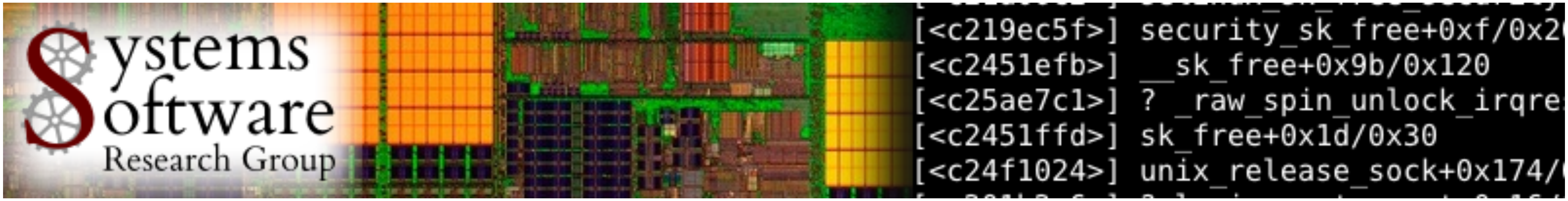


Improving Performance of Highly-Programmable Concurrent Applications by Leveraging Parallel Nesting and Weaker Isolation Levels

Thesis Defense—Master of Science

Duane Niles

Advisor: Binoy Ravindran

Systems Software Research Group

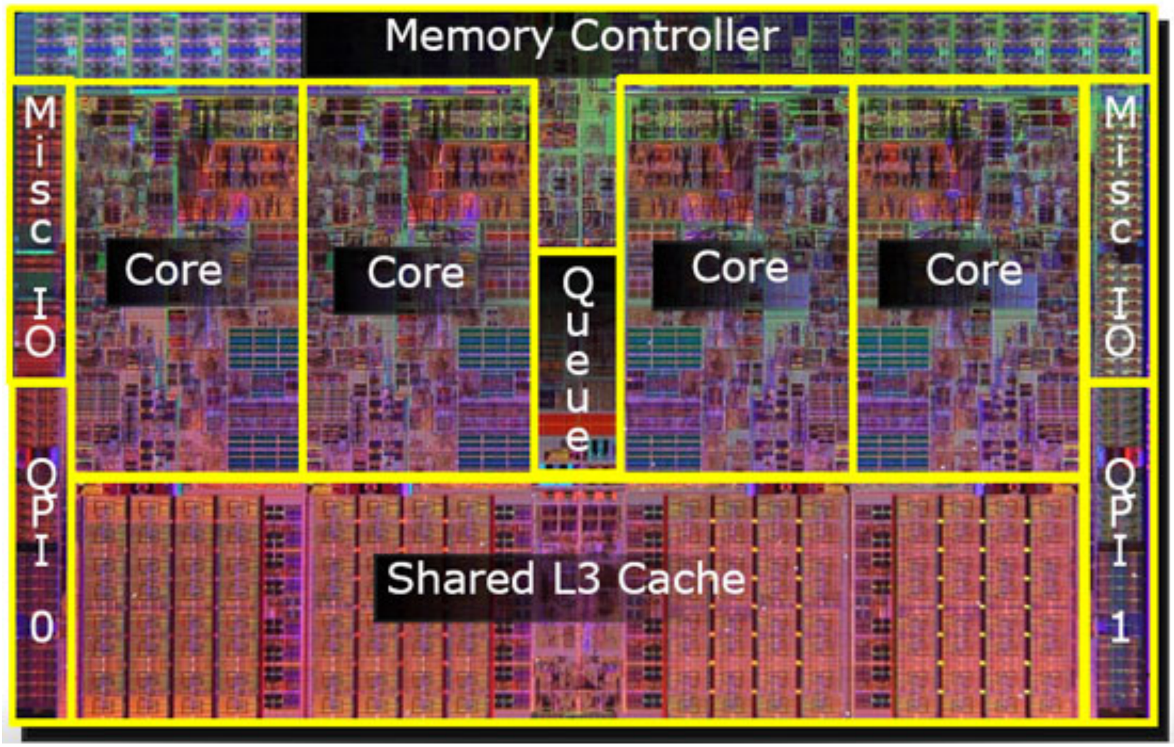
Virginia Tech

Overview

- Introduction
- Motivation
- Contributions
  - SPCN
  - AsR
- SPCN (Speculative Parallel Closed Nesting)
  - Strict
  - Relaxed
- Experimental Results
- Conclusions

Computer Hardware

- Multi-core architectures became focus after the turn of the century



Concurrent Programming

- New design paradigm — Parallelism
- Many approaches not designed for sharing data
  - E.g., MPI with separate, unique processes
- Require other forms to fully split one application
  - Most common: Lock-based Synchronization

