LINEARIZABILITY TESTING OF MULTI-WORD SYNCHRONIZATION PRIMITIVES

Presenter: Sakib Chowdhury

Joint work with: Diego Cepeda, Wojciech Golab, Nan Li, Raphael Lopez, Jeff Wang

August 2, 2019

OUTLINE

- Motivation
- Background
- The Algorithm
- Demonstration
- Conclusion

MOTIVATION

- Persistent memory will improve performance and reliability of data access
- We want to be sure that our behaviour is correct

MOTIVATION: CODE COMPLEXITY

- CASN
- Harris, 2002
- ~40 lines of code

```
word_t RDCSS (RDCSSDescriptor_t *d) {
  do {
    r = CAS1(d->a2, d->o2, d); /* C1 */
    if (IsDescriptor (r)) Complete(r); /* H1 */
  } while (IsDescriptor (r)); /* B1 */
  if (r == d->o2) Complete(d);
  return r;
}

word_t RDCSSRead (addr_t *addr) {
  do {
    r = *addr; /* R1 */
    if (IsDescriptor(r)) Complete(r); /* H2 */
  } while (IsDescriptor (r)); /* B2 */
 return r;
}

void Complete (RDCSSDescriptor_t *d) {
  v = *(d->a1); /* R2 */
  if (v==d->o1) CAS1(d->a2, d, d->n2); /*
    else CAS1(d->a2, d, d->o2); /* C3 */
}
```

```
bool CASN (CASNDescriptor_t *cd) {
  if (cd->status == UNDECIDED) { /* R4 */
phase_1: status = SUCCEEDED;
    for (i = 0; (i < cd->n) && (status == SUCCEEDED) ; i++) { /* L1 */
retry_entry: entry = cd->entry[i];
        val = RDCSS (new RDCSSDescriptor_t (&(cd->status), UNDECIDED,
                                  entry->addr, entry->old, cd)); /* X1 */
        if (IsCASNDescriptor_t (val)) {
          if (val != cd) {
            CASN (val); /* H3 */
            goto retry_entry;
          }
        } else if (val != entry->old) status = FAILED;
      }
    CAS1 (&(cd->status), UNDECIDED, status); /* C4 */
  }
phase_2: succeeded = (cd->status == SUCCEEDED);
  for (i = 0; i < cd->n; i ++)
    CAS1 (cd->entry[i].addr, cd,
        succeeded ? (cd->entry[i].new) : (cd->entry[i].old)); /* C5 */
  return succeeded;
}

word_t CASNRead (addr_t *addr) {
  do {
    r = RDCSSRead(addr); /* R5 */
    if (IsCASNDescriptor (r)) CASN (r); /* H4 */
  } while (IsCASNDescriptor (r)); /* B3 */
  return r;
}
```

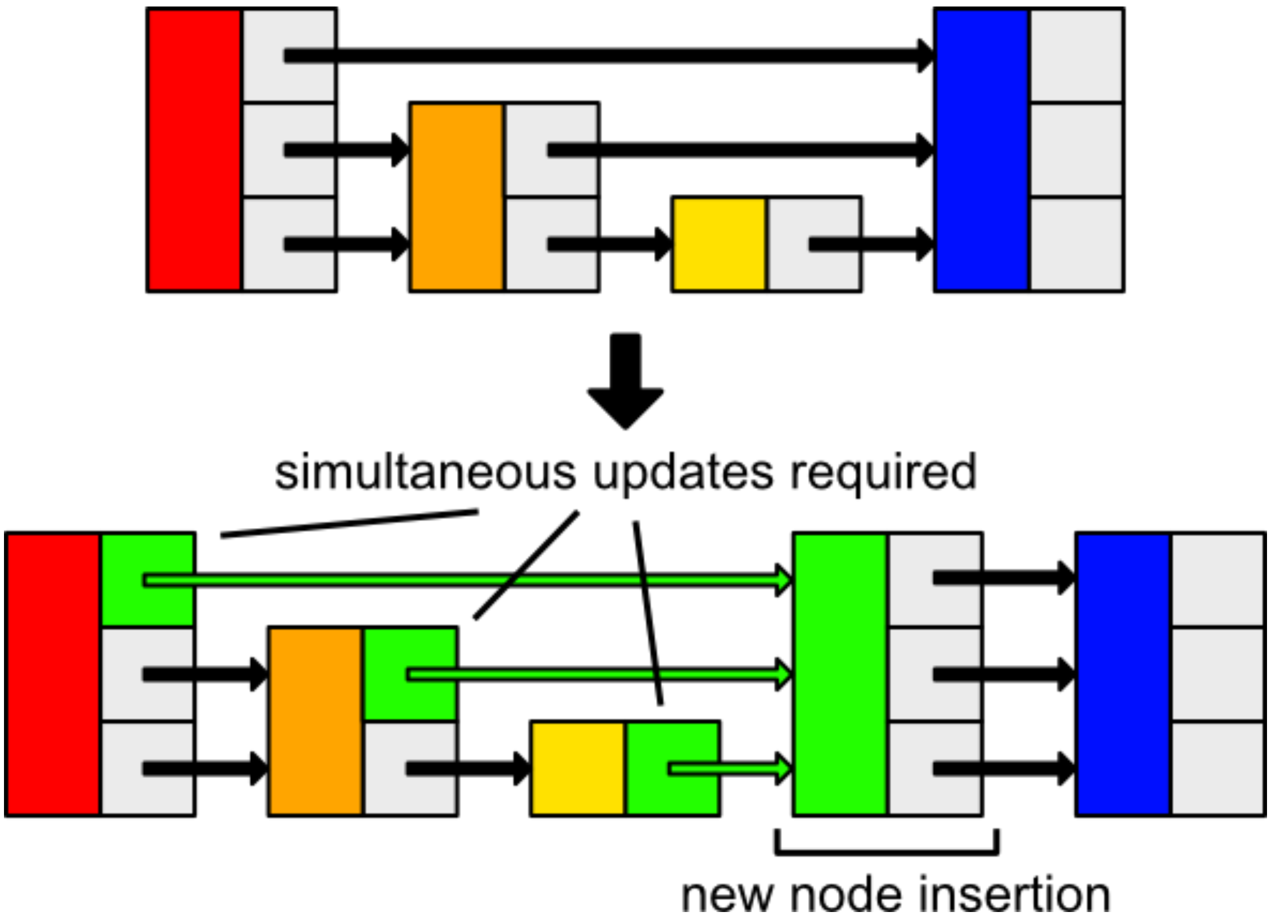
- PMwCAS
- Larson, Levandoski, Wang, 2017
- ~80 lines of code

```
def pmwcas_read(address):
retry:
  v = *address
  if v & RDCSSFlag:
    complete_install(v & AddressMask)
    goto retry

  if v & DirtyFlag:
    persist(address, v)
    v &= ~DirtyFlag

  if v & MwCASFlag:
    persistent_mwcas(v & AddressMask)
    goto retry
  return v

def install_mwcas_descriptor(word):
  ptr = word | RDCSSFlag
retry:
  val = CAS(word.address, word.old_value, ptr)
  if val & RDCSSFlag is not 0:
    # Hit a descriptor, help it finish
    complete_install(val & AddressMask)
    goto retry

  if val == desc.old_value:
    # Successfully installed the conditional CAS descriptor
    complete_install(word)
  return val

def complete_install(wdesc):
  mwcas_ptr = wdesc.mwcas_descriptor | MwCASFlag | DirtyFlag
  u = wdesc.mwcas_descriptor.status == Undecided
  CAS(word.address, val, u ? mwcas_ptr : wdesc.old_value)
```

```
def pcas_read(address):
  word = *address
  if word & DirtyFlag is not 0:
    persist(address, word)
  return word & ~DirtyFlag

def persistent_cas(address, old_value, new_value):
  pcas_read(address)
  # Conduct the CAS with dirty bit set on new value
  return CAS(address, old_value, new_value | DirtyFlag)

def persist(address, value):
  CLWB(address)
  CAS(address, value, value & ~DirtyFlag)

bool PMwCAS(mdesc):
 st = Succeeded
 for w in mdesc.words:
 retry:
  rval = install_mwcas_descriptor(w)
  if rval == w.old_value or rval & AddressMask == mdesc:
    # Descriptor successfully installed
    continue
  elif rval & MwCASFlag is not 0:
    if rval & DirtyFlag is not 0:
      persist(w.address, rval)
    # Clashed another on-going MwCAS, help it finish
    persistent_mwcas(rval.Address)
    goto retry
  else
    st = Failed
    break

 # Persist all target fields if Phase 1 succeeded
 if st == Succeeded:
  for w in mdesc.words:
    persist(w.address, mdesc | MwCASFlag | DirtyFlag)

 # Finalize the MwCAS's status
 CAS(mdesc.status, Undecided, st | StatusDirtyFlag)
 if mdesc.status & DirtyFlag:
  CLWB(&mdesc.status)
  mdesc.status &= ~DirtyFlag

 # Install the final values
 for w in mdesc.words:
  val = mdesc.status == Succeeded ? w.new_value : w.old_value
  expected = mdesc | MwCASFlag | DirtyFlag
  rval = CAS(w.address, expected, val | DirtyFlag)
  if rval == mdesc | MwCASFlag:
    CAS(w.address, expected & ~DirtyFlag, val)
  persist(w.address, val)
 return mdesc.status == Succeeded
```

