JPF test of concurrent linear hashing

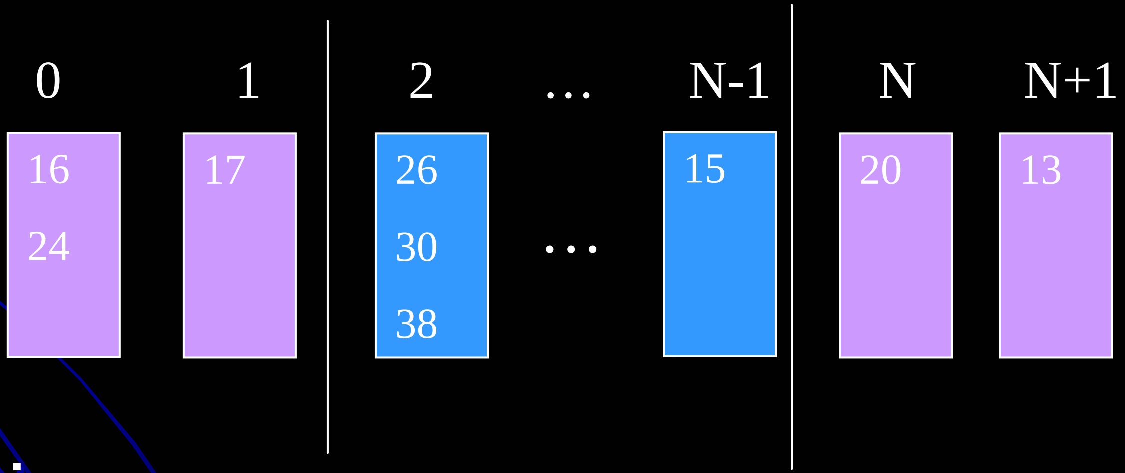
Huxia Shi

CSE, York University

May19, 2009

Linear Hashing Review

- A technique of dynamic hashing
- Data structure
  - Root variables: next and level
  - Sequence of bucket chains



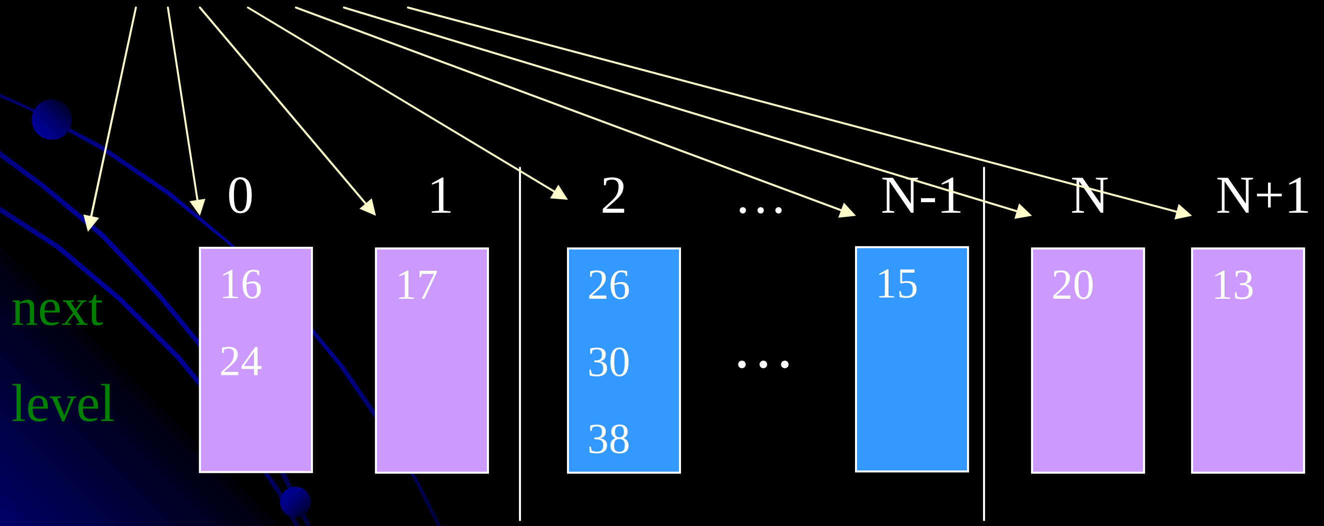
- Operations
  - Find, Insert, Delete, Split, Merge

