Toward Generating Reducible Replay Logs

Kyu Hyung Lee, Yunhui Zheng,
Nick Sumner, Xiangyu Zhang

PLDI 2011, June 4 - 7

PURDUE
UNIVERSITY

Introduction

- Recording and replay technique
  - Low overhead on logging phase
  - Fast and deterministic failure replay
- Fast replay techniques for long running applications
  - Checkpointing
    - Frequent checkpoints : high recording overhead
    - Infrequent checkpoints : long running replay
  - Log Reduction

Introduction

- Log reduction
  - Reduces a replay execution
  - Remove events from the replay log, but can still reproduce the failure
  - Limitations
    - Very expensive analysis to detect dependences between events
    - Does not guarantee replayability

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
```

Read personal information (name & age) and store the adult information to a linked-list

Log Reduction – conceptual example

*/*Read personal information (name & age) and store the adult information to a linked-list*/*

```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
```

Assert the linked-list has only one element

Log Reduction – conceptual example


```
/*Read personal information (name & age) and
 store the adult information to a linked-list*/
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
     /* Assert the list has one element */
```

Execute this program on top of the logging tool to record non-deterministic system calls

Log Reduction – conceptual example


```
/*Read personal information (name & age) and
 store the adult information to a linked-list*/
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
    /* Assert the list has one element */
```

Execute this program on top of the logging tool to record non-deterministic system calls

Log Reduction – conceptual example


```
1  int count=0;
2  Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5        n = new Node(name);
6        n->next = list;
7        list = n;
8        print(name);
       }
9    count++;
     }
10 print("count="+count);
11  Assert(list->next==null);
    /* Assert the list has one element */
```

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5        n = new Node(name);
6        n->next = list;
7        list = n;
8        print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8        print(name);
      }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
    /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)

8 Write(Brian)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)

8 Write(Brian)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5        n = new Node(name);
6        n->next = list;
7        list = n;
8        print(name);
       }
9    count++;
     }
10 print("count="+count);
11  Assert(list->next==null);
    /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)

Log Reduction – conceptual example

```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
     /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
3 Read(Amy, 6)
3 Read(Alex,15)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5        n = new Node(name);
6        n->next = list;
7        list = n;
8        print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
   /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
3 Read(Amy, 6)
3 Read(Alex,15)
3 Read(EOF)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11  Assert(list->next==null);
     /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
3 Read(Amy, 6)
3 Read(Alex,15)
3 Read(EOF)
10 Write(count=6)

Log Reduction – conceptual example


```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4      if(age > 18) {
5         n = new Node(name);
6         n->next = list;
7         list = n;
8         print(name);
       }
9    count++;
     }
10 print("count="+count);
11 Assert(list->next==null);
     /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
3 Read(Amy, 6)
3 Read(Alex,15)
3 Read(EOF)
10 Write(count=6)
11 ***Assertion Fails***

Log Reduction – conceptual example

```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
    /* Assert the list has one element */
```

LOG

```
3  Read(Brian, 25)
8  Write(Brian)
3  Read(John, 15)
3  Read(Tom, 21)
8  Write(Tom)
3  Read(Betty, 30)
8  Write(Betty)
3  Read(Amy, 6)
3  Read(Alex,15)
3  Read(EOF)
10 Write(count=6)
11 Assertion Fails
```

Replay Criterion

Log Reduction – conceptual example

```
1   int count=0;
2   Node *list = null;
3   while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9    count++;
    }
10 print("count="+count);
11 Assert(list->next==null);
    /* Assert the list has one element */
```

LOG

3 Read(Brian, 25)
8 Write(Brian)
~~3 Read(John, 15)~~
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
~~3 Read(Amy, 6)~~
~~3 Read(Alex, 15)~~
3 Read(EOF)
10 Write(count=6)
11 ***Assertion Fails***

Replay Criterion

Expensive analysis to find reduction at the first replay attempt

Goal

- Compiler based technique that generates a reducible replay log
  - Instrument programs to collect minimal additional information
- Reduction achieved though analyzing just the log
- Guarantee replayability of the reduced log

Challenges

- Reduction may induce a different control flow path

Challenges

- Reduction may induce a different control flow path

```
1   while(read(&x))
2     if(x) write(…);
```

Read(0)
Read(10)
Write(…)

Challenges

- Reduction may induce a different control flow path

```
1   while(read(&x))
2     if(x) write(…);
```

Read(0)
~~Read(10)~~
Write(…) // Replay FAIL!

Challenges

- Reduction may induce a different control flow path

```
1   while(read(&x))
2     if(x) write(…);
```

```
Read(0)
Read(10)
Write(…) // Replay FAIL!
```

Reduced log can not properly aligned with the execution

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency

```
1  While(read(&size))
2    socket_read(string, size);
```

```
Read(5)
Socket_read("abcde")
Read(3)
Socket_read("fgh")
```

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency

```
1   While(read(&size))
2     socket_read(string, size);
```

Read(5)
~~Socket_read("abcde")~~
~~Read(3)~~
Socket_read("fgh")

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency

```
1  While(read(&size))
2    socket_read(string, size);
```

```
Read(5)
Socket_read(“abcde”)
Read(3)
Socket_read(“fgh”) // Replay FAIL!
```

Number of bytes are changed that are supposed to be read

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency
- Structural constraints

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency
- Structural constraints

```
1  read(&x);          1  R(10)
2  print("x="+x);     2  W(x=10)
3  read(&y);          3  R(20)
4  print("y="+y);     4  W(y=20)
5  read(&z);          5  R(30)
6  print("z="+z);     6  W(z=30)
```

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency
- Structural constraints

```
1  read(&x);              1  R(10)
2  print("x="+x);         2  W(x=10)
3  read(&y);              3  R(20)
4  print("y="+y);         4  W(y=20)
5  read(&z);              5  R(30)
6  print("z="+z);         6  W(z=30)
```

Challenges

- Reduction may induce a different control flow path
- Reduction may change variable values, leading to inconsistency
- Structural constraints

```
1  read(&x);
2  print("x="+x);
3  read(&y);
4  print("y="+y);
5  read(&z);
6  print("z="+z);
```

```
1  R(10)
2  W(x=10)
3  R(20)
4  W(y=20)
5  R(30)
6  W(z=30)
```

Cannot reduce any event due to the **straight line structure**

Our Solutions

Divide execution into Units
"User annotation"

Instrumentation and optimization
"Static analysis"

Execute instrumented program

Reduction
"analyze just the log"

Replay reduced log

Execution UNIT

- User annotates "[UNIT]" on the event processing loop

```
1    int count=0;
2    Node *list = null;
3    while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

Execution UNIT

- User annotates "[UNIT]" on the event processing loop

```
1    int count=0;
2    Node *list = null;
3    while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

Execution UNIT

- User annotates "[UNIT]" on the event processing loop

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read
     (&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

Execution UNIT

- User annotates "[UNIT]" on the event processing loop
- Unit is an iteration of an event processing loop
- Reduction is only carried out at the unit level

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read
     (&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11   Assert(list->next==null);
```

Execution UNIT

- Why Unit?
  - Fewer dependences
  - Naturally solve the problem of structural constraint
  - Most long running programs are dominated by the event processing loop

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read
     (&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11   Assert(list->next==null);
```

Instrumentation


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

Instrumentation


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

- Find all variables which possibly have inter UNIT dependence
  - Static analysis by LLVM

Instrumentation


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

- Find all variables which possibly have inter UNIT dependence
  - Static analysis by LLVM

Instrumentation


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

- Find all variables which possibly have inter UNIT dependence
  - Static analysis by LLVM

Instrumentation


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9      count++;
       }
10   print("count="+count);
11   Assert(list->next==null);
```

- Instrument the program
  - Log accesses of inter-unit shared variable

Basic Reduction


```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name,
     &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9      count++;
       }
10   print("count="+count);
11   Assert(list->next==null);
```

