Volatile Variables and Memory Barriers
(Brainstorming Session)

http://d3s.mff.cuni.cz

Andrej Podzimek
podzimek@d3s.mff.cuni.cz
What has been said last time...

- A concurrent AVL tree algorithm exists
- The algorithm could be ported to a kernel environment after
  - Resolving memory ordering issues (volatile...)
  - “Replacing” the implicit garbage collection with
    - Explicit RCU read-side protection of all operations
    - Deferred destruction of deallocated nodes based on RCU
- We have (an insight into) a kernel that uses AVL trees
  - The UTS kernel used by Solaris
- We have a working RCU implementation for that kernel
  - The RCU algorithm presented >one year ago
- We could improve the kernel algorithms that use AVL trees
  - Provided that they can (be adjusted to) tolerate stale data
References

- Nathan G. Bronson, Jared Casper, Hassan Chafi, Kunle Olukotun: *A Practical Concurrent Binary Search Tree*
  Stanford University

- Abhik Roychoudhury: *Formal Reasoning about Hardware and Software Memory Models*

- Luc Bougé, Joaquim Gabbaró, Xavier Messeguer, Nicolas Schabanel: *Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries*

- Maurice P. Herlihy, Jeannette M. Wing: *Linearizability: A Correctness Condition for Concurrent Objects*
Outline

- Memory barriers and the JVM
  - Let's see what “volatile” actually means in Java
- Translating parallel algorithms from Java to C
  - Are all the memory barriers necessary?
  - More generally: Can we **compute** which barriers are needed?
- A possible approach to **automated** memory barrier insertion
  - Modeling (the non-blocking fragments) of an algorithm
  - Specification of correctness conditions
  - For all the **possible** reorderings of instructions...
- Do we have enough computational power?
  - Can this algorithm be more clever than just “brute force”? 
  - Yes, hopefully! Under certain assumptions...
- **Does the whole idea make sense at all?**
Parallel AVL algorithm

- Non-blocking readers (no exceptions...)
- Hand-over-hand traversal, hand-over-hand locking
  - Follow references that provably coexisted with previous ones
  - Backtrack and retry otherwise
  - (It is actually more complex, proof based on linearizability)
- Non-blocking communication using version numbers
  - Combined with bitwise tricks...
- Blocking synchronization among writers (when necessary)
- Key optimization idea: How does a rotation change a node?
  - Key interval enlarged: Dependent lookups can continue
  - Otherwise: Dependent lookups must backtrack one level
  - Independent lookups not affected, of course
- Combined with atomic CoW duplication and iterators (!)
Notes on Rotations

- Order of operations does matter
- Concurrent readers see all the intermediate states
- “Downward links originally pointing to a shrinking node must be changed last and downward links from a shrinking node must be changed first. A similar logic can be applied to the ordering of parent updates.”
- The growing/shrinking flags are vital for the correct operation
- Double rotation
  - Carried out as a single rotation (the nested one)
  - Rebalancing restarted at the node that has grown
- Stale height counters
  - Theoretical results show that they do not matter
  - The tree is a strict AVL tree whenever quiescent
The volatile keyword

volatile variable \texttt{a} : 0; volatile variable \texttt{b} : 0;

\begin{tabular}{c c}
threadOne \{ & threadTwo \{ \\
  \texttt{b} \leftarrow 1; & \texttt{a} \leftarrow 1; \\
  \texttt{output} \leftarrow \texttt{a}; & \texttt{output} \leftarrow \texttt{b}; \\
\}
\end{tabular}

- Possible outputs:
  - \textbf{C}: 00, 01, 10, 11
  - \textbf{Java}: 01, 11
- In C, volatile says
  - that the compiler should not reorder/omit memory accesses
  - \textbf{... nothing} about what the \texttt{processor} can do!
The volatile keyword

volatile variable a : 0; volatile variable b : 0;

threadOne {
    b ← 1;
    barrier(WRITE→READ);
    output ← a;
}

threadTwo {
    b ← 1;
    barrier(WRITE→READ);
    output ← b;
}

• In Java, volatile has the “happens-before” semantics
• Memory barriers are issued implicitly when needed (by the JVM)
  □ Between accesses to volatile variables
• The parallel AVL tree implementation relies on this
Notation

