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)

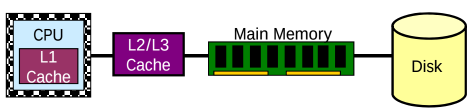
Twofold Speedup on a Quad-Core with 1 Month of Programmer Effort: A Case Study with BerkeleyDB

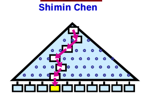
Todd C. Mowry
Carnegie Mellon University

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

What Have I Worked On in the Past?

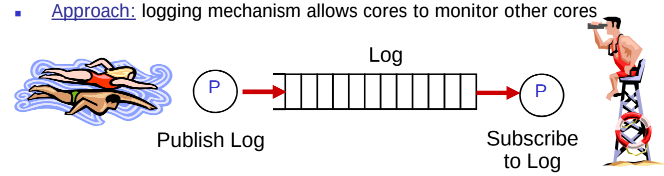
- Automatically extracting thread-level parallelism
- Smarter caching to better utilize deep memory hierarchies
  - SRAM to DRAM; DRAM to disk; local disk to remote web server
- Redesigning core database algorithms & data structures
  - to exploit modern processor architectures





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

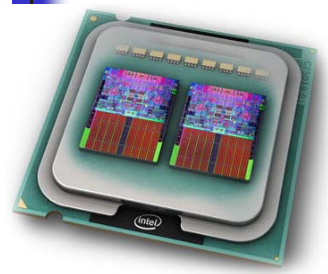


- Claytronics Project

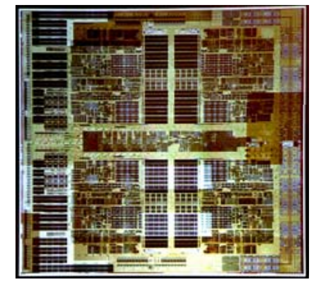
Today's Talk

- Chris Colohan's Ph.D. thesis work

Multicore is Here



Intel's Core 2 Quad



AMD's Quad-Core Opteron ("Barcelona")

- 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?

Exploiting Multicore

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%

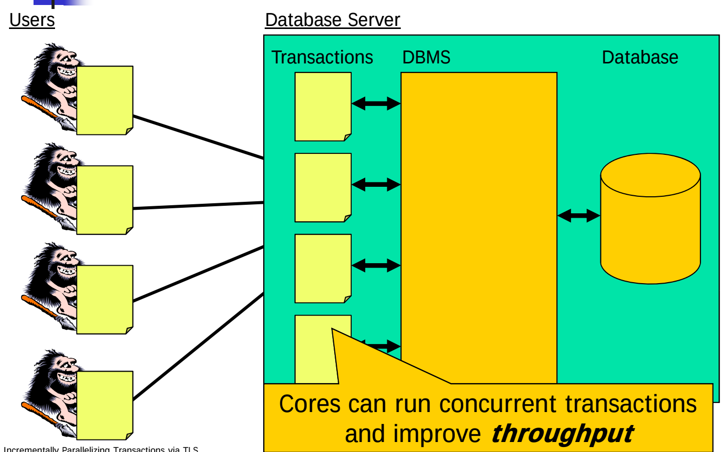
This work:

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

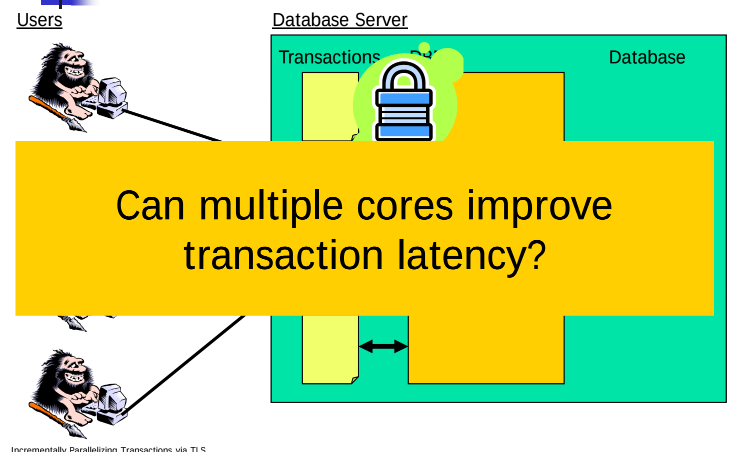
Case Study: BerkeleyDB

- 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

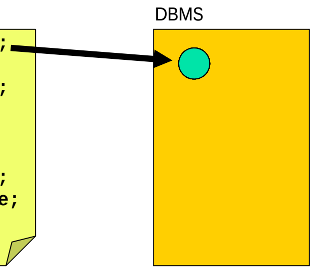


Multi-Core Enhances Throughput



Parallelizing transactions

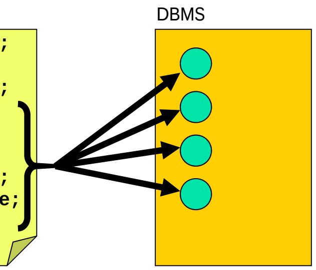
```
SELECT cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
    GET quantity FROM stock;
    quantity--;
    UPDATE stock WITH quantity;
    INSERT item INTO order_line;
}
```



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

Parallelizing transactions

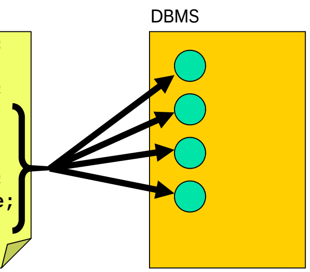
```
SELECT cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
foreach(item) {
    GET quantity FROM stock;
    quantity--;
    UPDATE stock WITH quantity;
    INSERT item INTO order_line;
}
```



- 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…

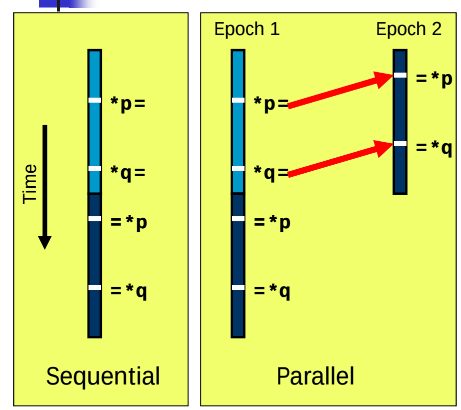
Parallelizing transactions

```
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;
}
```

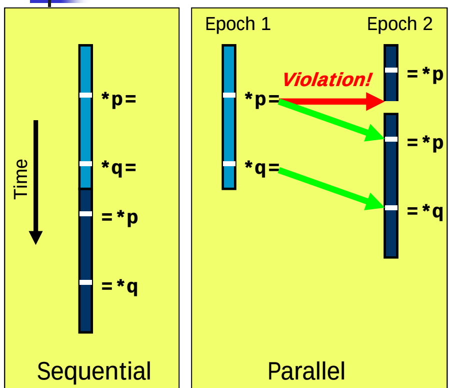


Thread Level Speculation (TLS) makes parallelization easier.

Thread Level Speculation (TLS)