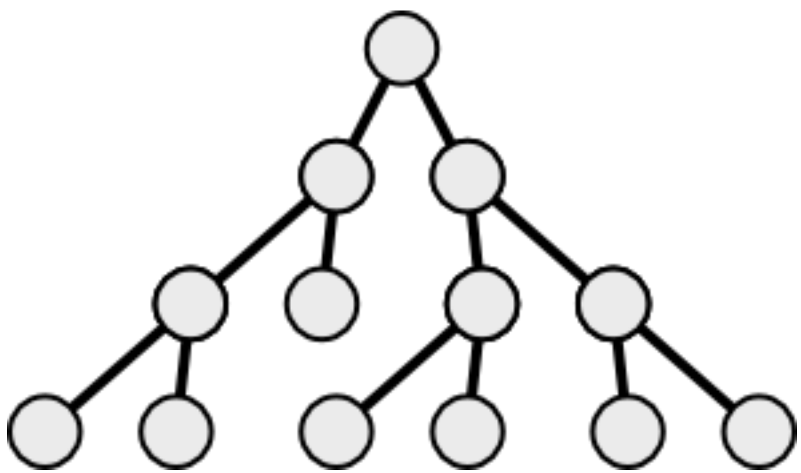
"Mo' [memory]… mo' problems."
— *Notorious B.I.G., 1997*

MOTIVATION: MWCAS AND PERSISTENT MEMORY



MOTIVATION: VERIFYING CORRECTNESS

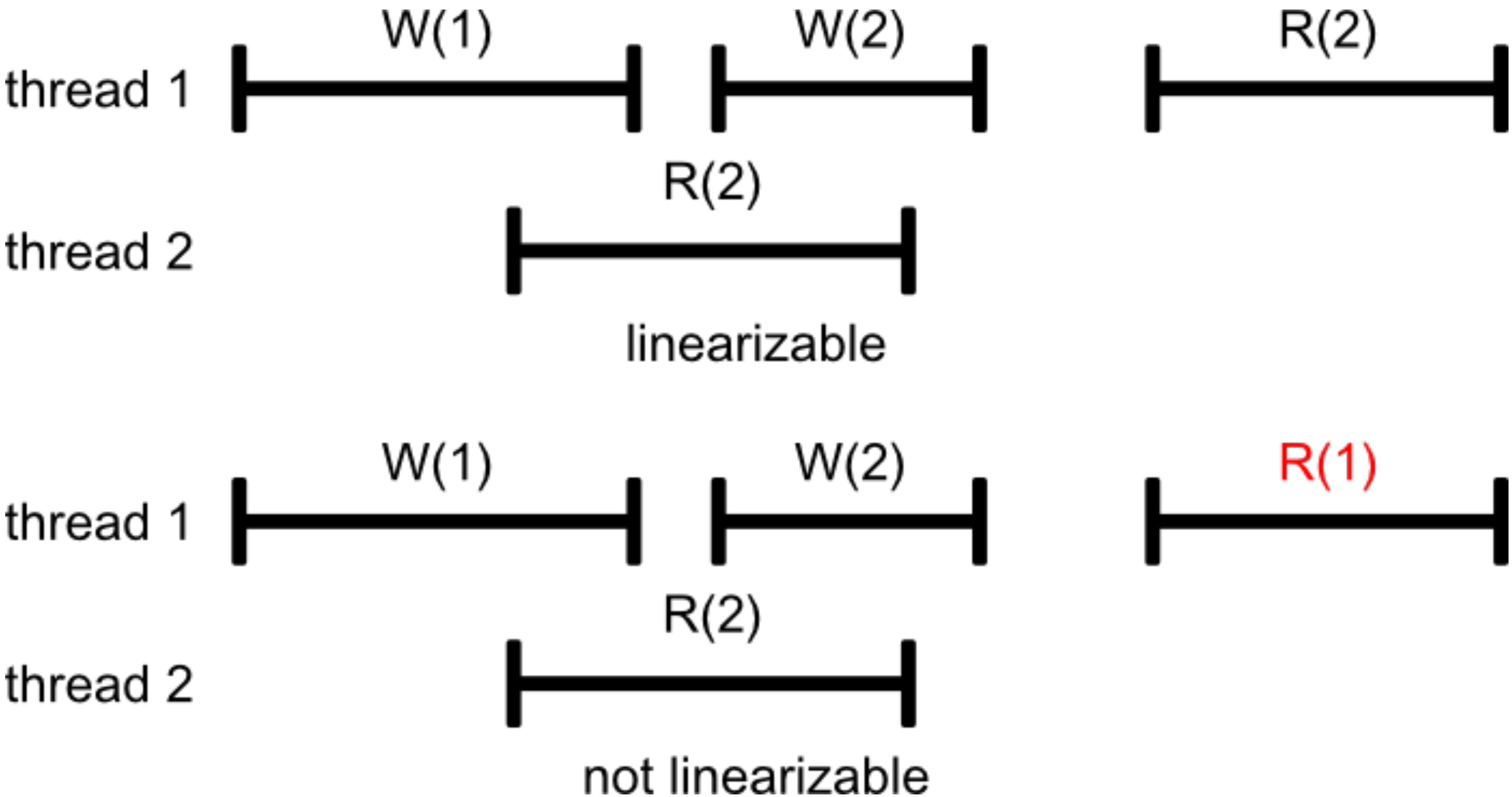
- Model checking: exhaustive search and test of all possible states of the system
  - Intractable due to state-space explosion as number of processes increases



MOTIVATION: VERIFYING CORRECTNESS

- Blackbox testing: because we use software for what it *does*
  - Can identify mistakes in behaviour of a system
  - Cannot prove that all behaviours are correct
  - Absence of evidence vs. evidence of absence