- @ stands for (“scheduling-relevant”) instructions
  - Distinguishable instructions are denoted A, B, C, ...
- - stands for potential (“inactive”) memory barriers
- | stands for chosen (“active”) memory barriers

A piece of code might be illustrated like this:
- @-@-@|@|@-@
- A-D|E-G
Modeling Dependencies Between Instructions

- **A binary relation**
  - Dependencies of the form *A must happen before B*
  - Too conservative in some cases
    - How does one say *Either X or Y must happen before Z?*
  - Modeling of reorderings troublesome
    - The relation is **not** known beforehand
    - Should be derived from formal verification results
    - Reordering one instruction might affect many related pairs
  - There is an obvious mismatch
    - Let $\mathbf{I}$ denote a sequence of instructions of an algorithm
    - # subsets of the set of *feasible* permutations on $\mathbf{I}$
    - # possible antirefl. antisym. binary relations on $\mathbf{I}$
    - Binary relations and sets of permutations don't match
Modeling Dependencies Between Instructions

- A set of valid permutations
  - The most general representation
  - Can be obtained using a verification tool and **Brute Force**
- Based on the input data...
  - Preconditions and environment characteristics
  - Representation of the algorithm (sequence of instructions)
  - Postconditions
- ...try the following:
  - For each subset of **potential** memory barriers B
    - For each permutation of instructions P
      - If P conforms to the environment characteristics and does not violate the barriers in B
        - Invoke a verification tool (NuSMV?)
        - If incorrect then move to **next** subset
      - All permutations passed verification → note the subset
Traversing the Sets of Barriers

- A simple scenario that can be expressed using a binary relation
- Subsets of potential barriers form a standard lattice
- Key observations:
  - Addition of a barrier cannot make a valid algorithm invalid
  - Removal of a barrier cannot make an invalid algorithm valid
- Consequently...
  - The lattice can be split into two disjoint parts
  - Both parts are coherent, separable with a single “borderline”
- We are looking for...
  - The lowest possible number of barriers
  - The lowest possible segment of the borderline
- But the borderline can be exponentially long
Optimizations

- Only the Matrix can handle this:
  - For each subset of **potential** memory barriers \( B \)
    - For each permutation of instructions \( P \)
      - If \( P \) conforms to the environment characteristics and does not violate the barriers in \( B \)
        - Invoke a verification tool (NuSMV?)
        - ... ... ...
  - Optimizing the generator of permutations
    - Knuth's Dancing Links method
    - Never traverse permutations that violate the constraints
    - Constraints are based on environment characteristics
  - Incremental computation of the set of barriers
    - **Start with all potential barriers activated**
    - Removing one-by-one right from the start ... would it work?
The Greedy Algorithm (1/13)

- Incremental computation of the set of barriers
  - **Start with all potential barriers activated**
  - Remove one-by-one right *from the start* ... is this correct???
  - Well, under certain assumptions...
- From the start ← What does this mean?
  - We need a valid permutation of instructions
    - Satisfies postconditions if preconditions are met
  - User input ← A working model (with fixed instructions ordering)
  - The search may **not** traverse all the permutations!
  - But may still compute the lowest possible number of barriers
    - **Under certain assumptions**...
The Greedy Algorithm (2/13)

- Key advantages
  - **Linear** (rather than exponential) in \# potential barriers
  - **Lazy** generating of permutations
    - Based on already chosen memory barriers
    - No need for brute-force traversal of all permutations

- Summary of the algorithm
  - Activate **all** the potential memory barriers
  - For each memory barrier \( B \) in the input order
    - Deactivate \( B \)
    - Verify **new** permutations obtained by deactivation of \( B \)
    - If an invalid permutation is found
      - Re-activate \( B \)
What is going on in the tree of permutations?