Solutions

- Concurrent solution
  - Carla Schlatter Ellis. Concurrency in linear hashing. ACM Transactions on Database Systems, 12(2): 195-217, June 1987
- Sequential solution

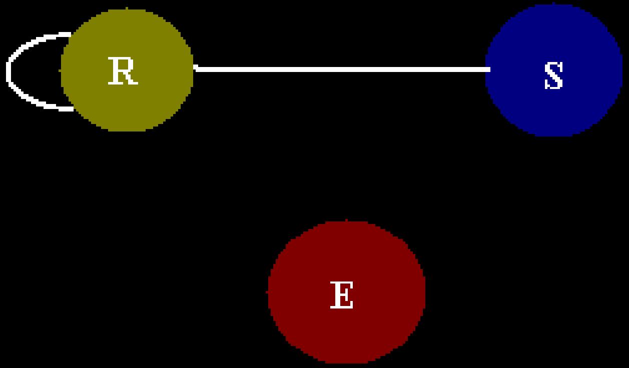
Concurrent Solution

- Three lock types
  - Read Lock
  - Selective Lock
  - Exclusive Lock



Concurrent Solution

	Existing lock		
Lock Request	Read lock	Selective lock	Exclusive lock
Read lock	yes	yes	no
Selective lock	yes	no	no
Exclusive lock	no	no	no