Concurrency with Locks

- **Coarse-grained**: Simpler, but vastly inefficient
- **Fine-grained**: Great performance, difficult to program
- Challenging to compose without low-level information (e.g., deadlock, livelock, etc.)

```
public boolean add(int item) {
  Node pred, curr;
  lock.lock();
  try {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item)
      node.next = curr;
      pred.next = node;
      return true;
    }
  } finally {
    lock.unlock();
  }
}
```

```
public boolean add(int item) {
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.val < item) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Concurrency with Transactions

- Originated from database systems
- *Atomic* operation, speculative
- Transaction context holds the data
- Programmable like coarse-grained locking
- Aimed towards fine-grained locking's performance
- Easily composable — *Nesting*

```
public boolean add(int item) {
  Node pred, curr;
  atomic {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  }
}
```

Motivation for Transaction Research

Problems

- Trade-off between programmability and generality
- Unable to utilize internal program knowledge

Research Goals

- **Broad**: Enhance performance while keeping programmability the same
- **Thesis**: Two approaches – SPCN and AsR

Research Contributions

- **SPCN: Speculative Parallel Closed Nesting**
  - Composed transactions are typically sequential
  - Parallelization can allow internal conflicts
  - Automatic processing improves the performance
- **AsR: As-Serializable Transactions**
  - *Serializability*: ordered synchronization of transactions (as if they were sequentially operated)
  - Too strict of a requirement in many systems
  - Keep application serializable while detecting inconsistencies with meta-data; relax the system itself

Nested Transactions

```
atomic A {
    atomic B1 {
      write(x)
      read(y)
      . . .
    }
    atomic B2 {
      read(x)
      read(z)
      . . .
    }
    atomic B3 {
      write(y)
      write(z)
      commit()
    }
}
```

Nested Transactions

Parent

```
atomic A {
    atomic B1 {
     write(x)
     read(y)
     . . .
    }
    atomic B2 {
     read(x)
     read(z)
     . . .
    }
    atomic B3 {
     write(y)
     write(z)
     commit()
    }
}
```

Nested Transactions

Parent

Nested

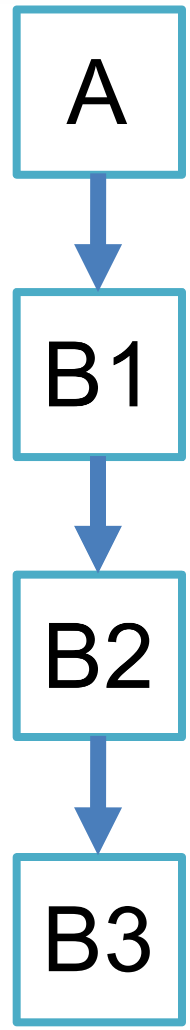
```
atomic A {
   atomic B1 {
    write(x)
    read(y)
    . . .
   }
   atomic B2 {
    read(x)
    read(z)
    . . .
   }
   atomic B3 {
    write(y)
    write(z)
    commit()
   }
}
```

Sequential Nesting

Parent

Nested

```
atomic A {
  atomic B1 {
   write(x)
   read(y)
   . . .
  }
  atomic B2 {
   read(x)
   read(z)
   . . .
  }
  atomic B3 {
   write(y)
   write(z)
   commit()
  }
}
```

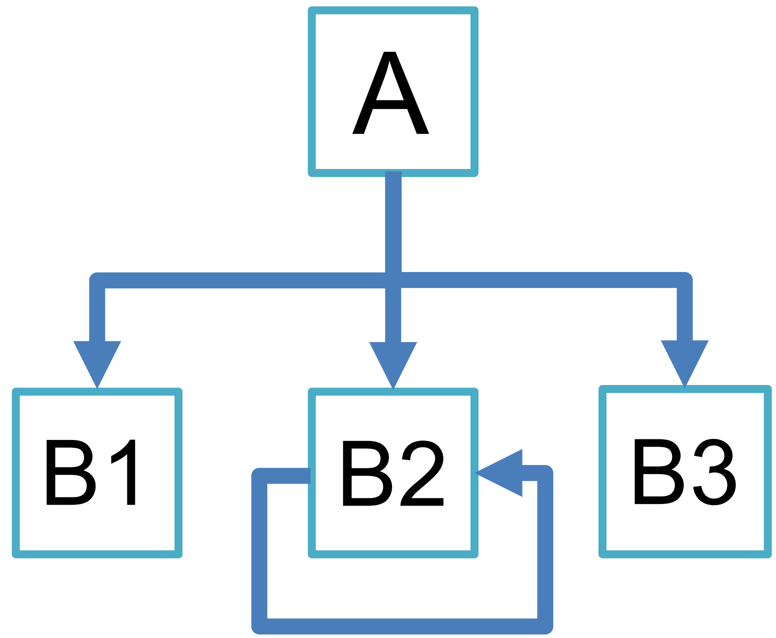


Sequential Nesting

- **Flat**: No proper nesting (single-level transaction)
- **Closed**: Transactions operate piece-by-piece (able to restart with some completed work)
- **Open**: Optimistic; nested transactions commit early—must be undone later if conflicting (using *abstract locks*)

Parallel Nesting

