Lightweight Analyses For Reliable Concurrency

Stephen Freund
Williams College

joint work with Cormac Flanagan (UCSC),
Shaz Qadeer (MSR)

Part 3: Beyond Reduction

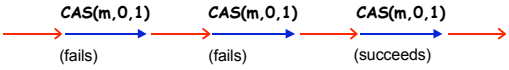
Busy Acquire

```
atomic void busy_acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```

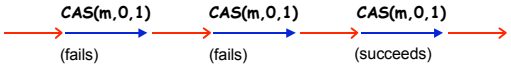
```
if (m == 0) {
  m = 1; return true;
} else {
  return false;
}
```

Busy Acquire

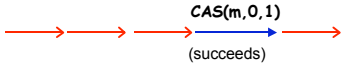
```
atomic void busy_acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```



- Non-Serial Execution:



- Serial Execution:

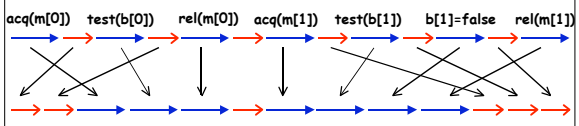


- Atomic but not reducible

alloc

```
boolean b[MAX]; // b[i]==true iff block i is free
Lock m[MAX];

atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    acquire(m[i]);
    if (b[i]) {
      b[i] = false;
      release(m[i]);
      return i;
    }
    release(m[i]);
    i++;
  }
  return -1;
}
```

alloc

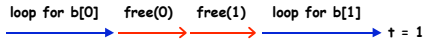



```
m[0] = m[1] = 0; b[0] = b[1] = false;

t = alloc();  ||  free(0); free(1);
```

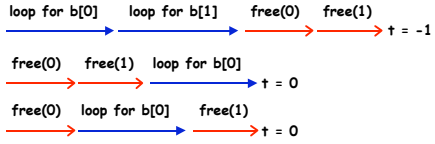
```
void free(int i) {
  acquire(m[i]);
  b[i] = true;
  release(m[i]);
}
```

```
m[0] = m[1] = 0; b[0] = b[1] = false;

t = alloc();  ||  free(0); free(1);
```

- Non-Serial Execution:



- Serial Executions:



Extending Atomicity

- Atomicity doesn't always hold for methods that are "intuitively atomic"
  - serializable but not reducible (busy_acquire)
  - not serializable (alloc)
- Examples
  - initialization
  - caches
  - resource allocation
  - commit/retry transactions
  - wait/notify

Abstraction

- Want to extend reduction-based tools to check atomicity at an abstract level
  - abstract semantics
    - admits more execution traces
    - hides some "irrelevant" details of execution (ie, failed iterations of alloc, busy_wait)
  - apply reduction to abstract semantics
- Must still be able to reason about important program properties in abstract semantics

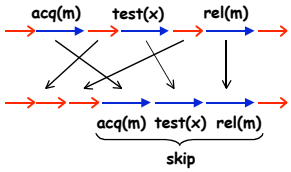
Pure Code Blocks

- Pure block: `pure { E }`
  - If `E` terminates normally, it does not update state visible outside of `E`
  - `E` is reducible
- Example

```
while (true) {
  pure {
    acquire(mx);
    if (x == 0) { x = 1; release(mx); break; }
    release(mx);
  }
}
```

Purity and Abstraction

- A `pure` block's behavior under normal termination is the same as `skip`



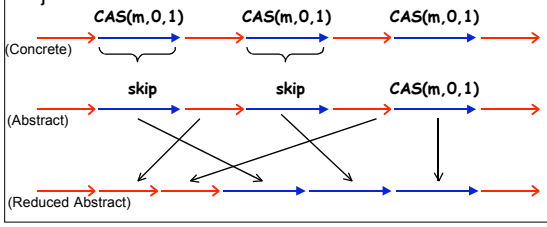
- Abstract execution semantics:
  - `pure` blocks can be skipped

