Incrementally Parallelizing Database Transactions with Thread-Level Speculation

Todd C. Mowry
Carnegie Mellon University

(in collaboration with Chris Colohan, J. Gregory Steffan, and Anastasia Ailamaki)

Twofold Speedup on a Quad-Core with 1 Month of Programmer Effort:
A Case Study with BerkeleyDB

Todd C. Mowry
Carnegie Mellon University

(in collaboration with Chris Colohan, J. Gregory Steffan, and Anastasia Ailamaki)

What Have I Worked On in the Past?

- Automatically extracting thread-level parallelism
- Smarter caching to better utilize deep memory hierarchies
  - SRAM to DRAM; DRAM to disk; local disk to remote web server
- Redesigning core database algorithms & data structures
  - to exploit modern processor architectures

What Am I Working on Now?

- Log-Based Architectures Project
  - **Motivation**: detect & fix software correctness problems in real time
  - **Approach**: logging mechanism allows cores to monitor other cores

- Claytronics Project
Today’s Talk

- Chris Colohan’s Ph.D. thesis work

Multicore is Here

- Quad-cores are now common
  - 8, 16, 32... cores expected in the future
  - Great for throughput, but what about latency?

Exploiting Multicore

One view:
- Don’t worry: everyone will write parallel software from now on
  - and it will all speed up nicely

Rebuttal:
- Writing parallel software is difficult
- Getting large speedups is also difficult
- What about legacy codes?

Another view:
- Don’t worry: the compiler will automatically parallelize everything
  - and it will all speed up nicely

Rebuttal:
- Beyond regular matrix-based codes, compilers really struggle with this
- Ambiguous dependences are a stumbling block
The Stampede Project @ CMU

Idea:
- Using novel hardware & compiler support, allow the compiler to optimistically create parallel threads
  - "Thread-Level Speculation" (TLS)
  - Rollback and recover if speculation fails

Our early work:
- Automatically parallelize SPEC Integer benchmarks
  - Resulted in speedups of roughly 20-35%

This work:
- Focus on large, legacy code that is hard to parallelize
  - "semi-automatic" approach: the programmer is involved

Case Study: BerkeleyDB

- We chose to parallelize individual transactions in BerkeleyDB
- The code was not written to support parallelism
  - Much the opposite: it takes advantage of the fact that there is never concurrency within a given transaction
- Rewriting the code to support intra-transaction parallelism would be extremely painful
  - Problems throughout the 200K lines of code
  - Would probably need to start over again from scratch

Transactions on Multi-Core

Cores can run concurrent transactions and improve throughput

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

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) makes parallelization easier.

Thread Level Speculation (TLS)

- Epoch 1
  - Sequential: $p = q$
- Epoch 2
  - Parallel: $p = q$
Thread Level Speculation (TLS)

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

Data dependences limit performance.

Violations as a Feedback Signal

Must Make Faster

TLS in Database Systems

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

Non-Database TLS

Concurrent transactions

TLS in Database Systems

Incrementally Parallelizing Transactions via TLS
Todd C. Mowry & Chris Colohan
Eliminating Violations

All-or-nothing execution makes optimization harder

Optimization may make slower?

Eliminate \( *p \) Dep.

Tolerating Violations: Sub-threads

Eliminate \( *p \) Dep. Sub-threads

Sub-threads

- Periodic checkpoints of a speculative thread
- Makes TLS work well with:
  - Large speculative threads
  - Unpredictable frequent dependences

Speed up database transaction response time by a factor of 1.9 to 2.9.

A Coordinated Effort

Transactions
- TPC-C

DBMS
- BerkeleyDB

Hardware
- Simulated machine
What's New

- Intra-transaction parallelism
  - Without changing the transactions
  - With minor changes to the DBMS
  - Without having to worry about locking
  - Without introducing concurrency bugs
  - With good performance

- Halve transaction latency on four cores

Outline

- Modifying the DBMS to exploit TLS
  - Dividing transactions into epochs
  - Removing bottlenecks in the DBMS

- 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)
Case Study: New Order (TPC-C)

```sql
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;
}
```

Outline

- Modifying the DBMS to exploit TLS
  - Dividing transactions into epochs
  - Removing bottlenecks in the DBMS
- Results
- Conclusions

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

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;
}
```

TLS ensures first epoch gets page first. Who cares?

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(); -> No violations while calling get_page()
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;
}
```

Incrementally Parallelizing Transactions via TLS
Todd C. Mowry & Chris Colohan
### 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,
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

- **Delay** operations 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

- Modifying the DBMS to exploit TLS
- Dividing transactions into epochs
- Removing bottlenecks in the DBMS

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

- This process took Chris 30 days and <1200 lines of code.
Other TPC-C Transactions

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

Conclusions

- A new form of parallelism for databases
- Tool for attacking transaction latency
- Intra-transaction parallelism
  - Without major changes to DBMS
- TLS can be applied to more than transactions
- Halve transaction latency by using 4 CPUs

Final Thoughts

- We achieved respectable speedups:
  - On a large piece of software that was written without parallelism in mind
  - With roughly a month of (non-expert) programmer effort
- To do this, we need TLS support plus:
  - Feedback on which instruction pairs cause dependence violations
  - Sub-thread support to minimize cost of failed speculation
- There is hope for large dusty-deck codes!!!