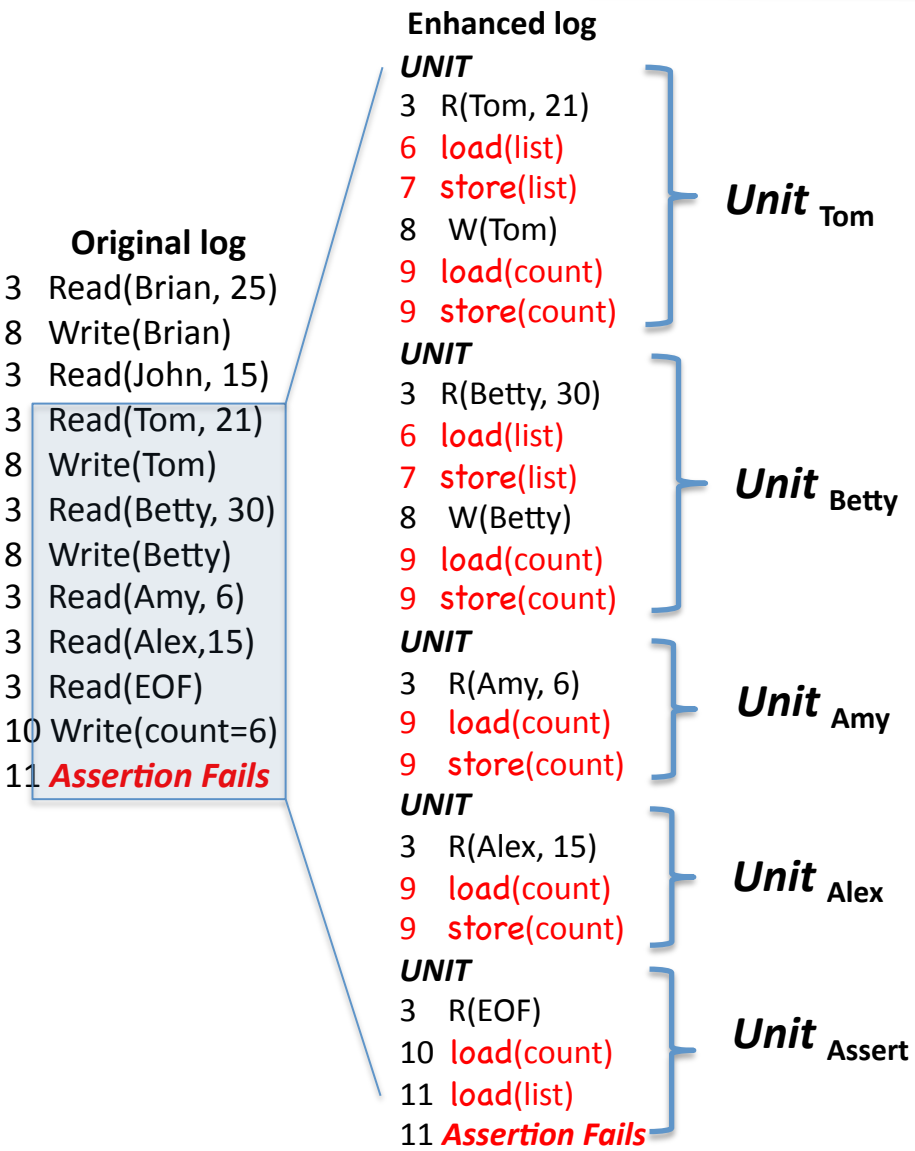
Original log

```
3  Read(Brian, 25)
8  Write(Brian)
3  Read(John, 15)
3  Read(Tom, 21)
8  Write(Tom)
3  Read(Betty, 30)
8  Write(Betty)
3  Read(Amy, 6)
3  Read(Alex,15)
3  Read(EOF)
10 Write(count=6)
11 Assertion Fails
```

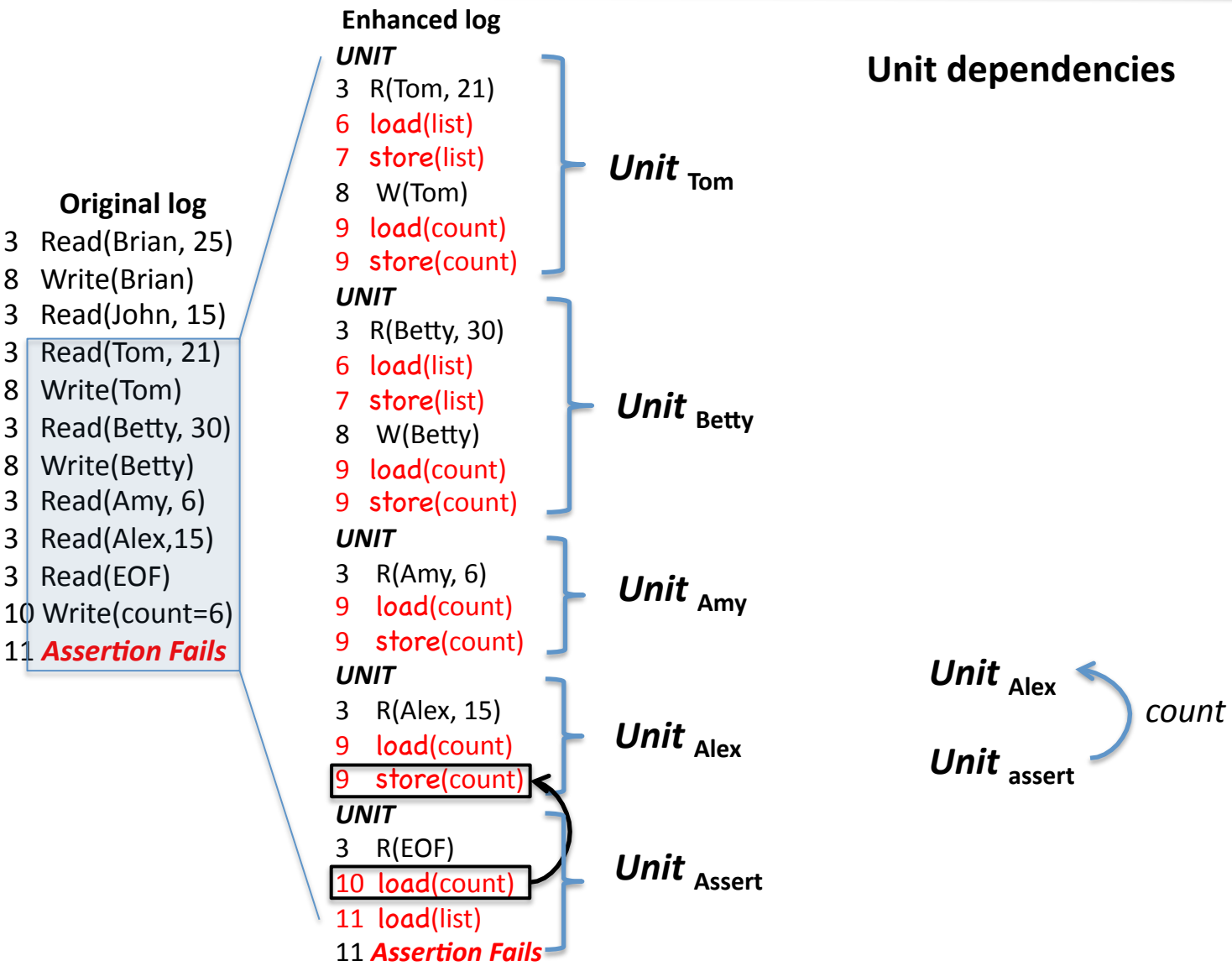
Unit **Tom**

```
3  R(Tom, 21)
6  load(list)
7  store(list)
8   W(Tom)
9  load(count)
9  store(count)
```