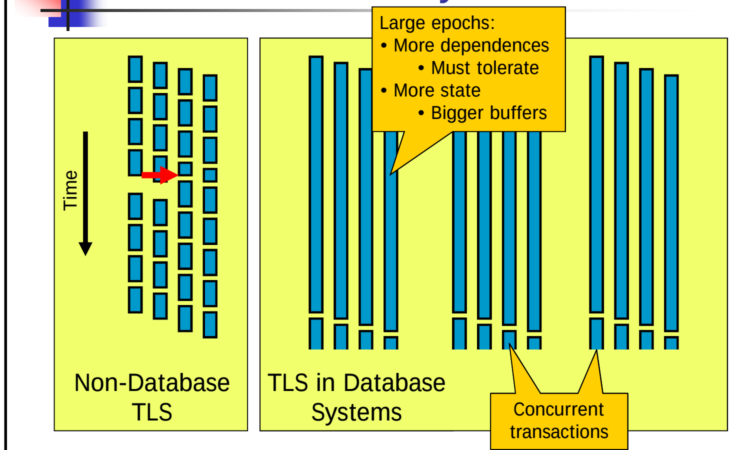
Thread Level Speculation (TLS)



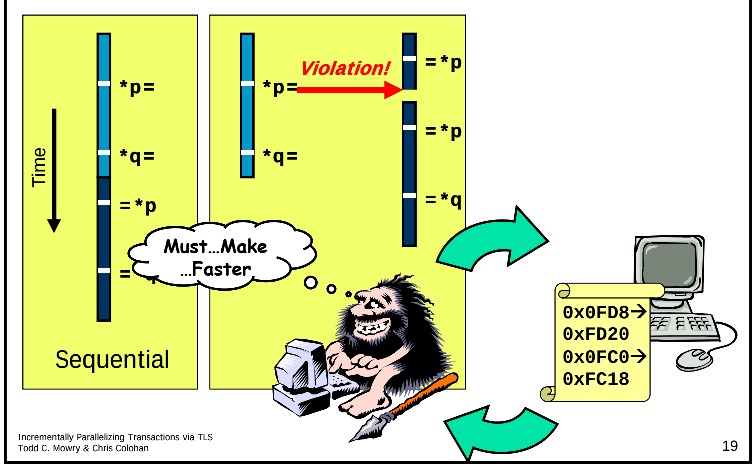
- Use *epochs*
- *Detect* violations
- *Restart* to recover
- *Buffer* state
- Worst case:
  - Sequential
- Best case:
  - Fully parallel

Data dependences limit performance.

