

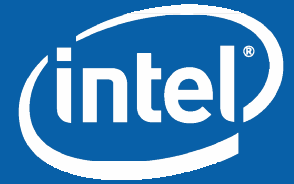
Agenda

Transactional Memory (TM)

- TM Introduction
- TM Implementation Overview
- Hardware TM Techniques
- Software TM Techniques



Q&A



Software Transactional Memory

Bratin Saha
Programming Systems Lab
Intel Corporation

Outline

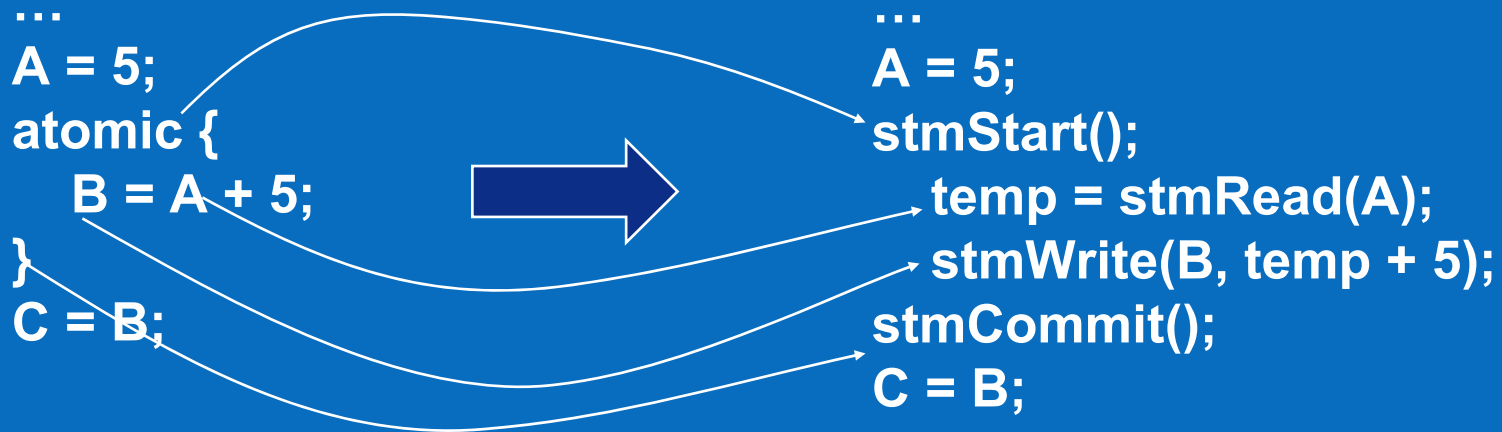
Software Transactional Memory

- Translating a language construct
- Runtime support
- Compiler support

Integrating with other system services

Open issues & conclusions

Compiling Atomic



Transactional memory accessed via STM read & write functions

- Compiler inserts appropriate calls
- Code generation, control flow, optimizations in later slides

STM tracks accesses & detects data conflicts

Runtime Data Structures

Per-data

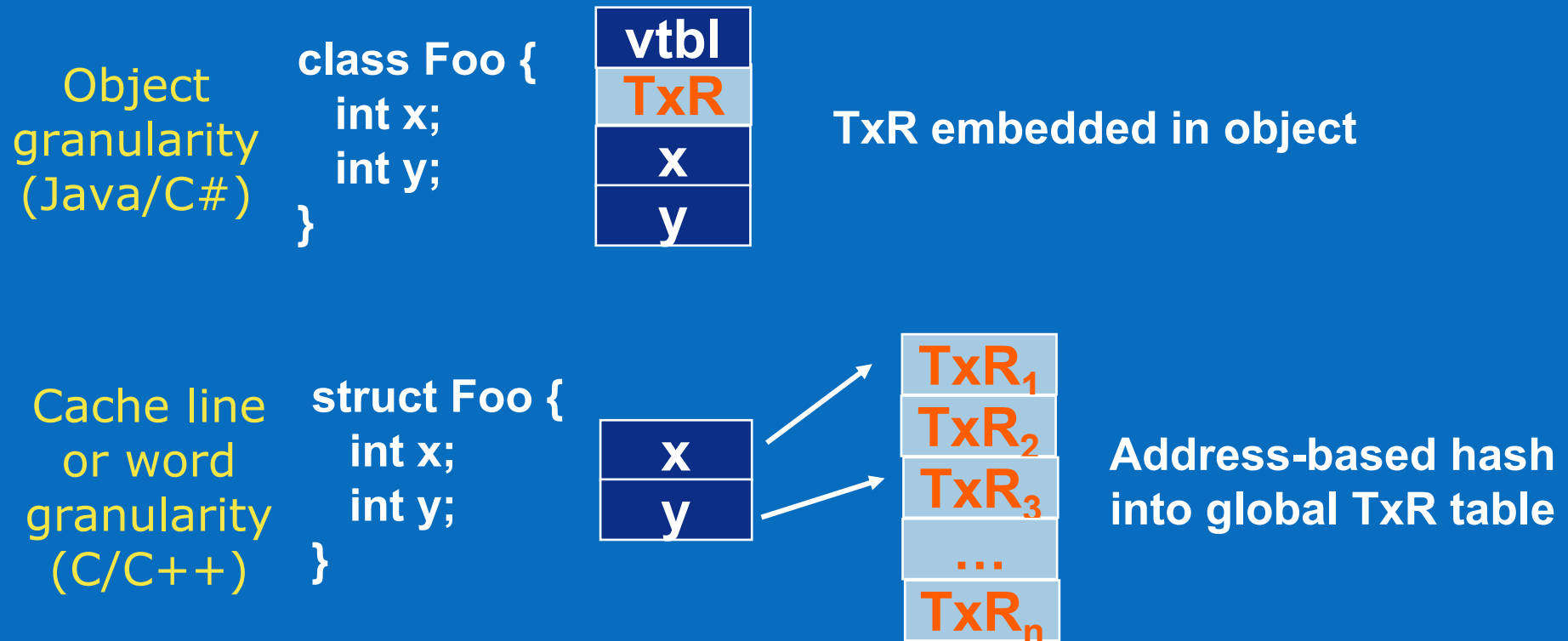
- Transaction Record (TxR)
 - Pointer-sized field guarding shared data
 - Track transactional state of data
 - Shared: Read-only access by multiple readers
 - Exclusive: write-only access by single owner

Per-thread

- Transaction Descriptor
 - Read set, write set, & log
 - For validation, commit, & rollback
- Transaction Memento
 - Checkpoint of transaction descriptor
 - For nesting & partial rollback

Mapping Data to Transaction Records

Every data item has an associated transaction record



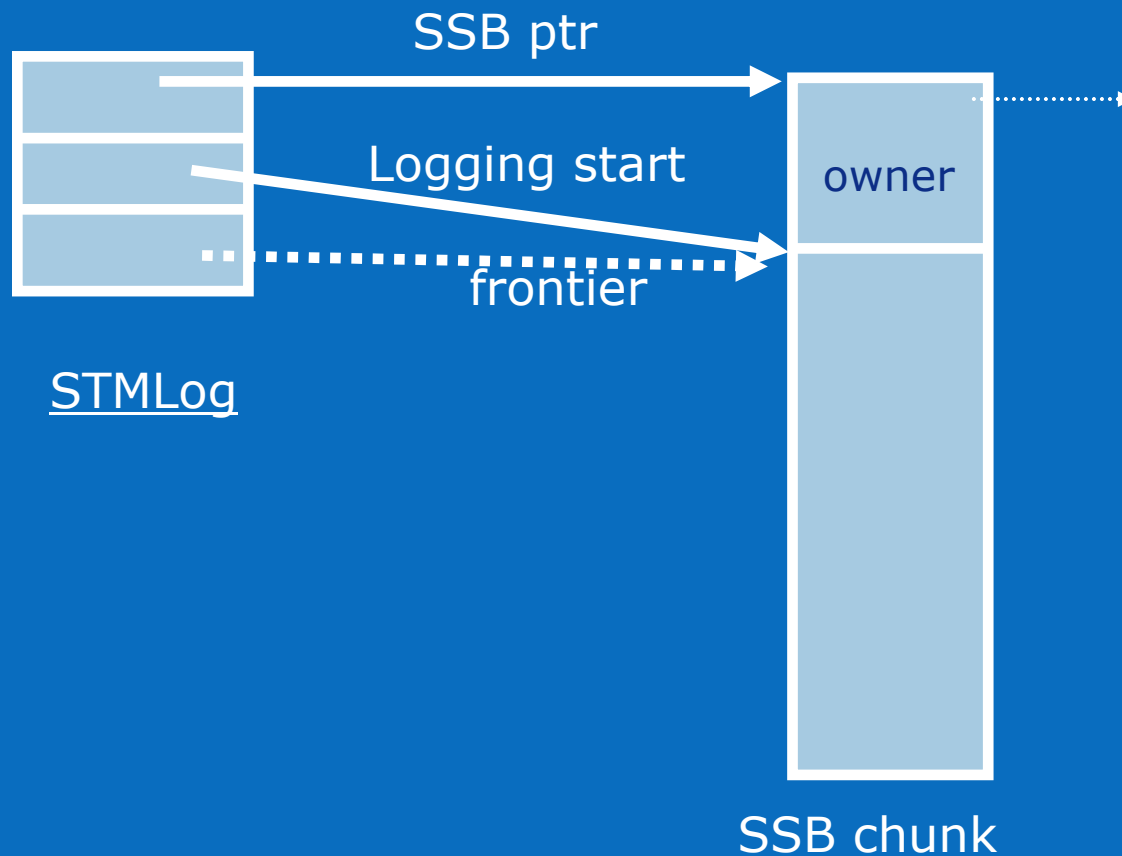
Transaction Descriptor

```
struct STMDescriptor {  
    STMState state;           /* state of transaction */  
    STMLog writeLock;         /* write locks acq */  
    STMLog readLock;          /* read versions acq */  
    STMLog writeLocations;     /* undo log */  
    /* other fields, for example, stats ... */  
};
```