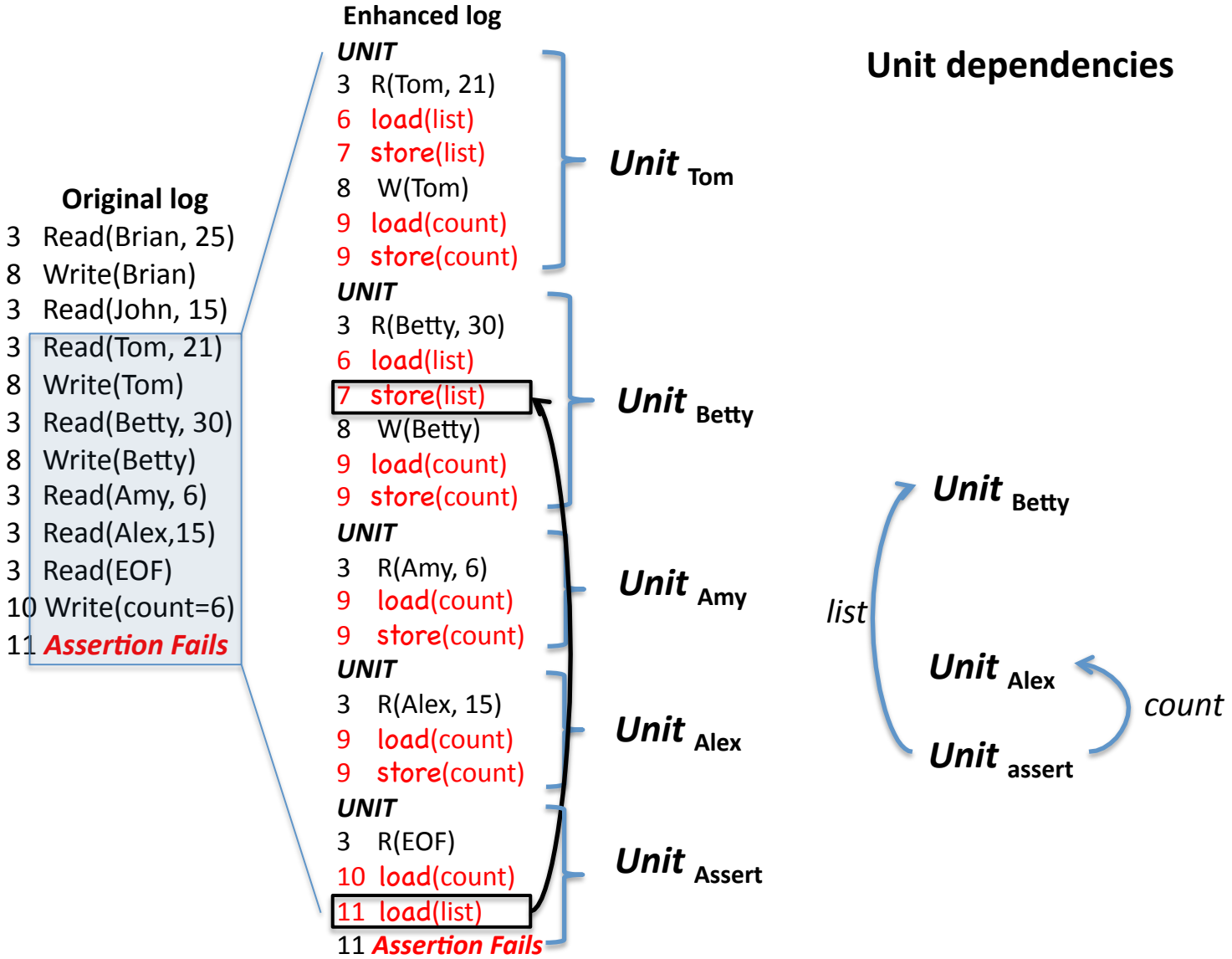
Basic Reduction



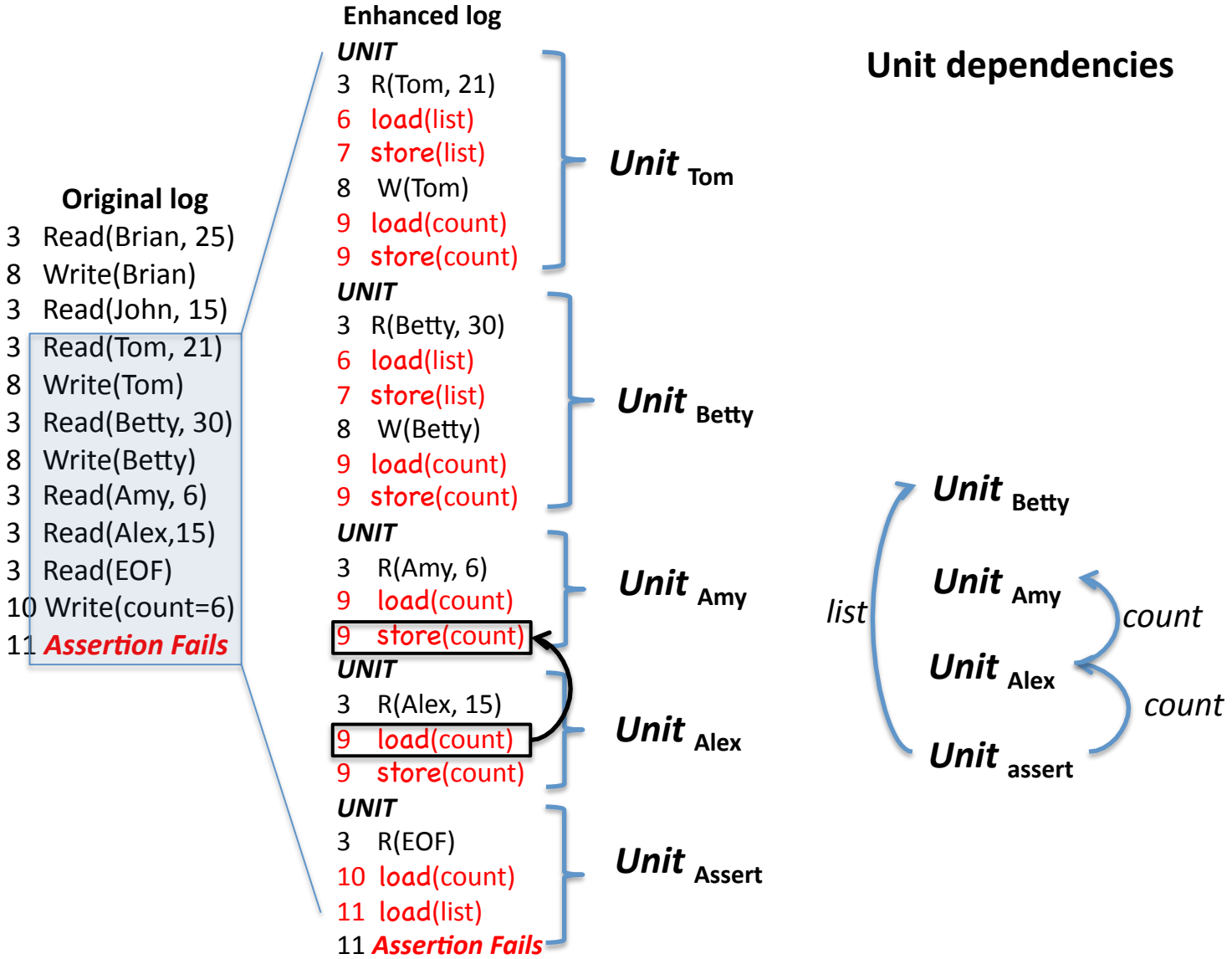
Basic Reduction



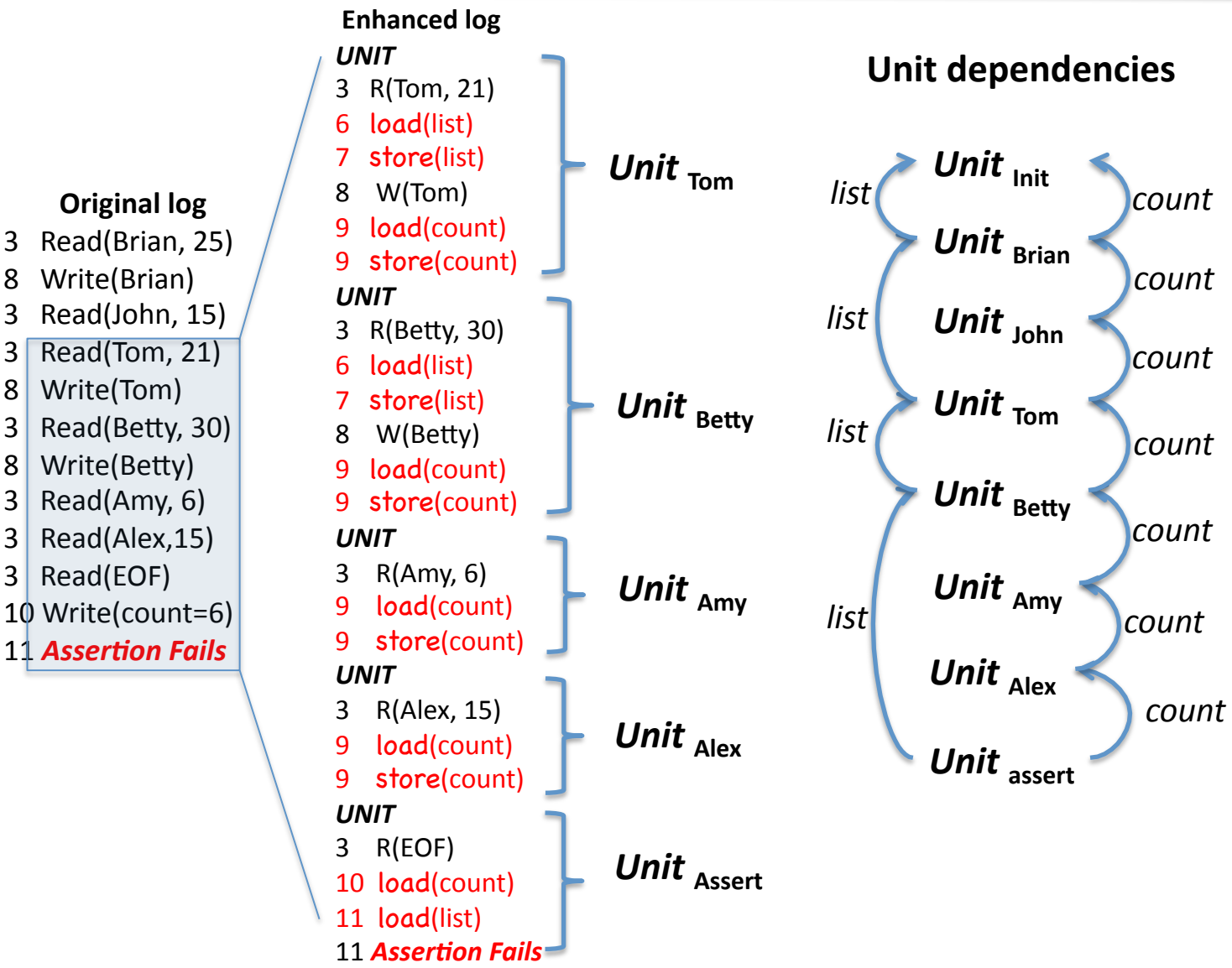
Basic Reduction



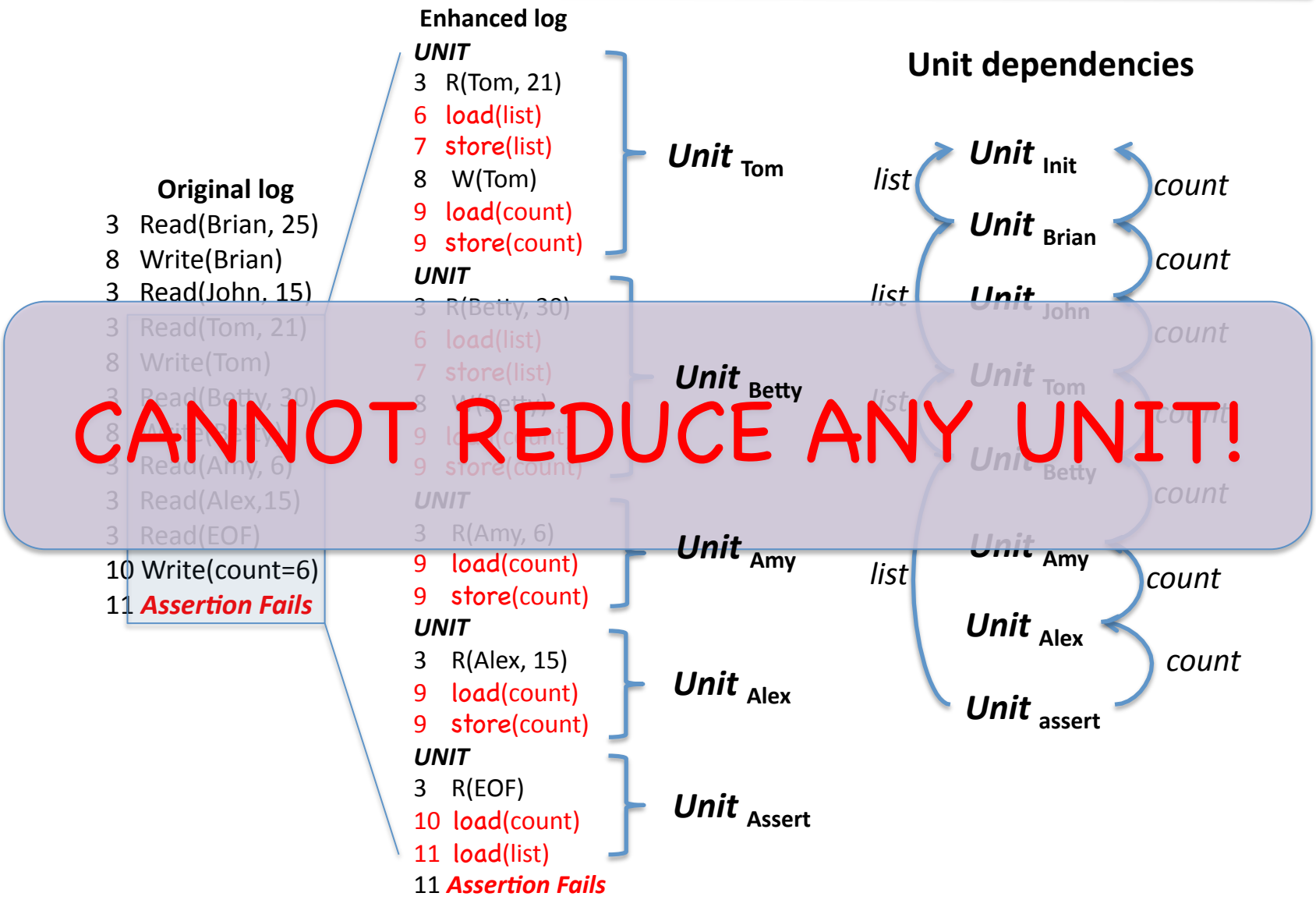
Basic Reduction



Basic Reduction



Basic Reduction



Weave two types of replay

- Replay by value
  - For primitive types
  - Record accesses and value
- Replay by dependence
  - For pointer types
  - Record accesses

Weave two types of replay

Original log

3 Read(Brian, 25)
8 Write(Brian)
3 Read(John, 15)
3 Read(Tom, 21)
8 Write(Tom)
3 Read(Betty, 30)
8 Write(Betty)
3 Read(Amy, 6)
3 Read(Alex,15)
3 Read(EOF)
10 Write(count=6)
11 ***Assertion Fails***

Enhanced log

UNIT
3 R(Tom, 21)
6 load(list)
7 store(list)
8 W(Tom)
9 load(count,2)
9 store(count)

Unit **Tom**

UNIT
3 R(Betty, 30)
6 load(list)
7 store(list)
8 W(Betty)
9 load(count,3)
9 store(count)

Unit **Betty**

UNIT
3 R(Amy, 6)
9 load(count,4)
9 store(count)