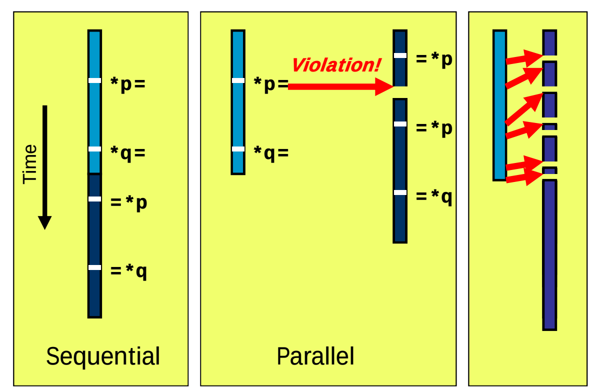
TLS in Database Systems



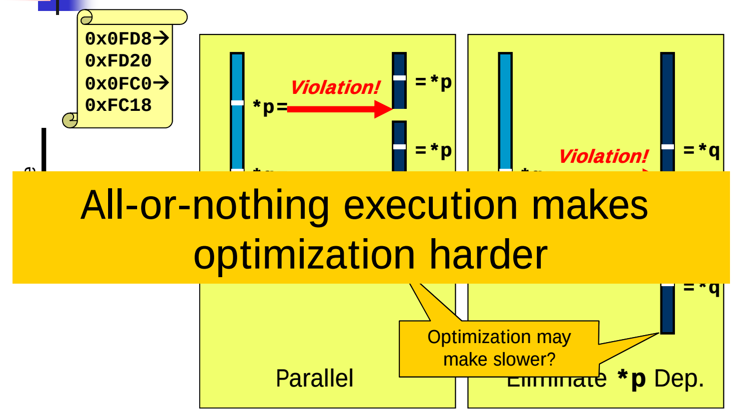
Violations as a Feedback Signal



Violations as a Feedback Signal

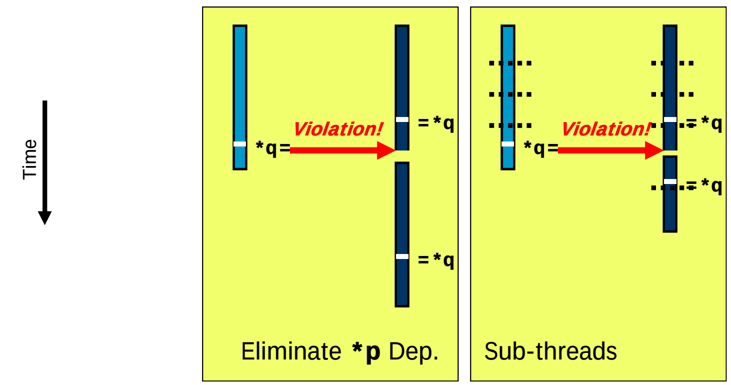


Eliminating Violations



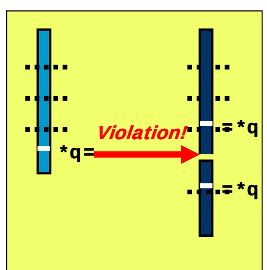
All-or-nothing execution makes optimization harder

Tolerating Violations: Sub-threads



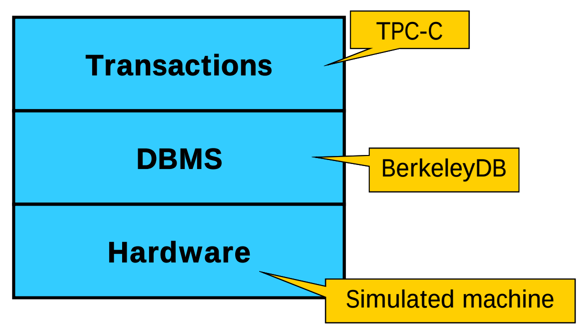
Sub-threads

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

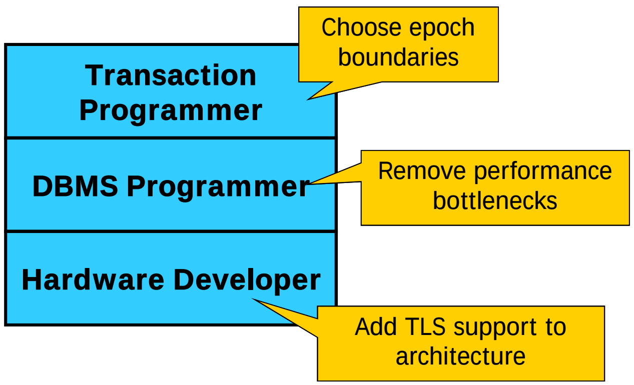


Speed up database transaction response time by a factor of *1.9 to 2.9.*

A Coordinated Effort



A Coordinated Effort

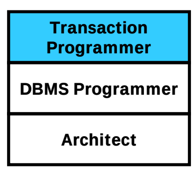


What's New

- 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;
}
```

78% of transaction execution time

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

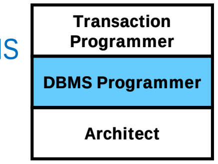
Case Study: New Order (TPC-C)