```
atomic A {
    atomic B1 {
      write(x)
      read(y)
      . . .
    }
    atomic B2 {
      read(x)
      read(z)
      . . .
    }
    atomic B3 {
      write(y)
      write(z)
      commit()
    }
}
```



SPCN: Speculative Parallel Closed Nesting

- Pessimism of closed nesting—no early commit
- Enforces order of operation
- Two versions
  - Strict: Hard boundary of commits; lighter processing
  - Relaxed: Out-of-order commits; more meta-data

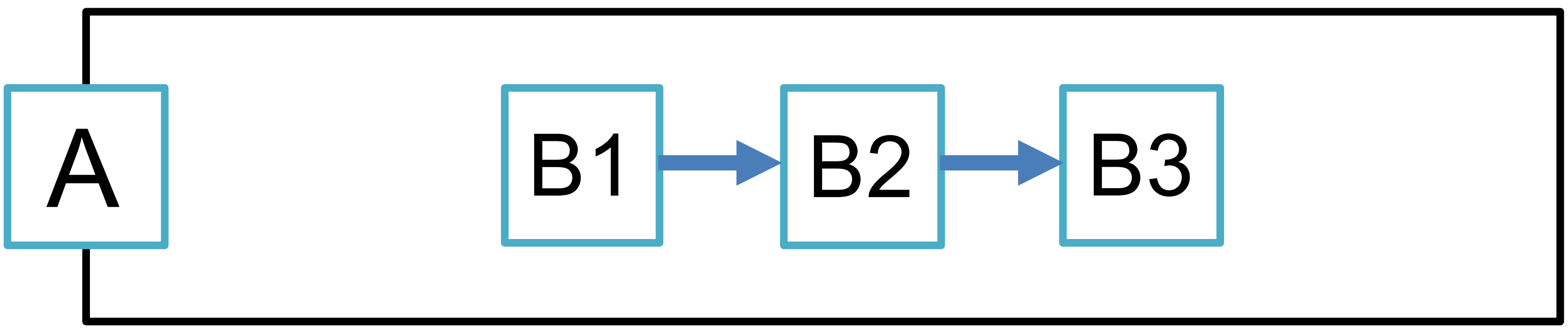
External Transaction Processing

- Store operations with **read-set** and **write-set**
- **Abort** if conflicts occur during locking or validation
- Validation utilized for correctness; varies per system (e.g., eager-locking, lazy validation, etc.)
- Correct validation allows **commit**
  - Make updates public and release locks
- Different contention schemes process conflicts in other manners

SPCN Strict

- Total order on nested transactions
- *Futures*: Scala primitive to allocate sub-transactions
- Validation performed after all previous siblings
- **Write-After-Read**: Conflict of sibling transactions

A (the root) begins all transactions, and A finishes all of them.



SPCN Strict

Parent

Nested

```
atomic A {
  atomic B1 {
    write(x)
    read(y)
    . . .
  }
  atomic B2 {
    read(x)
    read(z)
    . . .
  }
  atomic B3 {
    write(y)
    write(z)
    commit()
  }
}
```

Order of Operation

- All sub-transactions start
- B1 commits *(no errors)*
- B2 detects conflict—aborts, restarts *(immediate commit)*
- B3 commits *(no errors)*

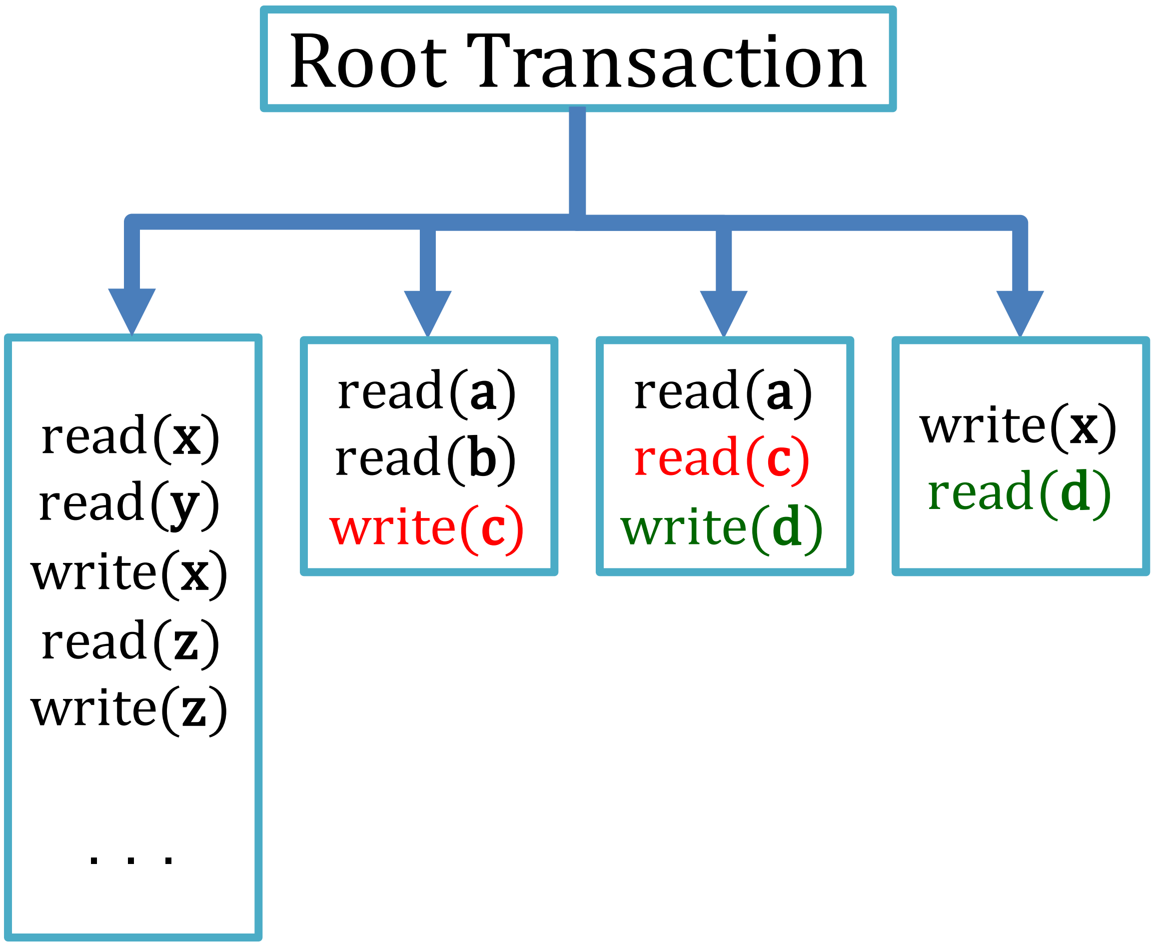
SPCN Strict – Good Example