Sequences of instructions == paths from the root to leaves

Root == an empty \textbf{tail} of the sequence

Initially, all barriers are active → one single path is valid

Barriers deactivated from the start (i.e., from the leaves)

- Subtrees of the tree need to be verified
- Lookup based on the fixed suffix (with all barriers active)

\textbf{0}-complete subtree

- \( k_0 \) initial barriers removed, all accessible permutations correct

\textbf{1}-complete subtree

- Active barrier on position \( k_0 \)
- \( k_1 \) inactive barriers after the first barrier
- All \( k_1! \) paths from current position lead to \textbf{0}-complete trees
- “Current position” well defined by the fixed suffix
The Greedy Algorithm (4/13)

- n-complete subtree
  - Active barriers on positions $k_0, k_1, \ldots, k_{(n-1)}$
  - For $m$ in $1 \ldots n$: $k_m$ inactive barriers after the $(m-1)$th barrier
  - All $k_n!$ paths from curr. position reach $k_{(n-1)}$-complete trees
  - Again, “current position” well defined by the fixed suffix

- The Greedy Algorithm again, just expressed otherwise

  - pos := 0
  - for each $m$ from $0$:
    - $k_m := 0 \leftarrow$ You obtain a trivial $m$-complete subtree
    - do { ++$k_m$, ++pos } while current subtree is $m$-complete and pos $<$ # barriers
      - Barrier deactivation and verification takes place here
    - Re-activate the barrier at position pos
    - if $k_m ==$ # barriers exit
    - ++$m$
For a potential barrier \( B \), let \( \text{left}(B) \) and \( \text{right}(B) \) denote the sets of potential barriers left and right of \( B \), respectively.

**Definition**

A potential barrier \( B \) is **strongly isolating** left-to-right iff the following implication holds:

For an arbitrary correct set of active barriers containing \( B \), activating arbitrary additional barriers from \( \text{left}(B) \) does not change the set of required barriers from \( \text{right}(B) \).

**Hypothesis**

For an algorithm model in which **all** potential barriers for **all** valid instruction reorderings are **strongly isolating** left-to-right, the left-to-right **Greedy Algorithm** will find a smallest (WRt inclusion) set of active barriers.
In most cases (and perhaps in all cases for certain memory models and algorithms), all barriers are strongly isolating. Else the Greedy Algorithm might not find the best solution.

An attempt to prove the hypothesis

Consider the output of the Greedy Algorithm:
- under the assumption of left-to-right strongly isolating barriers
- started with an arbitrary correct permutation of instructions of an arbitrary algorithm
- Let us denote the resulting set of active barriers $G$ (Greedy)
- Let us assume, for the sake of contradiction, that there exists a strictly smaller correct set of active barriers $E$ (Enemy)
“No matter what solution the enemy produces, our solution will be at least as good”

An outline of the proof follows:

First barrier $\mathcal{B}$ where solutions differ: Which one is active?

- Our $\mathcal{B}$ $\rightarrow$ enemy's solution is wrong
- Enemy's $\mathcal{B}$ $\rightarrow$ Look for next $\mathcal{B}'$ active in any solution
  - No $\mathcal{B}'$ $\rightarrow$ $\#G() + 1 = \#E()$
  - Both $\mathcal{B}'$ $\rightarrow$ activate $\mathcal{B}$; induction; $\#G() < \#E()$
  - Enemy's $\mathcal{B}'$ $\rightarrow$ enemy's solution is not minimal
- Our $\mathcal{B}'$ $\rightarrow$ Look for next $\mathcal{B}''$ active in any solution
  - No $\mathcal{B}''$ $\rightarrow$ $\#G() = \#E()$
  - Both $\mathcal{B}''$ $\rightarrow$ activate $\mathcal{B}$ and $\mathcal{B}';$ induction; $\#G() \leq \#E()$
  - Enemy's $\mathcal{B}''$ $\rightarrow$ activate $\mathcal{B}$ and $\mathcal{B}';$ induction; - $||$ -
- Our $\mathcal{B}''$ $\rightarrow$ enemy's solution is wrong
Let us consider the **first** potential barrier (from left to right) where the two solutions differ:

- If G activated a barrier $B$ inactive in the output of E:
  - ... $B$ ...
  - G: ... @-@|@-@|@ ...
  - E: ... @-@-@-@|@ ...

Let us consider the incorrect permutation $P$ that caused the Greedy Algorithm to reactivate $B$.

Then applying $P$ to $\text{left}(B)$ and ID to $\text{right}(B)$ yields a permutation valid according to E, but incorrect.

This contradicts the statement that E returned a correct solution.