BACKGROUND: LINEARIZABILITY



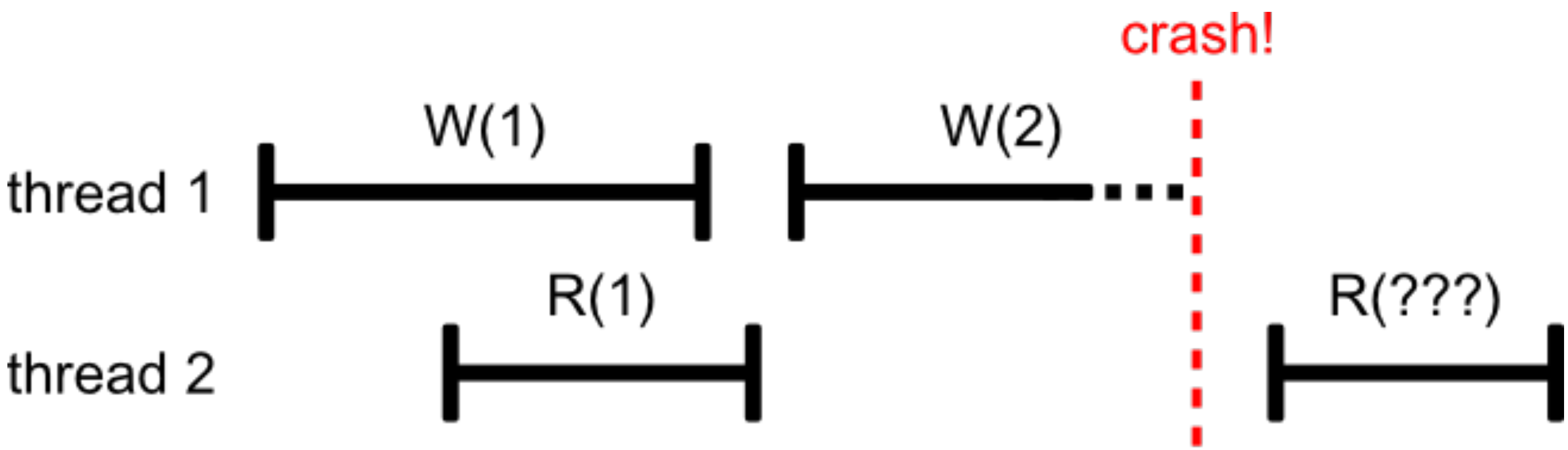
BACKGROUND: MULTI-WORD OPERATIONS

```
mwread(addresses) {
 values[n]
 atomic {
  for i in 1…n {
   values[i] = read(
                addresses[i])
  }
 }
 return values
}
```

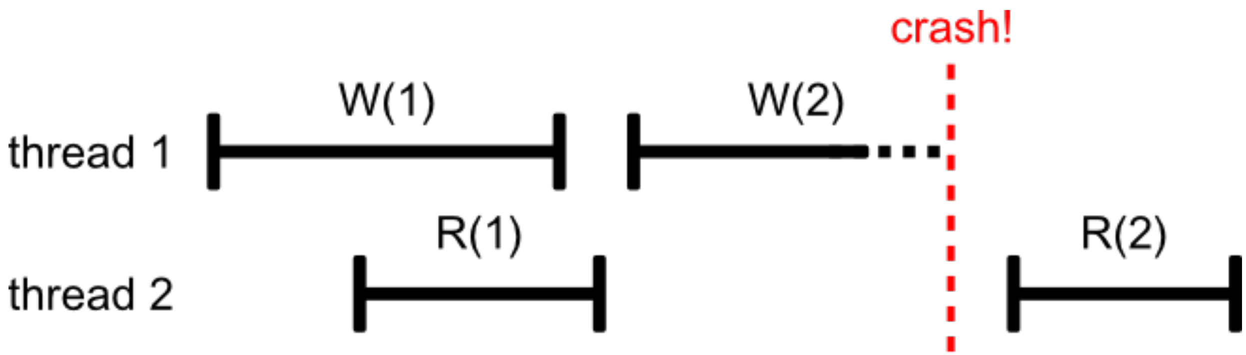
```
mwcas(addresses,
      expected-values,
      desired-values) {
 atomic {
  for i in 1…n {
   if read(addresses[i]) !=
      expected-values[i] {
    return false
   }
  }

  for i in 1…n {
   write(
    addresses[i],
    desired-values[i])
  }
 } return true
}
```

BACKGROUND: OPERATIONS

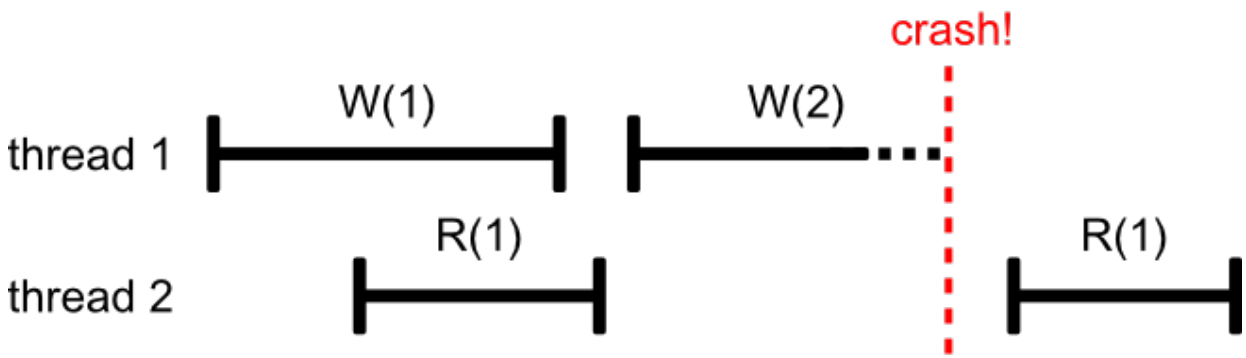
- Crash failure: when operations fail to return a response due to failure of their threads



BACKGROUND: RECOVERY



operation took effect



operation did not take effect (before read)

BACKGROUND: MODEL

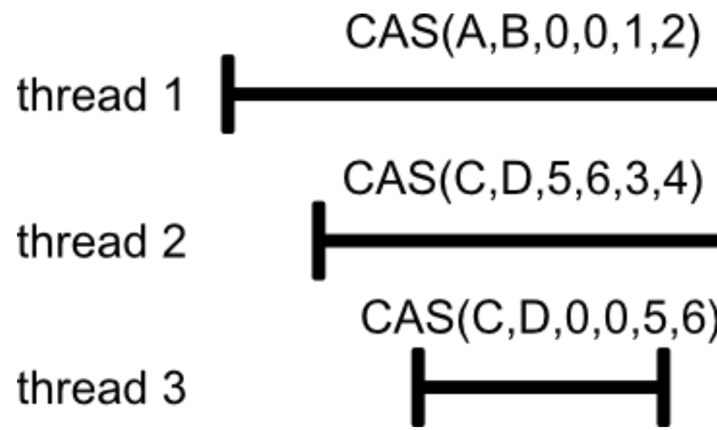
- Behaviour of a system is recorded as an execution history