```
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;
}
```

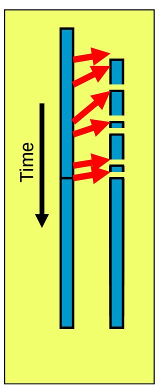
```
GET cust_info FROM customer;
UPDATE district WITH order_id;
INSERT order_id INTO new_order;
TLS_foreach(item) {
    GET quantity FROM stock
        WHERE i_id=item;
    UPDATE stock WITH quantity-1
        WHERE i_id=item;
    INSERT item INTO order_line;
}
```


Outline

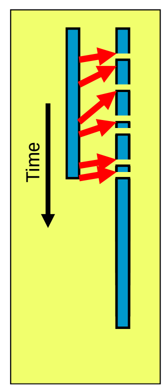
- Modifying the DBMS to exploit TLS
  - Dividing transactions into epochs
  - Removing bottlenecks in the DBMS
- Results
- Conclusions



Dependences in DBMS



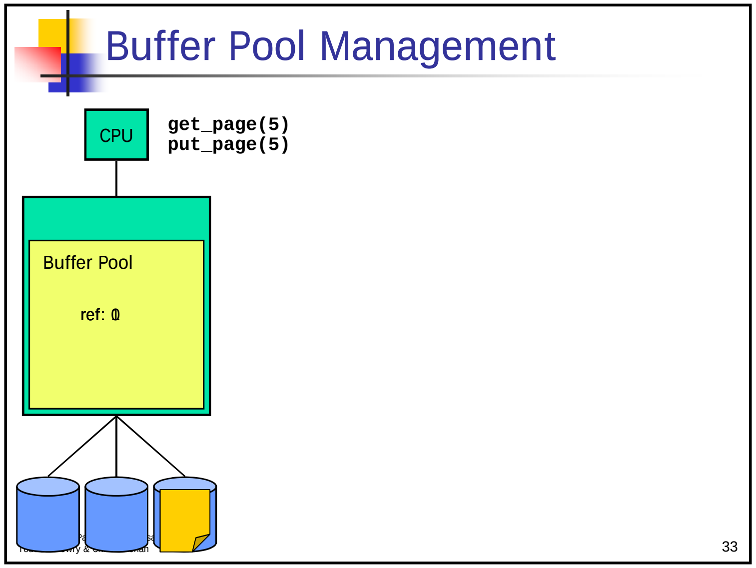
Dependences in DBMS

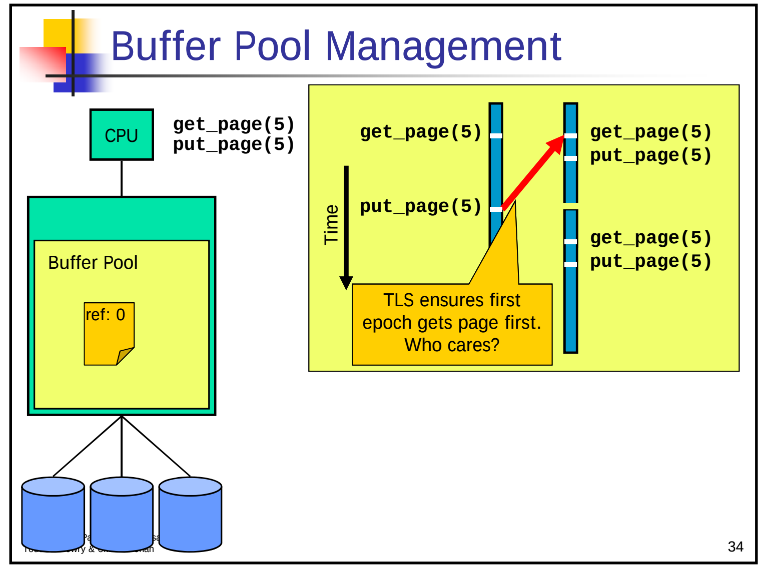


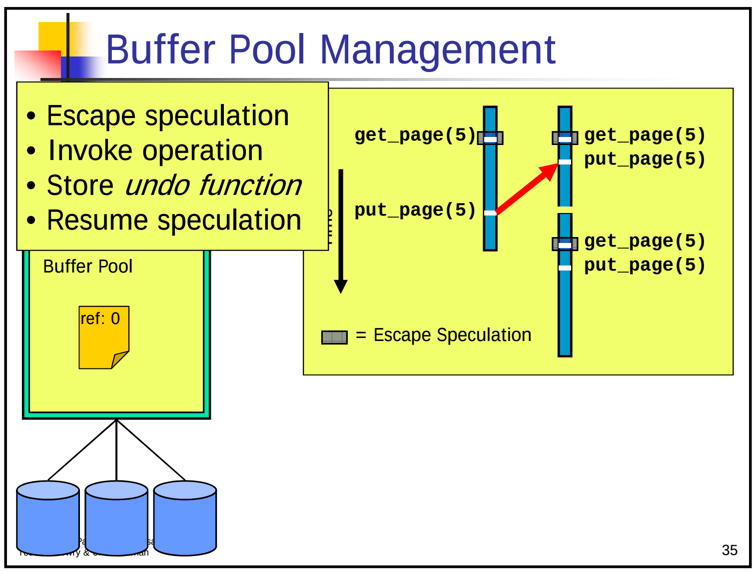
Dependences serialize execution!

Performance tuning:

- Profile execution
- Remove *bottleneck* dependence
- Repeat







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()**

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

36

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()`**

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!

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

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;
}
```

