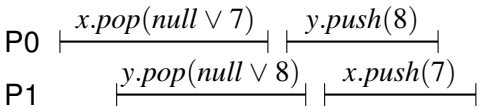


History H1: Concurrent history $H1$ on a single LIFO object x.

Serializable: Yes
Linearizable: No (or Yes with k-LIFO)
Quantifiable: Yes

Conditional Semantics Violates Type Constraints, Atomicity

Conditional pop assumed to make $H2$ Linearizable.

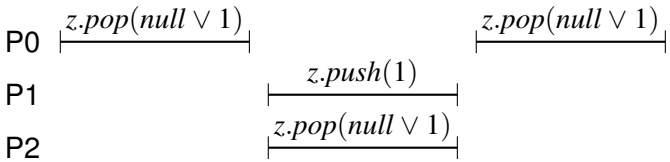


History H2: Concurrent history $H2$ on two LIFO objects x and y.

Serializable: Yes
Linearizable: Yes
Quantifiable: Yes

Conditional *pop* is inherently unfair

$P0$ keeps trying to *pop*, i.e. "progress without progress."

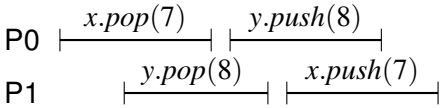


History H3: Concurrent history on a single LIFO object z.

Serializable: Yes
Linearizable: Yes
Quantifiable: Yes

Serializable, but not Linearizable

H4 works as a database transaction as some total order is valid.



History H4: Concurrent history $H3$ on two LIFO objects x and y.

Serializable: Yes
Linearizable: No
Quantifiable: Yes

Drivers for Change

- Architectural demands to utilize multicore resources
- General acceptance of relaxed semantics
- The intractable $O(n!)$ complexity of concurrent system models prompting the search for reductions
- Growth of distributed software applications: blockchain, distributed file systems, network apps

Desiderata

- Native concurrent correctness, independent of sequential history
- Timing and semantics in the type system available to applications
- Defined operationally on system state, not by a combinatorial search of the results
- Compositional*
- Free of Inherent Locking or Waiting*

*(Linearizability does fulfill the last two)

First Principles

Inspired by Descartes

Applying Cartesian thought to concurrent systems, we see that *first principles* describe what the program does (what is programmed), while *secondary principles* such as timing, order and arguments are modifiers on them.

"The conditions defined in secondary principles cannot be known without the first, whereas the reverse is not true."

– Descartes, Principles of Philosophy (preface to French edition of 1647)

Quantifiability

Introducing Quantifiability

Principles of Quantifiability

1. Methods are Conserved: Method calls are first class objects in the system that must succeed, remain pending, or be explicitly cancelled.

2. Methods Count: Every method call has a measurable impact on the system state.

Informal Definition of Quantifiability

A history H is *quantifiable* if each method call in H succeeds, remains pending, or is explicitly canceled, and the effect of each method call appears to execute atomically and in isolation. Furthermore, the effect of every completed method call makes a measurable contribution to the system state.

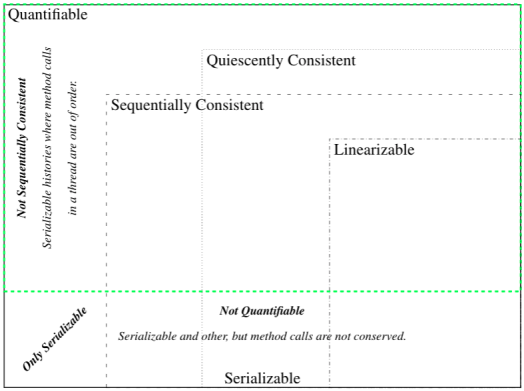


Figure 1: Venn diagram showing Quantifiability (green dashes) and other correctness conditions

Vector Space

System Model

Methods (M)

Methods are actions defined on concurrent objects.

Processes (P)

Processes are the actors that call the methods.

Objects (O)

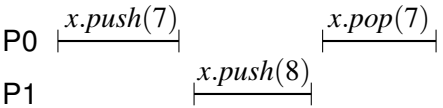
Objects are encapsulated containers of concurrent system.

Items (I)

Items are data passed as arguments to and returned as results from method calls.

