Workspace Consistency: A Programming Model for Shared Memory Parallelism

Amitai Aviram, Bryan Ford, and Yu Zhang
Yale University
http://dedis.cs.yale.edu/

WoDet '11, March 6, 2011
Question

Do we want just deterministic program execution, or a deterministic programming model?

This talk will not answer that question, but maybe it'll make you think about it.
Ex. 1: Variable Swap

Many languages have “parallel assignment”

\[ x, y := y, x \]

Not true parallelism, but similar in spirit to “two assignments in parallel”:

\[ x := y \quad \text{and} \quad y := x \]

*Implicit barrier: reads before writes*

Has predictable, deterministic semantics!
Workspace Consistency (WC)

Simplistic definition:
A consistency model where parallel threads
1. Read shared state, “check out” a workspace
2. Execute and mutate *private* workspace
3. Write or “check in” changes to shared state

Just like “parallel assignment” semantics (or version control – see previous talk!)
Ex. 1 (cont): Variable Swap

Suppose we write a “parallel assignment” with real threads running assignments in parallel:

```
{ x := y } // { y := x }
```

How does this code behave under:

- Sequential consistency (SC)?
- SC with assignments in locks/transactions?
- Workspace consistency (WC)?
Ex. 1 (cont): Variable Swap

access orderings possible under sequential consistency

sequential consistency with atomicity

workplace consistency

{x := y} {y := x} //

{y := x} {x := y}

{y := x} {x := y}

{x := y} {y := x}

R y
W x

R y
W x

R x
W y

R x
W y

x overwrites y

y overwrites x

swap x and y
Ex. 2: Gaming/Simulation, Sequential Consistency

```c
struct actorstate actor[NACTORS];

void update_actor(int i) {
    ...examine state of other actors...
    ...update state of actor[i] in-place...
}

int main() {
    ...initialize state of all actors...
    for (int time = 0; ; time++) {
        thread t[NACTORS];
        for (i = 0; i < NACTORS; i++)
            t[i] = thread_fork(update_actor, i);
        for (i = 0; i < NACTORS; i++)
            thread_join(t[i]);
    }
}
```
Ex. 2: Gaming/Simulation, Sequential Consistency

```c
struct actorstate actor[NACTORS];

void update_actor(int i) {
  ...examine state of other actors...
  ...update state of actor[i] in-place...
}

int main() {
  ...initialize state of all actors...
  for (int time = 0; ; time++) {
    thread t[NACTORS];
    for (i = 0; i < NACTORS; i++)
      t[i] = thread_fork(update_actor, i);
    for (i = 0; i < NACTORS; i++)
      thread_join(t[i]);
  }
}
```

oops! corruption/crash due to race
Ex. 2: Gaming/Simulation, Workspace Consistency

struct actorstate actor[NACTORS];

void update_actor(int i) {
    ...examine state of other actors...
    ...update state of actor[i] in-place...
}

int main() {
    ...initialize state of all actors...
    for (int time = 0; ; time++) {
        thread t[NACTORS];
        for (i = 0; i < NACTORS; i++)
            t[i] = thread_fork(update_actor, i);
        for (i = 0; i < NACTORS; i++)
            thread_join(t[i]);
    }
}
Why Workspace Consistency?

Offers *naturally deterministic* parallel semantics

- Synchronization semantics always *fully define* which prior computed results *do* – and *don't* – affect which blocks of code.
- Behavior depends *only* on structure of code, not on any imposed schedule, *real or artificial*.
- *Less familiar* than sequential consistency, but perhaps equally or *more intuitive*?

(wild subjective claim alert!)
In the simple **fork/join** model, this is *not* new

- Early: Parallel Fortran systems
  - Burroughs FMP [Schwartz '80]
  - Myrias Parallel Computer [Beltrametti '88]
- ... 
- Recent: Language and OS mechanisms
  - Revision/Isolation Types [Burckhardt '10]
  - Determinator OS [Aviram '10]
Many Dimensions of Determinism

<table>
<thead>
<tr>
<th>Deterministic <strong>synchronization semantics</strong></th>
<th>Cilk</th>
<th>Kendo</th>
<th>Grace</th>
<th>CoreDet</th>
<th>Determinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic <strong>synchronization behavior</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deterministic <strong>memory access behavior</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deterministic <strong>speculation-free execution</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Workspace Consistency
Defining and Generalizing WC

This paper attempts to:

- Give this programming model a name :) 
- Define WC more precisely
  - Identify exact conditions for determinism
- Generalize to high-level parallel abstractions
  - Including *non-hierarchical*: pipelines, futures, arbitrary dependency graphs
- Map high-level constructs to simple primitives
Defining Workspace Consistency

1. Deterministic synchronization semantics:
   - Synch occurs at release and acquire events
   - Program logic alone uniquely defines:
     • At what points release, acquire events occur
     • Which release event “feeds” each acquire

2. Deterministic memory behavior:
   - Updates propagate between threads when, and only when, mandated by synchronization
   - Conflicting parallel writes handled deterministically at relevant acquire
Synchronization in WC

“Primitive” approach:
- Label each **release** (threadID, release#)
- Each **acquire** explicitly names specific **release**
- **acquire** waits for, merges updates from that **release**
Why is WC deterministic?