Busy Acquire

```
atomic void busy_acquire() {
  while (true) {
    pure { if (CAS(m,0,1)) break; }
  }
}
```

Abstract Execution of Busy Acquire

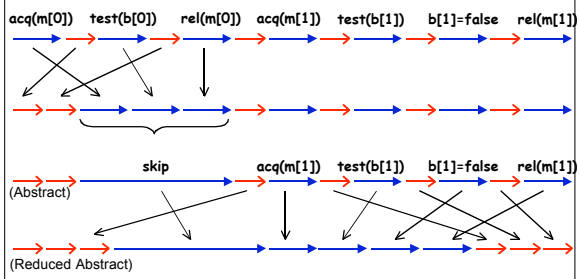
```
atomic void busy_acquire() {
  while (true) {
    pure { if (CAS(m,0,1)) break; }
  }
}
```



alloc

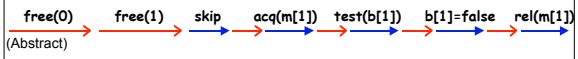
```
atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    pure {
      acquire(m[i]);
      if (b[i]) {
        b[i] = false;
        release(m[i]);
        return i;
      }
      release(m[i]);
    }
    i++;
  }
  return -1;
}
```

Abstract Execution of alloc



Abstraction

- Abstract semantics admits more executions



- Can still reason about important properties
  - "`alloc` returns either the index of a freshly allocated block or -1"
  - cannot guarantee "`alloc` returns smallest possible index"
    - but what does this really mean anyway???

Type Checking

```
atomic void deposit(int n) {
  acquire(this);        R
  int j = bal;          B
  bal = j + n;          B
  release(this);        L
}
```

((R;B);B);L =
(R;B);L =
R;L =
A

```
atomic void depositLoop() {
  while (true) {
    deposit(10);        A
  }
}
```

(A)* = C ⇒ ERROR

Normal and Exceptional Atomicities

- Track pair of atomicities for expression e:

$X \uparrow Y$

normal exceptional

  - X: atomicity if e is evaluated to completion
  - Y: atomicity if e terminates early (because of a **break** or **return** statement)

- Examples:
  - $A \uparrow \bot$ $B \uparrow A$ $\bot \uparrow A$

Type Checking with Purity

```
atomic int alloc() {
  int i = 0;
  while (i < MAX) {
    pure {
      acquire(m[i]);
      if (b[i]) {
        b[i] = false;
        release(m[i]);
        return i;
      }
      release(m[i]);
    }
    i++;
  }
  return -1;
}
```

A↑A B↑A (B*;A) ↑ ⊥

Atomicity Algebra

- $e_1;e_2$:
  - $(u \uparrow v) ; (x \uparrow y) = u;x \uparrow (v \sqcup u;y)$
- **while (true) e**:
  - $(x^* ; y) \uparrow \bot$, where e has atomicity $x \uparrow y$
- **break**
  - $\bot \uparrow B$
- **pure e**
  - $B \uparrow b$, where e has atomicity $a \uparrow b$ and $a,b \sqsubseteq A$

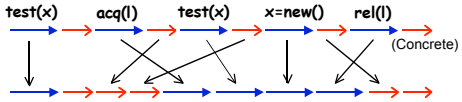
Double Checked Initialization

```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null)
    x = new();
  release(l);
}
```

Double Checked Initialization

```
atomic void init() {
  if (x != null) return;
  acquire(l);
  if (x == null)
    x = new();
  release(l);
}
```

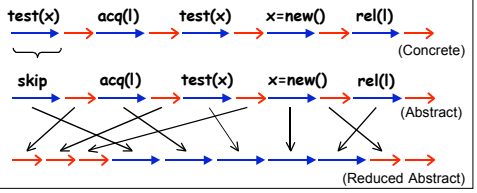
conflicting accesses



Double Checked Initialization

```
atomic void init() {
  pure {
    if (x != null) return;
  }
  acquire(l);
  if (x == null)
    x = new();
  release(l);
}
```

