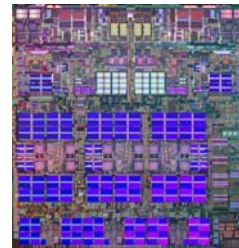


Applying Thread Level Speculation to Database Transactions

Chris Colohan

(Adapted from his Thesis Defense talk)

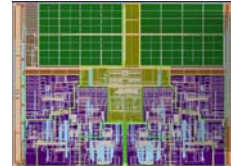
Chip Multiprocessors are Here!



AMD Opteron



IBM Power 5



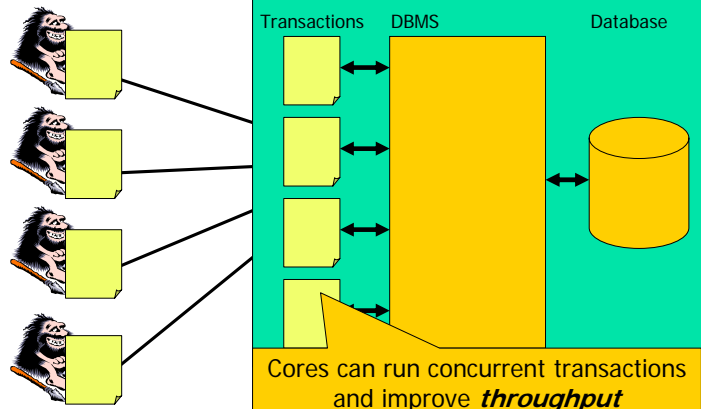
Intel Yonah

- 2 cores now, soon will have 4, 8, 16, or 32
- Multiple threads per core
- How do we best use them?

2

Multi-Core Enhances Throughput

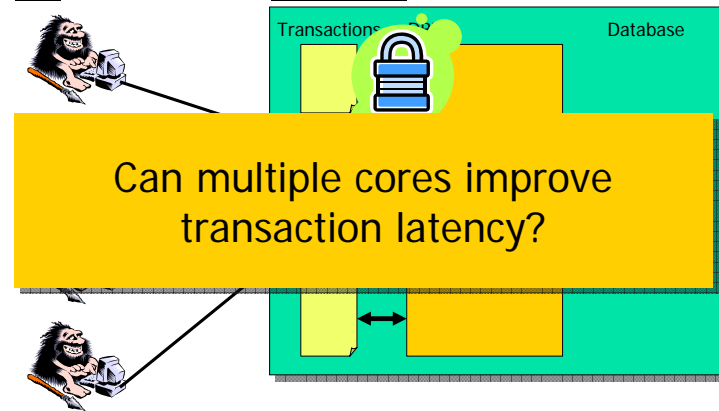
Users



3

Multi-Core Enhances Throughput

Users



4

Parallelizing transactions

```

SELECT cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
  GET quantity FROM stock;
  quantity--;
  UPDATE stock WITH quantity;
  INSERT item INTO order_line;
}
  
```

DBMS

- Intra-query parallelism
 - Used for long-running queries (decision support)
 - Does not work for short queries
- Short queries dominate in commercial workloads

5

Parallelizing transactions

```

SELECT cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
  GET quantity FROM stock;
  quantity--;
  UPDATE stock WITH quantity;
  INSERT item INTO order_line;
}
  
```

DBMS

- Intra-transaction parallelism
 - Each thread spans multiple queries
- Hard to add to existing systems!
 - Need to change interface, add latches and locks, worry about correctness of parallel execution...

6

Parallelizing transactions

```

SELECT cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
  GET quantity FROM stock;
  quantity--;
  UPDATE stock WITH quantity;
  INSERT item INTO order_line;
}
  
```

DBMS

Thread Level Speculation (TLS)
makes parallelization easier.

7

Thread Level Speculation (TLS)

Time ↓

Sequential

Parallel

Epoch 1

Epoch 2

*p=

*q=

=*p

=*q

=*p

=*q

8

Thread Level Speculation (TLS)

- Use *epochs*
- Detect violations
- Restart to recover
- Buffer state
- Worst case:
 - Sequential
- Best case:
 - Fully parallel

Time ↓

Sequential Parallel

Data dependences limit performance.