Transaction descriptor stores transaction related info

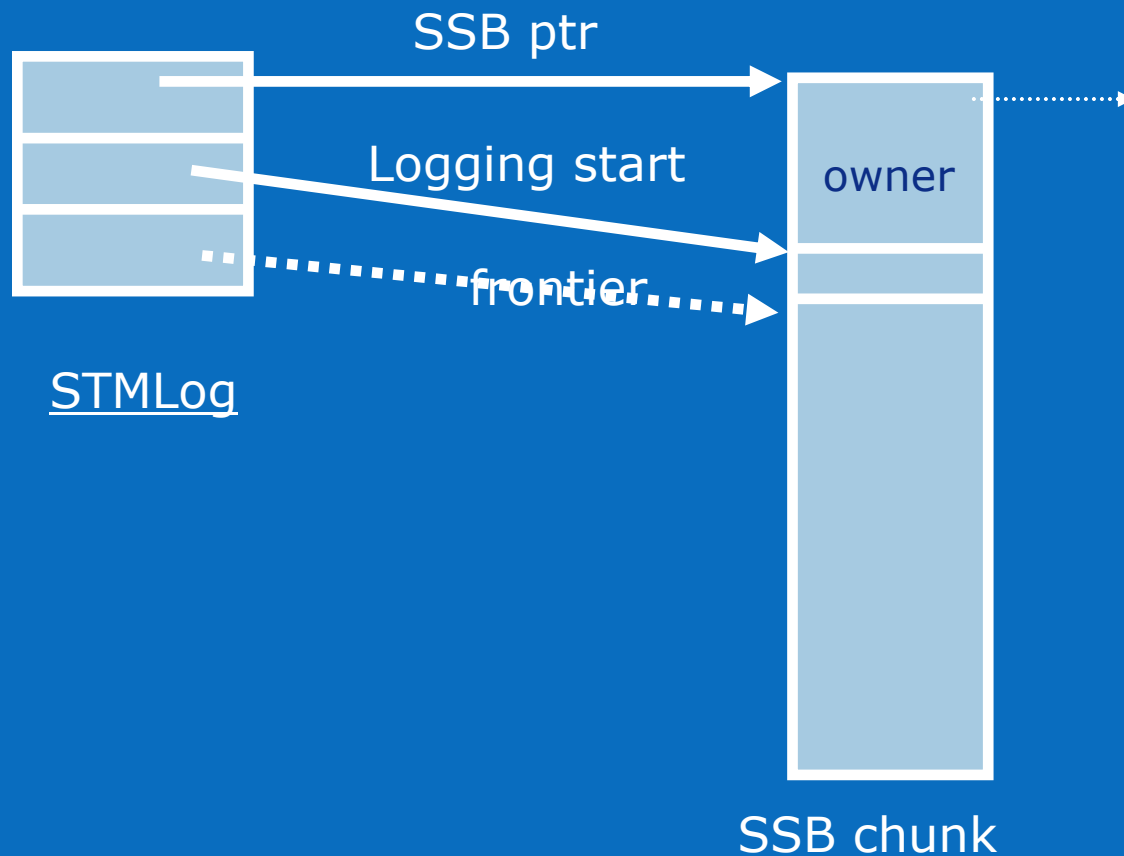
- Usually a thread local data structure

Transaction Log: An Efficient Design



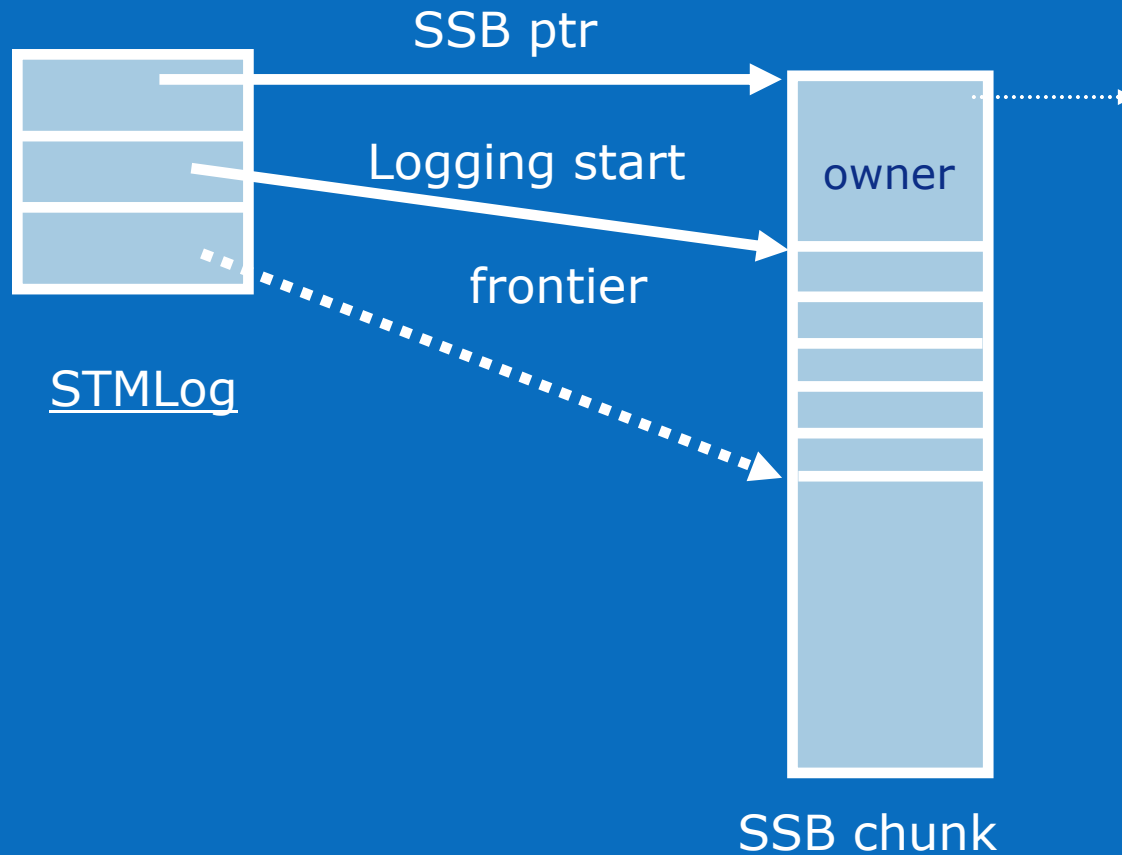
Allocate a chunk of space like a sequential store buffer
Aligned on a 2^k boundary

Transaction Log: An Efficient Design



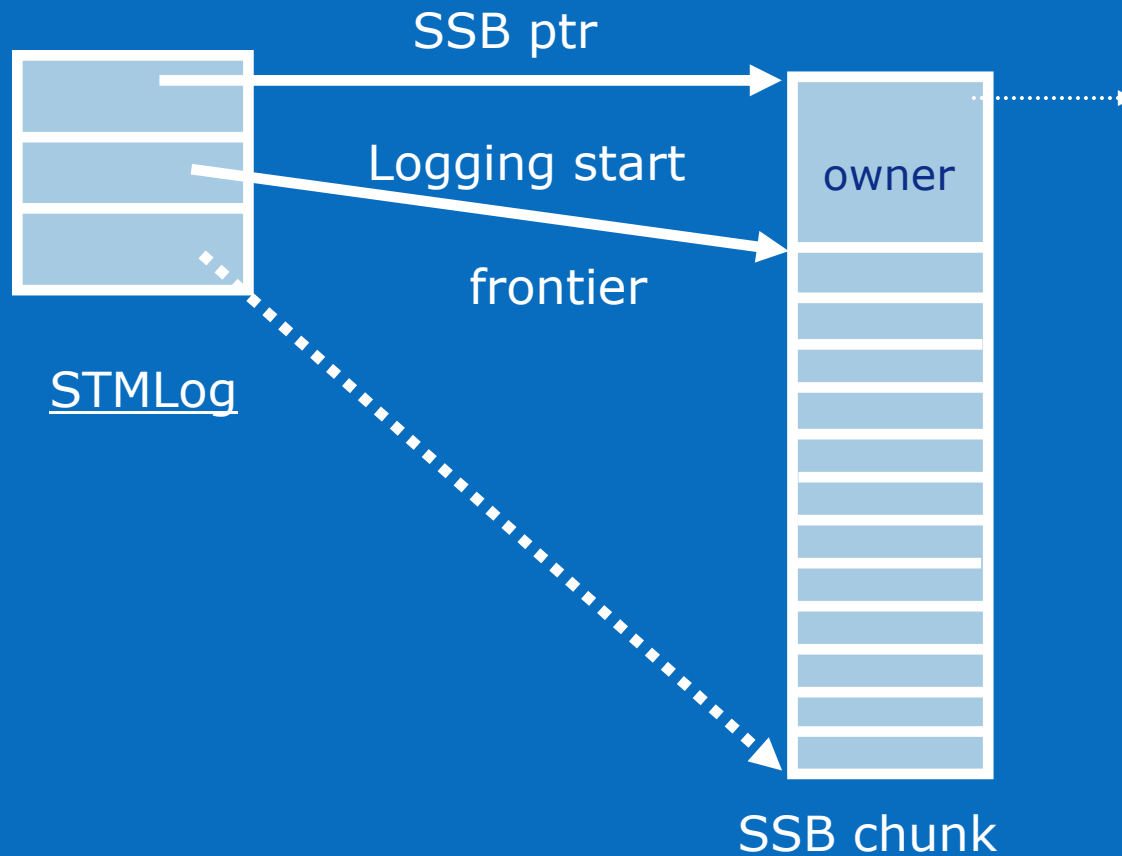
Bump the frontier when logging
Frontier can be in register, or accessed quickly off TLS