event
```
1 t1 INV A B 1 2
2 t1 RES A B 0 1 0
```

time | thread | event type | addresses | values | operation type

BACKGROUND: EXAMPLE MULTI-WORD HISTORY

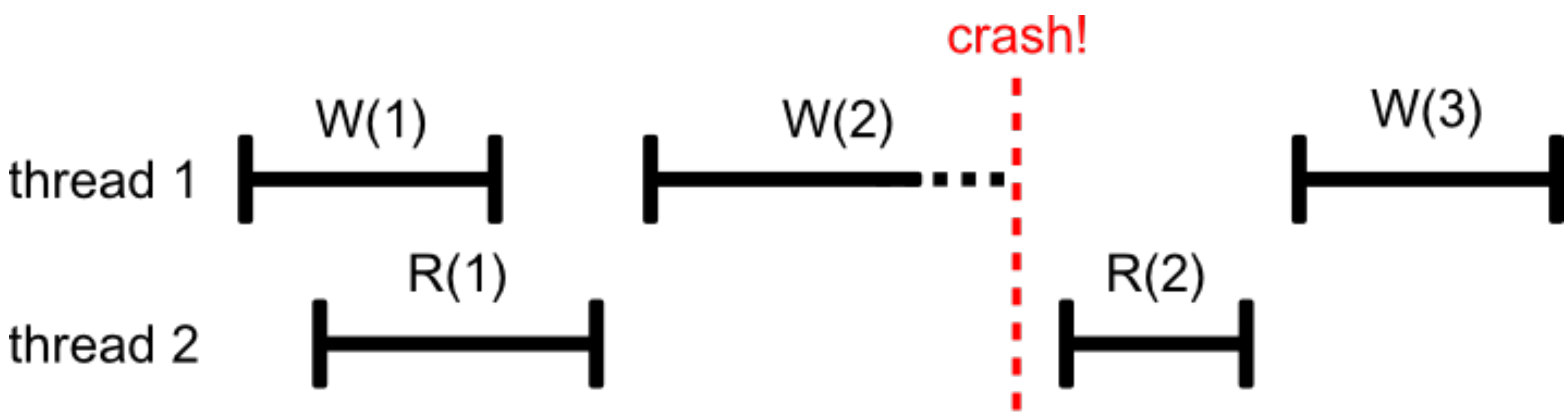
```
1 I t1 A B 1 2
2 I t2 C D 3 4
3 I t3 C D 5 6
4 R t3 C D 0 0 0
5 R t2 C D 5 6 0
6 R t1 A B 0 0 0
```



- 0 is success, $1 \le i \le n$ is failure due to comparison i
- R is for reads

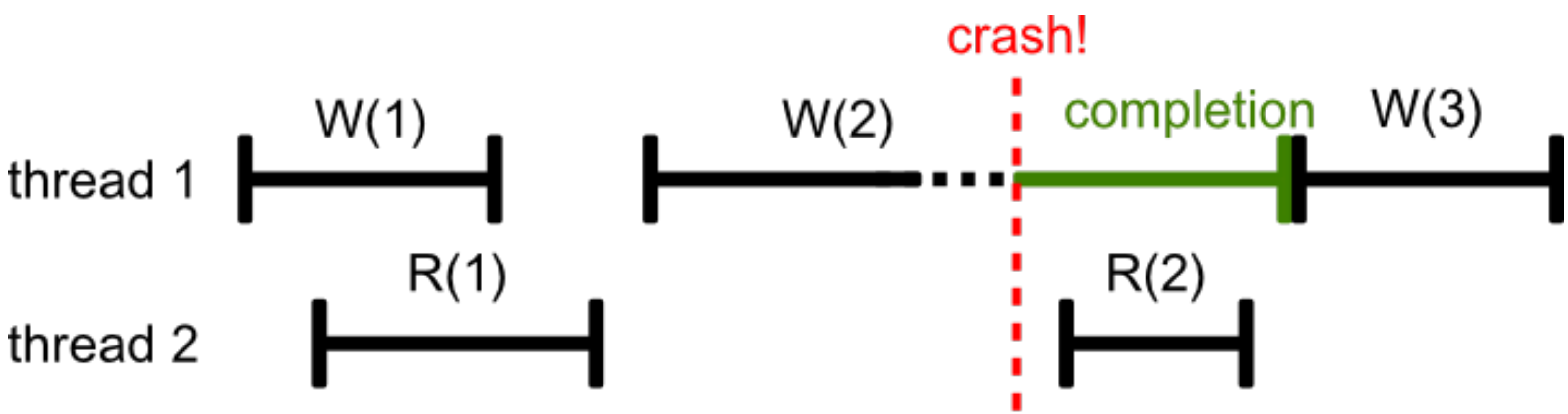
BACKGROUND: LINEARIZING AFTER FAILURE

- No response? No problem!
- Let’s add one if it looks like the operation took effect
- But where?



BACKGROUND: LINEARIZING AFTER FAILURE

- Recoverable linearizability
  - » The operation must have occurred before the next operation on the same object by the same thread



BACKGROUND: ANALYSIS

- Linearizability testing of arbitrary histories involving reads, writes is **NP-complete**.
  - » *Gibbons, Korach. 1997.*
- How can we do blackbox testing in polynomial time?

BACKGROUND: ANALYSIS

- ~~Solution: prove P=NP~~
- Check: is it easier with swaps?
  - » given the previous value in the register during a write (i.e. a swap), the problem is still NP-complete
  - » *Gibbons, Korach. 1997.*
- Solution: Given the read-mapping, the problem is O(nlogn) for a history of n operations
  - » *Gibbons, Korach. 1997.*
  - » We can infer the read-mapping from the log of successful swaps if all values are unique

BACKGROUND: ANALYSIS TECHNIQUES

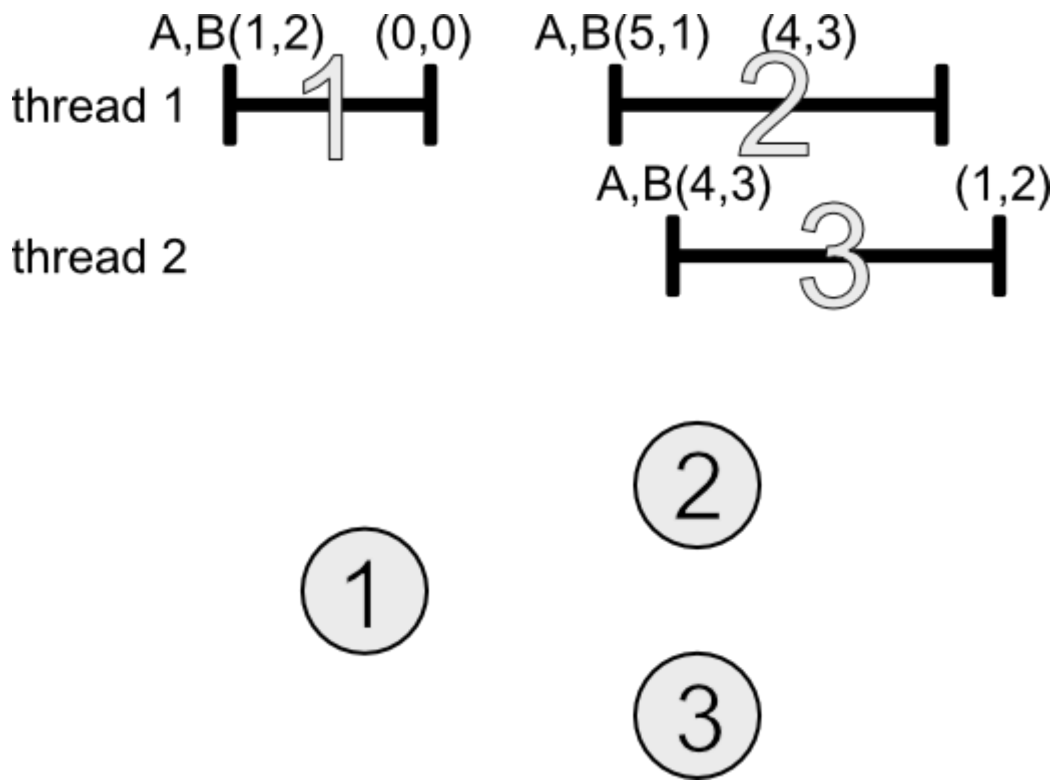
- Zone-based testing
  - » Proposed by Gibbons, Korach
  - » Defines interval of time, or “zone”, over which a value is the latest value of an object
- Graph-based testing
  - » Build precedence graph of operations and check for cycles and consistency

BACKGROUND: ANALYSIS TECHNIQUES

- Hitting families
  - » Order small groups of operations linearizably first
    - *Ozkan, Majumdar, Niksic. 2019.*
- Data-structure-specific methods
  - » P-time algorithms exist for abstract collections
    - *Emmi, Enea. 2018.*
  - » Can reduce some models to simpler models
    - *Bouajjani et al. 2015.*
  - » Local view arguments to easily linearize search operations
    - *Feldman et al. 2018.*

