Quantitative Quiescent Consistency

JOINT WORK WITH JAMES RIELY

SCHOOL OF COMPUTING, DEPAUL UNIVERSITY
Outline of talk

- What is an interface for a concurrent object?
- Linearizability
- Counting networks and Quiescent consistency
- This Talk: Quantitative Quiescent consistency
Interfaces for Concurrent Objects

(A QUICK TOUR FOLLOWING HERLIHY AND SHAVIT)
Sequential specifications

Abstractly, a set of traces. (EG) COUNTER with only an increment method

\[ [^+]_0 [^+]_1 \cdots [^+]_n \]

Concretely:
PRE/POST conditions for methods

Linear in size of interface. Interactions between methods are succinctly characterized by the side effects on the object state.

Counter: int i.

```java
int inc() {
    int j = i;
    i = i+1;
    return j;
}
```
Criteria

1. Method calls should appear to happen in a one-at-a-time, sequential order.
2. Method calls separated by a period of quiescence should appear to take effect in their real-time order.
3. Method calls should appear to take effect in program order.
4. Each method call should appear to take effect instantaneously at some moment between its invocation and response.

(1) +(3): Sequential consistency. ‘Interleaving semantics’

(1) +(2): Quiescent consistency.

(1—4): linearizability
Linearizability

NOT LINEARIZABLE

\[ (+ [^+]_1 [^+]_0 )_2^+ \]

\[ (+ \{ + \}^+_1 [^+]_2 )_0^+ \]

LINEARIZABLE

\[ [^+]_0^+ [^+]_1^+ \cdots [^+]_n^+ \]

\[ (+ [^+]_0^+ [^+]_1^+)_2^+ \]

\[ (+ \{ + \}_2^+ )_1^+ \]_0^+ \]

\[ (+ [^+]_1^+ \{ + \}_2^+ )_0^+ \]
Linearizability

1. Compositionality.

Composition of the histories of two non-interfering linearizable objects is linearizable

SC is NOT compositional in this sense.
Linearizability

1. Compositionality [Herlihy/Wing]

Composition of the histories of two non-interfering linearizable objects is linearizable

2. Operational refinement. [Filipovic/Ohearn/Rinetzky/Yang]

From a client perspective, the set of linearizations of a linearizable object is an operational refinement of the specification of the object. The client is unable to distinguish the implementation from the specification.

\[ C[\text{Impl}] \leq C[\text{Spec}] \]
BUT INTRINSICALLY INEFFICIENT

[Aspnes, Dwork, Herlihy, Shavit, Touitou, Waarts]

Contestion * computation length >= \( \Omega(n) \)

Contestion: maximum number of concurrent accesses
Computation length: number of distinct vars in execution path
\( n \): number of concurrent “threads”

Counter: \( \text{int } i \).

    int inc() {
        int j = i;
        while (!cas(i, j, j+1)) {
        }
    }
Context of this talk

- Shavit. Data structures in the multicore age [CACM]
- Our focus: What is the “right” correctness criterion?
- This talk: Quantitative quiescent consistency
Rest of the talk

- Counting networks
- Formalizing quantitative quiescent consistency
- Properties and expressiveness
Counting networks
Balancers
Balancers
Counters

Inc():
- flip routing bit
- go down old path
- increment cntr by 2
- return old cntr val
Counters

**Inc()**: flip routing bit, go down old path, increment cntr by 2, return old cntr val

RETURN 0, $[+]_0^+$
Counters

Inc():
1: flip routing bit
2: go down old path
3: increment cntr by 2
4: return old cntr val
Counters

Inc():
flip routing bit
go down old path
increment cntr by 2
return old cntr val
The Counter

Not linearizable. \( (+ [+]_1 [+]_0)_{2}^{+} \)

Odd intermediate states. \( (+ [+]_1^{+} \)

“Balance” restored at quiescence \( 0 \leq J-I \leq 1 \)
Hierarchical counters

“Balance” restored at quiescence
0 ≤ J1 – J0 ≤ 1
0 ≤ J2 – J1 ≤ 1
0 ≤ J3 – J2 ≤ 1

(↑[↑]↑1[↑]↑2)0
Counters with inc() and dec(): more later

Inc():
- flip routing bit
- go down old path
- increment cntr by 2
- return old cntr val

dec():
- flip routing bit
- go down NEW path
- decrement cntr by 2
- return NEW cntr val

As before: “Balance” restored at quiescence $0 \leq J - I \leq 1$

However.....
Stacks

Push(o):
- flip routing bit
- go down old path
- push o onto stack

pop():
- flip routing bit
- go down NEW path
- pop from stack
Addressing contention: Diffracting prisms
"Balance" restored at quiescence
\[ 0 \leq \text{size(left)} - \text{size(right)} \leq 1 \]