Concurrent Solution

- Lock-coupling protocols

  add lock on first element, then next element

  release lock on first element, then next element

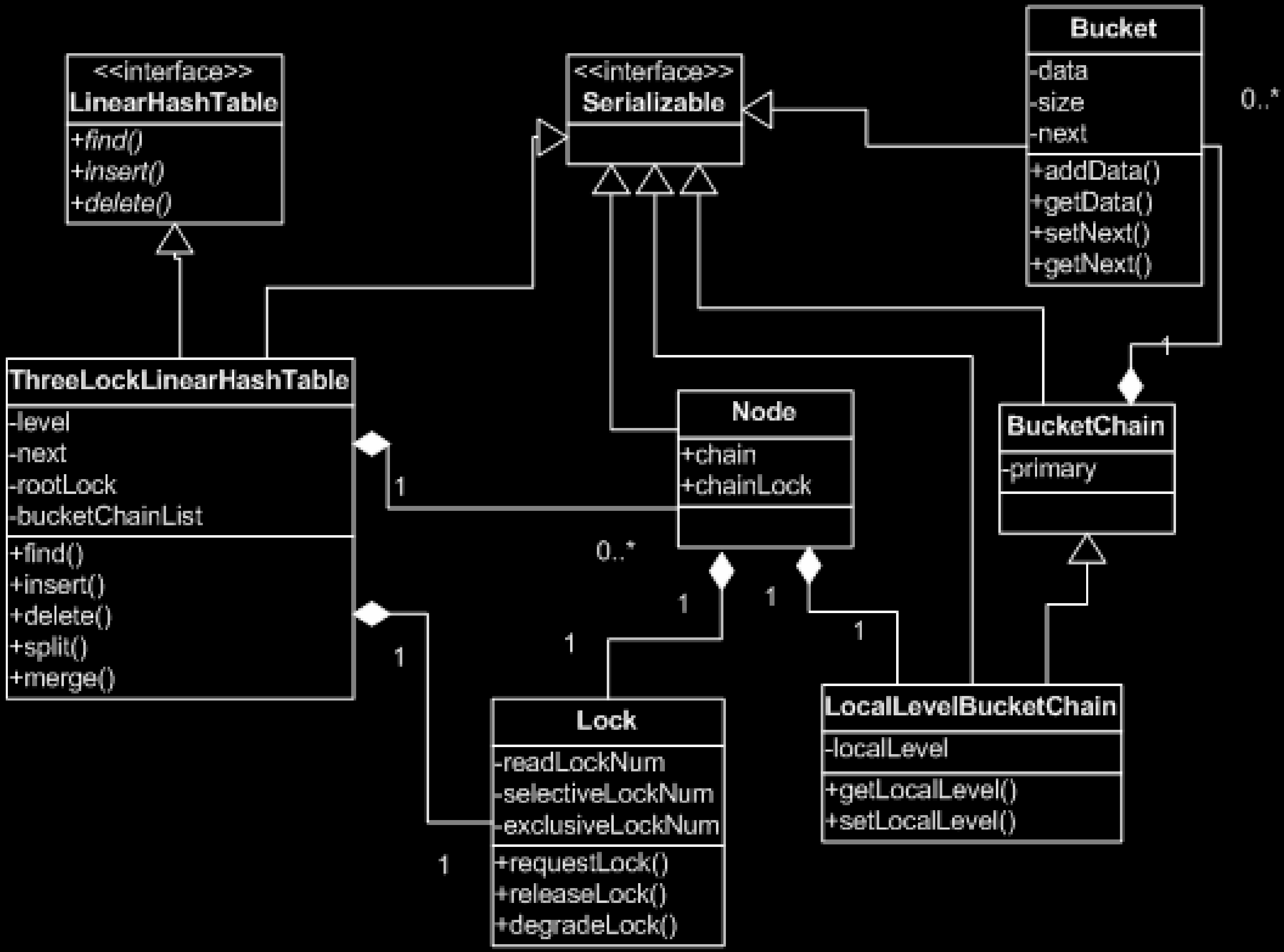
- Local level technique

  A duplicated local level at bucket chain

  Allow concurrent access to root variables

  (find, insert, delete, split)

Concurrent Implementation



Sequential Solution

- All operations are serialized

```
public synchronized boolean find(int key) { … }

public synchronized void insert(int key) { … }

public synchronized void delete(int key) { … }
```

Test setting

- Bucket size: 2
- Each thread inserts/deletes/finds 4 numbers
- Different types of threads use same data

2 find threads:	0-3	4-7
2 insert threads:	0-3	4-7
2 delete threads:	0-3	4-7

- Max memory for JPF test

  2.5G

Test Plan

- Deadlock freedom
- Data Race
- Check lock number consistency

(The last item is only verified in concurrent solution)

Uncaught exception

- One uncaught exception in concurrent solution

  Exception: Array index out of range

  Root cause: run merge on hash map with init root variables (level==0 and next==0)