A Coordinated Effort

Transactions (TPC-C)

DBMS (BerkeleyDB)

Hardware (Simulated machine)

10

A Coordinated Effort

Transaction Programmer (Choose epoch boundaries)

DBMS Programmer (Remove performance bottlenecks)

Hardware Developer (Add TLS support to architecture)

11

Outline

- Introduction
- Related work
- **Dividing transactions into epochs**
- Removing bottlenecks in the DBMS
- Hardware Support
- Results
- Conclusions

12

Case Study: New Order (TPC-C)

```
GET cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
    GET quantity FROM stock
    WHERE i_id=item;
    UPDATE stock WITH quantity-1
    WHERE i_id=item;
    INSERT item INTO order_line;
}
```

*78% of transaction
execution time*

- Only dependence is the **quantity** field
 - Very unlikely to occur (1/100,000)

13

Case Study: New Order (TPC-C)

```
GET cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
    GET quantity FROM stock
    WHERE i_id=item;
    UPDATE stock WITH quantity-1
    WHERE i_id=i
    INSERT item INTO
}
```

```
GET cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
TLS_foreach(item) {
    GET quantity FROM stock
    WHERE i_id=item;
    UPDATE stock WITH quantity-1
    WHERE i_id=item;
    INSERT item INTO order_line;
}
```

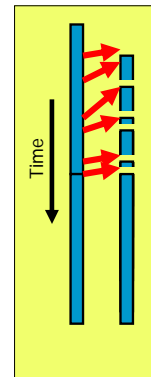
14

Outline

- Introduction
- Related work
- Dividing transactions into epochs
- **Removing bottlenecks in the DBMS**
- Hardware support
- Results
- Conclusions

15

Dependencies in DBMS



16

Dependences in DBMS

Dependences serialize execution!

Performance tuning:

- Profile execution
- Remove *bottleneck* dependence
- Repeat

17

Buffer Pool Management

CPU `get_page(5)`
`put_page(5)`

Buffer Pool
ref: 0

18

Buffer Pool Management

CPU `get_page(5)`
`put_page(5)`

Buffer Pool
ref: 0

Time

`get_page(5)`
`put_page(5)`

`get_page(5)`
`put_page(5)`

get_page(5)
put_page(5)

get_page(5)
put_page(5)

TLS ensures first epoch gets page first. Who cares?

19

Buffer Pool Management

- Escape speculation
- Invoke operation
- Store *undo function*
- Resume speculation

Buffer Pool
ref: 0

Time

`get_page(5)`
`put_page(5)`

`get_page(5)`
`put_page(5)`

`get_page(5)`
`put_page(5)`

`get_page(5)`
`put_page(5)`

☐ = Escape Speculation

20

get_page() wrapper

```

page_t *get_page_wrapper(pageid_t id) {
    static tls_mutex mut;
    page_t *ret;

    tls_escape_speculation();
    check_get_arguments(id);
    tls_acquire_mutex(&mut);

    ret = get_page(id);
    tls_release_mutex(&mut);
    tls_on_violation(put, ret);
    tls_resume_speculation()

    return ret;
}

```

→ Wraps get_page()

21

get_page() wrapper

```

page_t *get_page_wrapper(pageid_t id) {
    static tls_mutex mut;
    page_t *ret;

    tls_escape_speculation();
    check_get_arguments(id);
    tls_acquire_mutex(&mut);

    ret = get_page(id);

    tls_release_mutex(&mut);
    tls_on_violation(put, ret);
    tls_resume_speculation()

    return ret;
}

```

→ No violations while calling get_page()

22

get_page() wrapper

```

page_t *get_page_wrapper(pageid_t id) {
    static tls_mutex mut;
    page_t *ret;

    tls_escape_speculation();
    check_get_arguments(id);
    tls_acquire_mutex(&mut);

    ret = get_page(id);

    tls_release_mutex(&mut);
    tls_on_violation(put, ret);
    tls_resume_speculation()

    return ret;
}

```

→ May get bad input data from speculative thread!