Unit **Amy**

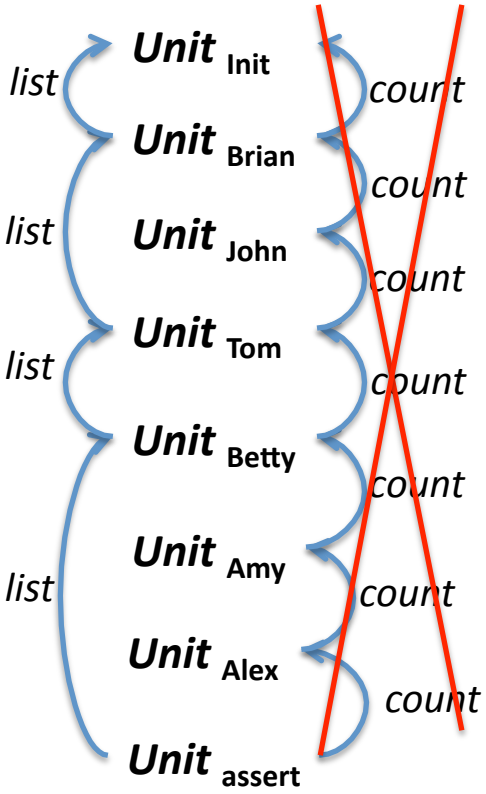
UNIT
3 R(Alex, 15)
9 load(count,5)
9 store(count)

Unit **Alex**

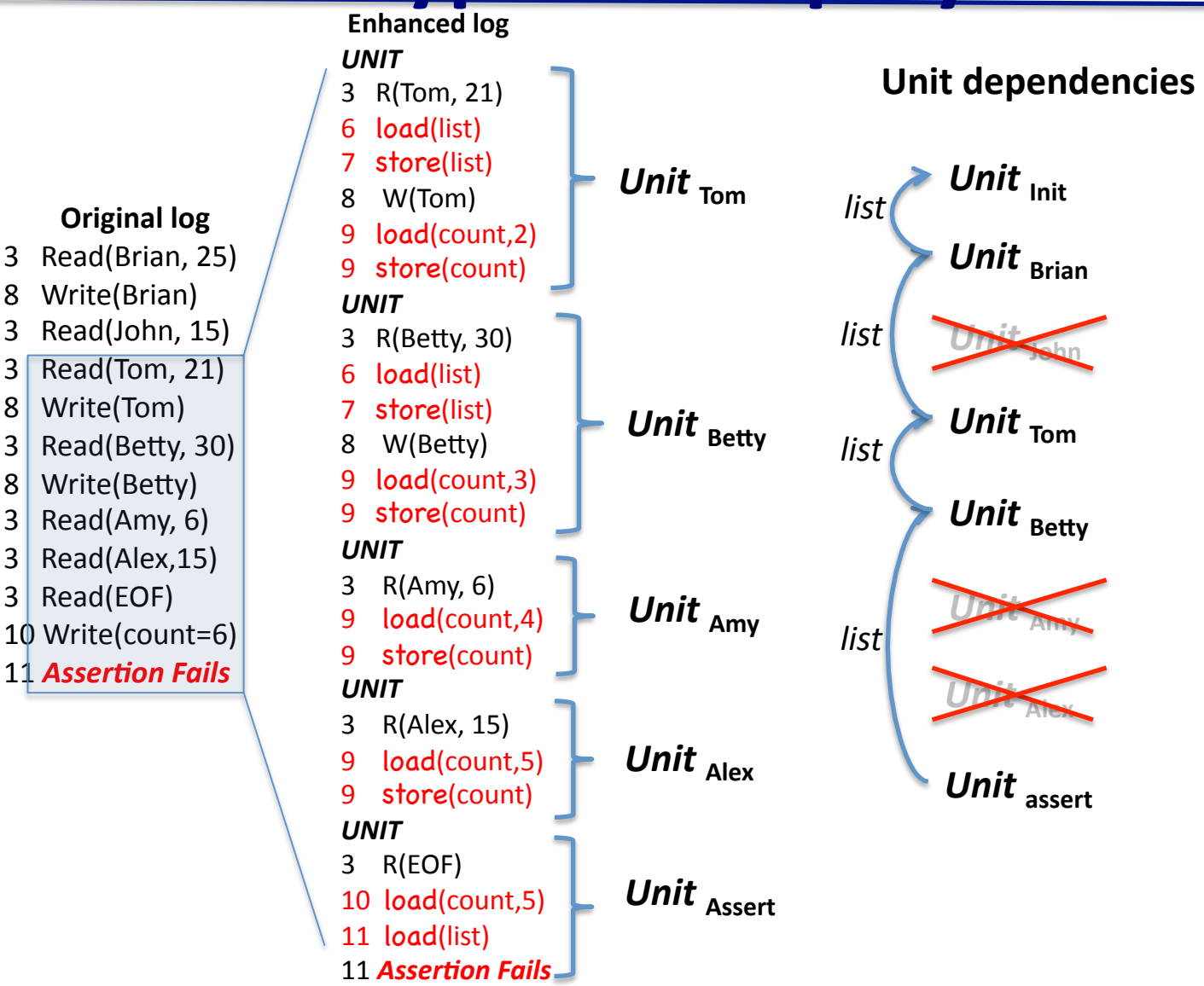
UNIT
3 R(EOF)
10 load(count,5)
11 load(list)
11 ***Assertion Fails***

Unit **Assert**

Unit dependencies



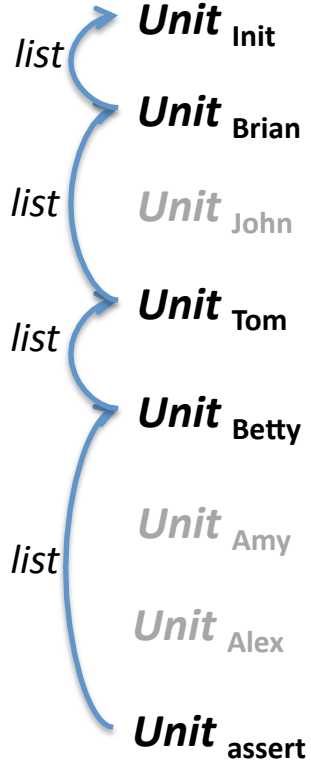
Weave two types of replay



Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11   Assert(list->next==null);
```

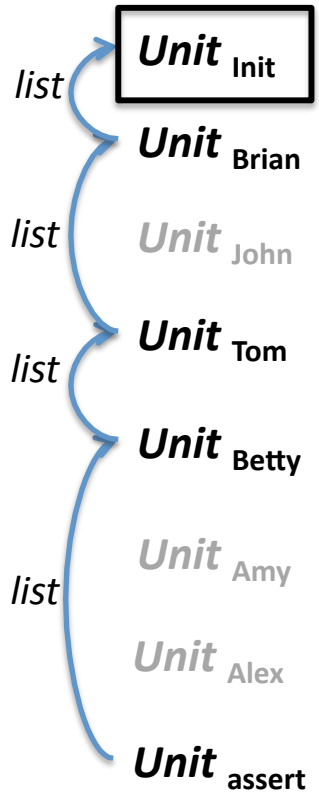
Unit dependencies



Basic replay

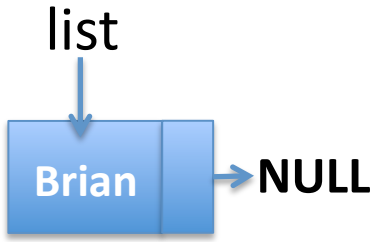
```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