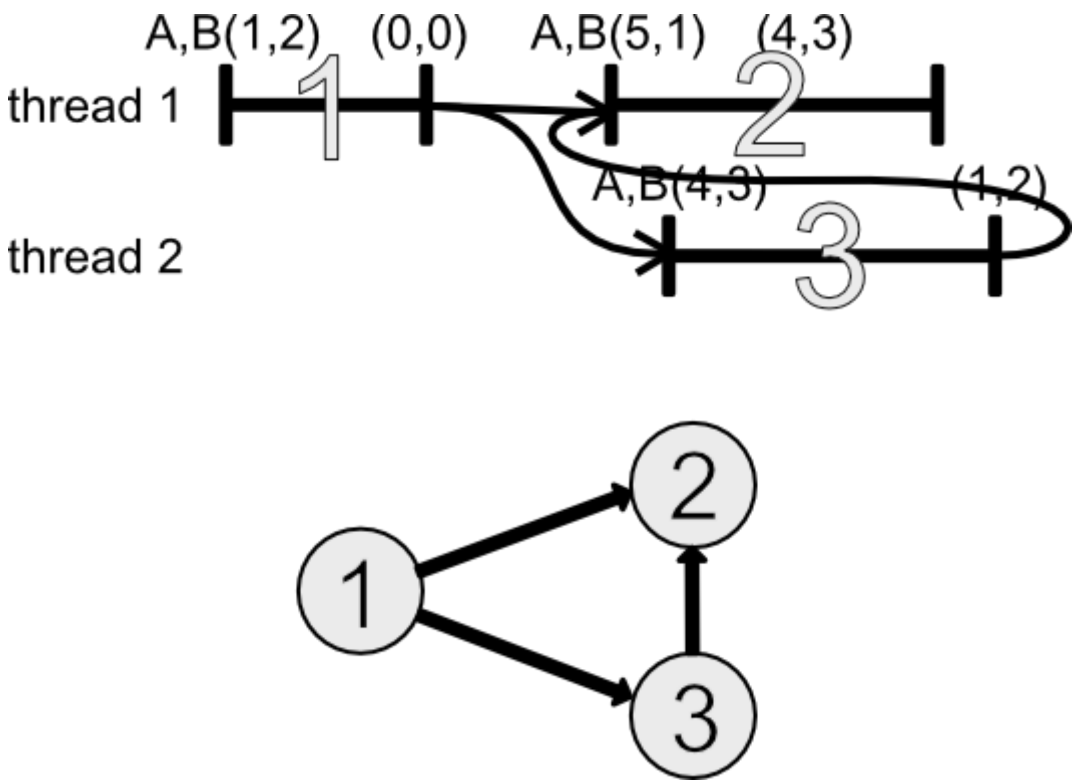
THE ALGORITHM: THE GRAPH

- Vertices represent each operation



THE ALGORITHM: THE GRAPH

- Directed edges establish order in which they must be linearized

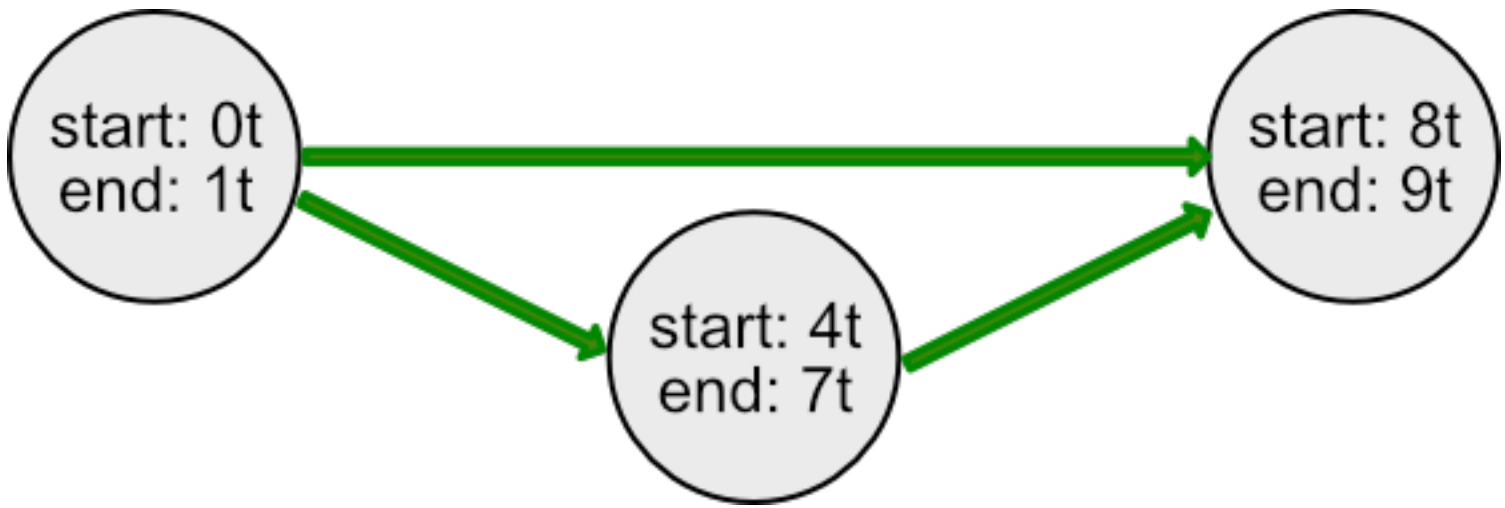


THE ALGORITHM: EDGE BUILDING

- Reads-from edge: read of a value must occur after it is written

