Tight bounds for anonymous adopt-commit objects

James Aspnes\textsuperscript{1}    Faith Ellen\textsuperscript{2}

\textsuperscript{1}Yale
\textsuperscript{2}Toronto

June 6th, 2011
Motivation

What we really care about is shared-memory consensus:

- **Termination**: All non-faulty processes terminate.
- **Validity**: Every output value is somebody’s input.
- **Agreement**: All output values are equal.
Usual asynchronous shared-memory model:

- \( n \) concurrent processes.
- Communication by reading and writing atomic registers.
- Asynchronous, with timing controlled by an adversary scheduler.
- **Wait-free**: each process finishes in a finite number of steps.

We will be considering **anonymous** algorithms in which all processes run the same code.
Implementing consensus

- Typical implementation: use some randomized process that produces agreement with some probability, and commit to a return value when we detect agreement.
- But how to detect agreement?
(Gafni, PODC 1998; Mostefaoui et al., SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody’s input.
- **Agreement:** All output values are equal.
- **Coherence:** All output values are equal if some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.
(Gafni, PODC 1998; Mostefaoui et al., SICOMP 2008)

**Termination:** All non-faulty processes terminate.

**Validity:** Every output value is somebody’s input.

**Agreement:** All output values are equal.

**Coherence:** All output values are equal if some process commits.

**Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.
Adopt-commit objects

(Gafni, PODC 1998; Mostefaoui et al., SICOMP 2008)

- **Termination:** All non-faulty processes terminate.
- **Validity:** Every output value is somebody’s input.
- **Agreement:** All output values are equal.
- **Coherence:** All output values are equal if some process commits.
- **Acceptance:** All processes commit if all inputs are equal.

Any consensus object is also an adopt-commit object.
Any consensus object is also an adopt-commit object.

(Gafni, PODC 1998; Mostefaoui et al., SICOMP 2008)

- **Termination**: All non-faulty processes terminate.
- **Validity**: Every output value is somebody’s input.
- **Agreement**: All output values are equal.
- **Coherence**: All output values are equal if some process commits.
- **Acceptance**: All processes commit if all inputs are equal.
We show that adopt-commit is equivalent (up to small constants) to a conflict detector:

- Two operations: write and read.
- The read operation returns true if distinct values have previously been written, otherwise false.
Conflict detectors

We show that adopt-commit is equivalent (up to small constants) to a conflict detector:

- Two operations: write and read.
- The read operation returns true if distinct values have previously been written, otherwise false.
procedure write(v)
begin
    if adoptCommit(v) \neq (commit, v) then
        conflict \leftarrow true
    end
end

procedure read()
begin
    return conflict
end
procedure write(v)
begin
    if adoptCommit(v) \neq (commit, v) then
        conflict ← true
    end
end

procedure read()
begin
    return conflict
end
procedure adoptCommit(v)
begin
    conflict.write(v)
    $u \leftarrow \text{proposal}$
    if $u = \bot$ then
        proposal $\leftarrow v$
    else
        $v \leftarrow u$
    end
    if conflict.read() = false then
        return (commit, v)
    else
        return (adopt, v)
    end
end
procedure adoptCommit(v)
begin
    conflict.write(v)
    u ← proposal
    if u = ⊥ then
        proposal ← v
    else
        v ← u
    end
    if conflict.read() = false then
        return (commit, v)
    else
        return (adopt, v)
    end
end
procedure adoptCommit(v)
begin
    conflict.write(v)
    u ← proposal
    if \( u = \bot \) then
        proposal ← v
    else
        v ← u
    end
    if conflict.read() = false
    then
        return (commit, v)
    else
        return (adopt, v)
    end
end
Assign unique write quorum $W_v$ of $k$ out of $2k$ registers to each value $v$, where $k = \Theta(\log m)$ satisfies $\binom{2k}{k} \geq m$.

Write $v$ by writing all registers in $W_v$.

Check for $v' \neq v$ by reading all registers in $\overline{W}_v$.

I always see you if you finish writing $W_{v'}$.