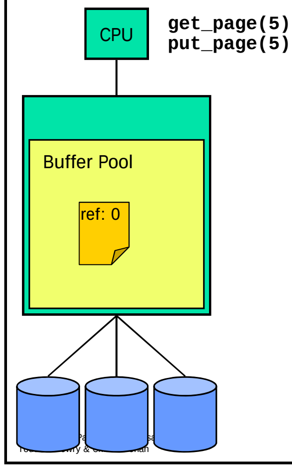
→ How to undo **`get_page()`**

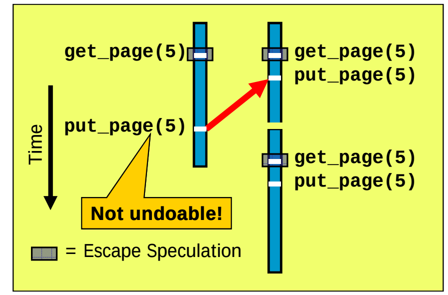
get_page() wrapper

```
page_t *get_page_wrappe
    static tls_mutex mut
    page_t *ret;

    tls_escape_speculat
    check_get_arguments(
    tls_acquire_mutex(&m

    ret = get_page(id);

    tls_release_mutex(&m
    tls_on_violation(put
    tls_resume_speculat

    return ret;
}
```

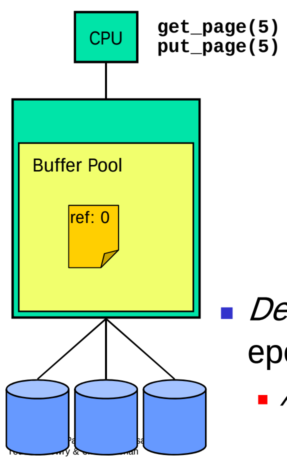
- Isolated
  - Undoing this operation does not cause cascading aborts
- Undoable
  - Easy way to return system to initial state
- Can also be used for:
  - Cursor management
  - **`malloc()`**

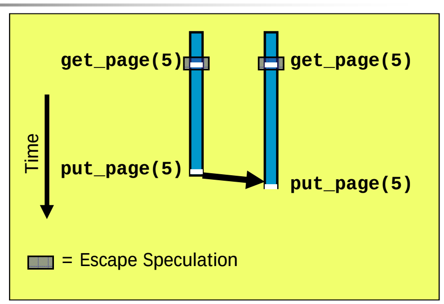
Buffer Pool Management





Buffer Pool Management





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

Removing Bottleneck Dependences

We introduce three techniques:

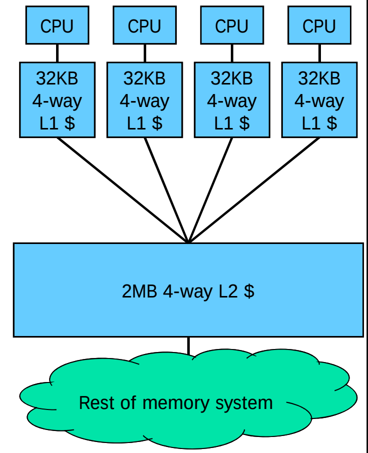
- **Delay operations** until non-speculative
  - Mutex and lock *acquire* and *release*
  - Buffer pool, memory, and cursor *release*
  - Log sequence number assignment
- **Escape speculation**
  - Buffer pool, memory, and cursor *allocation*
- **Traditional parallelization**
  - Memory allocation, cursor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

Outline

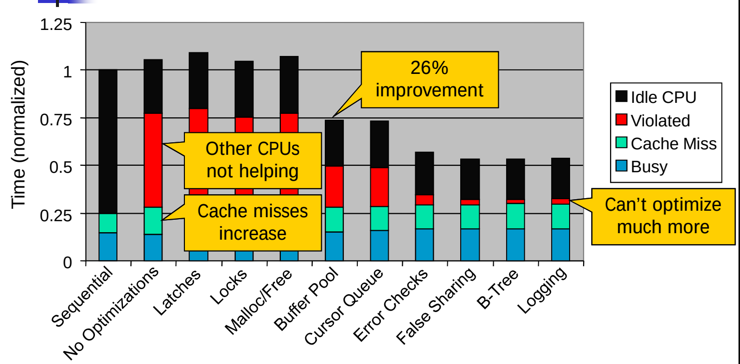
- Modifying the DBMS to exploit TLS
  - Dividing transactions into epochs
  - Removing bottlenecks in the DBMS
- Results
- Conclusions

Experimental Setup

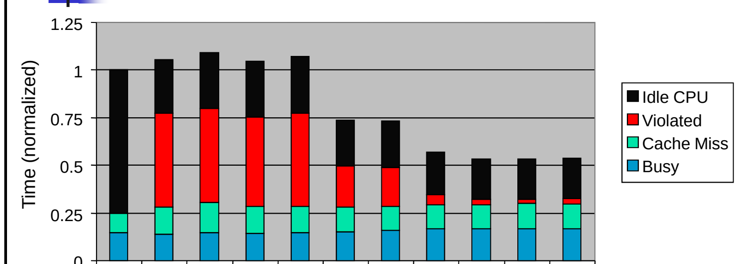
- Detailed simulation
  - Superscalar, out-of-order, 128 entry reorder buffer
  - Memory hierarchy modeled in detail
- TPC-C transactions on BerkeleyDB
  - In-core database
  - Single user
  - Single warehouse
  - Measure interval of 100 transactions
  - Measuring *latency* not throughput



Optimizing the DBMS: New Order

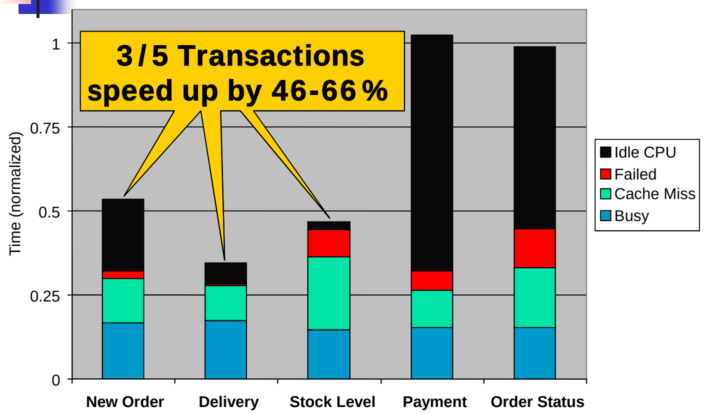


Optimizing the DBMS: New Order



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

Other TPC-C Transactions



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!!!