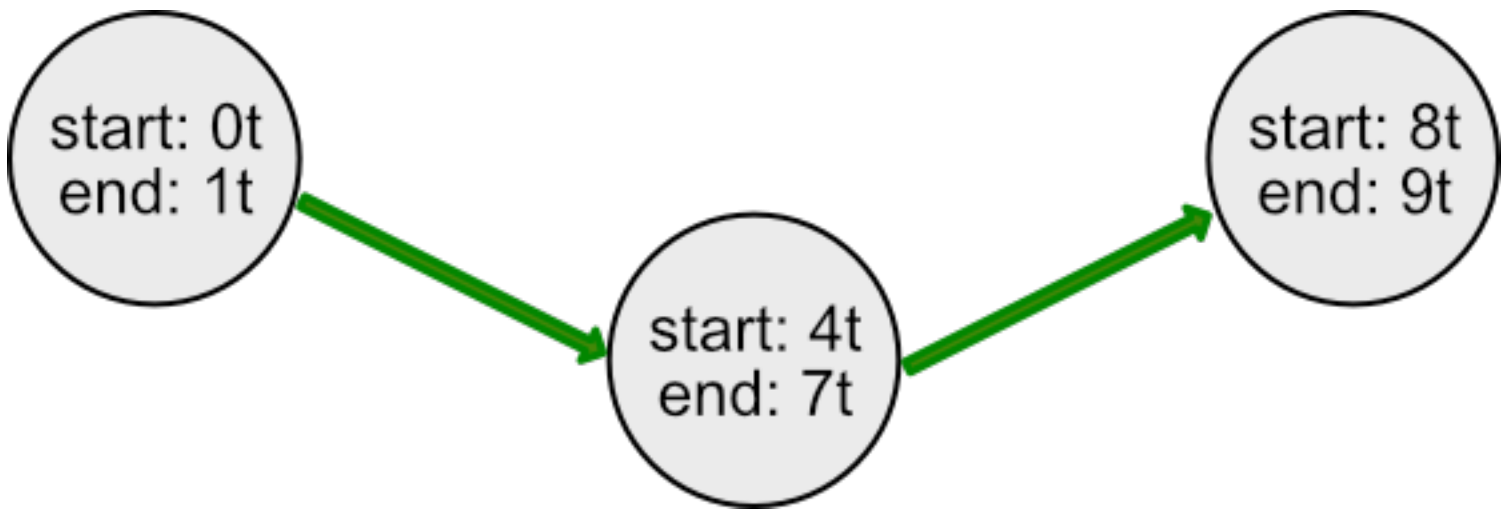
THE ALGORITHM: EDGE BUILDING

- Time-precedence edge: non-concurrent operations are totally ordered

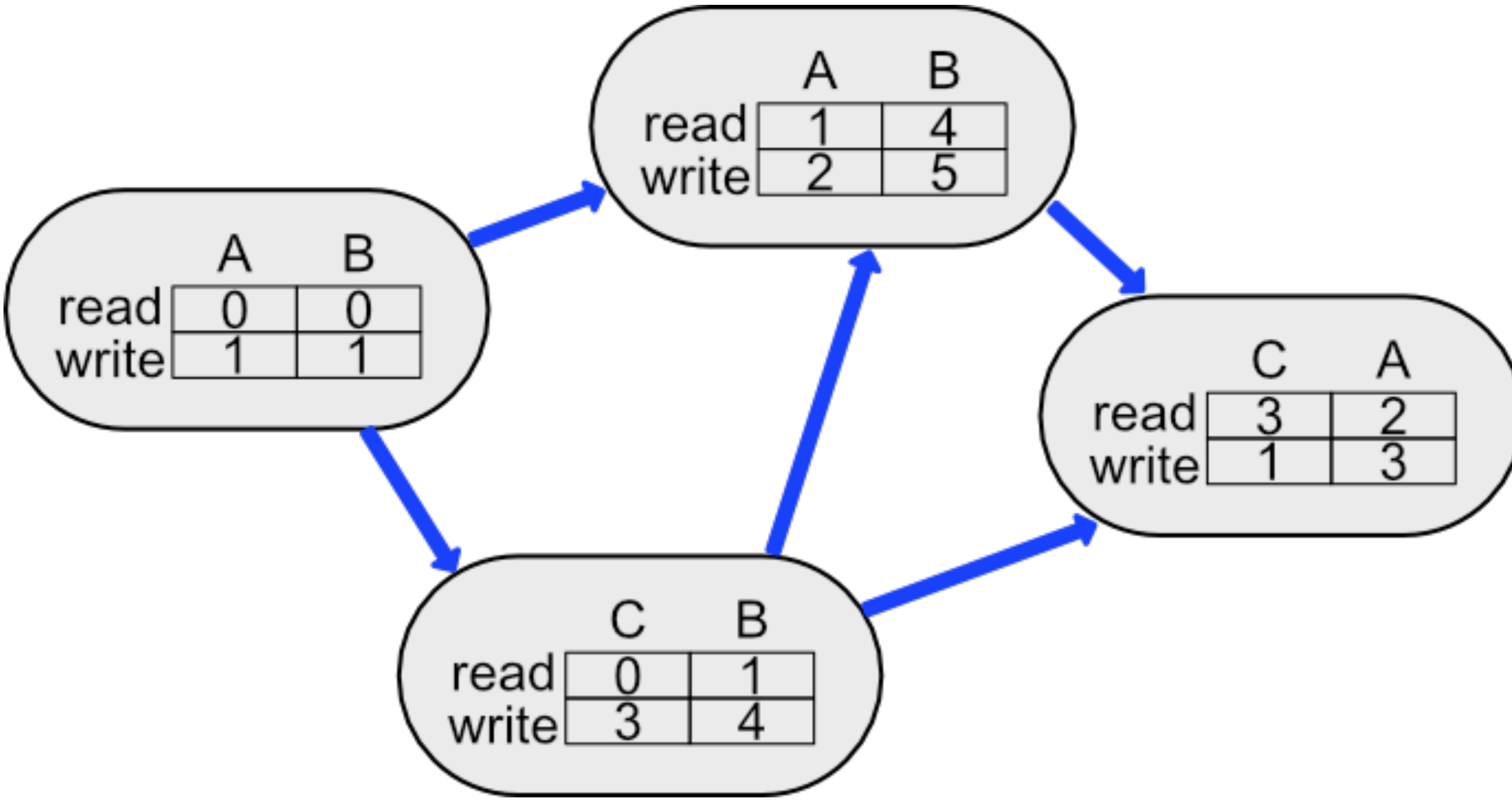


THE ALGORITHM: EDGE BUILDING

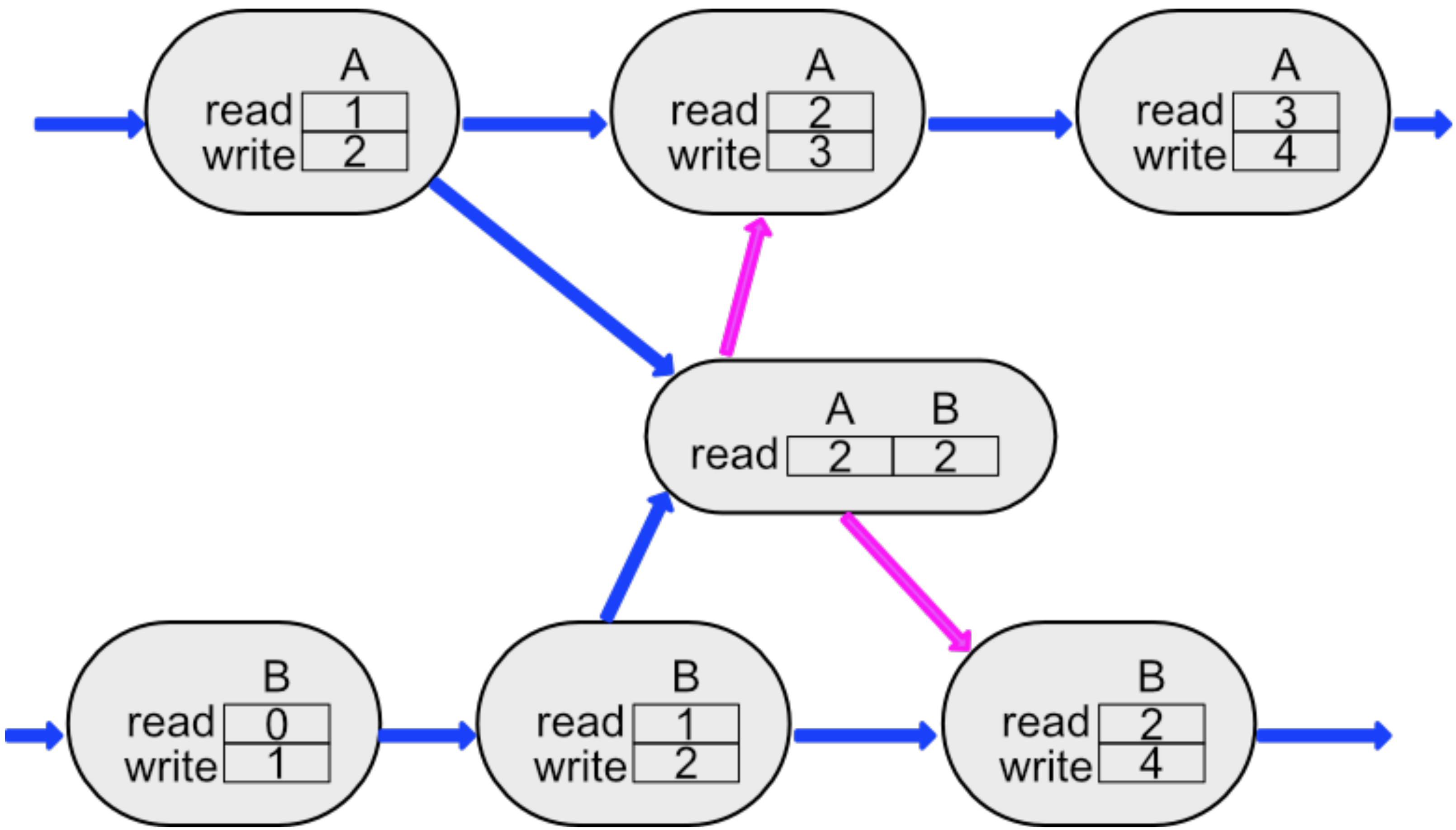
- Greatly reduce the number of time edges by using an algorithm to select necessary edges