Let $\#G(X)$ and $\#E(X)$ denote the **sizes** of the sets of active barriers (over a subsequence $X$) returned by G and E.
Let us consider the **first** potential barrier (from left to right) where the two solutions differ:

- If E activated a barrier \( B \) inactive in the output of G:
- Let us consider the earliest subsequent barrier \( B' \), active in any of the two solutions:

\[
\begin{array}{ccc}
\ldots & B & B' \ldots \\
G: & \ldots @-@-@- -@?@ \ldots \\
E: & \ldots @-@ | @- -@?@ \ldots \\
\end{array}
\]

- If there is no \( B' \), then \( \#G() < \#E() \), which contradicts the enemy's statement that \( \#G() > \#E() \)
- If \( B' \) is active according to both G and E, activate \( B \) in G (based on strong isolation by \( B' \)) and apply induction to \( \text{right}(B') \)

For sequences of 1 instruction and 2 instructions, G **always** returns the optimum solution.
Let us consider the **first** potential barrier (from left to right) where the two solutions differ:

- If E activated a barrier $B$ inactive in the output of G:
- Let us consider the earliest subsequent barrier $B'$, active in any of the two solutions:

  
  - $\ldots | @-@-@-\ldots @?@ \ldots$
  - $\ldots @-@-@-\ldots @?@ \ldots$

  If $B'$ is active only according to E, then $B$ can be safely deactivated in E based on (1) the correctness of G and (2) on strong isolation ($(re)activation$ of $B$ must have no effect on $right(B')$ and therefore the same holds for deactivation). This yields a contradiction to the minimality of $\mathcal{E}$.

  - If $B'$ is active only according to G, let us look for the first active $B''$ following $B'$.
Let us consider the **first** potential barrier (from left to right) where the two solutions differ:

- If E activated a barrier B inactive in the output of G:
- Let us consider the earliest subsequent barrier B', active in any of the two solutions:
- If B' is active according to G, let us look for the first active B'' following B':

  - ... B B' B'' ...
  - G: ... @-@-@- - -@ | @- - -@?@ ...  
  - E: ... @-@ | @- - -@-@ - -@?@ ...  

  If B'' is active only according to E, then activate B in G and B' in E (both of which does not affect right(B'')) and adds one barrier evenly to both solutions) and apply induction on right(B'), based on the fact that \( \#E(left(B'')) = \#G(left(B'')) \), yielding \( \#G() \leq \#E() \).
Let us consider the first potential barrier (from left to right) where the two solutions differ:

If E activated a barrier $B$ inactive in the output of $G$:

Let us consider the earliest subsequent barrier $B'$, active in any of the two solutions:

If $B'$ is active according to $G$, let us look for the first active $B''$ following $B'$:

```
... B B' B'' ...
```

- $G$: ...
  ... @-@-@-@-...@|@-...@?@ ... 
- $E$: ...
  ... @-@|@-...@-@-...@?@ ... 

If $B''$ is active only according to $G$, then consider the incorrect permutation $P$ that caused the Greedy Algorithm to reactivate $B''$. Consider an incorrect permutation $P'$ composed of $[\text{ID}({\text{left}(B')}) + P({\text{right}(B') \cap \text{left}(B''))} + \text{ID}({\text{right}(B''))}]$. (It is easy to show that this mapping is a permutation.) Then $P'$ is valid according to $E$ and incorrect.
Let us consider the first potential barrier (from left to right) where the two solutions differ:

If E activated a barrier $B$ inactive in the output of G:

Let us consider the earliest subsequent barrier $B'$, active in any of the two solutions:

If $B'$ is active according to G, let us look for the first active $B''$ following $B'$:

- $\ldots B \quad B' \quad B'' \ldots$
- $G: \ldots @-@-@-@-@|@-\ldots\?\?@ \ldots$
- $E: \ldots @-@|@-\ldots @-@-@-@\ldots\?\?@ \ldots$

If there is no $B''$, then $\#E() = \#G()$, which contradicts the enemy's statement that $\#E() < \#G()$

If $B''$ is active according to both E and G, activate $B$ in G and $B'$ in E and apply recursion to $right(B'')$, yielding $\#G() \leq \#E()$
The Greedy Algorithm: Outline (revisited)

○ “No matter what solution the enemy produces, our solution will be at least as good”

○ An outline of the proof follows:

○ First barrier \( B \) where solutions differ: Which one is active?
  - Our \( B \) → enemy's solution is wrong
  - Enemy's \( B \) → Look for next \( B' \) active in any solution
    - No \( B' \) → \#G() + 1 == \#E()
    - Both \( B' \) → activate \( B \); induction; \#G() < \#E()
    - Enemy's \( B' \) → enemy's solution is not minimal
    - Our \( B' \) → Look for next \( B'' \) active in any solution
      - No \( B'' \) → \#G() == \#E()
      - Both \( B'' \) → activate \( B \) and \( B' \); induction; \#G() <= \#E()
      - Enemy's \( B'' \) → activate \( B \) and \( B' \); induction; - || -
      - Our \( B'' \) → enemy's solution is wrong
Preconditions **do matter**
- Induction only applied with equal sets of barriers...
- ...on the left side (in the default algorithm variant)
- Otherwise we may yield a different set of initial states!
- Additional barriers must be added “symmetrically”

Is there a proof without the strong isolation assumption?
- Dunno
- The presented proof fails without strong isolation
Open Problems (1/3)

- A suitable modeling language
  - Low level Load/Store operations, preferably pointer-aware
  - Awareness of cache lines...?

- Verification backend
  - NuSMV, Promela/SPIN, something else?
  - Separate calls per permutation vs. permutations in the model

- Most of the mappings
  - (memory model, algorithm) → X
  - X → a set of formal models to verify
  - Preferably, X should be constructed incrementally...
  - ...based on previous verification results
  - ...so that the tree of permutations can be pruned
Open Problems (2/3)

- “Memory-model-aware” generator of permutations
  - Exploiting Knuth's Dancing Links as far as possible
  - Avoiding infeasible instruction reorderings
  - Extensible for multiple memory models (SPARC, Java)

- Representation of the tree of permutations
  - Efficiently compressed subtrees
  - Subtrees instantiated on-demand, based on Greedy Algorithm
  - Heuristics to disprove (sub)tree completeness quickly
    - Highest number of transpositions first...?

- Types of barriers
  - JVM only uses one on Intel, the “ultimate” one
  - The Greedy Algorithm also knows only one barrier type :-(
  - SPARC has 15 types of barriers...
Open Problems (3/3)

- Branching
  - How do we represent speculative execution?
  - The number of permutations may explode

- Cycles
  - No counter-based cycles in the AVL tree
  - Infinite structure traversal – can it be represented?
  - Abstract key intervals and related correctness conditions

- Number of iterations / recursion levels to model and verify
  - Readers never race against each other
  - Neither do writers (hand-over-hand locking proven separately)
  - Readers race against writers
    - Readers are “invisible”
    - A reader may observe multiple operations...
    - ...performed by multiple writers
Looking for a Feasible Goal

- **Extreme simplification**
  - Only one writer
  - Arbitrary number of readers
  - Readers run in RCU read-side critical sections
  - The writer waits for \( \geq \) a grace period after each operation

- **Why?**
  - It would be possible to restrict the model...
  - ...to one simple **non-recursive** operation without cycles
  - Testing the approach on the elementary scenarios
    - Readers vs. one balancing rotation
    - Readers vs. one addition / removal of a node
  
- **Transformation** of the model into a working NuSMV input
Looking for a **Feasible** Goal

- A reasonable generator of instruction reorderings

**First stage**
- **Input:**
  - Abstract memory model representation
  - Representation of the algorithm
- **Output:**
  - A set of rules and tokens for the second stage

**Second stage** – a generator of permutations (Dancing Links)
- **Input:**
  - Output from the first stage...
  - An optional set of ordering **heuristics**
- **Output:**
  - A list of possible reorderings
Conclusion

- Unnecessary memory barriers are extremely costly
- Optimizations tasks are mostly beyond human capabilities
  - What is the lowest possible number of memory barriers?
  - Are the barriers placed correctly?
  - Are the barriers too strong / too weak?
- Model-checking tools take the order of operations for granted
  - McKenney's model of QRCU in Promela: Explicit reordering
  - Is it possible to generate and verify possible reorderings?
- How does one translate the parallel AVL algorithm into C?
  - By adding memory barriers conservatively, just like JVM
  - By intuition (!) (a good servant, but a poor master)
  - Is there a more sophisticated approach :-)