Applying Thread Level Speculation to Database Transactions

Chris Colohan
(Adapted from his Thesis Defense talk)

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

Multi-Core Enhances Throughput

Can multiple cores improve transaction latency?
Parallelizing transactions

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

Thread Level Speculation (TLS) makes parallelization easier.

**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...

```
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**

```
*p = *q = *p
=q = *p
=q = *q
```

Sequential

Epoch 1

Parallel

Epoch 2

Thread Level Speculation (TLS) makes parallelization easier.
Thread Level Speculation (TLS)

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

Data dependences limit performance.

A Coordinated Effort

- Transactions
- DBMS
- Hardware

A Coordinated Effort

- Choose epoch boundaries
- Remove performance bottlenecks
- Add TLS support to architecture

Outline

- Introduction
- Related work
- Dividing transactions into epochs
- Removing bottlenecks in the DBMS
- Hardware Support
- Results
- Conclusions
Case Study: New Order (TPC-C)

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

78% of transaction execution time

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

Dependences in DBMS
**Dependences in DBMS**

*Dependences serialize execution!*

Performance tuning:
- Profile execution
- Remove *bottleneck* dependence
- Repeat

**Buffer Pool Management**

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

\[\text{get\_page}(5)\]

\[\text{put\_page}(5)\]
get_page() wrapper

```c
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()`

No violations while calling `get_page()`

Only one epoch per transaction at a time

May get bad input data from speculative thread!
get_page() wrapper

```c
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;
}
```

get_page() wrapper

```c
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;
}
```

### Buffer Pool Management

- **get_page(5)**
- **put_page(5)**
- **get_page(5)**
- **put_page(5)**
- **get_page(5)**
- **put_page(5)**
- **Not undoable!**
- **= Escape Speculation**

### Buffer Pool Management

- **get_page(5)**
- **put_page(5)**
- **get_page(5)**
- **put_page(5)**
- **get_page(5)**
- **get_page(5)**
- **= Escape Speculation**

- **Delay put_page until end of epoch**
- **Avoid dependence**

Isolated

- Undoing this operation does not cause cascading aborts

Undoable

- Easy way to return system to initial state

Can also be used for:
- Cursor management
- malloc()
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

Outline

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

TLS in Database Systems

Large epochs:
- More dependences
- Must tolerate
- More state
- Bigger buffers

Feedback Loop

I know this is parallel!

```c
for() {
  do_work();
}
```

```c
par_for() {
  do_work();
}
```

Must...
Make...
Faster
Violations == Feedback

Sequential

Violations

Parallel

Eliminating Violations

Sequential

Eliminating Violations

Parallel

Eliminating *p Dep.

Sub-epochs

Tolerating Violations: Sub-epochs

Sequential

Eliminating *p Dep.

Parallel

Eliminating *p Dep.

Sub-epochs

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
Old TLS Design

- **Problems:**
  - L1 cache not large enough
  - Later epochs only get values on commit

New Cache Design

- **Speculative writes immediately visible to L2 (and later epochs)**
- **Restart by invalidating speculative lines**
- **Buffer speculative and non-speculative state for all epochs in L2**
- **Detect violations at lookup time**
- **Invalidation coherence between L2 caches**

New Features

- Speculative state in L1 and L2 cache
- Speculative victim cache
- Data dependence tracking within cache
- Cache line replication (versions)

Outline

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

### Optimizing the DBMS: New Order

- 3/5 Transactions speed up by 46-66%

---

**This process took me 30 days and <1200 lines of code.**
**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