Method Calls in Vector Space

Concurrent histories are represented in vector space. Each method call has a value over a basis vector uniquely defined by the system model [M, P, O, I].



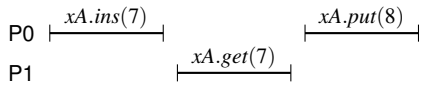
History H1

$$basis = \begin{bmatrix} push,\ P0,\ x,\ 7 \\ push,\ P0,\ x,\ 8 \\ push,\ P1,\ x,\ 7 \\ push,\ P1,\ x,\ 8 \\ pop,\ P0,\ x,\ 7 \\ pop,\ P0,\ x,\ 8 \\ pop,\ P1,\ x,\ 7 \\ pop,\ P1,\ x,\ 8 \end{bmatrix} \quad H1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ -1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (1)$$

History Vector Value Assignment

Let $\vec{H}$ be a history vector, let $\vec{H_{prev}}$ be the previous state of the history vector, and let i be the position in the basis vector corresponding to a method call M on item I by a process P in object O in a history.

- Producer: $\vec{H}[i] = \vec{H_{prev}}[i] + 1$
- Consumer: $\vec{H}[i] = \vec{H_{prev}}[i] - 1$
- Reader: for the jth read, $\vec{H}[i] = \vec{H_{prev}}[i] - \frac{1}{2^j}$
  geometric series $\sum_{j=0}^{\infty} \frac{1}{2^j} = 1$
- Writer: let k be the position in the basis vector corresponding to an old item to be overwritten in object O, $\vec{H}[i] = \vec{H_{prev}}[i] + 1$, $\vec{H}[k] = \vec{H_{prev}}[k] - 1$



History H6: Quantifiable History for a Concurrent Map. xA is a specific key in the map.

$$basis = \begin{bmatrix} ins,\ P0,\ xA,\ 7 \\ ins,\ P0,\ xA,\ 8 \\ ins,\ P1,\ xA,\ 7 \\ ins,\ P1,\ xA,\ 8 \\ put,\ P0,\ xA,\ 7 \\ put,\ P0,\ xA,\ 8 \\ put,\ P1,\ xA,\ 7 \\ put,\ P1,\ xA,\ 8 \\ get,\ P0,\ xA,\ 7 \\ get,\ P0,\ xA,\ 8 \\ get,\ P1,\ xA,\ 7 \\ get,\ P1,\ xA,\ 8 \end{bmatrix} \qquad H6 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -\frac{1}{2} \\ 0 \end{bmatrix} \tag{2}$$

Reshaping $H6$ into a 2 by 6 matrix such that each row corresponds to an object/item combination yields the following:

$$H6 = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & -\frac{1}{2} \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad sum = \begin{bmatrix} \lceil 1 - \frac{1}{2} \rceil - 1 \\ 1 \end{bmatrix} \qquad = \begin{bmatrix} 1 - 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3}$$

- Top row corresponds to item 7
- Bottom row corresponds to item 8
- For each row:
  - Add producers and readers, then take the ceiling of the sum
  - Add consumers
  - If the sum for each row is greater than or equal to zero, the sum is quantifiable

Methods count: Verification becomes a matrix operation

- Let n be the total number of method calls in a history
- Let i be the total number of input/output combinations
- Let j be the total number of objects
- It takes $O(n)$ time to iterate through all methods in the method call set
  - Update sum for each method call
- It takes $O(i \cdot j)$ time to iterate through all possible configurations
  - Check that each position in vector is greater than or equal to zero
- The total time complexity is $O(n + i \cdot j)$

Proving that a Data Structure is Quantifiably Correct

Visibility Point

- Identify an instruction in which the entire effects of the method become visible
- Demonstrates:
  - Atomicity
  - Isolation

Method Call Conservation

- Proof by cases to demonstrate
  - A method call completes its operation on the successful code path
  - A method call's pending request is stored in the data structure on the unsuccessful code path
  - A method call's pending request is fulfilled by the corresponding inverse method