Unit dependencies

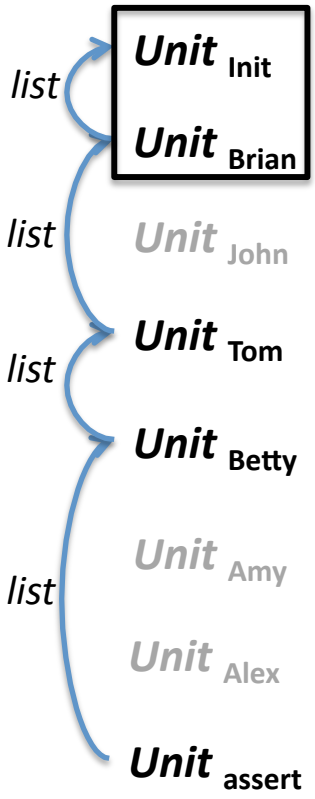


Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

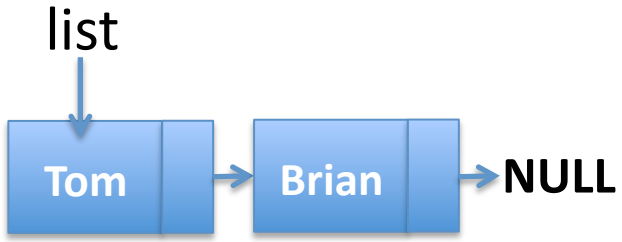


Unit dependencies

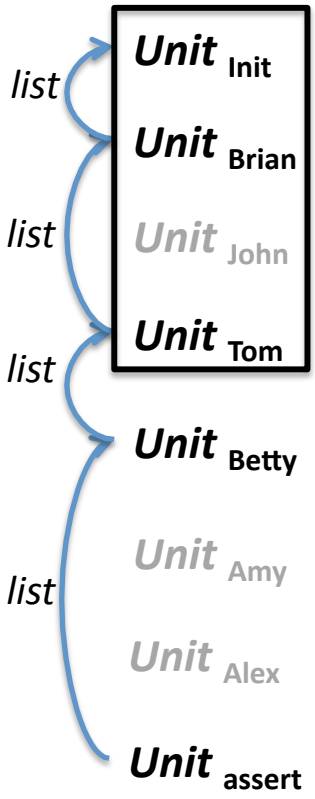


Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

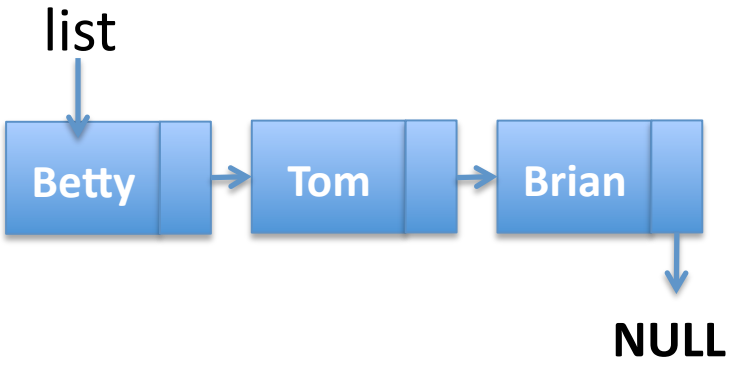


Unit dependencies

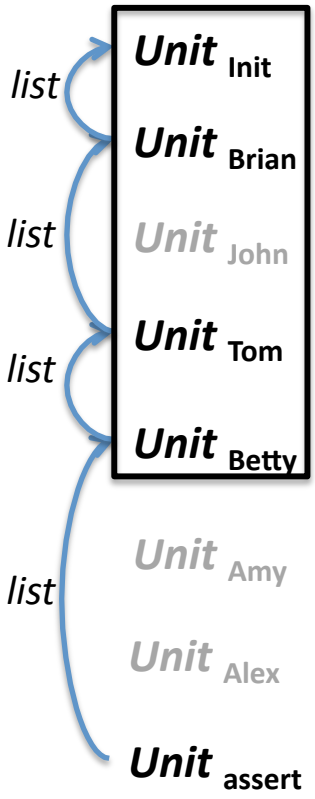


Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11   Assert(list->next==null);
```

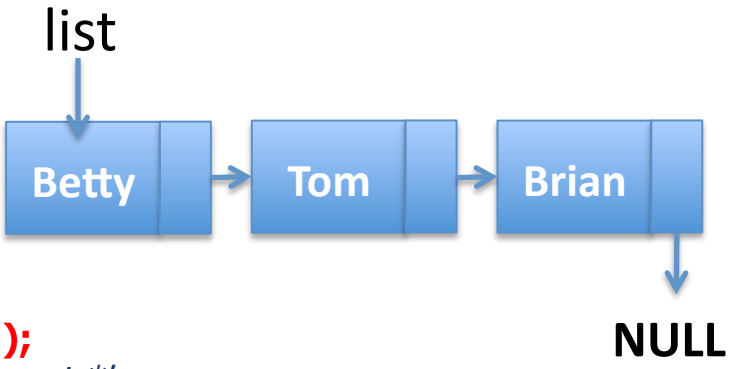


Unit dependencies

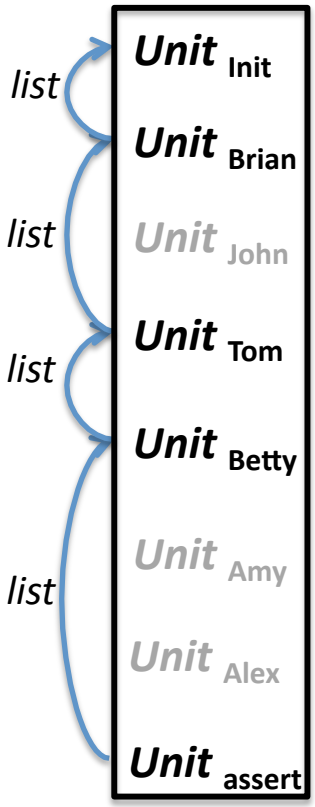


Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11  Assert(list->next==null);
   /* Assert the list has one element */
```



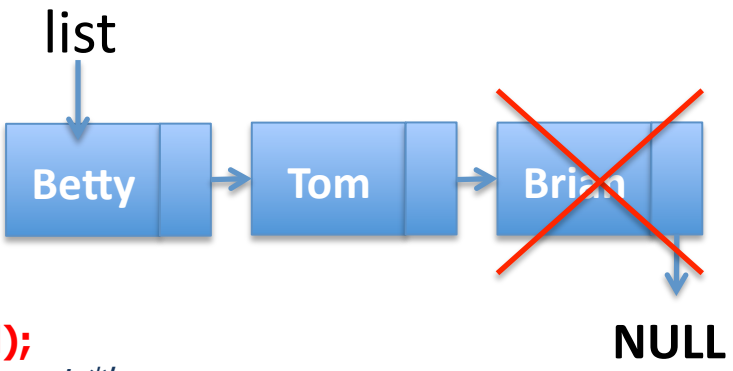
Unit dependencies



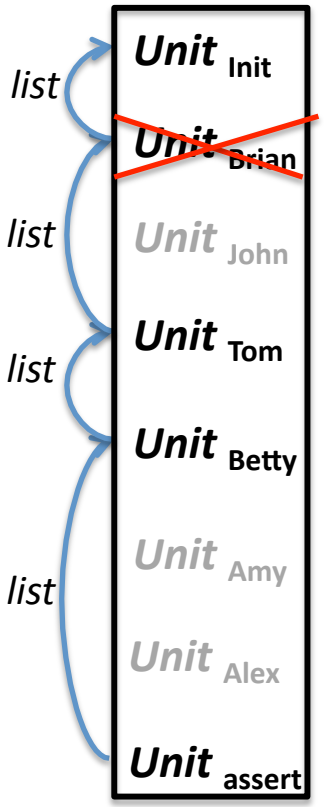
Assertion fail reproduced!