```
for (k <- 1 to lines.length) {
  atomic { implicit txn =>
    // Parse order line.
    val ol = lines(k)
    val item = Hyflow.dir.open[TpccItem](Name.I(ol))

    // Get item info.
    val I_PRICE = item.I_PRICE()
    val I_NAME = item.I_NAME()
    val I_DATA = item.I_DATA()

    // Get stock info.
    . . .
  }
}
```

	RS	WS	Prev
T_1	Item 1	*Empty*	*Empty*
T_2	Item 2	*Empty*	T_1
⋮			
T_N	Item N	*Empty*	T_{N-1}

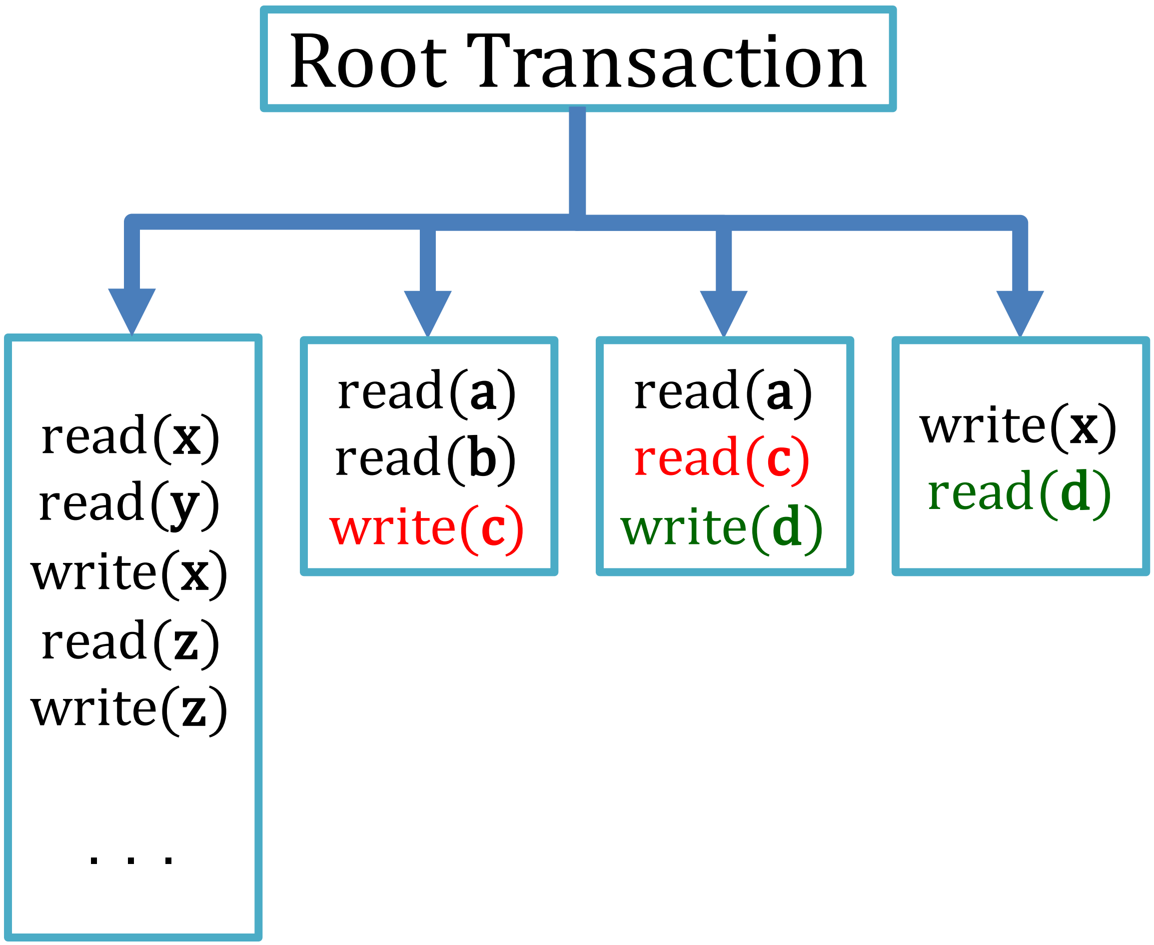
- Transactions used to create parts of an order—easily split the work

SPCN Strict – Bad Example