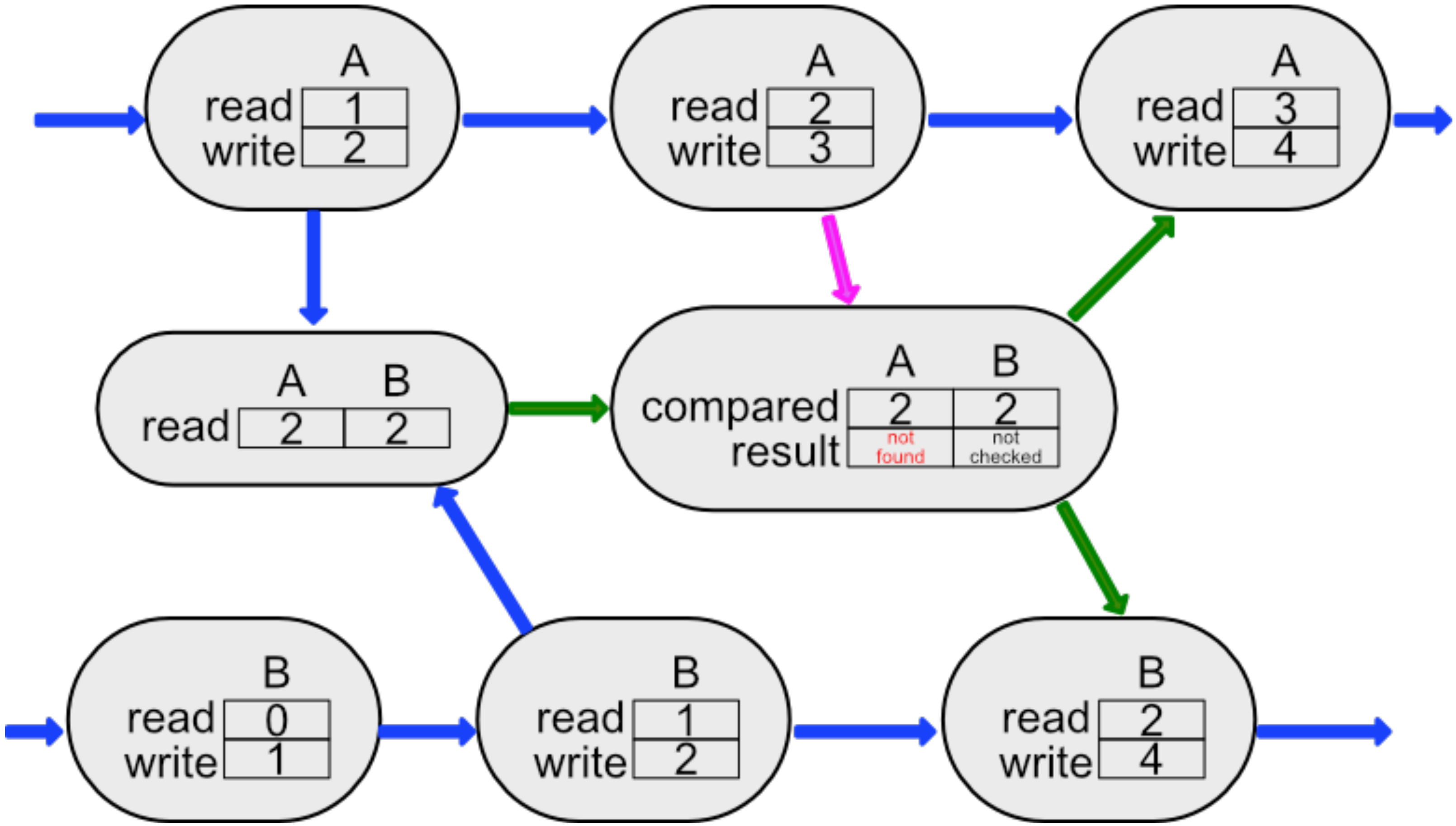
THE ALGORITHM: UNCONDITIONAL MULTI-WORD SWAPS



THE ALGORITHM: MULTI-WORD READS

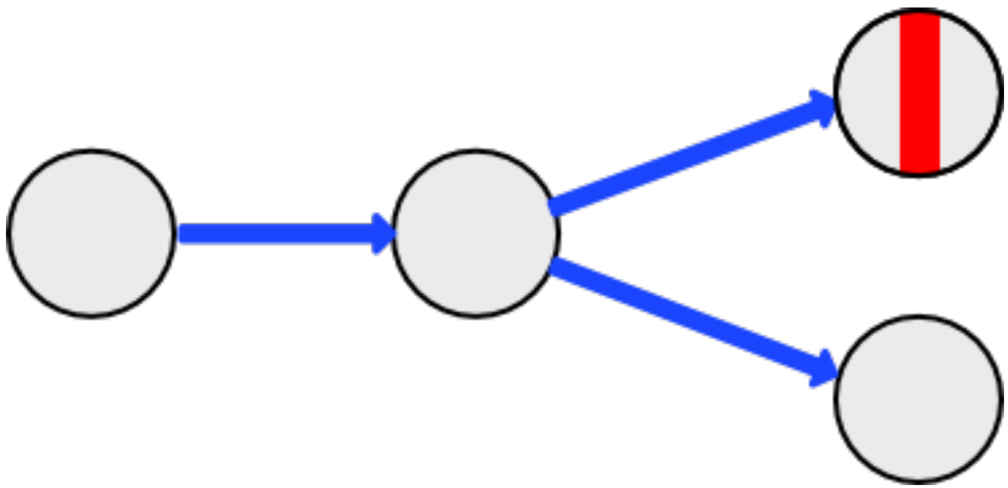


THE ALGORITHM: UNSUCCESSFUL CAS OPERATIONS



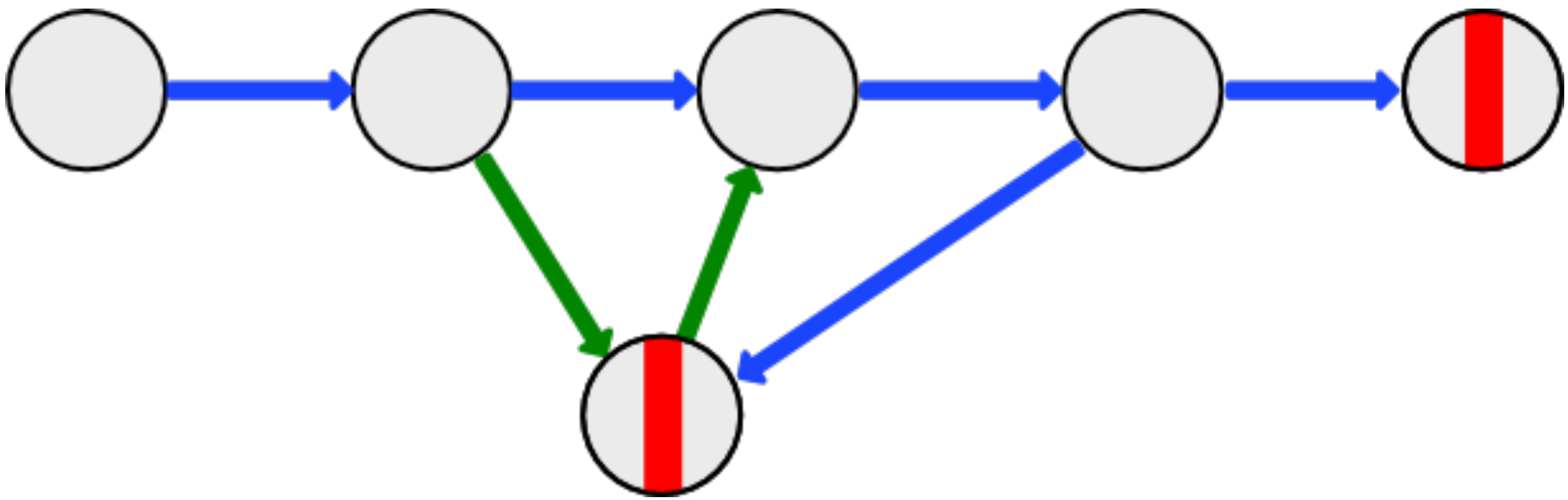
THE ALGORITHM: TYPES OF LINEARIZABILITY ERRORS