Transaction Log: An Efficient Design



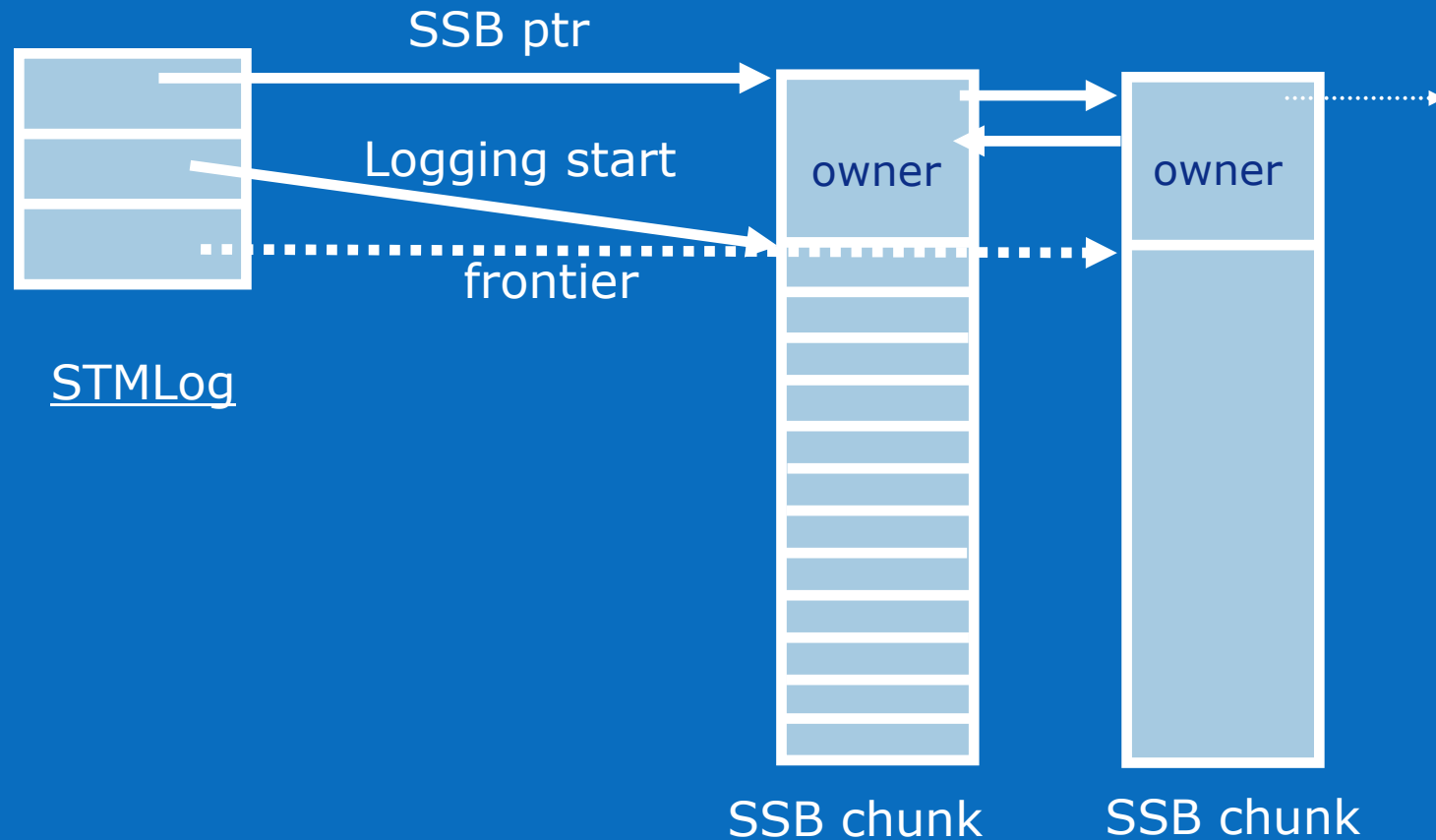
Fast logging sequence, 6 ISA instructions
Good ILP in the sequence

Transaction Log: An Efficient Design



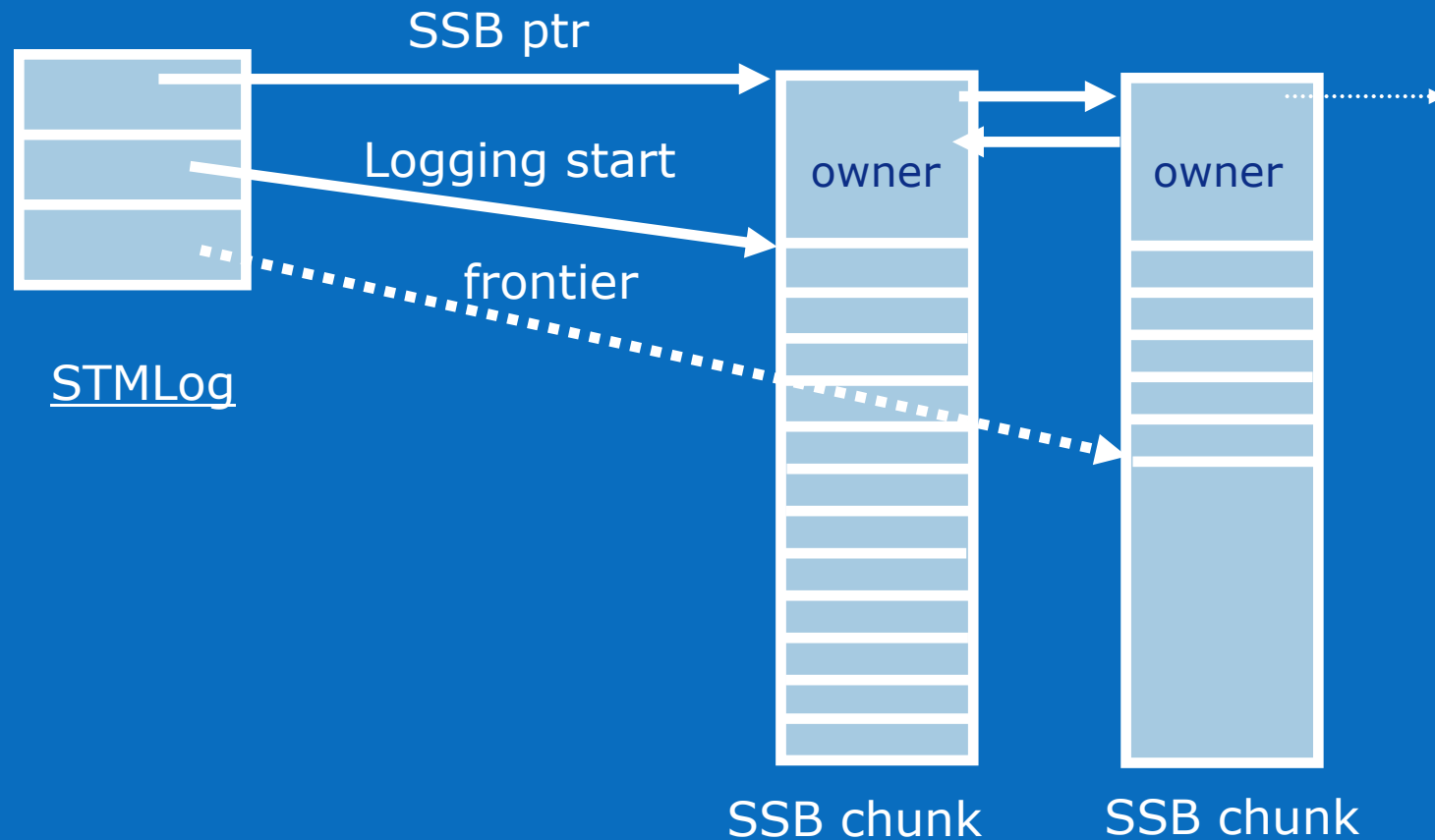
Overflow happens when the lower address bits are zero
Check is a single ISA instruction