23

get_page() wrapper

```

page_t *get_page_wrapper(pageid_t id) {
    static tls_mutex mut;
    page_t *ret;

    tls_escape_speculation();
    check_get_arguments(id);
    tls_acquire_mutex(&mut);

    ret = get_page(id);

    tls_release_mutex(&mut);
    tls_on_violation(put, ret);
    tls_resume_speculation()

    return ret;
}

```

→ Only one epoch per transaction at a time

24

Removing Bottleneck Dependences

We introduce three techniques:

- **Delay operations** until non-speculative
 - Mutex and lock *acquire* and *release*
 - Buffer pool, memory, and cursor *release*
 - Log sequence number assignment
- **Escape speculation**
 - Buffer pool, memory, and cursor *allocation*
- **Traditional parallelization**
 - Memory allocation, cursor pool, error checks, false sharing

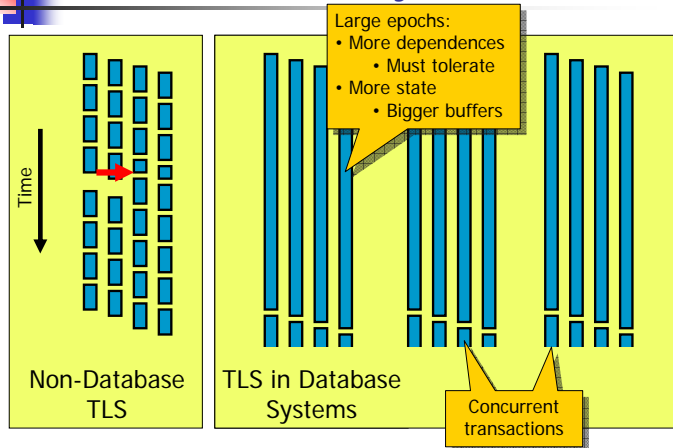
29

Outline

- Introduction
- Related work
- Dividing transactions into epochs
- Removing bottlenecks in the DBMS
- **Hardware support**
- Results
- Conclusions

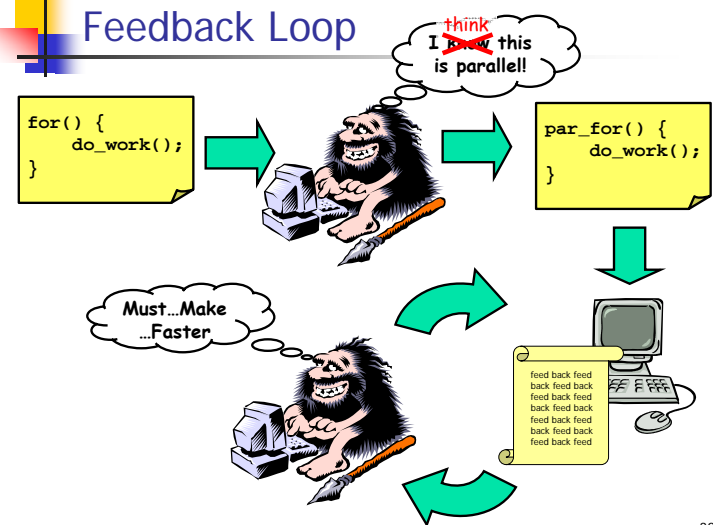
30

TLS in Database Systems



31

Feedback Loop



32

Violations == Feedback

Sequential

Violation!

Must...Make...Faster

0x0FD8
0x0FD20
0x0FC0
0x0FC18

33

Eliminating Violations

Parallel

Eliminate *p Dep.

Optimization may make slower?

0x0FD8
0x0FD20
0x0FC0
0x0FC18

34

Tolerating Violations: Sub-epochs

Eliminate *p Dep.