Inductive argument (informal):

- **Assuming:**
  - Sequential code is deterministic
  - No cyclic dependencies (deadlock-free)
- Program state at each **release** determined uniquely by *causally prior* code, data, events
- State feeding into each **acquire** determined uniquely by own state + paired **release**

Yields *unique* synchronization graph
Example Primitive Thread API

Primitive thread/synchronization operations:

- \( t \leftarrow \text{fork}(\text{fun}) \): create new thread \( t \) at version 0
- \text{rel}(): release current thread's next version #
- \text{acq}(t, \nu): acquire version \( \nu \) from thread \( t \)
- \text{exit}(): stop thread and release version \( \infty \)

First \text{release}() in thread \( t \) releases version 1, etc.
High-level Synchronization

What is interesting about this primitive API?
1. It satisfies WC's conditions for determinism
2. Many high-level synchronization abstractions are easily constructed atop it.

Examples:
- Hierarchical: fork/join, barrier, task, reduction
- Non-hierarchical: pipeline, future
Example: Fork/Join, Futures

(as in Revision/Isolation Types model)
Example: Barrier Synchronization

\[ \text{barrier}() \]

\[ \text{compute} \]

\[ \text{acq}(t1,n) \]
\[ \text{rel}() \]
\[ \text{acq}(t0,n) \]

\[ \text{barrier}() \]

\[ \text{compute} \]

\[ \text{acq}(t2,n) \]
\[ \text{rel}() \]
\[ \text{acq}(t0,n) \]

\[ \text{barrier}() \]

\[ \text{compute} \]

\[ \text{acq}(t1,n) \]
Example: Partially Ordered Loops

OpenMP feature:
parallelize a loop body across iterations, but order an interior block by iteration count

```c
#pragma omp parallel for
for (i = 0; i < 3; i++) {

  parallel block

  #pragma omp ordered
  {
    serial block
  }

  parallel block

  }
```
Example: Partially Ordered Loops

\[ \text{acq}(t_0, 1) \]

\[ \text{rel}() \]

\[ \text{serial} \]

(Ordered block)

\[ \text{acq}(t_1, 1) \]

\[ \text{rel}() \]

\[ \text{serial} \]

Parallel

Serial
Example: Simple Pipelines

Stage 1
rel()

Stage 2
rel()

Stage 3
rel()

Stage 1
acq(t0,1)

Stage 2
acq(t0,2)

Stage 3
acq(t0,3)

Stage 1
acq(t1,1)

Stage 2
acq(t1,2)

Stage 3
acq(t1,3)
Tasks with Explicit Dependencies

- **I-frame**
  - slice 1
  - slice 2
  - slice 3

- **B-frame**
  - slice 1
  - slice 2
  - slice 3

- **B-frame**
  - slice 1
  - slice 2
  - slice 3

- **P-frame**
  - slice 1
  - slice 2
  - slice 3

Output video frames:

- slice 1
- slice 2
- slice 3
What does WC not support?

Nondeterministic synchronization: e.g., mutexes
What else does WC not support?

Not sure – Future Work!
Determinizing Memory Access

WC assumes *serial code* is deterministic
Only true if we either:

- Assume code is race-free (weak determinism)
- Determinize regular memory accesses too

For strong determinism, we must “diff-&-merge” memory over *arbitrary* synchronization graphs

- Not necessarily straightforward!
Determinizing Memory Access

Build on Lazy Release Consistency (LRC)

- Created for distributed shared memory (DSM)
- Supports version-based update propagation via release/acquire pairing, as WC requires
- **Key change for WC:** propagate updates at – and never before – acquire/release pairs
  - No “eager update pushes” as in usual LRC
- Practical? Remains to be seen!
Implementation Work In Progress

High-level: **Deterministic OpenMP (DOMP)**

- Builds on OpenMP for familiarity, convenience
- Enriches OpenMP's deterministic abstractions
  - General user-defined *reductions*
  - *Pipelines* and multiple *ordered* blocks
  - Explicit, dynamic *task synchronization*
- Reduce programmers' need to “escape” to non-deterministic synchronization
  - Further details to appear in HotPar '11
Low-level: **Determinator OS** [OSDI '10]

- Add *non-hierarchical synchronization* via producer/consumer virtual memory.

![Diagram of memory and synchronization]
Conclusion

Workspace Consistency gives programs deterministic **semantics**, not just **execution**

- Simple memory state propagation model
  - As in parallel assignment, version control
- Can support interesting high-level abstractions
  - Fork/join, future, barriers, pipelines, etc.
- Reduce to powerful **release/acquire** primitives

**Determinator**-based prototype in progress

http://dedis.cs.yale.edu