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

Users

Database Server

Transactions DBMS Database

Cores can run concurrent transactions and improve 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

**DBMS**

```
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;
}
```

**Thread Level Speculation (TLS)**

**Database System**

```
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;
}
```

**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...

**Thread Level Speculation (TLS)**

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

**Sequential**

**Parallel**

```
Epoch 1
*p=
*q=
=*p
=*q
```

```
Epoch 2
*p=
*q=
=*p
=*q
```
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

Transactions

- TPC-C
- BerkeleyDB
- Simulated machine

A Coordinated Effort

- Transaction Programmer
- DBMS Programmer
- Hardware Developer

Outline

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

- 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

CPU

get_page(5)
put_page(5)

Buffer Pool

ref: 0

get_page(5)
put_page(5)

Time

get_page(5)
put_page(5)

TLS ensures first epoch gets page first. Who cares?

get_page(5)
put_page(5)

• Escape speculation
• Invoke operation
• Store undo function
• Resume speculation

Buffer Pool

ref: 0

get_page(5)
put_page(5)

= Escape Speculation
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
Buffer Pool Management

- get_page(5)
- put_page(5)

```
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;
}
```

- 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()

Buffer Pool Management

```
Buffer Pool

ref: 0

CPU

get_page(5)
put_page(5)

get_page(5)
put_page(5)

Time

get_page(5)
put_page(5)

Not undoable!

= Escape Speculation

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

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

Think: this is parallel!
Violations == Feedback

Sequential

Must...Make...Faster

Parallel

Optimization may make slower?

Eliminating Violations

Sub-epochs

Tolerating Violations: 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

Restart by invalidating speculative lines

Detect violations through invalidations

Rest of system only sees committed data

New Cache Design

Speculative writes immediately visible to L2 (and later epochs)

Invalidation coherence between L2 caches

Speculative writes immediately visible to L2 (and later epochs)

Detect violations at lookup time

Invalidation coherence between L2 caches

New Features

Speculative state in L1 and L2 cache

Cache line replication (versions)

Data dependence tracking within cache

Speculative victim cache

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

Optimizing the DBMS: New Order

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