Transaction Log: An Efficient Design



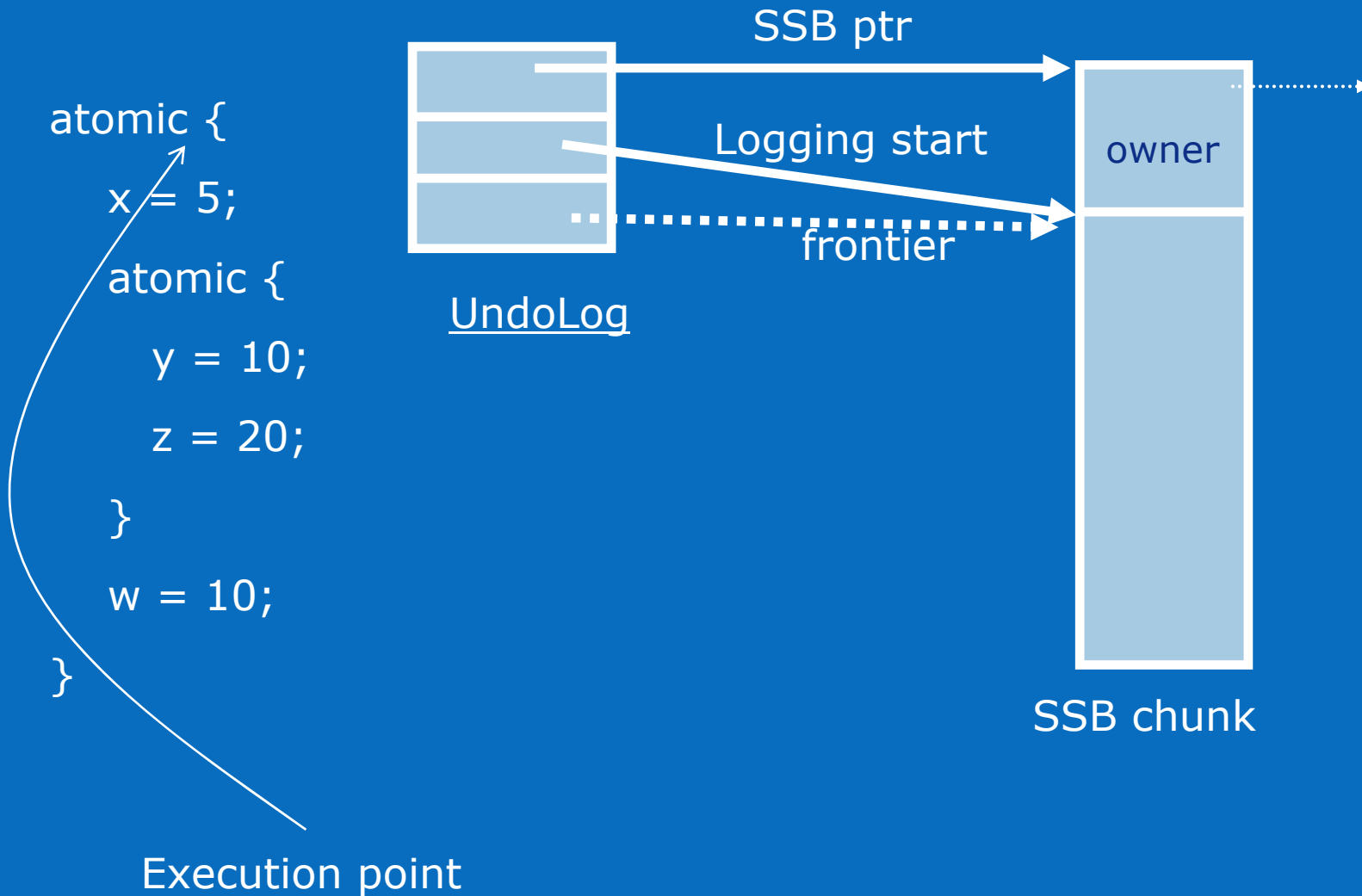
Allocate a similar chunk on overflow

Transaction Log: An Efficient Design

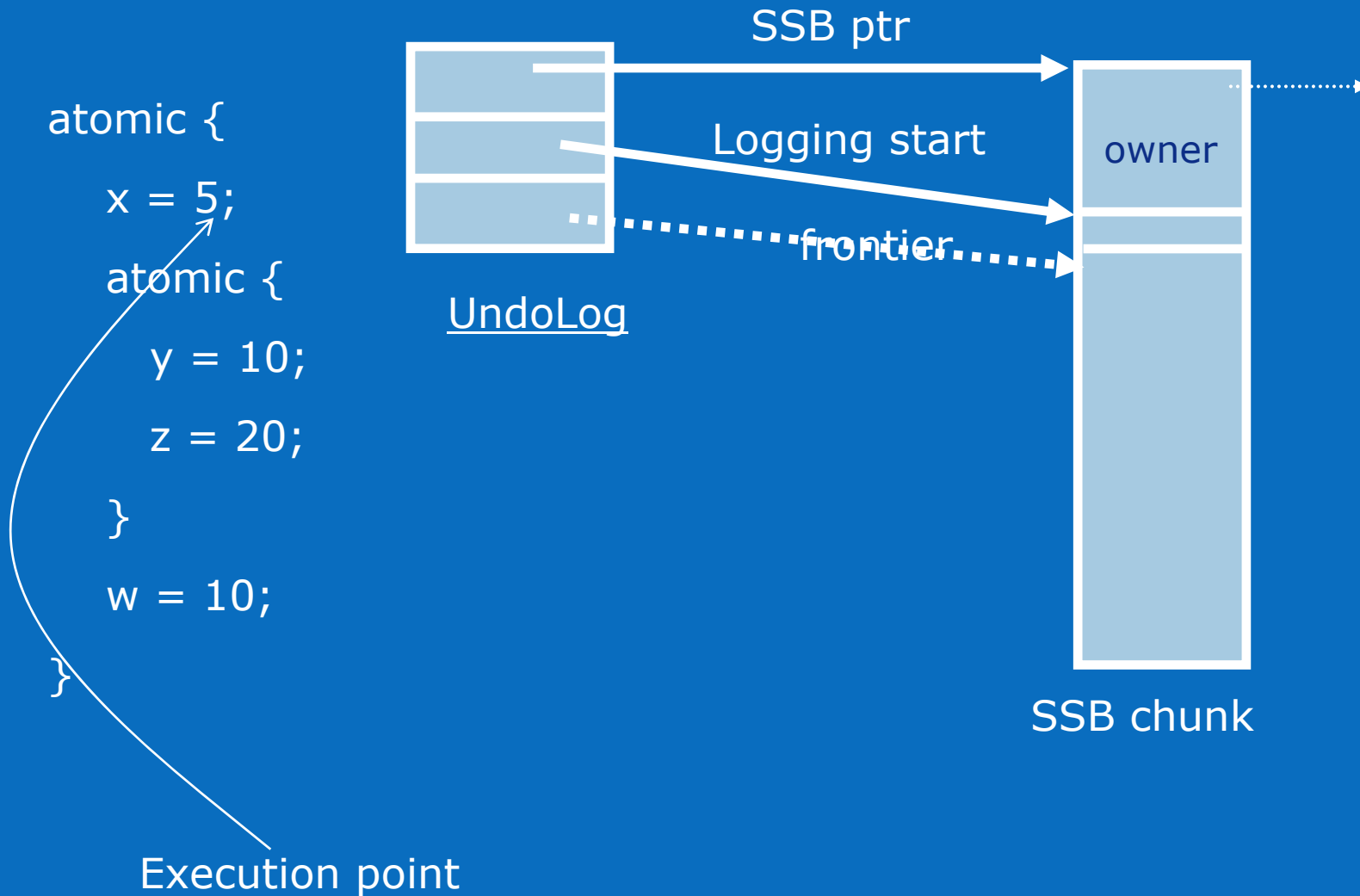


Logging is fast and unbounded

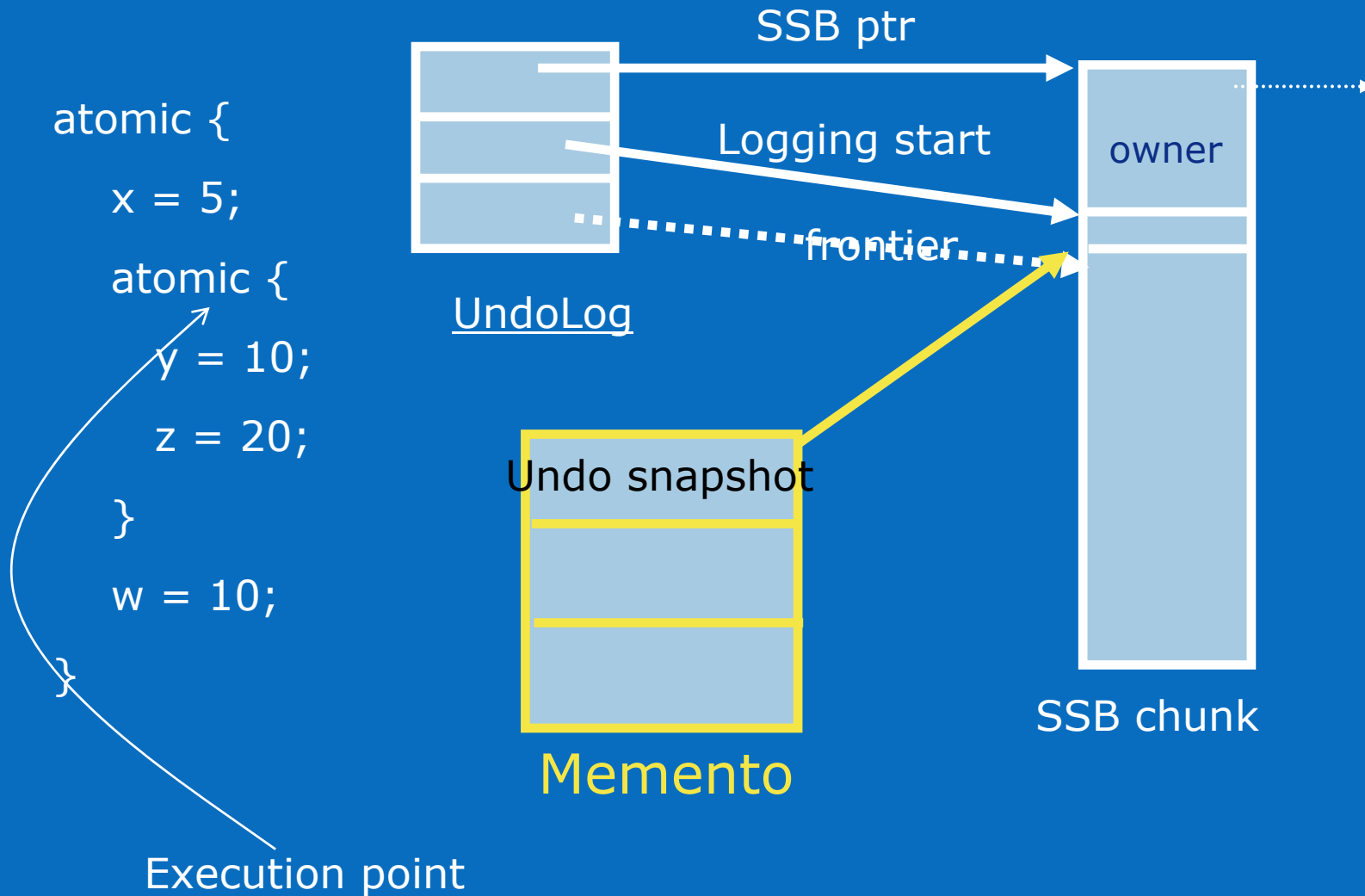
Transaction Mementos: Nesting



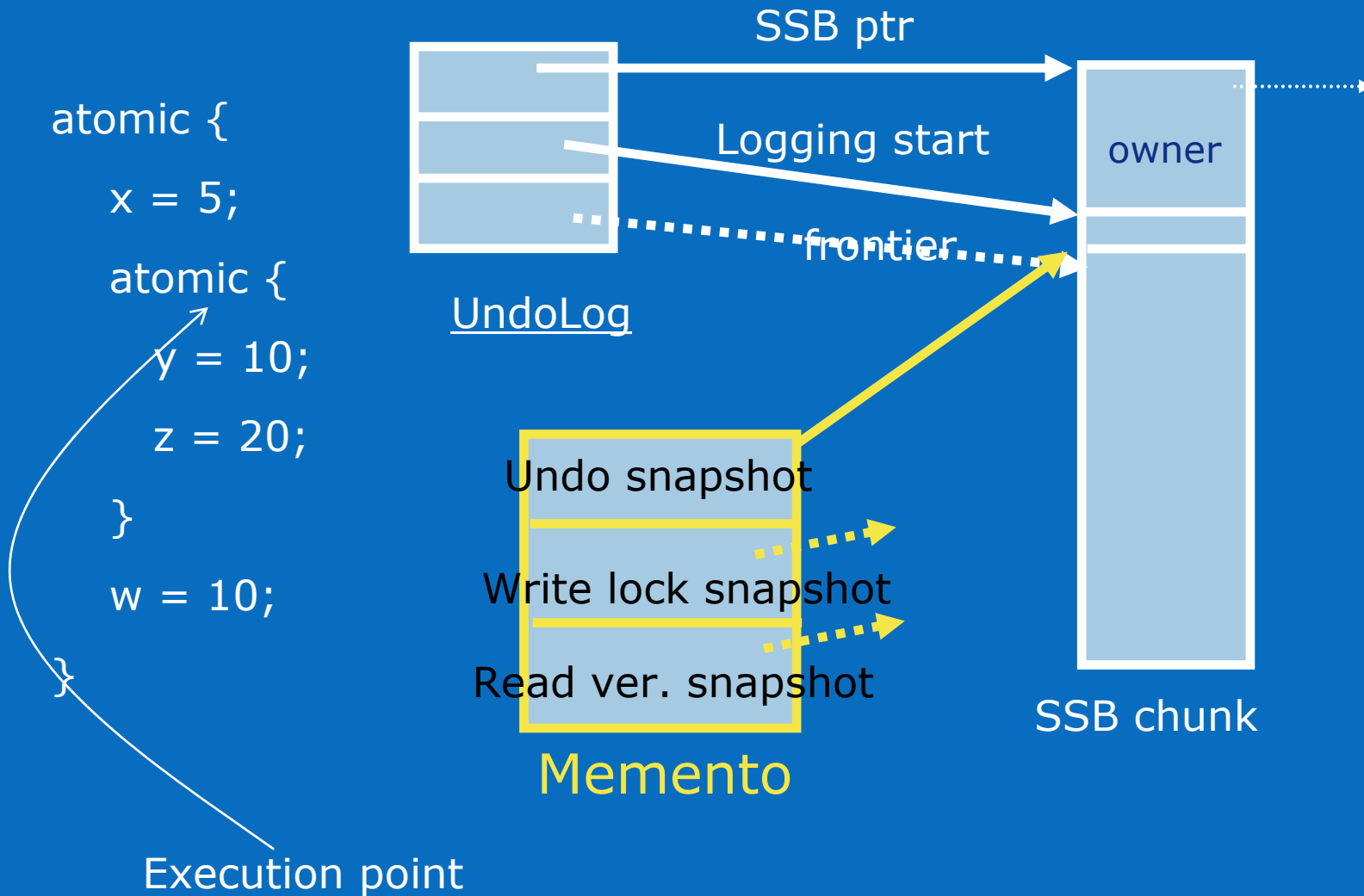
Transaction Mementos: Nesting



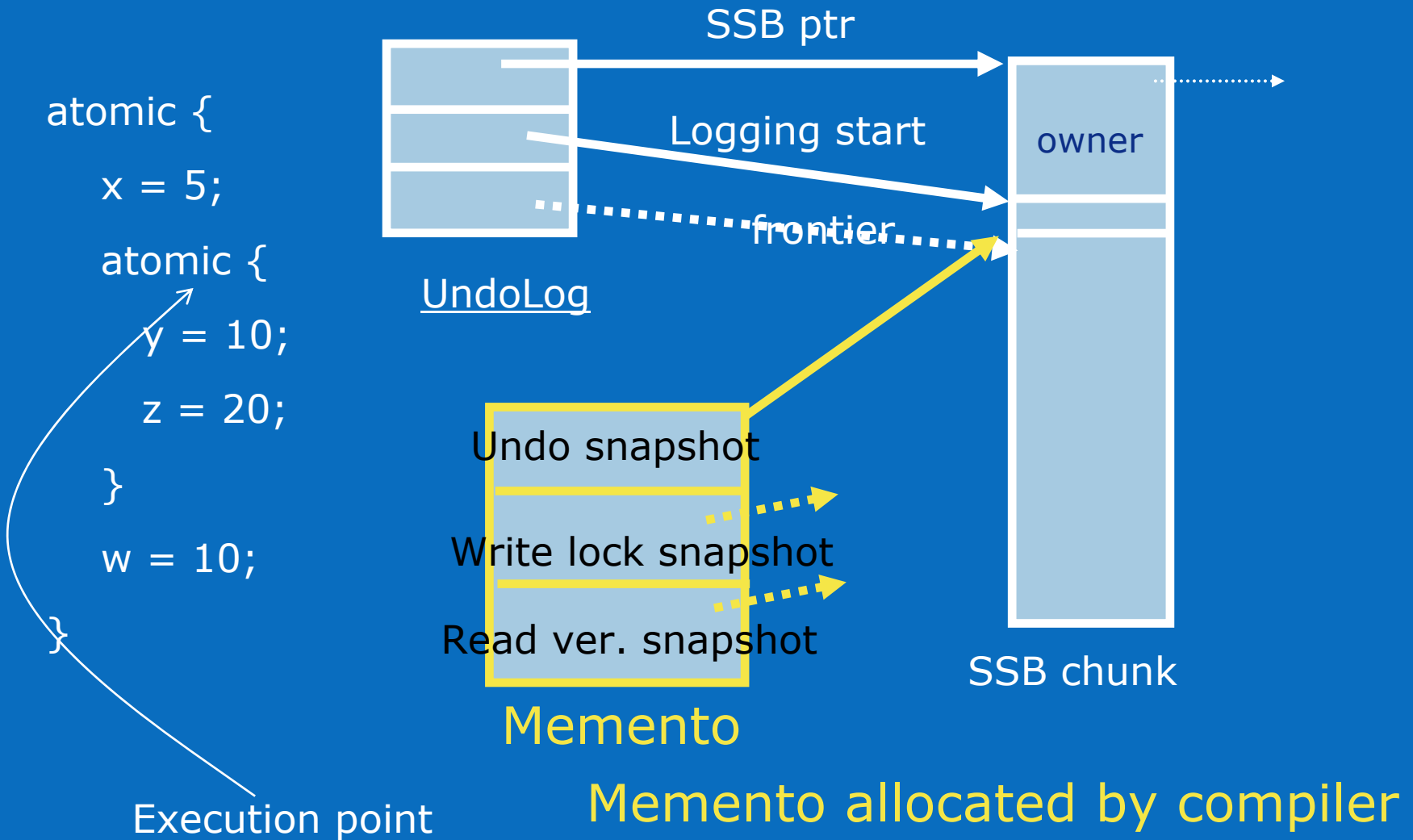