Cost: $\Theta(\log m)$ individual work and $\Theta(\log m)$ space. Can we do better?
Conflict detector using subsets

(Aspnes, PODC 2010)

- Assign unique write quorum $W_v$ of $k$ out of $2k$ registers to each value $v$, where $k = \Theta(\log m)$ satisfies $\binom{2k}{k} \geq m$.
- Write $v$ by writing all registers in $W_v$.
- Check for $v' \neq v$ by reading all registers in $\overline{W}_v$.
- I always see you if you finish writing $W_{v'}$.

Cost: $\Theta(\log m)$ individual work and $\Theta(\log m)$ space.
Can we do better?
Conflict detector using permutations

With 2 values:

- Processes with 1 write $r_1$ then read $r_2$.
- Processes with 2 write $r_2$ then read $r_1$.
- With a conflict, whoever writes last sees the other value.
Conflicting detector using permutations

With 2 values:

- Processes with 1 write $r_1$ then read $r_2$.
- Processes with 2 write $r_2$ then read $r_1$
- With a conflict, whoever writes last sees the other value.

SPAA 2011
Tight bounds for anonymous adopt-commit objects
Conflict detector using permutations

With 2 values:

- Processes with 1 write \( r_1 \) then read \( r_2 \).
- Processes with 2 write \( r_2 \) then read \( r_1 \).
- With a conflict, whoever writes last sees the other value.
With 2 values:

- Processes with 1 write $r_1$ then read $r_2$.
- Processes with 2 write $r_2$ then read $r_1$.
- With a conflict, whoever writes last sees the other value.
With 2 values:

- Processes with 1 write $r_1$ then read $r_2$.
- Processes with 2 write $r_2$ then read $r_1$
- With a conflict, whoever writes last sees the other value.
Conflict detector using permutations

With $m$ values:

- Use $k$ registers with $k! \geq m$.
- Each value $v$ gets a distinct permutation $\pi_v$.
- Processes execute the following code:

  ```
  for $i$ in $\pi_v$ do
    $r \leftarrow r_i$
    if $r = \bot$ then
      $r_i \leftarrow v$
    else if $r \neq v$ then
      conflict $\leftarrow$ true
    end
  end
  ```

- Any distinct permutations invert some pair $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log \log m)$.
Conflict detector using permutations

With $m$ values:

- Use $k$ registers with $k! \geq m$.
- Each value $v$ gets a distinct permutation $\pi_v$.
- Processes execute the following code:

  ```
  for $i$ in $\pi_v$ do
    $r \leftarrow r_i$
    if $r = \perp$ then
      $r_i \leftarrow v$
    else if $r \neq v$ then
      conflict $\leftarrow \text{true}$
  end
  end
  ```

- Any distinct permutations invert some pair $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log \log m)$. 

12345
52143
Conflict detector using permutations

With $m$ values:

- Use $k$ registers with $k! \geq m$.
- Each value $v$ gets a distinct permutation $\pi_v$.
- Processes execute the following code:

  ```
  for $i$ in $\pi_v$ do
    $r \leftarrow r_i$
    if $r = \bot$ then
      $r_i \leftarrow v$
    else if $r \neq v$ then
      conflict $\leftarrow$ true
  end
  end
  ```

- Any distinct permutations invert some pair $\Rightarrow$ conflict detected as in two-value version.
- Cost: $\Theta(\log m / \log \log m)$. 

12345

52143
We have reduced the cost of an $m$-valued adopt-commit from
\[ \Theta(\log m) \]
to
\[ \Theta(\log m / \log \log m). \]
This is not especially exciting on its own, but we also have a matching lower bound.
**Theorem:** Any anonymous deterministic conflict detector has an input that causes a process to take $\Omega(\log m / \log \log m)$ steps in a solo execution.

**Proof outline:**

1. For each input $v$, consider set of registers accessed in resulting solo execution $E_v$.
2. Define a permutation $\pi_v$ of this set based on order of accesses.
3. If $\pi_v$ and $\pi_{v'}$ agree on order of registers accessed in both $E_v$ and $E_{v'}$, then there exists an execution where $v \neq v'$ conflict is not detected.
4. Avoiding this requires longest $\pi_v$ to have at least $\Omega(\log m / \log \log m)$ elements.
We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they’re anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution $E_v$ into clone execution $E_v^*$. 
We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they’re anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution $E_v$ into clone execution $E_v^*$. 