Problematic races between pushes and pops: later.
Formalization
Model: traces

Labelled partial orders. Pomset model enriched with:

- polarity
- input/output
- bracketing
- call/return

Order generated by edges between opposite polarity: eg. input events cannot be related to other input events unless there is an intervening output event,

Operational traces have a global notion of time. The constraints between pairs of inputs and pairs of output indicate the limits of observability: it is impossible to tell which of two calls happened first.
Model: components

Component  [eg. “Stack”]
= set of traces.  [eg “all possible concurrent executions of a stack”]

“Internal” order. Determined by the component.  $\xrightarrow{?} s$

“External” order. Determined by the calling context.  $\xrightarrow{!} s$
Linearizability
\( s \sqsubseteq_{\text{lin}} t \)

\( s \) is a permutation of \( t \)

for any prefix \( s_1 \preceq_{\text{pre}} s \)

for any \( a_1 \in s_1 \) and \( a_2 \in (s - s_1) \)

if \( s[a_1] \Rightarrow s[a_2] \) then \( t[a_1] \Rightarrow t[a_2] \)

\( s \sqsubseteq_{\text{lin}} t \)

\( \exists s' \succ_{\text{pre}} s. \exists t' =_{\alpha} t. s' \sqsubseteq_{\text{lin}} t' \)
Quiescent Consistency
\[ s \sqsubset_{\text{qcin}} t \]

- \( s \) is a permutation of \( t \)
- for any prefix \( s_1 \preceq_{\text{pre}} s \)
- for any \( a_1 \in s_1 \) and \( a_2 \in (s - s_1) \)
  - if \( s[a_1] \not\Rightarrow s[a_2] \) then \( t[a_1] \Rightarrow t[a_2] \)

\[ s \sqsubset_{\text{qcin}} t \]

\[ \exists s' \succ_{\text{pre}} s. \exists t' = \alpha \ t. \ s' \sqsubset_{\text{qcin}} t' \]
Quantitative Quiescent Consistency
\( s \sqsubseteq t \)

\( s \) is a permutation of \( t \)

for any \( k \) and \( k \)-open prefix \( s_1 \leq_{\text{pre}} s \),

there exists \( r \) such that \( |r| \leq k \),

for any \( \alpha_1 \in s_1 \) and \( \alpha_2 \in ((s - s_1) - r) \)

if \( s[a_1] \not\Rightarrow s[a_2] \) then \( t[a_1] \Rightarrow t[a_2] \)

\( s \sqsubseteq t \)

\( \exists s' >_{\text{pre}} s. \exists t' =_\alpha t. s' \sqsubseteq \text{in} t' \)
Alternate Definition
\[ s \subseteq t \quad (a_1^?, a_1^!, \ldots a_n^?, a_n^!) \]

\[ \forall j. \ j \leq |\{ i \mid s[a_i^?] \Rightarrow s[a_j^!]\}| \]

\[ s \subseteq t \]

\[ \exists s' >_{\text{pre}} s. \ \exists t' =_\alpha t. \ s' \subseteq_{\text{in}} t' \]
Operational realization
TOWARDS UNDERSTANDING QQC
Curiosities

$$[I] = \{s \mid \exists t \in I. \ s \sqsubseteq t\}$$

$$\langle + \ [+]_1 ^+ \ {+ \ [+]_3 ^+} ^+ < ^+ \ [+]_5 ^+ \ d ^+ \ [- \ [+]_7 ^+} ^+ \ d ^+ \ D_0 ^+ \ D_2 ^+ > ^+ \ 4 \ 6 ^+\rangle ^+ _8 ^+$$

$$\langle + \ [+]_1 ^+ \ {+ \ [+]_3 ^+} ^+ < ^+ > _0 ^+ \ [+]_5 ^+ \ d ^+ \ D_2 ^+ \ 4 ^+ \rangle _6 ^+$$
CONCURRENT WORLD

SEQUENTIAL DATA STRUCTURE

MEDIATOR
Speculation

- As an actual mechanism for improving efficiency. eg PLDI 2013 paper. A Compiler Framework for Speculative Analysis and Optimizations