- **Strict** delays the short sub-transactions
- Conflicts are resolved slowly after the first sub-transaction

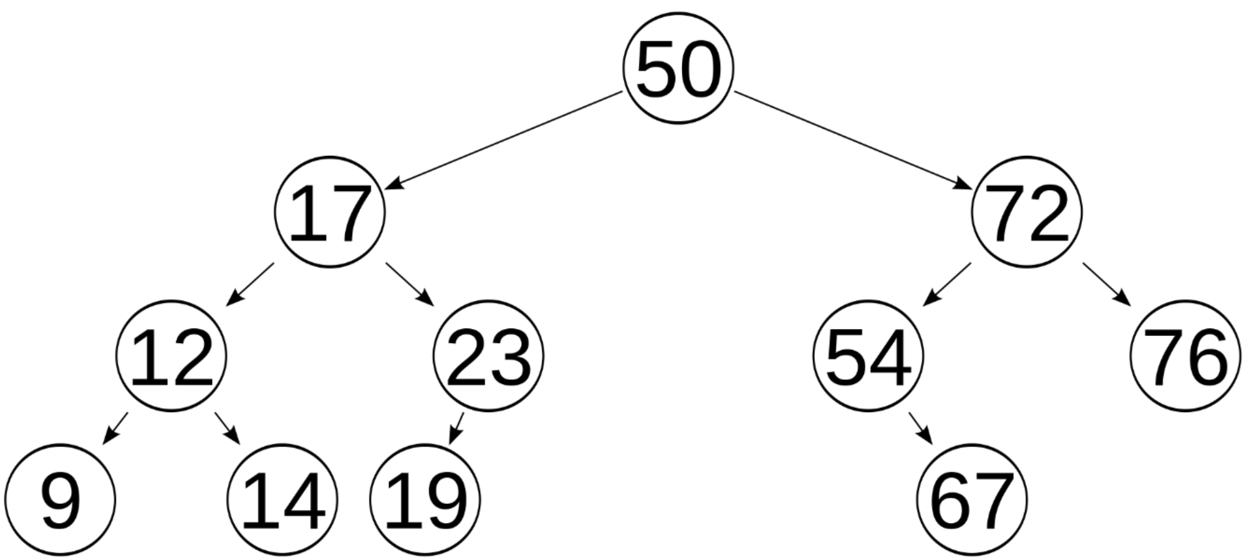
SPCN *Relaxed* – *Good* Example



- Allow them to work in a different order
- **Relaxed** can process the later transactions while the first runs

SPCN Relaxed

- Allows early completion (after validation)
- Requires multi-versioned data
- **ReadHash:** Track visible reads of sub-transactions
- **VerTree:** Track multiple versions via AVL Tree



SPCN Relaxed

Parent | Nested