- Fork



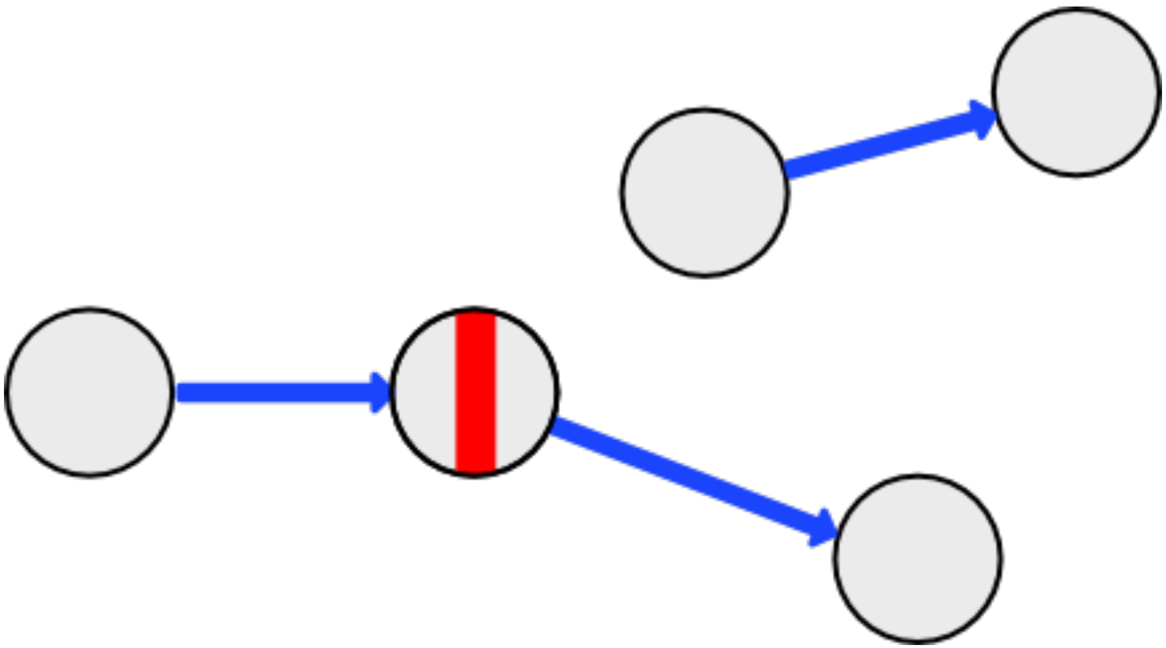
THE ALGORITHM: TYPES OF LINEARIZABILITY ERRORS

- Time travel

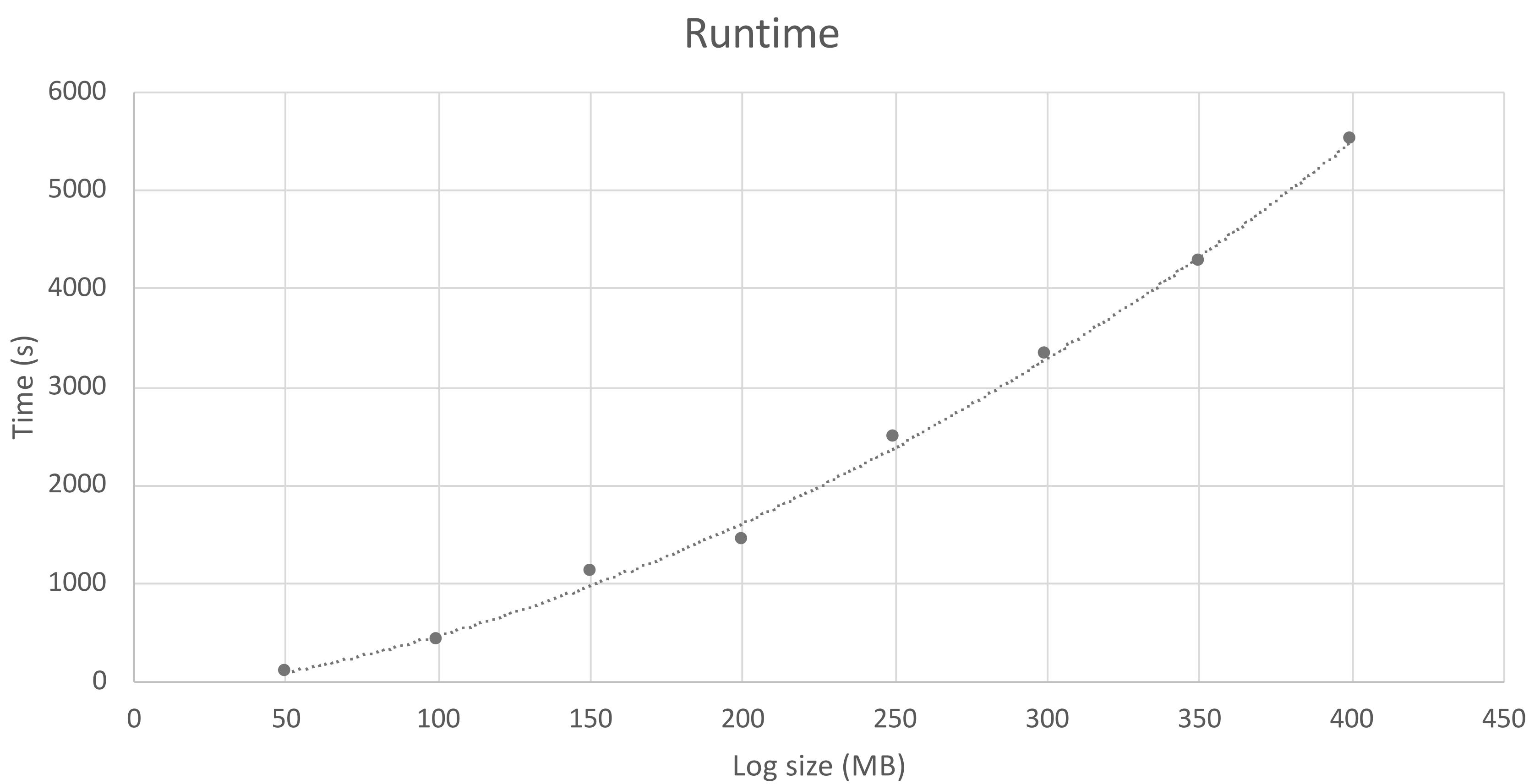


THE ALGORITHM: TYPES OF LINEARIZABILITY ERRORS

- Disconnect



THE ALGORITHM: PERFORMANCE



CONCLUSION

- Blackbox testing is a valuable tool when model checking becomes increasingly difficult
- Graph-based methods allow intuitive extension of linearizability testing to multiple words and new operations
- Testing recoverability of multi-word primitives is important because of their usefulness in building linked data structures

Questions?