Transaction Mementos: Nesting



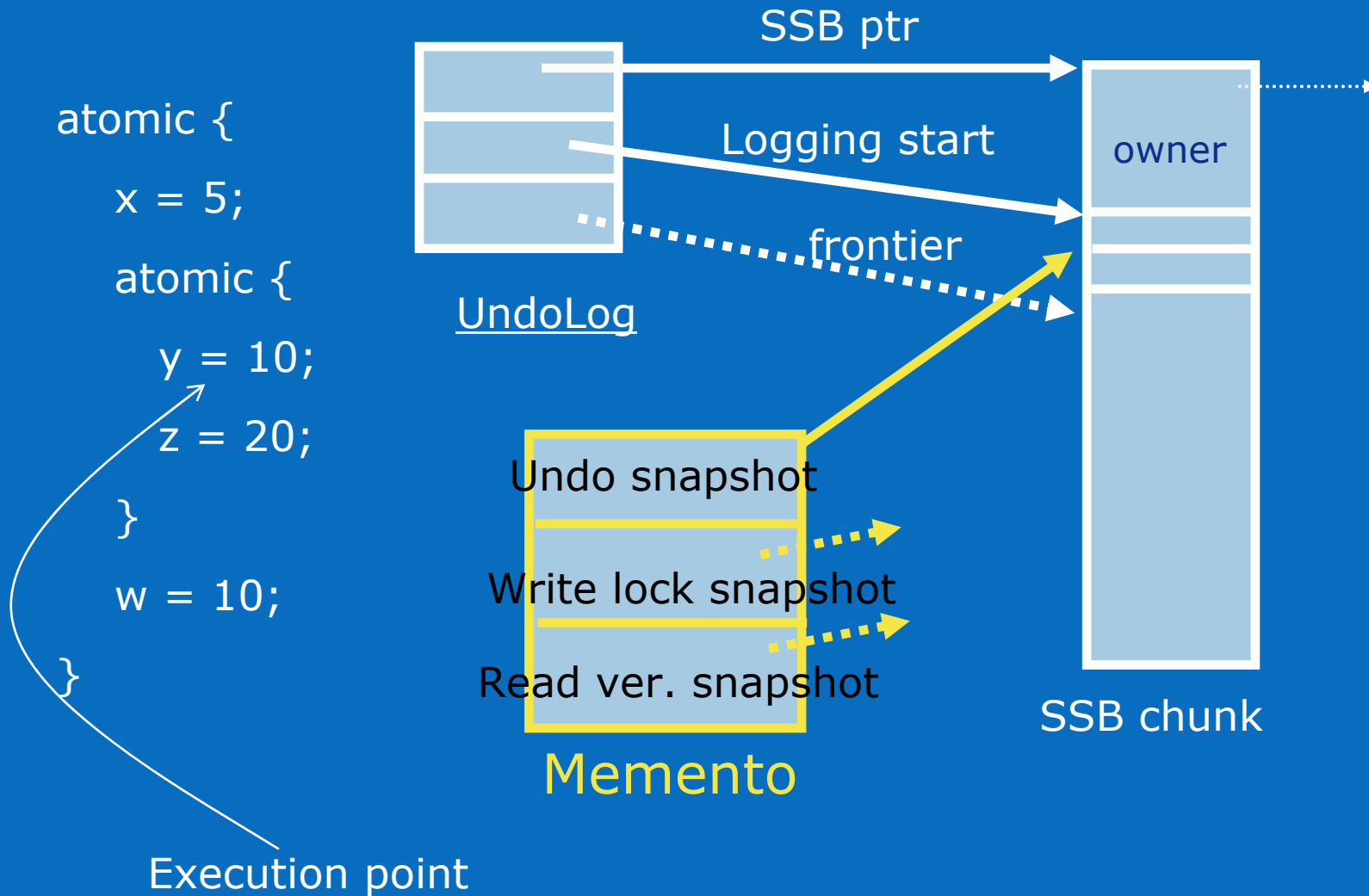
Transaction Mementos: Nesting



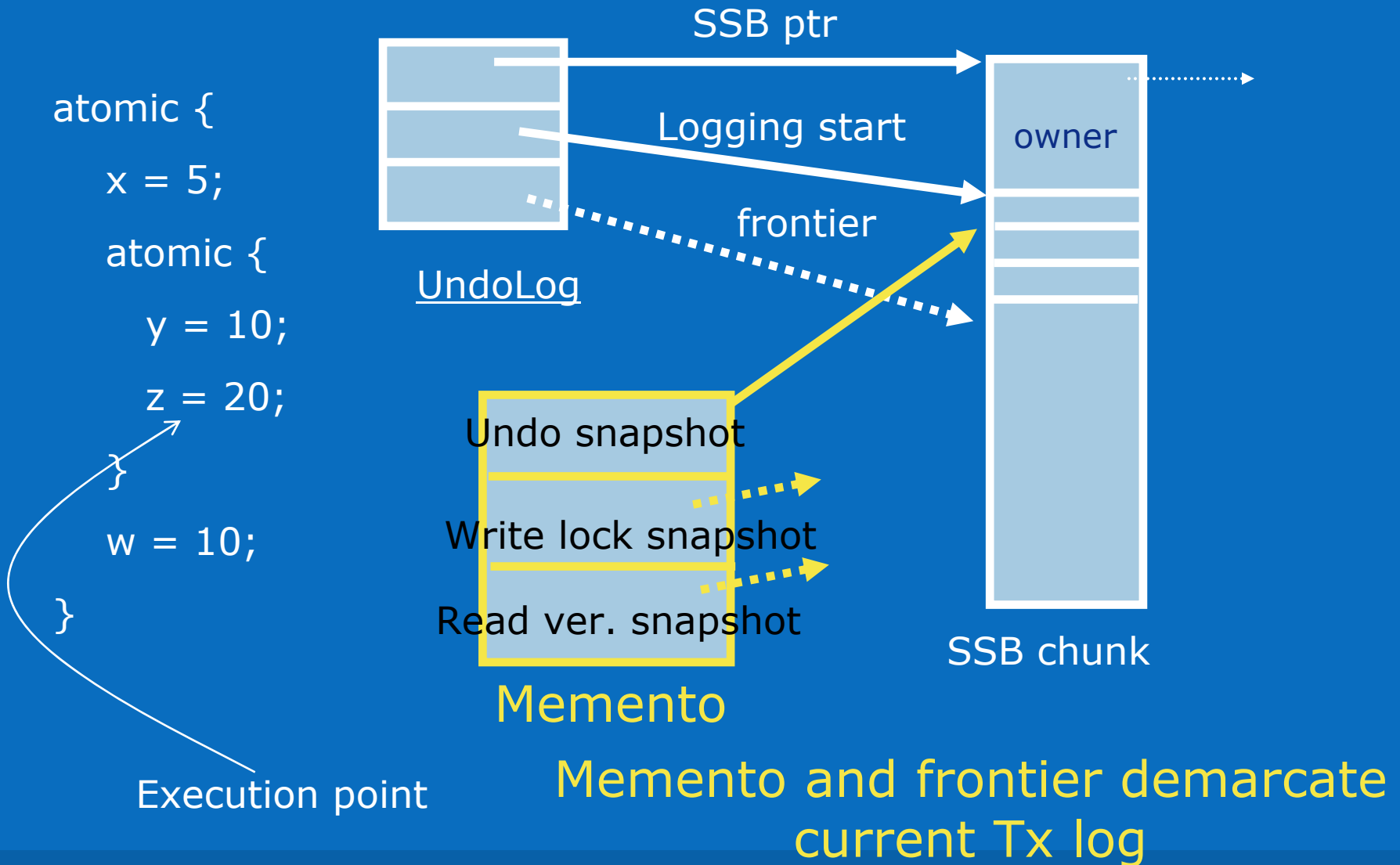
Transaction Mementos: Nesting



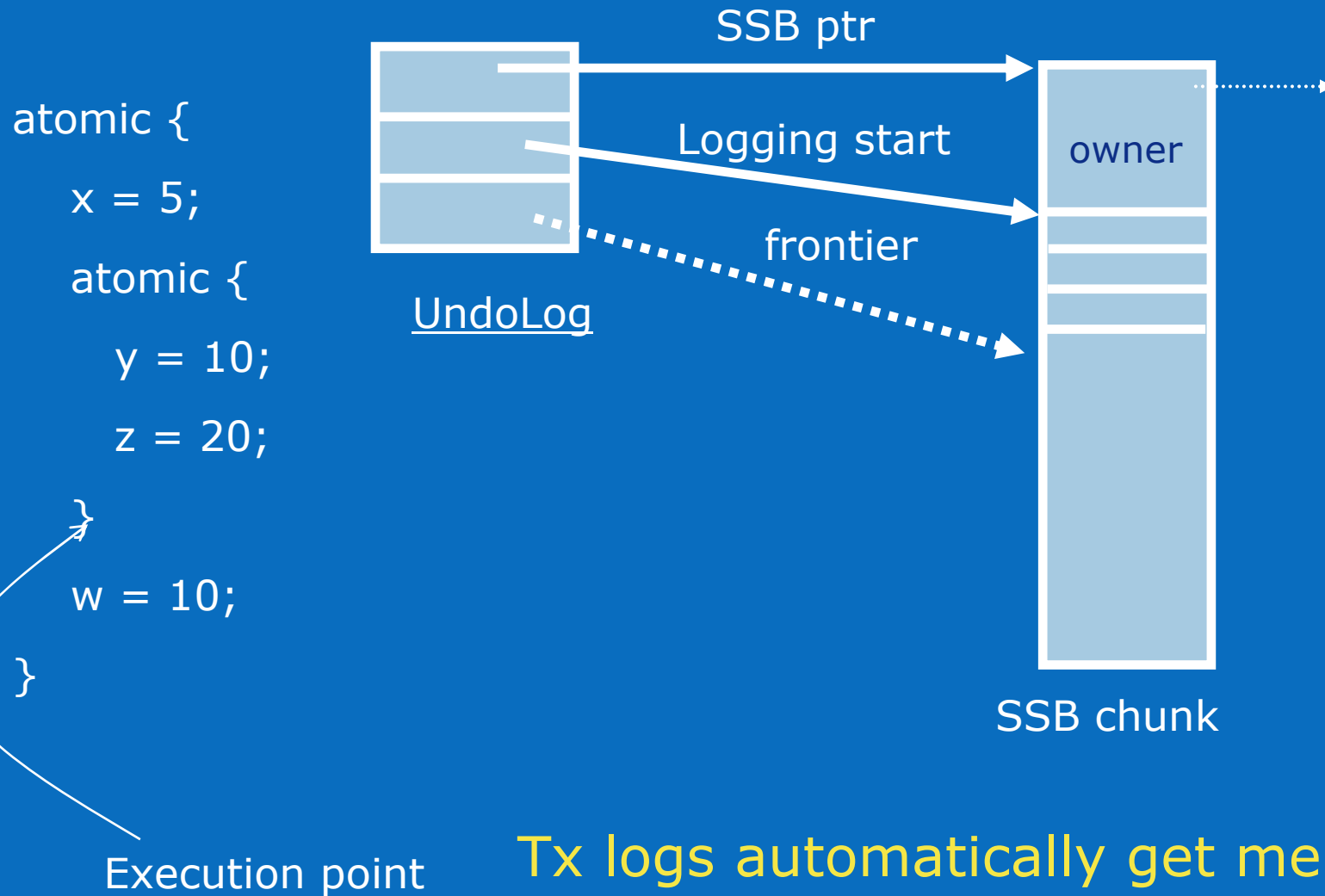
Transaction Mementos: Nesting



Transaction Mementos: Nesting



Transaction Mementos: Nesting



Implementing Atomicity: Example

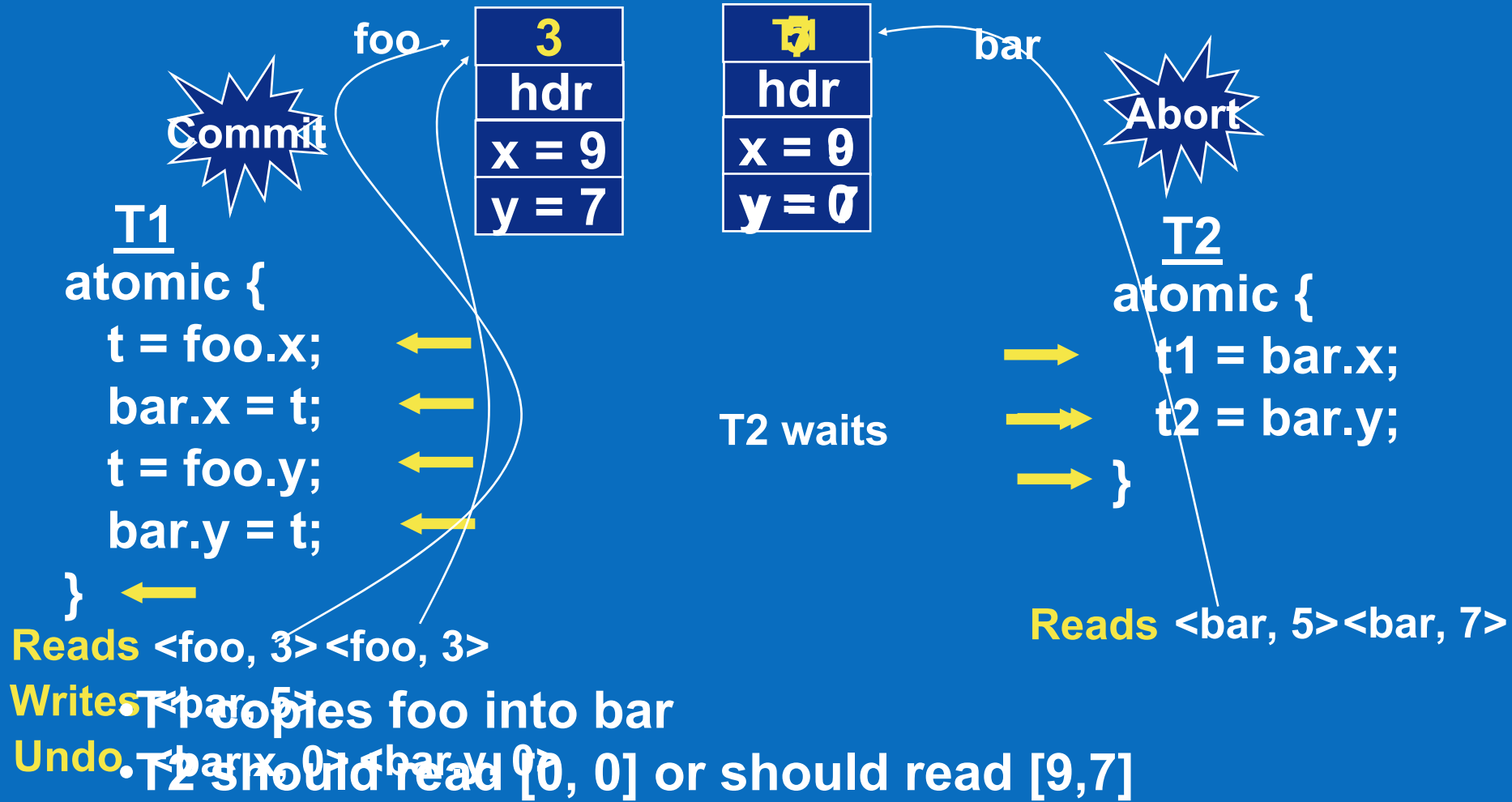
We will show one way to implement atomicity in a STM

Uses two phase locking for writes

Uses optimistic concurrency for reads

Illustrates how the different data structures are used

Example



Ensuring Atomicity: Options

Memory Ops → Mode ↓	Reads	Writes
Pessimistic Concurrency	Read lock on TxR (reader-writer lock or reader list)	
Optimistic Concurrency	Use versioning on TxR	

Ensuring Atomicity: Options

Memory Ops → Mode ↓	Reads	Writes
Pessimistic Concurrency	<ul style="list-style-type: none">- Caching effects- Lock operations	
Optimistic Concurrency	<ul style="list-style-type: none">+ Caching effects+ Avoids lock operations	

Ensuring Atomicity: Options

Memory Ops → Mode ↓	Reads	Writes
Pessimistic Concurrency		Write lock on TxR
Optimistic Concurrency		Buffer writes & acquire locks at commit