Using clones to hide writes
Using clones to hide writes

We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they’re anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution $E_v$ into clone execution $E_v^*$. 

2

1

Tight bounds for anonymous adopt-commit objects
We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they’re anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution $E_v$ into clone execution $E^*_v$. 
We are using a classic trick of (Fich, Herlihy, and Shavit, JACM 1998):

- Most clones do the same thing at the same time (they’re anonymous and deterministic).
- But we leave a few behind to cover any register we write.
- If we read the register again, we release a delayed write to restore our last value.
- This transforms solo execution $E_v$ into clone execution $E^*_v$. 

First-write/last-read permutation

\[ E_v = W1 \quad R2 \quad W1 \quad R3 \quad W2 \quad R1 \quad R3 \]

\[ \downarrow \quad \downarrow \quad \downarrow \]

\[ \pi_v = 1 \quad 2 \quad 3 \]

- For each register \( r \), pick the
  - First write to \( r \) if there is one, or
  - Last read from \( r \) otherwise.

- Let \( \pi_v \) list the registers in order of these operations.
Interleaved execution

Interleave $E_v^*$ and $E_{v'}^*$, according to $\pi_v \cup \pi_{v'}$ to make chosen operations on the same registers adjacent.

- Put last-reads before first-writes.
- Use delayed clones to rewrite registers before later reads.
Why the interleaving works

Restricting the view to a single register:

- If I *don’t* write to *r*, my last read of *r* comes before your first write:
  \[
  E^* \quad R2 \quad W2 \\
  E^*_v \quad R2 \quad R2
  \]

- If I *do* write to *r*, your first write happens at the same time as mine, so we can use cloned operations to mask it (and any subsequent writes):
  \[
  E^*_v \quad W1 \quad W1 \quad R1 \\
  E^*_v \quad W1 \quad R1 \quad (W1) \quad R1
  \]

⇒ Conflict detector doesn’t work unless \( \pi_v \) and \( \pi_{v'} \) are inconsistent for all \( v \neq v' \).
Claim: Any family of pairwise-inconsistent partial permutations \( \{\pi_v\} \) satisfies
\[
\sum_v \frac{1}{|\pi_v|!} \leq 1.
\]

Proof:
1. Pick a random ordering of all registers.
2. Let \( A_v \) be the event that \( \pi_v \) is increasing in this ordering.
3. \( \Pr[A_v] = \frac{1}{|\pi_v|!} \).
4. Observe that if \( \pi_v \) and \( \pi_{v'} \) are inconsistent, \( A_v \cap A_{v'} = \emptyset \).
5. \( \Rightarrow \sum \Pr[A_v] = \Pr[\bigcup A_v] \leq 1. \)

Corollary: Pigeonhole argument gives \( \frac{1}{|\pi_v|!} \leq \frac{1}{m} \) for some \( v \), which gives \( \max_v |\pi_v| = \Omega(\log m / \log \log m) \).
For a randomized conflict detector:

1. Define $E_v$ to be shortest solo execution that occurs with nonzero probability for input $v$.
2. Repeat same analysis as for deterministic executions.
3. If we can interleave $E_v^*$ and $E_{v'}^*$, there is a (small) nonzero probability that every clone flips its coins the right way, violating the spec.

So lower bound applies with probability 1 to solo executions of randomized algorithms as well.
Let $n$ be the number of processes.

- Interleaving consumes $O(1)$ clones per step.
- $\Rightarrow$ lower bound can’t exceed $\Omega(n)$.
- Can also get $O(n)$ upper bound.
- So real bound is:

$$\Theta \left( \min \left( \frac{\log m}{\log \log m}, n \right) \right)$$

Same lower bound applies for anonymous $m$-valued consensus.
Open problem

Does \( \Theta \left( \min \left( \frac{\log m}{\log \log m}, n \right) \right) \) bound hold without anonymity?

Progress so far (not in proceedings version):

- **Lower bound:**
  \[
  \Omega \left( \min \left( \frac{\log m}{\log \log m}, \frac{\sqrt{\log n}}{\log \log n} \right) \right)
  \]
  for deterministic implementations.

- **Upper bound:**
  \[
  O \left( \min \left( \frac{\log m}{\log \log m}, \log n \right) \right)
  \]