Sub-epochs

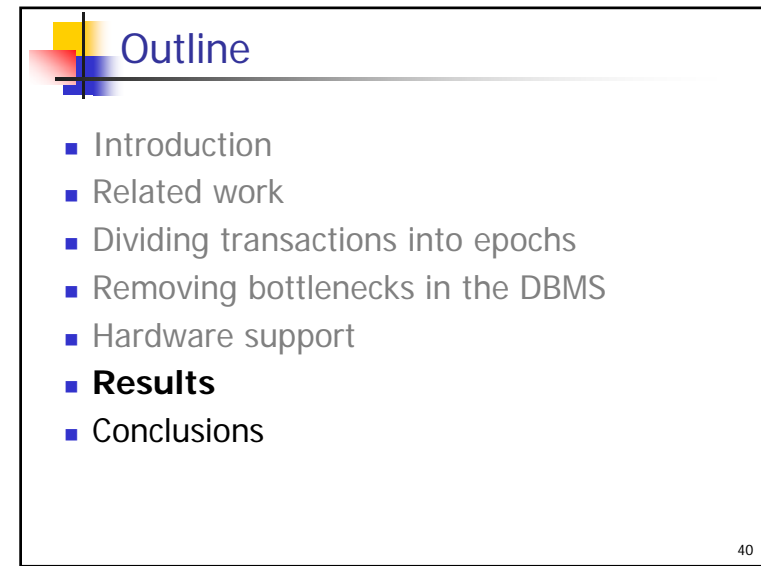
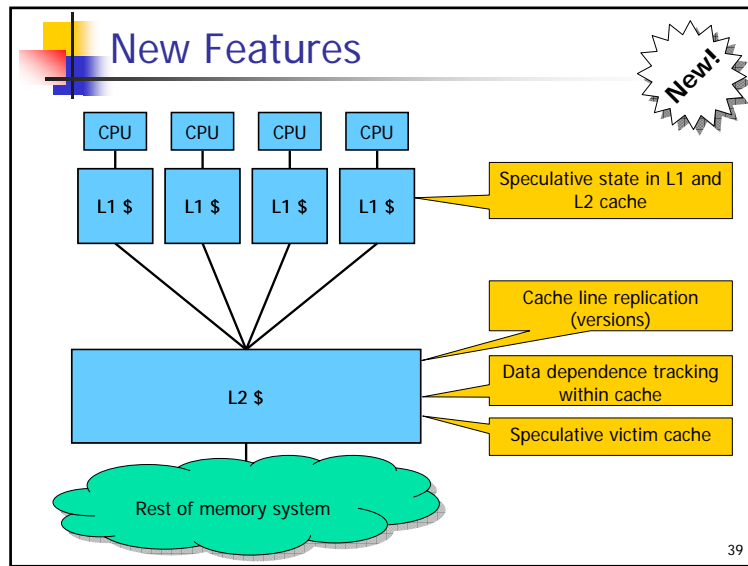
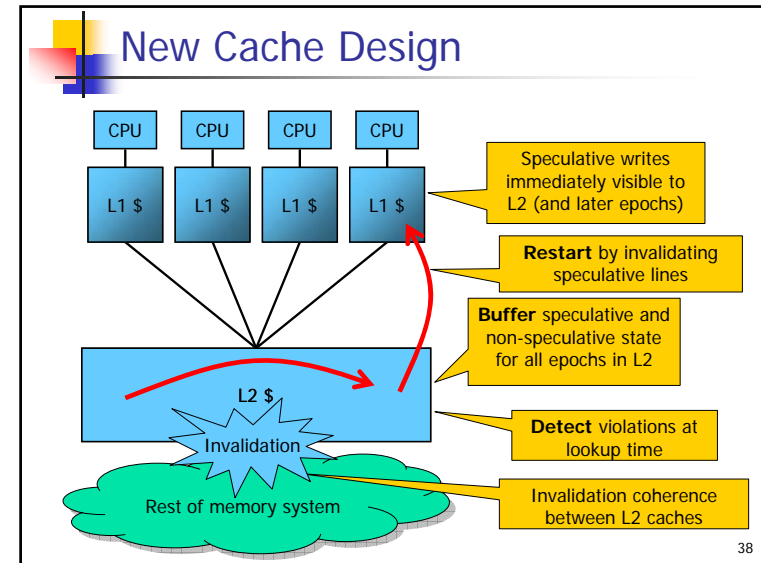
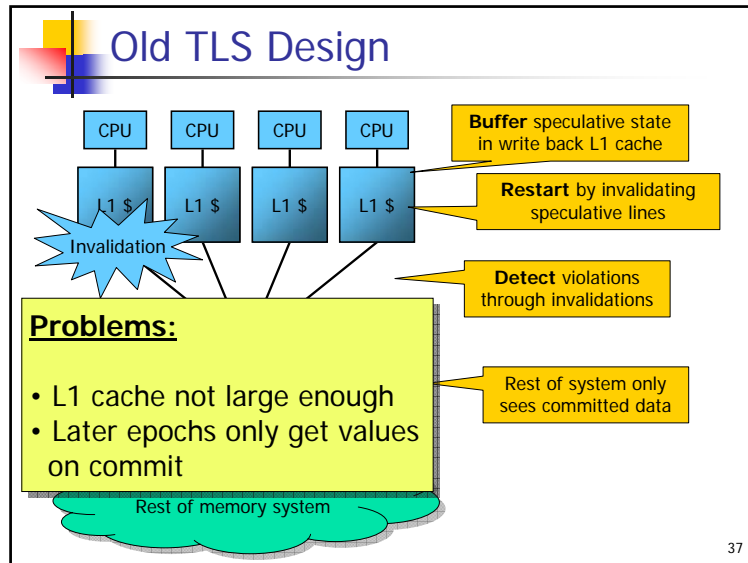
35