Entropy Measurement

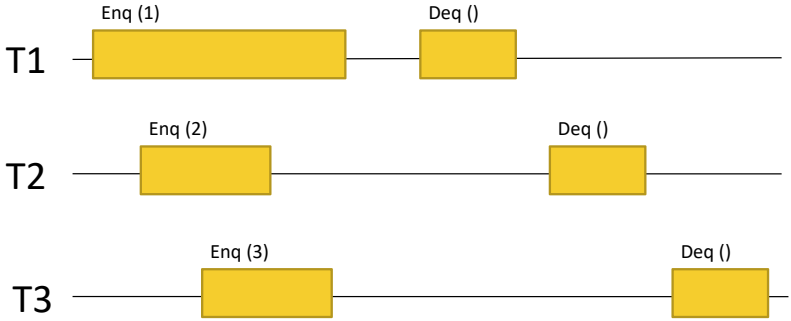
Relationship between Correctness and Performance

- Previous works focus on relaxed correctness conditions
- Relaxed semantics is a technique, not a measurement
- Need to measure what relaxed semantics and other techniques do
- How correct is the result? (not binary)

Problem

Existing metrics are not capable of measuring the performance effects of a correctness condition because they neglect the delays in method calls.

Disordered results are "correct"



Linearizable response values for Deq operations:

T1:Deq ():1	T1:Deq ():2	T1:Deq ():3	T1:Deq ():1	T1:Deq ():2	T1:Deq ():3
T2:Deq ():2	T2:Deq ():3	T2:Deq ():1	T2:Deq ():3	T2:Deq ():1	T2:Deq ():2
T3:Deq ():3	T3:Deq ():1	T3:Deq ():2	T3:Deq ():2	T3:Deq ():3	T3:Deq ():1

Figure 2: First-In-First-Out (FIFO) Queue

Knuth Inversion Count

The expected order of method calls is the *invocation order*.
(Think when you enter the restaurant or you hit the brakes on the 4 wheels of an electric car.)

Actual order in a queue or stack represents the *method call order*.

The deviations in the expected and actual ordering of method calls is measured by counting the *inversions*.

Solution

Disorder is entropy. Use the Knuth inversion count to create a probability mass function and Shannon entropy to measure it. Normalize it so it is generally applicable across many systems.

Shannon Entropy

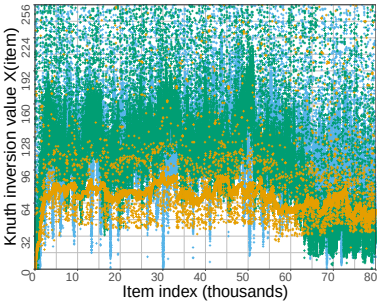
$$H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{4}$$

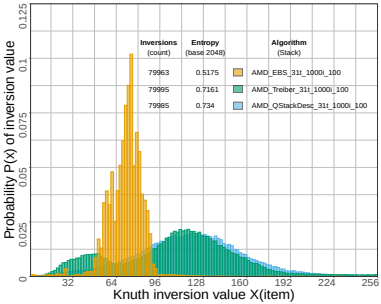
Equation 4 is Shannon entropy for a discrete random variable X with possible values $\{x_1, ..., x_n\}$ and probability mass function $P(X)$.

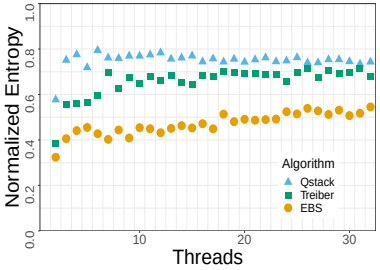
$P(X)$ varies for each of the abstract data types for concurrent data structures.

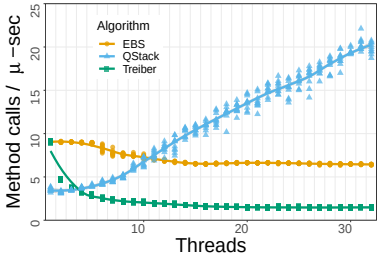
Stacks

- High Entropy
- Concurrent implementations are challenging
- EBS attains high throughput and low entropy


Knuth inversion value X(item)
Item index (thousands)


Inversions (count)
Entropy (base 2048)
Algorithm (Stack)
79963 0.5175 AMD_EBS_31i_1000i_100
79995 0.7161 AMD_Treiber_31i_1000i_100
79985 0.734 AMD_QStackDesc_31i_1000i_100
Probability P(x) of inversion value
Knuth inversion value X(item)