Basic replay

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
       }
9     count++;
     }
10   print("count="+count);
11  Assert(list->next==null);
   /* Assert the list has one element */
```



Unit dependencies



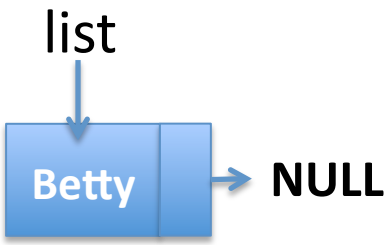
Assertion fail reproduced!

Aggressive Reduction

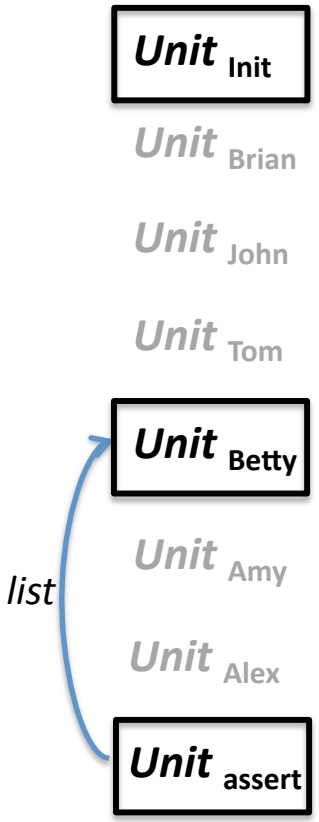
- Compatibility
  - Reconstruct part of execution states, but still can insure replayability
  - We use a BFS algorithm to find a compatible reduction

Aggressive Reduction

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11  Assert(list->next==null);
   /* Assert the list has one element */
```



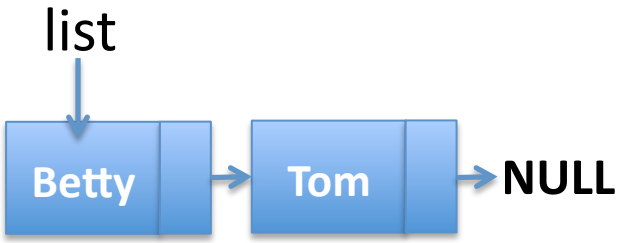
Unit dependencies



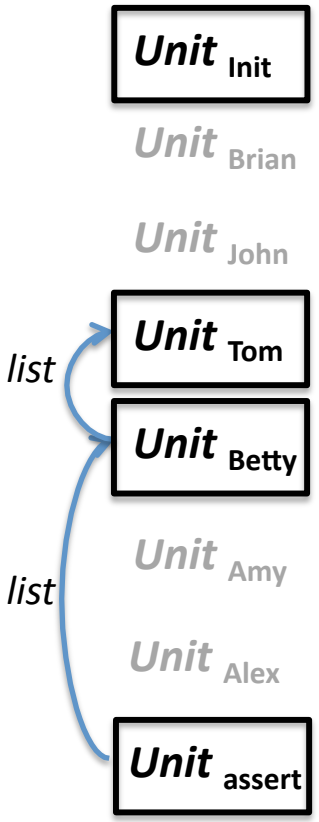
Assertion fail DOES NOT reproduced!

Aggressive Reduction

```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new Node(name);
6       n->next = list;
7       list = n;
8       print(name);
      }
9     count++;
    }
10   print("count="+count);
11  Assert(list->next==null);
   /* Assert the list has one element */
```



Unit dependencies



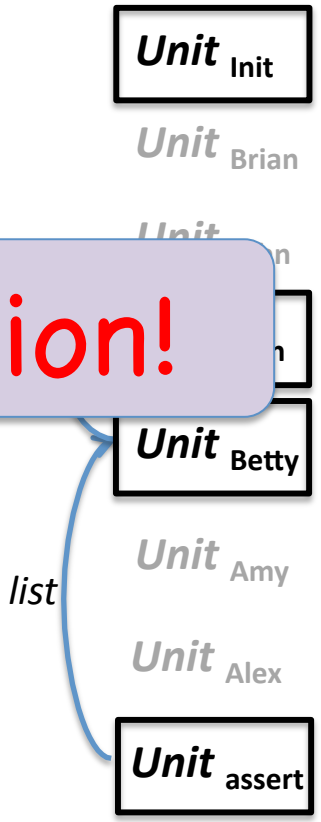
Assertion fail reproduced!

Aggressive Reduction

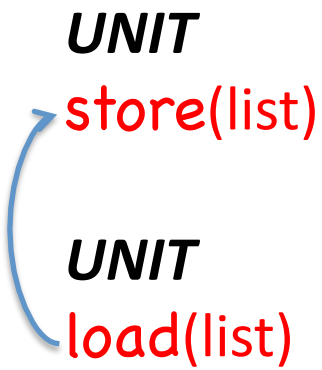
```
1    int count=0;
2    Node *list = null;
3    [UNIT] while(read(&name, &age)) {
4     if(age > 18) {
5       n = new
6       n->next
7       list = n;
8       print(na
      }
9     count++;
    }
10   print("count="+count);
11  Assert(list->next==null);
   /* Assert the list has one element */
```

Compatible Reduction!

Unit dependencies



Optimization



Optimization

UNIT

store(list)

load(list)

UNIT

load(list)

Does not make any dependence across units!

Optimization

UNIT

store(list)

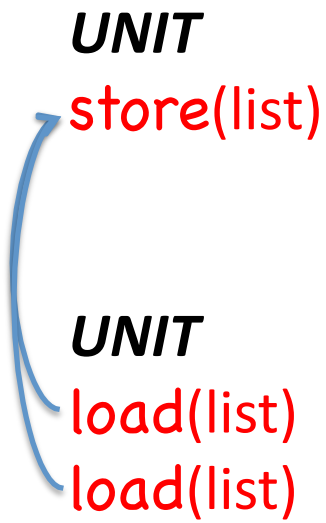
~~load(list)~~

UNIT

load(list)

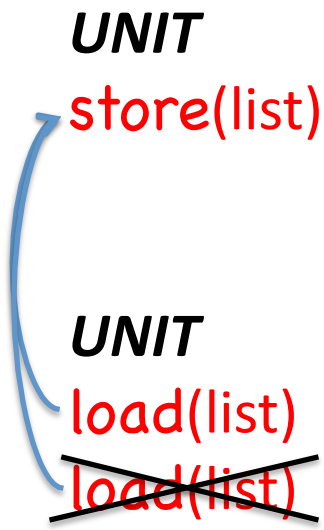
Does not make any dependence across units!

Optimization



Redundant dependence!

Optimization



Redundant dependence!

Evaluation

- Our system makes use of
  - LLVM – static analysis and instrumentation
  - Jockey – Logging system calls and extra information
  - Pin – Replay component
  - Reduction is implemented in C++
- Evaluate our technique with 11 real bugs from 8 applications

Evaluation – Bug description

Applications	LOC	# of threads	Bug description
Apache-2.0.48	157K	16	#1 : Atomicity violation #2 : Unprotected buffer #3 : Cache size problem
BerkeleyDB-4.7.25	172K	5	#1 : Failure in leader election #2 : Panic caused by out-dated messages
Squid-2.3.4	62K	1	Buffer overflow
MC-4.5.55	106K	1	Buffer overflow
W3M-0.5.2	51K	1	Out of memory
VIM-7.0	230K	1	Hangs (100% CPU usage)
DC-1.3	9.5K	1	Segmentation fault
YAFC-1.1.1	41K	1	Segmentation fault

Evaluation – Bug description

Applications	LOC	# of threads	Bug description
Apache-2.0.48	157K	16	#1 : Atomicity violation #2 : Unprotected buffer #3 : Cache size problem
BerkeleyDB-4.7.25	172K	5	#1 : Failure in leader election #2 : Panic caused by out-dated messages
Squid-2.3.4	62K	1	Buffer overflow
MC-4.5.55	106K	1	Buffer overflow
W3M-0.5.2	51K	1	Out of memory
VIM-7.0	230K	1	Hangs (100% CPU usage)
DC-1.3	9.5K	1	Segmentation fault
YAFC-1.1.1	41K	1	Segmentation fault

Evaluation - Instrumentation

Applications	Annotated loops	Data structure	# of Instrumentation	# of Instrumentation after optimization
Apache-2.0.48	2	32	2,333	1,775
BerkeleyDB-4.7.25	3	47	24,338	15,319
Squid-2.3.4	1	21	2,877	2,232
MC-4.5.55	1	18	1,967	1,460
W3M-0.5.2	1	12	5,506	3,609
VIM-7.0	1	25	12,040	8,375
DC-1.3	1	2	641	521
YAFC-1.1.1	1	8	1,785	1,337
Average	***1.37***	***20.6***	***6435***	***4328***