Sub-epochs

- Started periodically by hardware
 - How many?
 - When to start?
- Hardware implementation
 - Just like epochs
 - Use more epoch contexts
 - No need to check violations between sub-epochs within an epoch

Sub-epochs

36

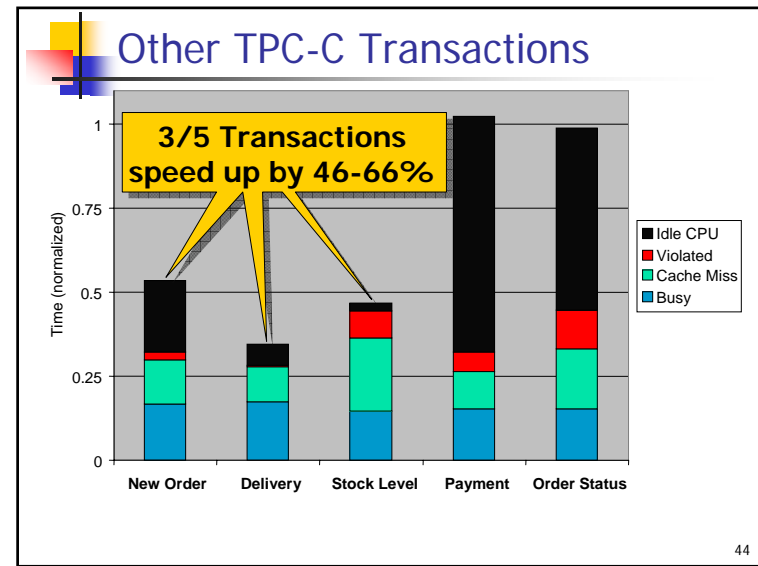
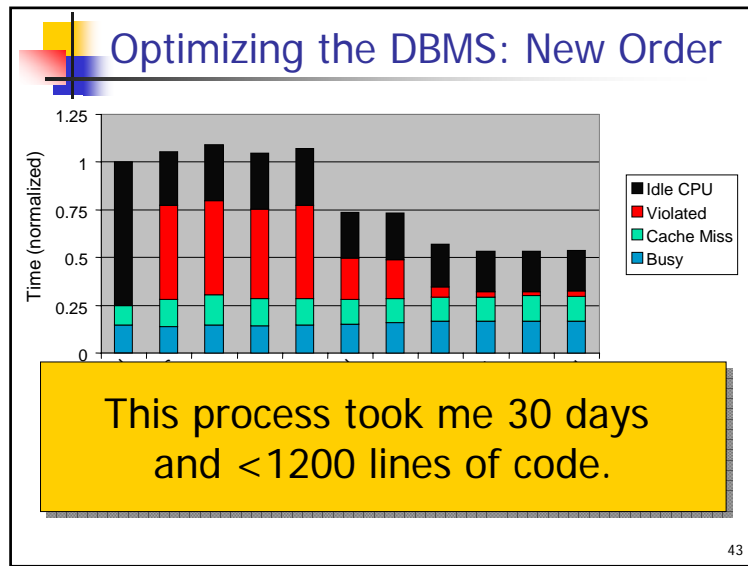
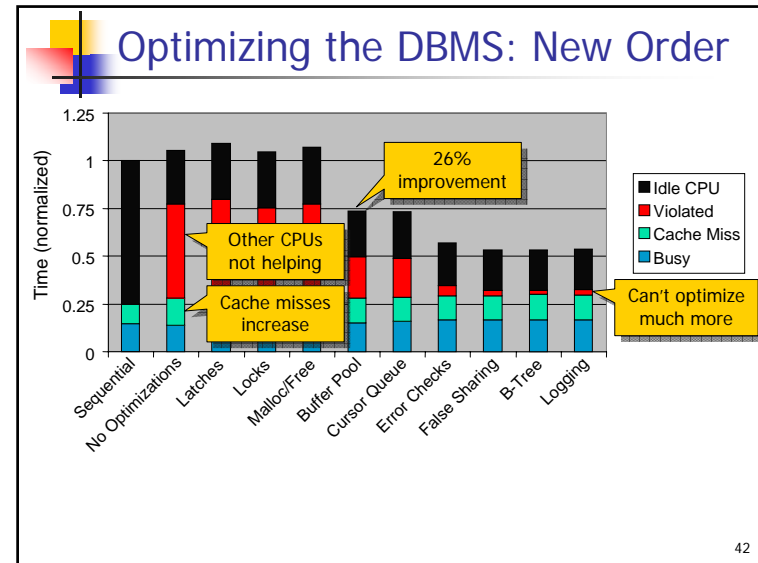


