Verifying Transactional Programs with Programmer-Defined Conflict Detection

Omer Subasi, Serdar Tasiran (Koç University)
Tim Harris (Microsoft Research)
Adrian Cristal, Osman Unsal (Barcelona SC)
Ruben Titos-Gil (University of Murcia)
Motivation: Linked List from Genome
Write-Read conflict!

Write-After-Read conflict!
• Conventional TM conflict detection
  – Insertions conflict often
• But, both insertions OK even if we ignore WAR conflict
⇒ Relaxed conflict detection
User-Defined Conflict Detection

- Ignore write-after-read conflicts (Titos et al):

  atomic[!WAR]{
  insert method body
}


Linked List: Insert

list_insert(list_t *listPtr, node_t *node) {
    atomic {
        *curr = listPtr->head;
        do {
            prev = curr;
            curr = curr->next;
        } while (curr != NULL &&
                curr->key < node->key);
        node->next = curr;
        prev->next = node;
    }
}
Linked List: Insert

```c
list_insert(list_t *listPtr, node_t *node) {
  atomic {
    *curr = listPtr->head;
    do {
      prev = curr;
      curr = curr->next;
    } while (curr != NULL &&
              curr->key < node->key);

    node->next = curr;
    prev->next = node;
  }
  assert(node is in the list && list is sorted);
}
```

Strict conflict detection:
Can reason about transaction code sequentially.
Linked List: Insert

```c
list_insert(list_t *listPtr, node_t *node) {
    atomic {[!]WAR]{

        *curr = listPtr->head;
        do {
            prev = curr;
            curr = curr->next;
        } while (curr != NULL &&
            curr->key < node->key);

        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Ignore Write-After-Read conflicts:

- Writes by others can occur between read phase and write phase
- Lost ability to reason sequentially
Making !WAR block atomic

```c
list_insert(list_t *listPtr, node_t *node) {
    atomic [!WAR] {
        *curr = listPtr->head;
        do {
            prev = curr;
            curr = curr->next;
        } while (curr != NULL && curr->key < node->key);
        node->next = curr;
        prev->next = node;
        assert(node is in the list && list is sorted);
    }
}
```

Would like this action to be "right mover"

- Can commute to the right of any action by another thread
Right Mover

$\alpha$ commutes to the right of $\beta$ if $\alpha ; \beta$ goes to state $S$ then $\beta ; \alpha$ goes to same state $S$.

If $\alpha$ is right-mover:

$\alpha ; \gamma \rightarrow \alpha;\gamma$
Making !WAR block atomic

```c
list_insert(list_t *listPtr, node_t *node) {
  atomic ![WAR] {
    *curr = listPtr->head;
    do {
      prev = curr;
      ![WAR] curr_T1 = curr_T1->next;
    } while (curr != NULL &&
      curr->key < node->key);

    node->next = curr;
    prev->next = node;
    assert(node is in the list && list is sorted);
  }
}
```
Abstraction intuition

READ 6; WRITE 5

WRITE 5; READ 5

ABSTRACT READ 6; WRITE 5

WRITE 5; ABSTRACT READ 6

ABSTRACT READ: Read anything forward but do not pass the key.

Want to read 6 again!

Need to jump over 5.
Abstraction intuition

1. READ 6; WRITE 5
   - curr = curr->next;
   - with curr = curr->next*;
   - but don’t go past key.

2. WRITE 5; READ 5
   - Need to jump over 5.

3. ABSTRACT READ 6; WRITE 5

4. WRITE 5; ABSTRACT READ 6
   - Want to read 6 again!
list_insert(list_t *listPtr, node_t *node) {
  atomic [!WAR] {
    *curr = listPtr->head;
    do {
      prev = curr;
      curr = curr->next;
    } while (curr != NULL && curr->key < node->key);

    node->next = curr;
    prev->next = node;
    assert(node is in the list && list is sorted);
  }
}
1. Sequentially prove the original code

```c
list_insert(list_t *listPtr, node_t *node) {

    *curr = listPtr->head;
    do {
        prev = curr;
        curr = curr->next;
    } while (curr != NULL &&
        curr->key < node->key);

    node->next = curr;
    prev->next = node;

    assert(node is in the list && list is sorted);
}
```
2. Apply the program transformation

```c
list_insert(list_t *listPtr, node_t *node) {
    *curr = listPtr->head;
    do {
        prev = curr;
        curr = curr->next*;
    } while (curr != NULL &&
              curr->key < node->key);
    node->next = curr;
    prev->next = node;
    assert(node is in the list && list is sorted);
}
```

Do global read abstractions

⇒ Abstract transaction becomes atomic.

19
3. Prove sequentially properties on abstract code

```c
list_insert(list_t *listPtr, node_t *node) {

    *curr = listPtr->head;
    do {
        prev = curr;
        curr = curr->next*;
    } while (curr != NULL &&
             curr->key < node->key);

    node->next = curr;
    prev->next = node;
    assert(node is in the list && list is sorted);

}
}
```

Finally: Soundness theorem says properties hold on original code, with !WAR ignored.
Other proofs

• Labyrinth from STAMP.
• StringBuffer pool.
• Can apply to any program that has the pattern where:
  • a large portion of the shared data is read first
  • local computations is done and then
  • a small portion of shared data is updated