A↑A B↑A A↑⊥ B;A ↑ (A ⊔ B;⊥) = A↑A



Transactions and Optimistic Concurrency Control

```
global int x;

atomic void apply_f() {
  acq(m);
  x = f(x);
  rel(m);
}
```

- Interesting case:
  - reads of x more frequent than updates
  - f(x) takes a long time to compute
  - want updates to x to appear atomically
- Similar problem to database update, etc.

Transactions and Optimistic Concurrency Control (2nd Attempt)

```
global int x;

atomic void apply_f() {
  int l, fl;
  acq(m);
  l = x;          } read(x)
  rel(m);
  fl = f(l);
  acq(m);
  x = fl;         } write(fl)
  rel(m);
}
```

Not serializable:

Transactions and Optimistic Concurrency Control (3rd Attempt)

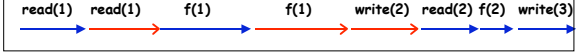
```
atomic void apply_f() {
  int l, fl;
  while (true) {
    acq(m);
    l = x;
    rel(m);

    fl = f(l);

    acq(m);
    if (l == x) {
      x = fl;
      rel(m);
      break;
    }
    rel(m);
  }
}
```

l is local copy of x

fl = f(l);

if x is still l, then x = fl and exit (else retry)

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    acq(m);
    l = x;
    rel(m);

    fl = f(l);

    acq(m);
    if (l == x) {
      x = fl;
      rel(m);
      break;
    }
    rel(m);
  }
}
```




```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);
      l = x;
      rel(m);
    }

    fl = f(l);

    pure {
      acq(m);
      if (l == x) {
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

Are these blocks pure?

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);
      l = x;
      rel(m);
    }

    fl = f(l);

    pure {
      acq(m);
      if (l == x) {
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

≈

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);
      l = *;
      rel(m);
    }

    fl = f(l);

    pure {
      acq(m);
      if (l == x) {
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

Local Variables and Abstraction

- Abstraction rule:
  - pure blocks may be skipped
  - pure blocks can change local vars
  - reads of global (shared) vars yield arbitrary value

```
global g;
local l;

pure {
  acq(m);
  l = g;
  rel(m);
}
```

≈

```
global g;
local l;

pure {
  acq(m);
  l = *;
  rel(m);
}
```

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);
      l = x;
      rel(m);
    }

    fl = f(l);

    pure {
      acq(m);
      if (l == x) {
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

Are these blocks atomic?

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);     R↑⊥
      l = x;      B↑⊥   } A↑⊥
      rel(m);     L↑⊥
    }

    fl = f(l);

    pure {
      acq(m);
      if (l == x) {
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

A↑A

```
atomic void apply_f() {
  int l, fl;
  while (true) {
    pure {
      acq(m);     R↑⊥
      l = x;      B↑⊥   } A↑⊥   } B↑⊥
      rel(m);     L↑⊥
    }

    fl = f(l);                        } B↑⊥

    pure {
      acq(m);
      if (l == x) {                   } B↑A
        x = fl;
        rel(m);
        break;
      }
      rel(m);
    }
  }
}
```

A↑A

B↑A

(B*;A)↑⊥ = A↑⊥

Transaction Summary

- The pure blocks allow us to prove apply_f is abstractly atomic
- We can prove on the abstraction that x is updated to f(x) atomically

Unstable Variables

- Variables with intentional race conditions
  - performance counters
  - global flags
- Example:

```
int packetCount;
final Queue packets;

atomic void enqueue(Queue q, Data packet) {...}

atomic void receive(Data packet) {
  packetCount++;                A
  enqueue(packets, packet);     A
}
```

Unstable Variables

- Variables with intentional race conditions
  - performance counters
  - global flags
- Example:

Write stores and read returns arbitrary value

```
unstable int packetCount;
final Queue packets;

atomic void enqueue(Queue q, Data packet) {...}

atomic void receive(Data packet) {
  packetCount = *;              B
  enqueue(packets, packet);     A
}
```

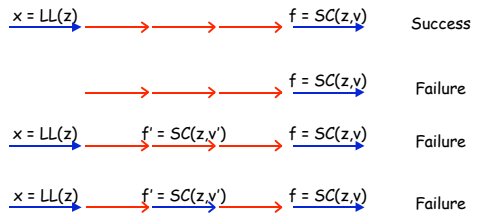
Lock-Free Synchronization

- Locking is expensive
  - requires thread coordination
  - may trigger memory barriers
  - contention and other issues
- Provide lower-level primitives for thread coordination
  - example: CAS
  - can implement locking with this "atomic" primitive
  - other instructions exist

LL/SC Instructions

- Alpha, and other architectures
  - (vaguely similar to optimistic concurrency)
- Load-linked: x = LL(z)
  - loads the value of z into x
- Store-conditional: f = SC(z,v)
  - if no SC has happened since the last LL by this thread
    - store the value of v into z and set f to true
  - otherwise
    - set f to false

Scenarios



Lock-Free Atomic Increment

```
atomic void increment() {
  int x;
  while (true) {
    x = LL(z);
    x = x + 1;
    if (SC(z,x)) break;
  }
}
```

Assume LL/SC are always used in this idiom

Commuting LL and SC for increment

- A successful SC operation is a left mover
- Unsuccessful SC operation is a both mover
  - equal to "false"
- The LL operation corresponding to a successful SC operation is a right mover

Lock-Free Atomic Increment

```
atomic void increment() {
  int x;
  while (true) {

    x = LL(z);
    x = x + 1;
    if (SC(z,x)) break;

  }
}
```

Lock-Free Atomic Increment

```
atomic void increment() {
  int x;
  while (true) {

    x = LL(z);
    x = x + 1;
    if (false ☐ SC-Success(z,x)) break;

  }
}
```

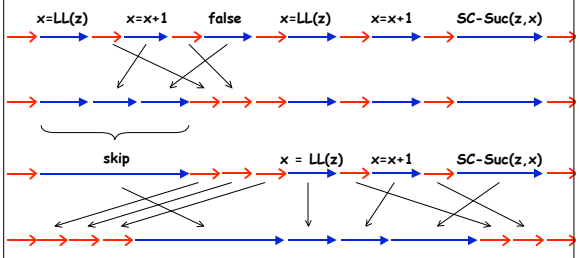
⟨z=x;true⟩

Lock-Free Atomic Increment

```
atomic void increment() {
  int x;
  while (true) {
    pure {
      x = LL(z);
      x = x + 1;
      if (false ☐ SC-Success(z,x)) break;
    }
  }
}
```

⟨z=x;true⟩

Reduction



Atomicity and Purity Effect System

- Enforces properties for abstract semantics
  - pure blocks are reducible and side-effect free
  - atomic blocks are reducible
- Leverages other analyses
  - race-freedom
  - control-flow
  - side-effect
- Additional notions of abstraction
  - unstable reads/writes, other notions of purity, LL/SC

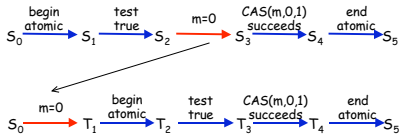
Summary

- Atomicity
  - enables sequential analysis
  - common in practice
- Purity enables reasoning about atomicity at an abstract level
  - techniques:
    - purity
    - instability
  - matches programmer intuition
  - more effective checkers

