Quantitative relaxation of concurrent data structures

Christoph-Simon Senjak

Lehr- und Forschungseinheit für Theoretische Informatik
Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr.67, 80538 München

Oberseminarvortrag 21. Dezember 2012

Foreword

- This talk bases on "Quantitative Relaxations of Concurrent Data Structures" by Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, Ana Sokolova.
- Some definitions are slightly adapted and not made in full generality.
- I am responsible for any mistake in this talk.

Concurrent Data Structures

- Shared structures between several processes/threads (Stack, Fifo, Deque, Heap)
- Important for parallel programming
- Synchronized, for example via compare-and-swap
- May impose threads to wait $\Rightarrow$ bottleneck in many cases

Example

```
Time | Thread 1          | Thread 2          | Stack
=====*=================*=================*========
  1  | beginCalc         | beginCalc         |  ()
  2  | calc              | calc              |  ()
  3  | end, result: x | calc              |  ()
  4  | lock              | end, result: y | [()]
  5  | expand stack      | lock              | [(?)]    |
  6  | write x           | wait              | [(x)]    |wasted
  7  | release           | wait              | (x)      |time
  8  |                   | wait              | [(x)]    |
  9  |                   | expand stack      | [(? x)]
 10  |                   | write y           | [(y x)]
 11  |                   | release           | (y x)
```

Tradeoff

- The common shared structures have very strict semantics
- In many cases, the full strictness is not needed
- Relaxation of semantics may increase performance
- Problem: How to formalize relaxations?
- Idea: Transition system in which every violation of strict semantics has a certain cost
- $\Rightarrow$ Costs can be limited to limit the relaxation

Informal Example

```
Time | Thread 1          | Thread 2          | Stack
=====*=================*=================*========
  1  | beginCalc         | beginCalc         |  ()
  2  | calc              | calc              |  ()
  3  | end, result: x    | calc              |  ()
  4  | make-buffer       | end, result: y    | () ? ?
  5  | write x 1         | write y 2         | () x y
  6  | try-lock=>win     | try-lock=>fail    | [()] [x y]
  7  | push-buffer       | ...               | [(x y)]
  8  | release           |                   | (x y)
```

Formal framework - 1

- Usually, the structures are containers, so let $\mathcal{D}$ be the set of objects the structure can contain
  - For Deques we can use the contained objects
  - For synchronized counters, we use their integer range
- They have several operations that can manipulate the data structure, they usually either return or insert objects from $\mathcal{D}$. Call the set of these operations with the associated objects Σ, like $\Sigma_{\text{stack}} = \{f\ \ x \mid x \in \mathcal{D}, f \in \{\text{push}, \text{pop}\}\} \cup \{\text{pop}\ \mathit{null}\}$
- We call the set of legal sequences of such operations $\mathcal{S} \subseteq \Sigma^*$ the **sequential specification**. We require it to be $\prec$-closed, where $\prec$ shall denote the prefix relation

Formal Framework - 2

- The same state might be reachable by several sequences: push(a) pop(a), push(a) push(b) pop(b) pop(a).
- We do not want to rely on further underlying structures
- $\Rightarrow$ define state-equality extensionally:

$$s =_S t :\Leftrightarrow \forall_{u \in \Sigma^*} (su \in \mathcal{S} \Leftrightarrow tu \in \mathcal{S})$$

- The **states** of the structure can then be identified with the elements $[q]_S$ of the set $\mathcal{S}/_{=_S}$
- The **kernel** of a state is the set of its shortest sequences, $\ker[q]_S = \{t \in [q]_S \mid t \text{ has minimal length}\}$

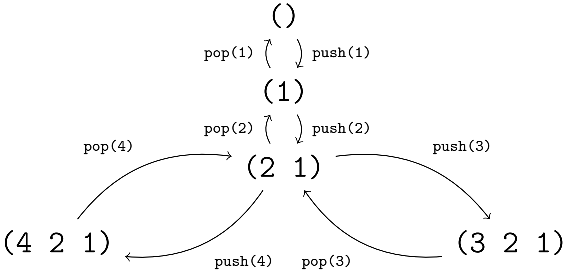
Example

Example: Stack (+ = push, - = pop)

```
Stack    | non-kernel sequences      | kernel sequences
=========*===========================*===================
 ()      | +a-a, +a+b-a-b,           | ()
         | +a+b-b+c+d-d-c-a, ...     |
---------+---------------------------+-------------------
 (2 1)   | +1+2+1+1-1-1-2+2, ...     | +1+2
---------+---------------------------+-------------------
 (1 2 3) | +a+b-b-a+3+2+c-c+1, ... | +1+2+3
```

Labelled transition systems

- Define a labelled transition relation $[p]_S \xrightarrow{m} [q]_S$ on $[p]_s, [q]_S \in \mathcal{S}/_{=_S}, [pm]_S = [q]_S, m \in \Sigma$ to model the state-changes of the structures
- Example: Stack