  Result: next pointer become -1, out of array range

  Solution: Block merge in this case

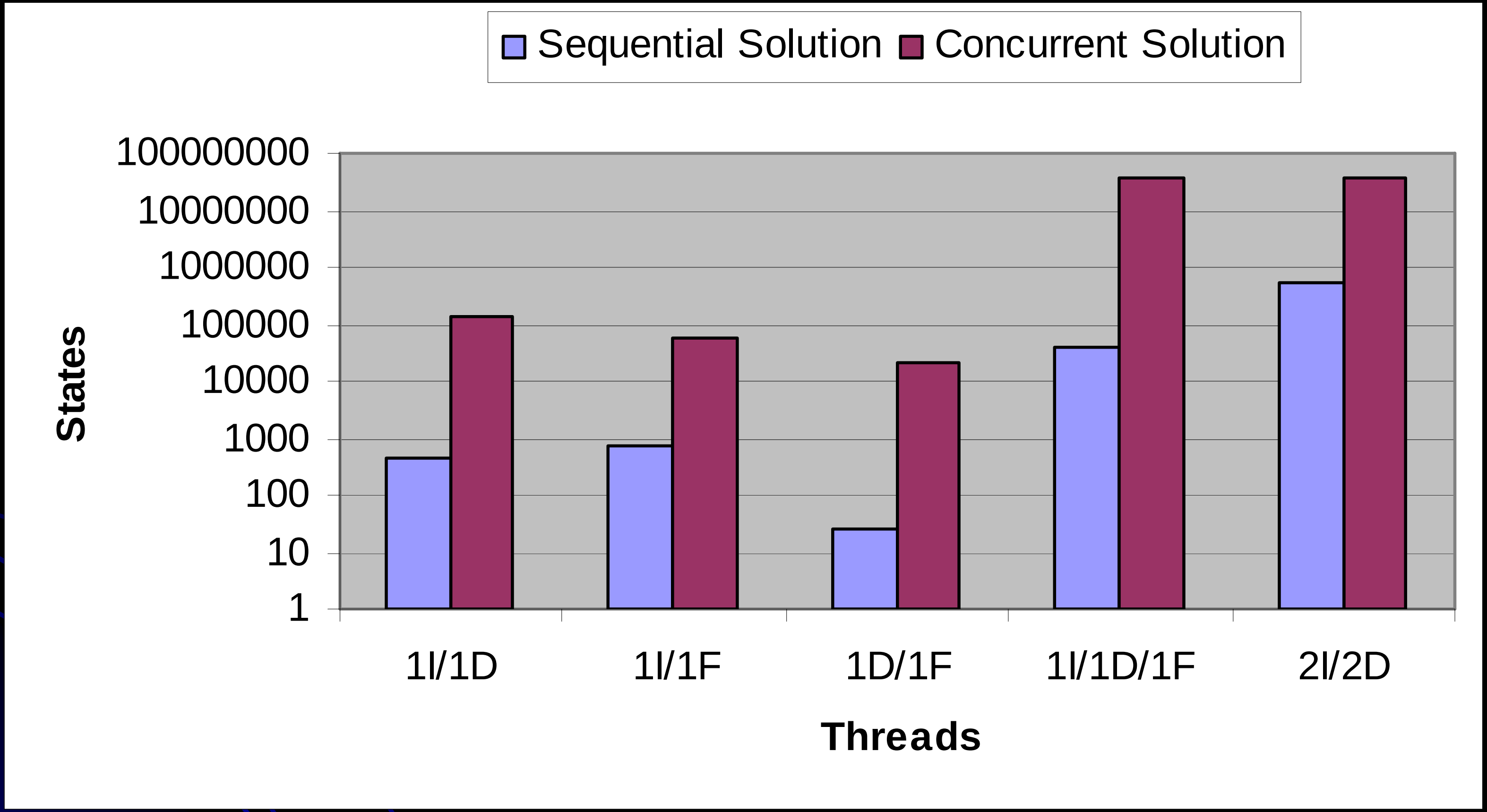
Deadlock Test - Sequential Solution

Insert Threads	Delete Threads	Find Threads	Time	States
1	1	0	0:00:02	443
1	0	1	0:00:01	718
0	1	1	0:00:01	27
1	1	1	0:00:14	41688
2	0	0	0:00:06	467
0	2	0	0:00:01	197
0	0	2	0:00:01	469
2	2	0	0:03:12	514939
2	0	2	0:02:13	394213
0	2	2	0:00:30	51966
2	2	2	10:58:20	Out of memory