Checking Atomicity

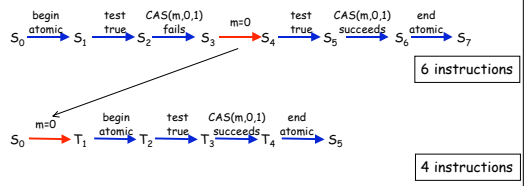
- Atomic via Reducibility
  - An non-serialized execution is atomic if it is *reducible* to a serialized execution

Limitations of Reduction




```
int m;  // lock, 0 if not held

atomic void acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```

Limitations of Reduction




```
int m;  // lock, 0 if not held

atomic void acquire() {
  while (true) {
    if (CAS(m,0,1)) break;
  }
}
```

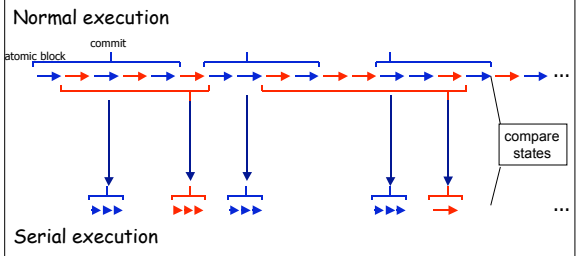
Checking Atomicity

- Atomic via Reducibility
  - An non-serialized execution is atomic if it is *reducible* to a serialized execution
- Atomic via Commit Atomicity
  - An non-serialized execution is atomic if it *has the same behavior* as a serialized execution
  - There must be a serial execution that ends in same state, but can take different steps

Commit-Atomic

- Run *normal* and *serial* executions of program concurrently, on separate stores
- Normal execution runs as normal
  - threads execute atomic blocks
  - each atomic block has *commit* point
- Serial execution
  - runs on separate *shadow* store
  - when normal execution *commits* an atomic block, serial execution runs entire atomic block serially
- Check two executions yield same behavior

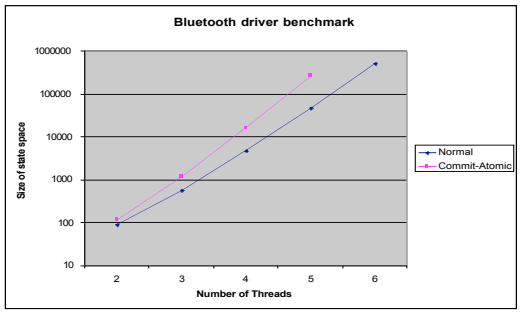
Commit-Atomic



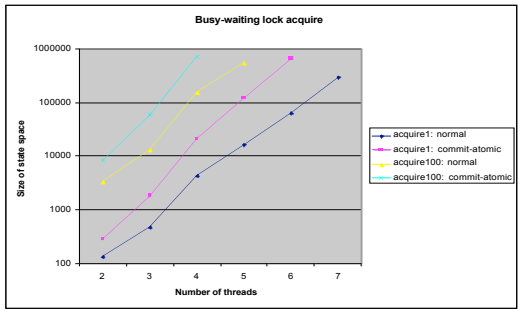
Preliminary Evaluation

- Some small benchmarks
  - Bluetooth device driver
    - atomicity violation due to error
  - Busy-waiting lock acquire
    - acquire1: 1 line of code in critical section
    - acquire100: 100 lines of code in critical section
- Hand translated to PROMELA code
  - Two versions, with and without commit-atomic
  - Model check with SPIN

Performance: Bluetooth device driver



Performance: acquire1 and acquire100



VYRD

- Dynamic checker for variant of commit-atomicity [Elmas-Tasiran-Qadeer 05]
- Use separate thread for serial execution
  - When transaction commits, run serialized version in separate thread
  - Check whether same state can be reached
- Also introduces abstraction

Summary

- Atomicity
  - concise, semantically deep partial specification
- Reduction
  - lightweight technique for verifying atomicity
- Commit-Atomicity
  - more general technique
- Future work
  - combine reduction and commit-atomic
  - generalizing atomicity
    - *temporal logics for determinism?*