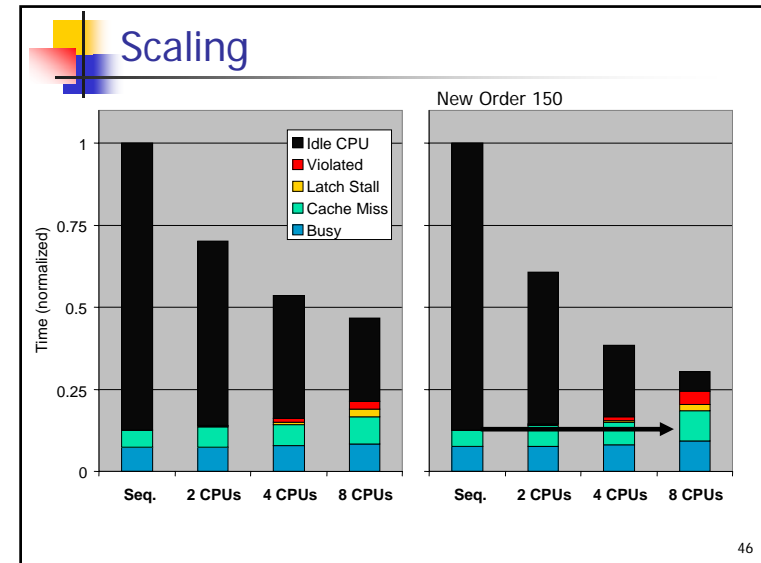
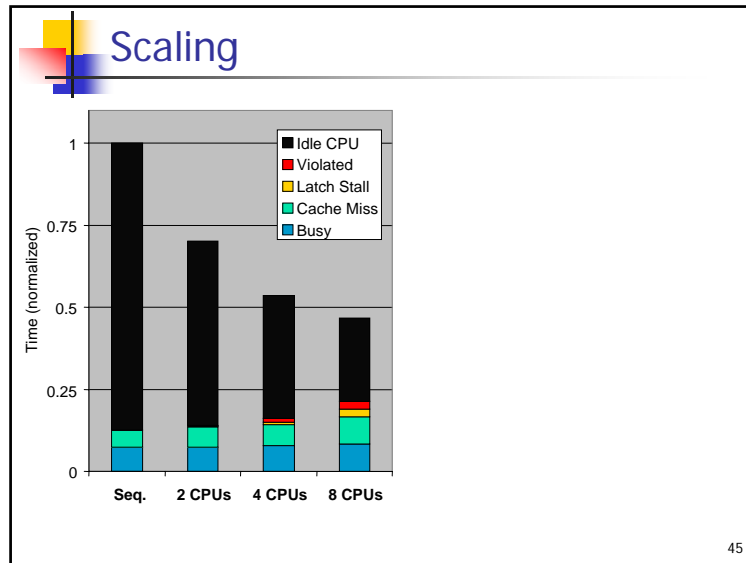
Experimental Setup

- Detailed simulation
 - Superscalar, out-of-order, 128 entry reorder buffer
 - Memory hierarchy modeled in detail
- TPC-C transactions on BerkeleyDB
 - In-core database
 - Single user
 - Single warehouse
 - Measure interval of 100 transactions
 - Measuring *latency* not throughput

The diagram shows four CPUs at the top, each with a 32KB 4-way L1 cache. These are connected to a central 2MB 4-way L2 cache. Below the L2 cache is a cloud representing the 'Rest of memory system'.

41





- ## Conclusions
- A new form of parallelism for databases
 - Tool for attacking transaction **latency**
 - Intra-transaction parallelism
 - Without major changes to DBMS
 - With feasible new hardware
 - TLS can be applied to more than transactions

 - Halve transaction latency by using 4 CPUs
- 47