Normalized Entropy
Algorithm
Qstack
Treiber
EBS
Threads


Algorithm
EBS
QStack
Treiber
Method calls / µ −sec
Threads

Case Study: Engineering trade-offs

- Correctness is not enough
- Entropy is an actionable measure
- Applications have different requirements

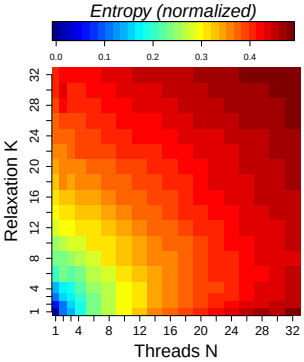


Figure 3: Entropy heatmap

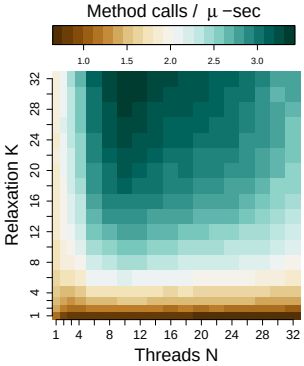


Figure 4: Performance is cool

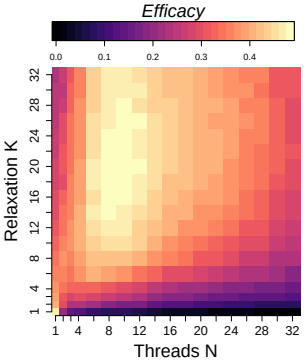


Figure 5: Sweet spot!

Design and Implementation of Quantifiable Stack/Queue

Requirements

- All method calls must be conserved
- All method call must be atomic

Design Goals

- Prioritize avoiding contention over resolving it

Conservation: pending pop is a first class object

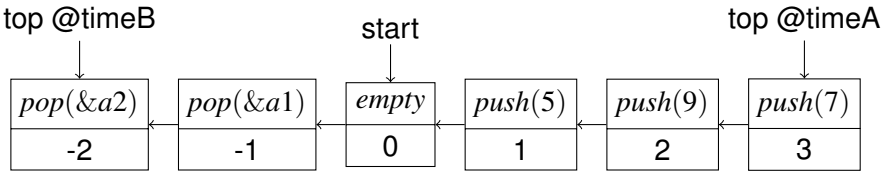


Figure 6: Negative stack formed @timeB after 3 *push* followed by 5 *pop* calls.

Providing Atomicity

Compare-And-Swap (CAS) atomically updates an object given an expected value. If the expected value is incorrect, no update occurs.

C++ Compare-Exchange

```
1: obj->compare_exchange_weak(*expected, desired)
```

Quantifiable Stack

- In the case of contention at a node, create a fork
- *Push* and *pop* insert or remove at any “leaf” node
- To make it easy to check for leaf nodes, we maintain a doubly-linked stack

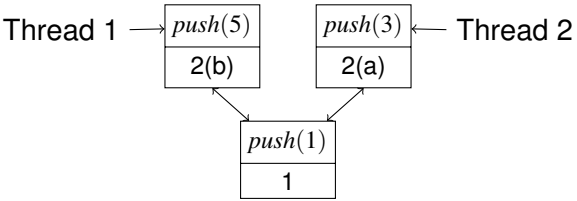


Figure 7: Thread 1 and Thread 2 concurrently perform push operations. An incoming pop may return 3 or 5, either is correct

Descriptor Objects

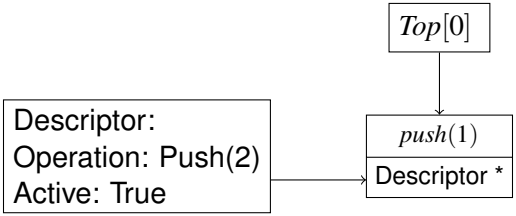
- Some operations must update multiple pointers
  - We can use descriptor objects to make this process appear atomic
- A descriptor object contains all necessary information for an arbitrary thread to complete an operation

Descriptor Object Example

```
1: struct Descriptor
2:     Operation {fork, push, pop}
3:     Active {true, false}
```