Deadlock Test - Concurrent Solution

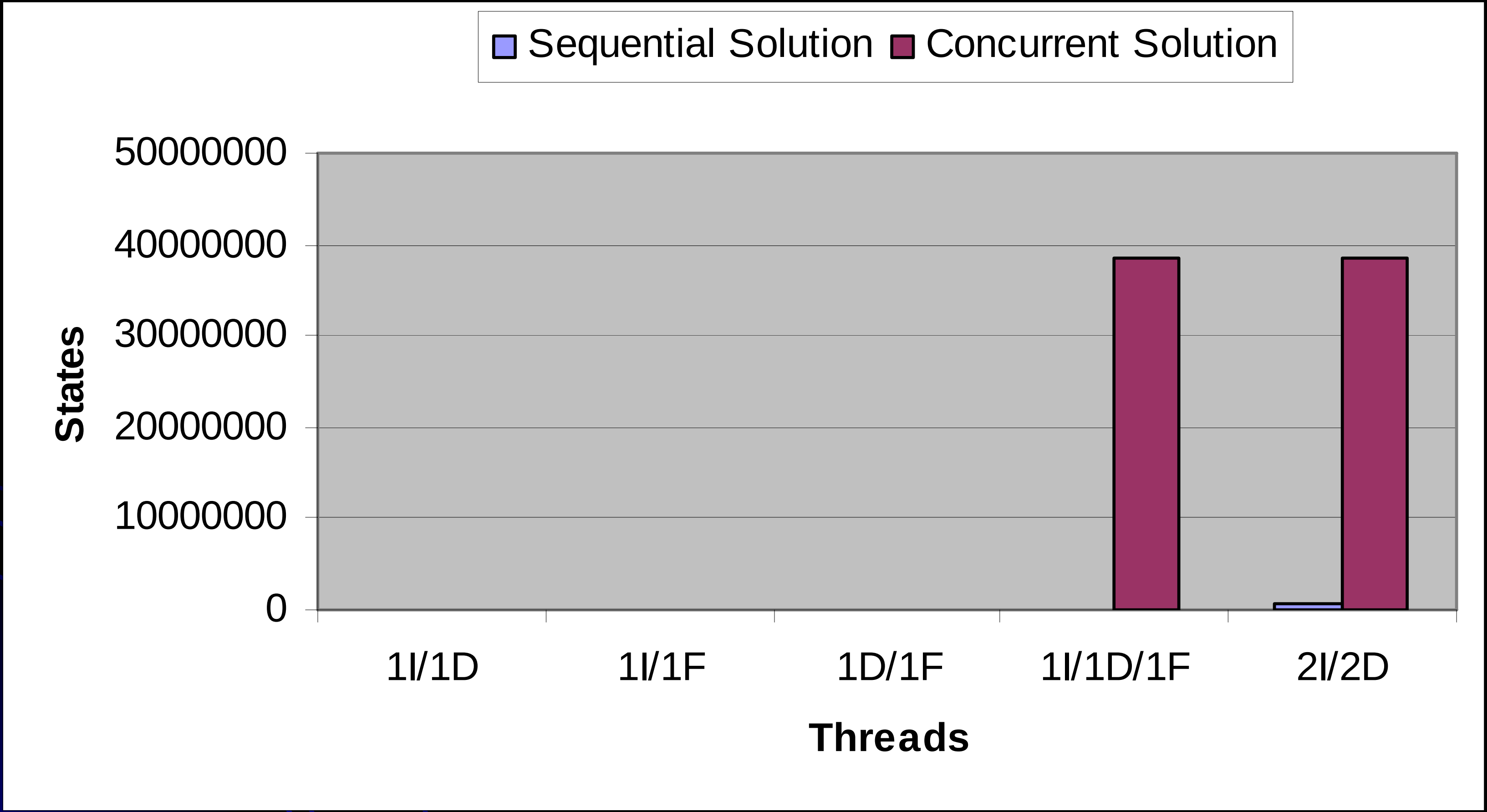
Insert Threads	Delete Threads	Find Threads	Time	States
1	1	0	0:00:42	144096
1	0	1	0:00:18	56910
0	1	1	0:00:08	20571
1	1	1	4:32:58	38550712
2	0	0	0:01:02	202505
0	2	0	0:00:08	21256
0	0	2	0:00:35	115250
2	2	0	4:50:43	38636347
2	0	2	14:33:51	Out of Memory
0	2	2	18:08:37	Out of Memory
2	2	2	19:05:26	Out of Memory

State Space



Sequential Solution
Concurrent Solution
States
100000000
10000000
1000000
100000
10000
1000
100
10
1
1I/1D
1I/1F
1D/1F
1I/1D/1F
2I/2D
Threads

State Space



Sequential Solution
Concurrent Solution
50000000
40000000
30000000
20000000
10000000
0
States
1I/1D
1I/1F
1D/1F
1I/1D/1F
2I/2D
Threads

Test Plan

- Deadlock freedom
- Data Race
- Check lock number consistency

(The last item is only verified in concurrent solution)

Data Race

- Sequential Solution

  No data race found

- Current Solution

  Data race is found

  Root cause: Split and find/insert/delete threads access root variables level at the same time

  split: this.level++

  locate: int lev = this.level

  Result: locate wrong bucket chain

  Solution: Local level technique handles this problem

Test Plan

- Deadlock freedom
- Data Race
- Check lock number consistency

(The last item is only verified in concurrent solution)

Lock number consistency

- After getting read lock

```
assert exclusiveLockNum == 0;
```

- After getting selective lock

```
assert exclusiveLockNum == 0;
assert selectiveLockNum == 1;
```

- After getting exclusive lock

```
assert readLockNum == 0;
assert selectiveLockNum == 0;
assert exclusiveLockNum == 1;
```

Experience with JPF

- The join in main method has strong influence on JPF run time

	1I/1D	1I/1F	1D/1F
New states (without join)	443	718	27
New states (with join)	21472	117844	8286

Conclusion

- Sequential solution
  - Simple in implementation
  - Small state space
  - More efficient (only memory operations)

- Concurrent solution
  - Complicated in implementation
  - Large state space
  - Worse efficiency (only memory operations)
  - Better performance with a lot of disk IO

End

Q&A

Thanks