```
atomic A {
  atomic B1 {
    write(x)
    read(y)
    …
  }
  atomic B2 {
    read(x)
    read(z)
    …
  }
  atomic B3 {
    write(y)
    write(z)
    commit()
  }
}
```

Order of Operation

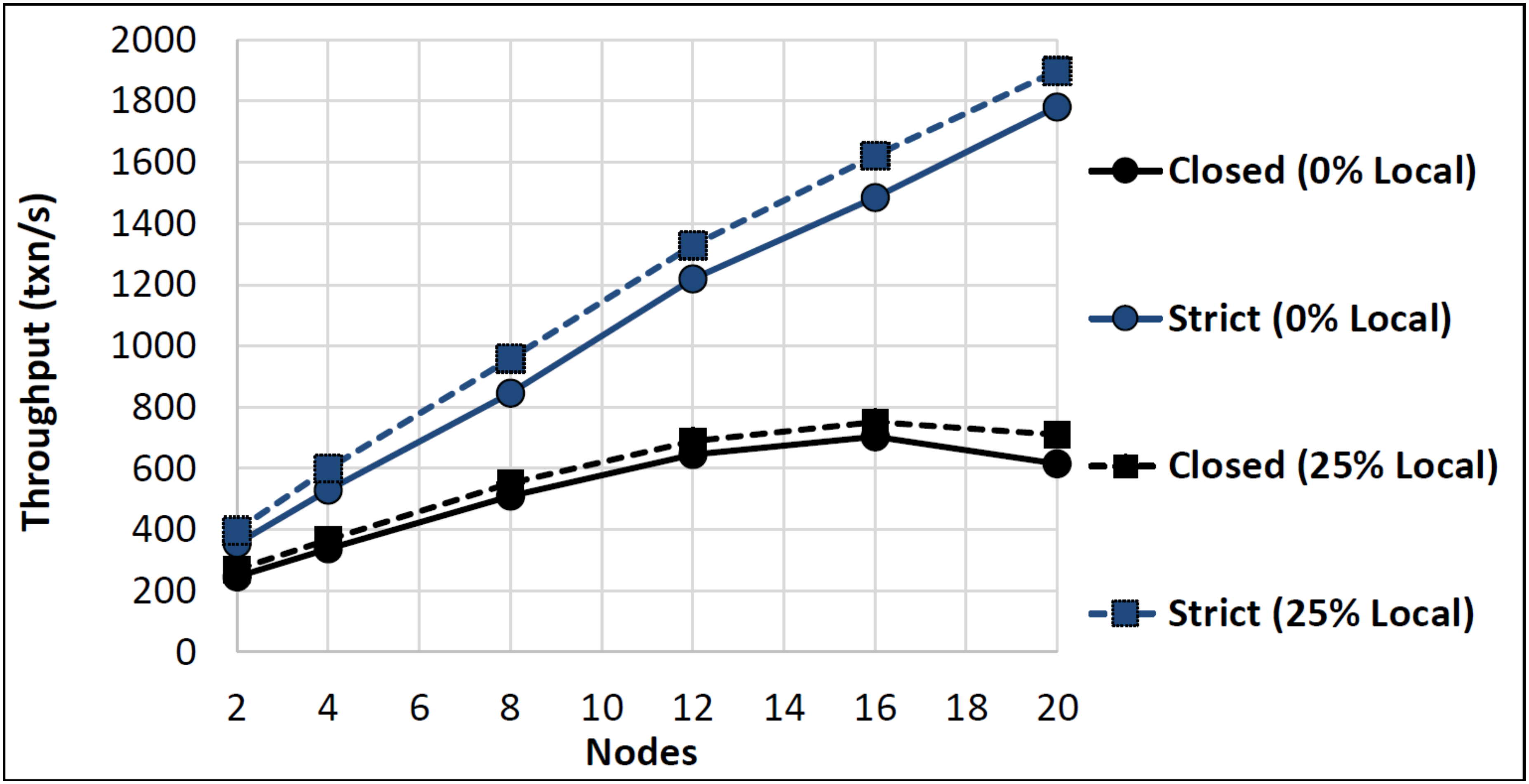
- All sub-transactions start
- Can commit in any order
- If B2 commits before B1:
  - B1 signals conflict
  - B2's data is removed
  - B2 is restarted
- B3 can commit with no problems