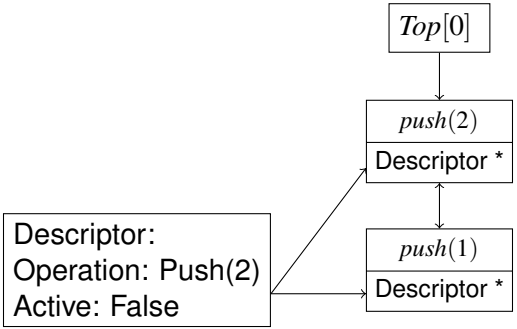
Push Operation With Descriptors

- To push a node, a thread updates a node's descriptor pointer using CAS
- This makes the intended operation visible to all threads



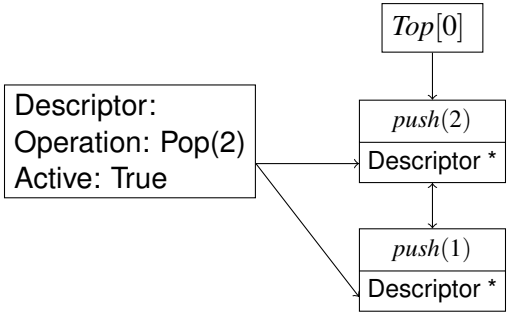
Push Operation With Descriptors

- Afterwards, an arbitrary thread may insert a new doubly-linked node, and update the top pointer
- Once this is done, the descriptor may be marked inactive



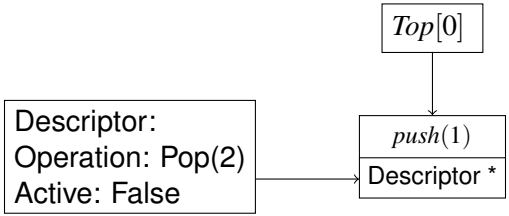
Pop Operation With Descriptors

- The pop operation requires two descriptor pointers to be updated



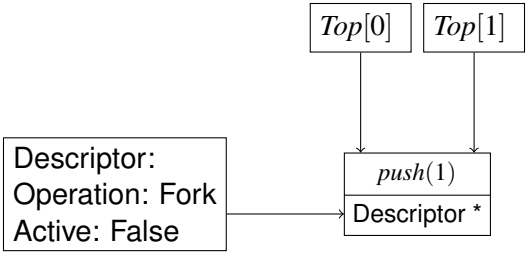
Pop Operation With Descriptors

- Afterwards, the links can be removed, and the top pointer updated



Fork Operation With Descriptors

- To create a fork, a thread updates a node's descriptor pointer
- Afterwards, any thread may initialize a new top pointer, and point it at that node



Reducing Entropy

- reduce time between new forks
- limit maximum number of branches
- Enforce maximum "height" disparity between branches

Quantifiable Stack

- Reduce entropy by keeping branches balanced

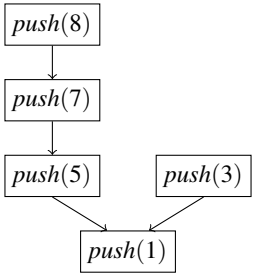
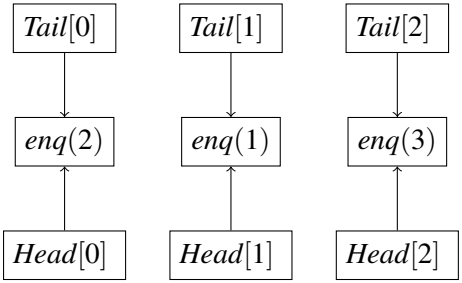


Figure 8: Branches with a height disparity of 2

Quantifiable Queue

- The aforementioned branching technique is not very useful in a queue
- Instead, we implement a multi-queue to minimize contention between threads



Live Demonstration

Demonstration Settings

Processor

- AMD EPYC 7501 @ 2 GHz
  - Cores: 32, Logical Processors: 64

Conclusion

Key Take-Aways

- Quantifiability enables highly scalable data structures by permitting relaxed semantics
- The vector space model facilitates an efficient verification technique for checking the correctness of a concurrent history
- The entropy metric provides designers with the ability to analyze the trade-off between correctness and performance

Source Code

- QStack and QQueue: https://github.com/RioVic/quantifiable.git
- Vector Space Verification: https://github.com/CLPeterson/VectorSpace/tree/gnuplots