- As a unifying conceptual framework to understand compiler optimizations. eg. Generative Operational Semantics for Relaxed Memory Models. ESOP 2010. (with Corin Pitcher and James Riely)
choose:

if unprocessed.notEmpty():
    received.add(unprocessed.remove())
    let i = object.predict()
    let r = object.run(i)
    executed.add(i, r)

or:

if exists i in received.keys() intersect executed.keys()
    let s = received.remove(i)
    let r = executed.remove(i)
    returnable.add(s, r)
\[
(\mathcal{G} \mathcal{M}_{\mathcal{E}} \mathcal{D})_{0}^{1} (\mathcal{G} \mathcal{M}_{\mathcal{R}} \mathcal{D})_{2}^{1}
\]

```python
choose:

\[\begin{align*}
\text{if } \text{unprocessed}.\text{notEmpty}(): \\
&\quad \text{received}.\text{add}(\text{unprocessed}.\text{remove}()) \\
&\quad \text{let } i = \text{object}.\text{predict}() \\
&\quad \text{let } r = \text{object}.\text{run}(i) \\
&\quad \text{executed}.\text{add}(i, r)
\end{align*}\]

or:

\[\begin{align*}
\text{if } \exists i \in \text{received}.\text{keys}() \cap \text{executed}.\text{keys}(): \\
&\quad \text{let } s = \text{received}.\text{remove}(i) \\
&\quad \text{let } r = \text{executed}.\text{remove}(i) \\
&\quad \text{returnable}.\text{add}(s, r)
\end{align*}\]
```
IMPOSSIBILITY

|RECEIVED| = |EXECUTED|

choose:
- if unprocessed.notEmpty():
  - received.add(unprocessed.remove())
  - let i = object.predict()
  - let r = object.run(i)
  - executed.add(i, r)
- or:
  - if exists i in received.keys() intersect executed.keys()
  - let s = received.remove(i)
  - let r = executed.remove(i)
  - returnable.add(s, r)
Correspondences

For counter, operational view coincides with semantic view, i.e. for any trace generated by opsem there is a counter of some width that generates it.

More generally: for ANY interface, (* with mild restrictions on speculation *), operational view is sound for semantics.
Compositionality

Let $T$, $U$ be interfaces with disjoint vocabularies. If

$ s \upharpoonright T $ is $\text{QQC wrt } t $ in $T$

$ s \upharpoonright U $ is $\text{QQC wrt } u $ in $T$

Then,

$ s $ is $\text{QQC wrt some interleaving of } t, u$
Comment on Proof

\[
\begin{align*}
s \cap T & \text{ is } \text{QQC wrt } t = t_1 \ldots t_n \\
s \cap U & \text{ is } \text{QQC wrt } u = u_1 \ldots u_m \\
\text{Then:} \\
& s \text{ is } \text{QQC wrt some interleaving of } t, u \\
\text{Of course, the interleaving is determined by } s
\end{align*}
\]

Linearizability proof: CAN be done by accumulating local constraints between \( t \) and \( u \) using \( s \). eg. Order \( t_i \) before \( u_j \) if so ordered by \( s \).

QQC: Global constraints that are solvable because of ``flow'' properties.
Winding down...
Contributions

1. QUIESCENT STATES ➔ INTERMEDIATE STATES
Hierarchical counters

“Balance” restored at quiescence

\[ 0 \leq J_1 - J_0 \leq 1 \]
\[ 0 \leq J_2 - J_1 \leq 1 \]
\[ 0 \leq J_3 - J_2 \leq 1 \]

\( (^+ [^+ ]_1 ^+ [^+ ]_2 ^+ )_0 ^+ \)
Contributions

1. QUIESCENT STATES ➔ INTERMEDIATE STATES

2. CLARIFY SPECIFICATIONS
Stacks

“Balance” restored at quiescence
0 ≤ size(left) − size(right) ≤ 1
INTERMEDIATE STATES in “NON-MONOTONE” DATA STRUCTURES

For “monotone” data structures, (eg) COUNTER with ONLY inc(), our criterion provides a measure of correctness.

BUT: for others

Stacks: PUSH and POP
Counters: Inc and Dec
Counters with inc() and dec()

Inc():
flip routing bit
go down old path
increment cntr by 2
return old cntr val

dec():
flip routing bit
go down NEW path
decrement cntr by 2
return NEW cntr val

Order that works: first inc, then dec (both return 0)

Order that DOES NOT work: first dec (returns -2)
then inc! (returns -2)
Stacks

Sequential POP:
1, 0 in that order
STACKS: PATHOLOGY

IF PUSH GOES FIRST:  POP RETURNS 2, THIS IS OK

IF POP GOES FIRST:    POP RETURNS 1. NOT OK!
Contributions

1. QUIESCENT STATES \(\rightarrow\) INTERMEDIATE STATES

2. CLARIFY SPECIFICATIONS: IDENTIFICATION of constraint needed on implementation to satisfy our criterion
Contributions

1. QUIESCENT STATES ➔ INTERMEDIATE STATES

2. CLARIFY SPECIFICATIONS: IDENTIFICATION of constraint needed on implementation to satisfy our criterion

3. A general model for “relaxed data structures”

   FUTURE: Proof principles. Does this refined analysis have any utility??