Experimental Results

- Amazon EC2 Cluster
- Up to 20 *c3.8xlarge* nodes
- Intel Xeon E5-2680 v2 (Ivy Bridge) processors
- 32 vCPU, 60 GB of memory

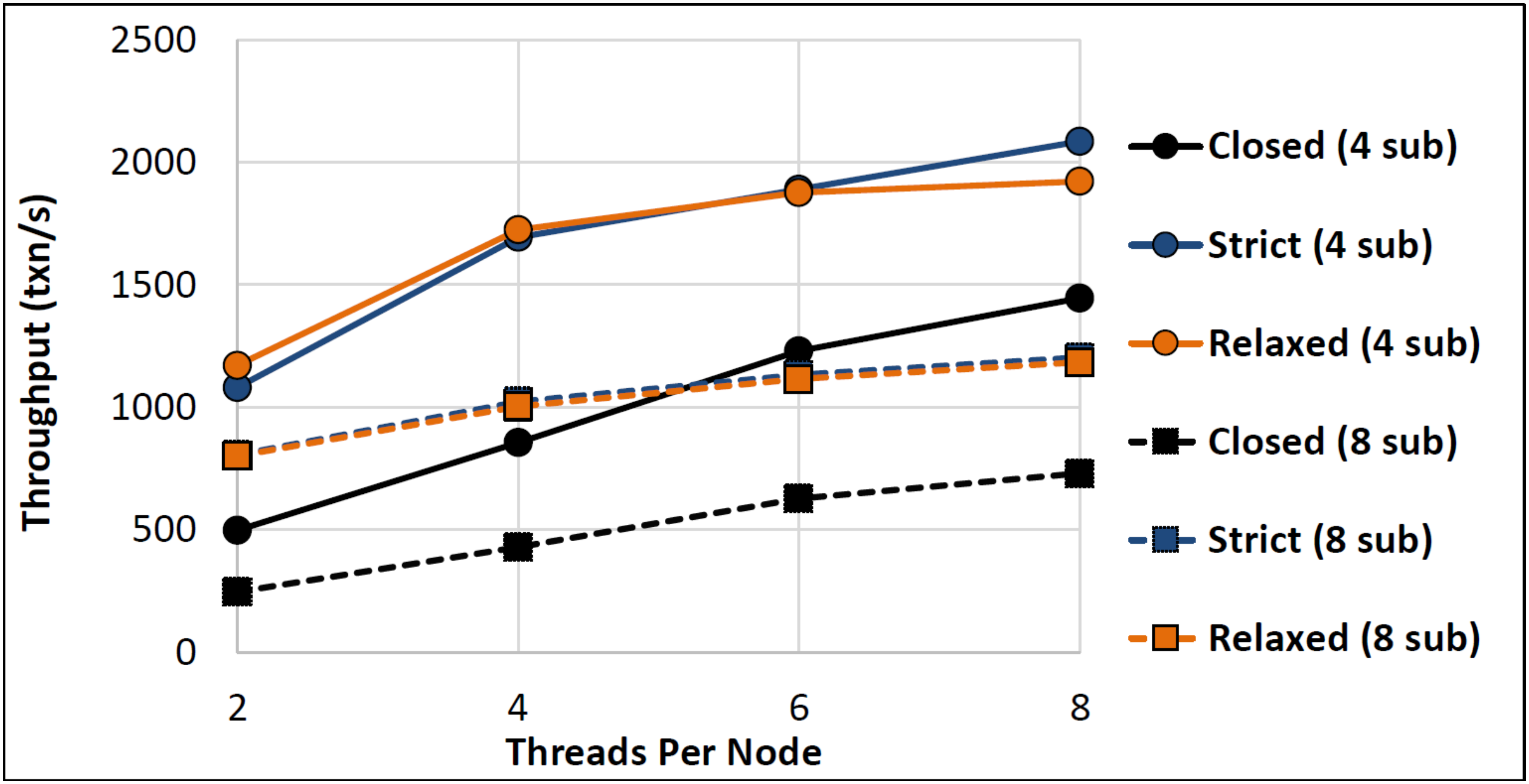
- **Benchmarks**: Bank, TPC-C, STMBench7, YCSB

TPC-C: Scalability



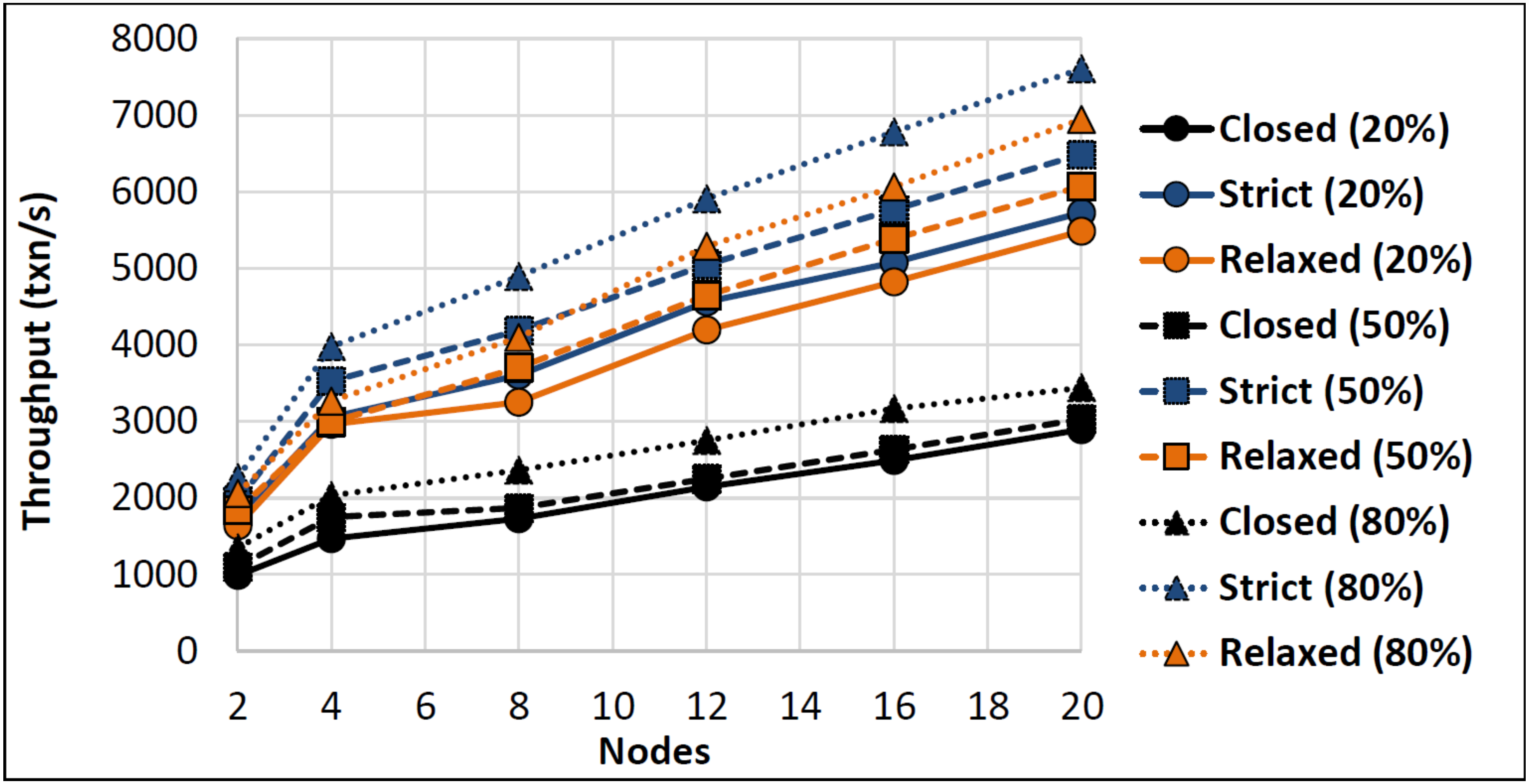
8 threads per node. Varying locality of operations.

TPC-C: Read-Only



10 nodes. Varying number of sub-transactions.