Ensuring Atomicity: Options

Memory Ops → Mode ↓	Reads	Writes
Pessimistic Concurrency		+ In place updates + Fast commits + Fast reads
Optimistic Concurrency		- Slow commits - Reads have to search for latest value

Java Virtual Machine Support

On-demand cloning of methods called inside transactions

JIT compiler automatically inserts read/write barriers

- Maps barriers to first class opcodes in intermediate representation
- Good compiler representation → greater optimization opportunities
- Determine conflict detection granularity on per-type basis

Garbage collection support

- Enumeration of references in STM data structures
- Filtering to remove redundant log entries
- Mappings are valid across moving GC

Representing Read/Write Barriers

Coarse-grain barriers hide redundant locking/logging

atomic {

 a.x = t1

 a.y = t2

 if(a.z == 0) {

 a.x = 0

 a.z = t3

 }

}

...

stmWr(&a.x, t1)

stmWr(&a.y, t2)

if(**stmRd**(&a.z) != 0) {

stmWr(&a.x, 0);

stmWr(&a.z, t3)

}

An STM IR for Optimization

Redundancies exposed:

```
atomic {  
    a.x = t1  
    a.y = t2  
    if(a.z == 0) {  
        a.x = 0  
        a.z = t3  
    }  
}
```

```
txnOpenForWrite(a)  
txnLogObjectInt(&a.x, a)  
a.x = t1  
txnOpenForWrite(a)  
txnLogObjectInt(&a.y, a)  
a.y = t2  
txnOpenForRead(a)  
if(a.z != 0) {  
    txnOpenForWrite(a)  
    txnLogObjectInt(&a.x, a)  
    a.x = 0  
    txnOpenForWrite(a)  
    txnLogObjectInt(&a.z, a)  
    a.z = t3  
}
```

Optimized Code

Fewer & cheaper STM operations

```
atomic {  
    a.x = t1  
    a.y = t2  
    if(a.z == 0) {  
        a.x = 0  
        a.z = t3  
    }  
}
```

```
txnOpenForWrite(a)  
txnLogObjectInt(&a.x, a)  
a.x = t1  
txnLogObjectInt(&a.y, a)  
a.y = t2  
if(a.z != 0) {  
    a.x = 0  
    txnLogObjectInt(&a.z, a)  
    a.y = t3  
}
```

Compiler Optimizations for Transactions

Standard optimizations

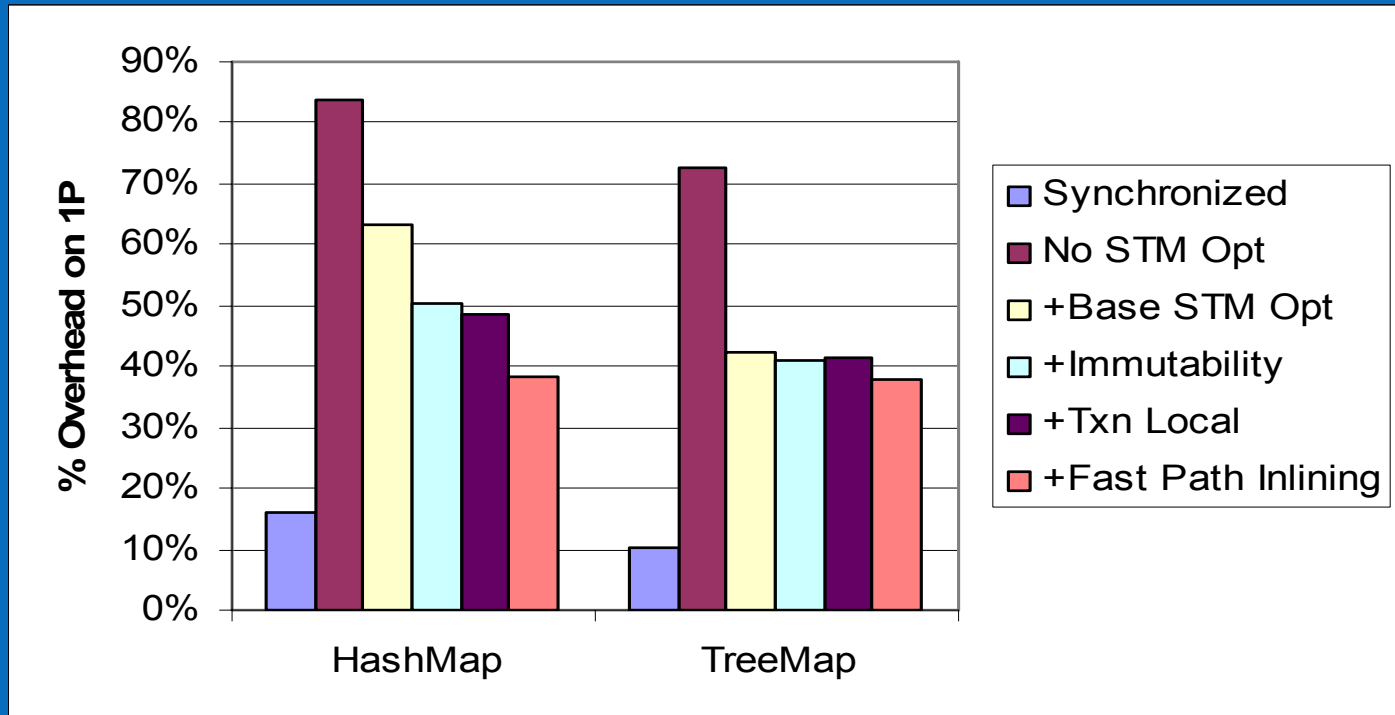
- CSE, Dead-code-elimination, ...
- Careful IR representation exposes opportunities and enables optimizations with almost no modifications
- Subtle in presence of nesting

STM-specific optimizations

- Immutable field / class detection & barrier removal (vtable/String)
- Transaction-local object detection & barrier removal
- Partial inlining of STM fast paths to eliminate call overhead

Effect of Compiler Optimizations

1P overheads over thread-unsafe baseline



Prior STMs typically incur $\sim 2\times$ on 1P

With compiler optimizations:

- < 40% over no concurrency control
- < 30% over synchronization

Outline

Software Transactional Memory

- Translating a language construct
- Runtime support
- Compiler support

Integrating with system services (malloc/free)

Open issues & conclusions



Malloc/free with Transactions

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
        while( temp1->next->key != key ) {  
            temp1 = temp1->next;  
        } /* search node with a key */  
        temp2 = temp1->next;  
        /* rearrange list pointers */  
        temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

Malloc/free with Transactions

```
nodeDelete(int key) {  
→ temp1 = head of list;  
  atomic {  
    while( temp1->next->key != key ) {  
      temp1 = temp1->next;  
    } /* search node with a key */  
    temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
  }  
  free(temp2);  
}
```

```
nodeDelete(int key) {  
→ temp1 = head of list;  
  atomic {  
    while( temp1->next->key != key ) {  
      temp1 = temp1->next;  
    } /* search node with a key */  
    temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
  }  
  free(temp2);  
}
```

Malloc/free with Transactions

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
→ while( temp1->next->key != key ) {  
        temp1 = temp1->next;  
    } /* search node with a key */  
    temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
→ while( temp1->next->key != key ) {  
        temp1 = temp1->next;  
    } /* search node with a key */  
    temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

Malloc/free with Transactions

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
        while( temp1->next->key != key ) {  
            temp1 = temp1->next;  
        } /* search node with a key */  
→ temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
        while( temp1->next->key != key ) {  
            temp1 = temp1->next;  
        } /* search node with a key */  
→ temp2 = temp1->next;  
    /* rearrange list pointers */  
    temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

Only reads till now

Malloc/free with Transactions

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
        while( temp1->next->key != key ) {  
            temp1 = temp1->next;  
        } /* search node with a key */  
        temp2 = temp1->next;  
        /* rearrange list pointers */  
        temp1->next = temp1->next->next;  
    }  
    → free(temp2);  
}
```

```
nodeDelete(int key) {  
    temp1 = head of list;  
    atomic {  
        while( temp1->next->key != key ) {  
            temp1 = temp1->next;  
        } /* search node with a key */  
        → temp2 = temp1->next;  
        /* rearrange list pointers */  
        temp1->next = temp1->next->next;  
    }  
    free(temp2);  
}
```

One transaction finishes, **when can it free?**

Integrating with Malloc/free

The memory manager uses epochs

- Epochs determine when it is safe to deallocate
- Implemented in a high performance memory allocator (McRT Malloc)

See ISMM 2006 paper by Hudson, Saha, Adl-Tabatabai & Hertzberg

Transactions: Research challenges

Performance

- Right mix of HW & SW components
- Good diagnostics & contention management

Semantics

- I/O & communication
- Nested parallelism

Memory Model

- Language level guarantees

Debugging & performance analysis tools

System integration

Conclusions

Multi-core architectures: an inflection point in mainstream SW development

Navigating inflection requires new parallel programming abstractions

Transactions are a better synchronization abstraction than locks

- Software engineering and performance benefits

Lots of research on implementation and semantics issues

- Great progress, but there are still open problems