Evaluation – Logging overhead

Applications	Time Overhead			Space Overhead (MB)	
	Time (sec)	Jockey overhead	Our overhead	Jockey log	Our log
Apache-2.0.48	210.0	3.46%	0.17%	8.79	47.24
BerkeleyDB-4.7.25	48.3	2.94%	4.20%	6.3	3.43
Squid-2.3.4	79.04	2.53%	1.1%	22.18	30.35
MC-4.5.55	58.1	3.25%	2.26%	12.9	4.34
W3M-0.5.2	N/A	N/A	N/A	4.42	5.09
VIM-7.0	55.26	0.94%	3.93%	4.14	23.27
DC-1.3	35.63	7.11%	4.8%	12.45	5.04
YAFC-1.1.1	63.03	3.48%	1.82%	49.11	1.78

- **Overhead by our technique is 2.6% average.**
- **Total overhead including Jockey is 5.04 % average.**

Evaluation – Logging overhead

Applications	Time Overhead			Space Overhead (MB)	
	Time (sec)	Jockey overhead	Our overhead	Jockey log	Our log
Apache-2.0.48	210.0	3.46%	0.17%	8.79	47.24
BerkeleyDB-4.7.25	48.3	2.94%	4.20%	6.3	3.43
Squid-2.3.4	79.04	2.53%	1.1%	22.18	30.35
MC-4.5.55	58.1	3.25%	2.26%	12.9	4.34
W3M-0.5.2	N/A	N/A	N/A	4.42	5.09
VIM-7.0	55.26	0.94%	3.93%	4.14	23.27
DC-1.3	35.63	7.11%	4.8%	12.45	5.04
YAFC-1.1.1	63.03	3.48%	1.82%	49.11	1.78

- **Overhead by our technique is 2.6% average.**
- **Total overhead including Jockey is 5.04 % average.**

Evaluation – Reduction result

Bugs	Log before reduction		Reduced log with search	
	# of units	Log Size(MB)	# of units	Our log(MB)
Apache	15,005	55.84	20	0.86
DB	2,033K	170.6	2	0.06
Squid	8,000	52.6	91	0.69
MC	5,348	17.75	5	1.62
W3M	961	5.65	2	0.051
VIM	2,544	27.41	12	0.314
DC	6,580	17.49	14	1.59
YAFC	196	113.06	2	0.111
Average	***259K***	***57.55***	***18.5***	***0.66***

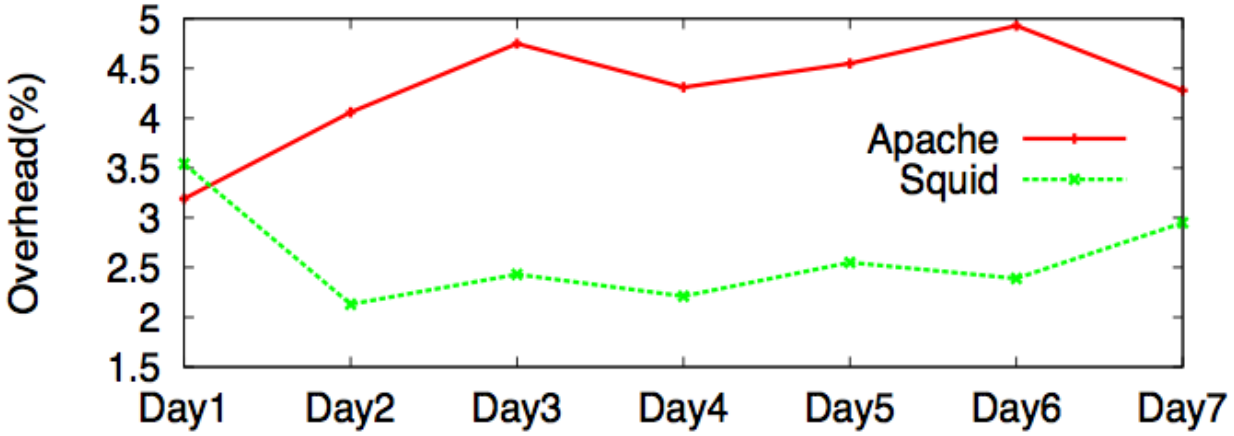
Evaluation – Reduction result

Bugs	Log before reduction		Reduced log with search	
	# of units	Log Size(MB)	# of units	Our log(MB)
Apache	15,005	55.84	20	0.86
DB	2,033K	170.6	2	0.06
Squid	8,000	52.6	91	0.69
MC	5,348	17.75	5	1.62
W3M	961	5.65	2	0.051
VIM	2,544	27.41	12	0.314
DC	6,580	17.49	14	1.59
YAFC	196	113.06	2	0.111
Average	***259K***	***57.55***	***18.5***	***0.66***

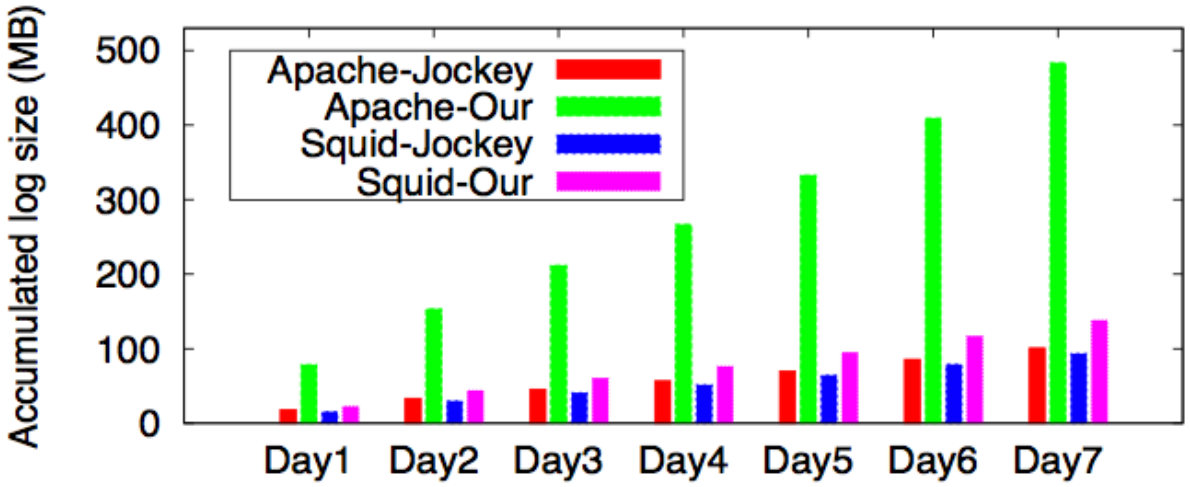
Evaluation – Reduction result

Bugs	Log before reduction		Reduced log with search	
	# of units	Log Size(MB)	# of units	Our log(MB)
Apache	15,005	55.84	20	0.86
DB	2,033K	170.6	2	0.06
Squid	8,000	52.6	91	0.69
MC	5,348	17.75	5	1.62
W3M	961	5.65	2	0.051
VIM	2,544	27.41	12	0.314
DC	6,580	17.49	14	1.59
YAFC	196	113.06	2	0.111
Average	***259K***	***57.55***	***18.5***	***0.66***

Evaluation – with real requests



(a) Runtime overhead



(b) Space overhead

Related Works

- Execution Fast Forwarding

  [S.Tallam et al. '07], [X.Zhang et al. '06]

  Expensive off-line analysis between events

- Checkpointing

  [K.M.Chandy et al. '07], [G. Xu et al. '07],

  [G.Broneevetsky et al. '05]

  Our technique allows fine-grained reduction

Related Works

- Language based replay system
  [M.Wu et al. '10], [R.Xue et al. '09]
  [S.Joshi. '07]
- Software based replay system
  [G.W.Dunlap et al. '08], [A.Ayers et al. '05]
  [Z.Guo et al. '08], [S.T.King et al. '05]
- Debugging concurrent programs
  [G.Altekar et al. '09], [P.Joshi et al. '09],
  [M.Musuvathi et al. '07], [S.Park et al. '09]

Conclusion

- Compiler based technique that generates a reducible replay log
  - Divides execution into UNITs
  - Instrument programs to collect minimal additional information
  - Reduction can be achieved through analyzing just the log
- Average additional cost by our technique is 2.6%
- Reduce executions with up to 2,033K UNITs to less than 91 Units

Q & A