Bank: Scalability

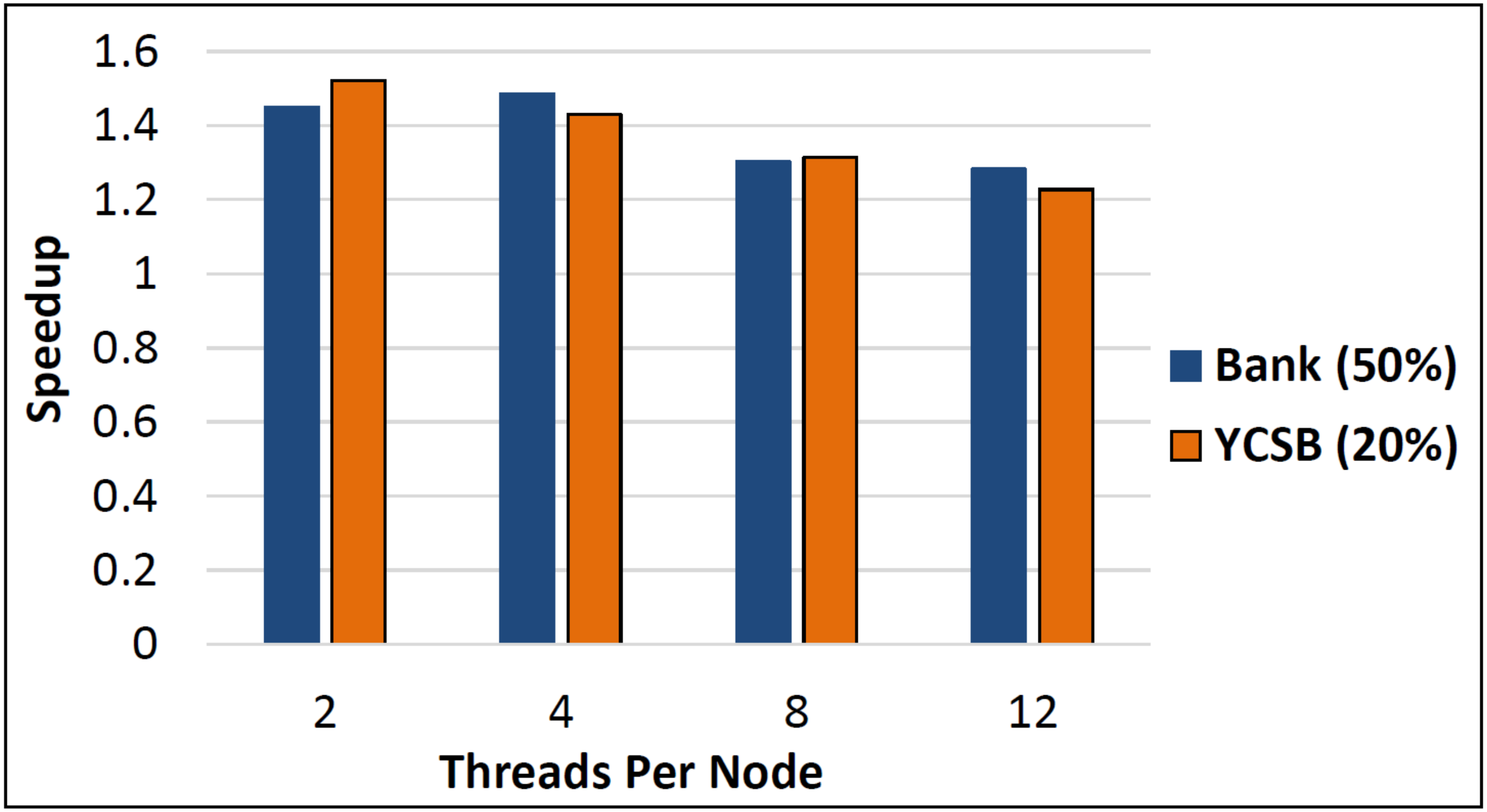


500k accounts. 8 threads per node. 8 operations per transaction.

Systems Software Research Group

VirginiaTech Invent the Future

Bank and YCSB: Contention



20 nodes.

Conclusions

- Contributions
  - SPCN
  - AsR
- Large performance increases
- Great accessibility for developers
- Improved parallelism for multi-core systems

Thank you for your time!
Any questions?