Quantified labelled transition systems

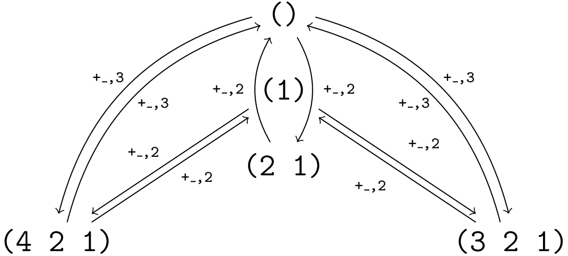
- To relax a structure, we will allow the sequences outside of $\mathcal{S}$
- Let α be some ordinal number. It denotes the set of possible **costs** of a transition.
- Define a **cost function**

$$\text{cost} : \mathcal{S}/_{=_S} \to \Sigma \to \mathcal{S}/_{=_S} \to \alpha$$

  between any two states such that $\text{cost}(p, m, q) = 0$ for $p \xrightarrow{m} q$
- Gain the **quantified labelled transition system** of the structure and the cost function by defining the quantified labelled transition relation by $p \xrightarrow{m,\text{cost}(p,m,q)} q$ for all valid p, m, q.

Example

Example: Stack (+=push,-=pop) with values describing the number of left-out elements (only pushs with value $\neq 0$ are drawn).



Path cost functions

- **Quantitative paths** are paths in our quantified labelled transition system. They are of the form

$$\kappa = q_1 \xrightarrow{m_1, k_1} \dots \xrightarrow{m_n, k_n} q_{n+1}$$

- Their **quantitative trace** is the sequence of transition labels $(\langle m_i, k_i \rangle)_i$ and their **trace** is the sequence of the applied operations $(m_i)_i$
- <u>Notice:</u> There might be multiple quantitative traces per trace! The set of quantitative traces of a trace u beginning from the initial state is denoted by $\mathrm{qtr}(u)$, and $\mathrm{qtr}(\mathcal{S}) = \bigcup_{m \in \Sigma^*} \mathrm{qtr}(m)$
- A **path cost function** is a function of type $\mathrm{qtr}(\mathcal{S}) \to \alpha$ which is $\prec$-$\subseteq$-preserving

Relaxations

- Now let Σ, $\mathcal{S}$, α, cost and a path cost function pcost be given.
- Define the **distance function** $d : \Sigma^* \to \alpha$ by $du = \min\{\mathrm{pcost}(\tau) \mid \tau \in \mathrm{qtr}(u)\}$.
- The k-**relaxed specification** $\mathcal{S}_k$, $k \in \alpha$, can now be defined as

$$\mathcal{S}_k = \{u \in \Sigma^* \mid du \leq k\}$$

Stuttering Relaxation

- One possible general relaxation is the allowance of repetitions of the same operation, the **stuttering relaxation**
- The cost function is defined by

$$\mathrm{stcost}(q, m, q') = \begin{cases} 0 & \text{for } q \xrightarrow{m} q' \\ 1 & \text{for } q = q', q \overset{m}{\not\to} q, \exists_r q \xrightarrow{m} r \\ \omega & \text{otherwise} \end{cases}$$

  A transition costs 1 if it erroneously returns to the same state after applying m.
- The path cost function is the length of a maximal subsequence in which a state is erroneously preserved.

Example: Stuttering Relaxation of CAS

- Example: Stuttering relaxation of a CAS-Structure (Compare-And-Swap)
- We have $\mathcal{D} = \omega + 1$, we use $\omega \in \omega + 1$ as "uninitialized". $\Sigma = \{\mathsf{cas}(d, d', b) \mid d \in \omega + 1, d' \in \omega, b \in 2\}$.
- We define $\mathcal{S}$ inductively:
  - $\{(\mathsf{cas}(\omega, d, 1)) \mid d \in \omega\} \subseteq \mathcal{S}$
  - If $(\ldots, \mathsf{cas}(d, d', 1)) \in \mathcal{S}$ then $(\ldots, \mathsf{cas}(d, d', 1), \mathsf{cas}(d_1, d_1', 0), \ldots, \mathsf{cas}(d_n, d_n', 0)) \in \mathcal{S}$ and $(\ldots, \mathsf{cas}(d, d', 1), \mathsf{cas}(d_1, d_1', 0), \ldots, \mathsf{cas}(d_n, d_n', 0), (d', d'', 1)) \in \mathcal{S}$ for $d' \notin \{d_1, \ldots, d_n\}$

Example: Stuttering Relaxation of CAS

- We define the path cost function to be infinite if any of its transitions is infinite, and otherwise as the maximum length of a subsequence of that path that contains only correct transitions of the form $\mathrm{cas}(d, d', 0)$ and transitions with value 1:

$$\mathrm{pcost}((\langle m_i, k_i \rangle)_i) = \begin{cases} \omega & \text{for } \exists_i k_i = \omega \\ \max_{j-i+1} \forall_{i \le x \le j} . k_x \neq 0 \vee m_x = \mathrm{cas}(\_, \_, 0) & \text{otherwise} \end{cases}$$

- A k-Relaxation then allows a failed state to remain at most k operations.

Example: A simple relaxed FIFO

- There are very efficient implementations of relaxed FIFOs which do not fully guarantee the preserving of order
- One less efficient but very easy implementation is to use multiple strict FIFOs parallelly in a round-robin manner
- By iterating over the inputs until a non-locked FIFO for input can be found, and iterating over the outputs until a FIFO ready for output is found, when the number of parallel strict FIFOs is near the number of producing threads, and provided that the consumption of the elements on the other end is nearly immediately, there might not be waiting time at all, as always some FIFO is unlocked

Happy Doomsday!