Going Under the Hood with Intel's Next Generation Microarchitecture Codename Haswell

Ravi Rajwar
Intel Corporation

QCon San Francisco
Nov 9, 2012

What is Haswell?

45nm	32nm		22nm	
Nehalem	**Westmere**	**Sandy Bridge**	**Ivy Bridge**	**Haswell**
NEW Intel® Microarchitecture (Nehalem)	Intel Microarchitecture (Nehalem)	NEW Intel Microarchitecture (Sandy Bridge)	Intel Microarchitecture (Sandy Bridge)	NEW Intel Microarchitecture (Haswell)
TOCK	*TICK*	*TOCK*	*TICK*	*TOCK*

<u>Haswell CPU Family</u>

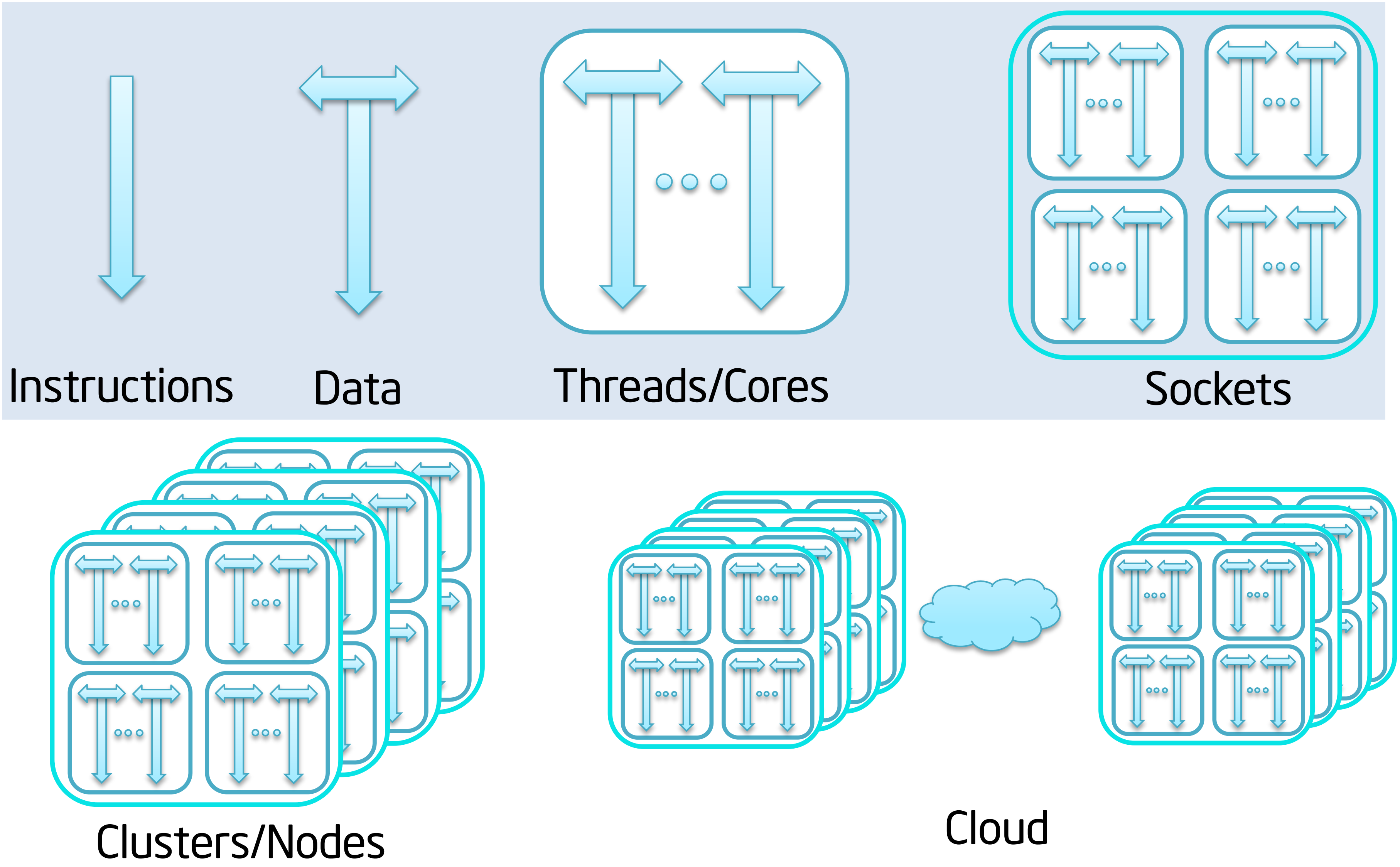
22nm Process Technology

Builds upon Innovations in the 2nd and 3rd Generation Intel® Core ™ Processors

TALK FOCUS:

PERFORMANCE THROUGH PARALLELISM

Parallelism: A Hardware Perspective



Instructions

Data

Threads/Cores

Sockets

Clusters/Nodes

Cloud

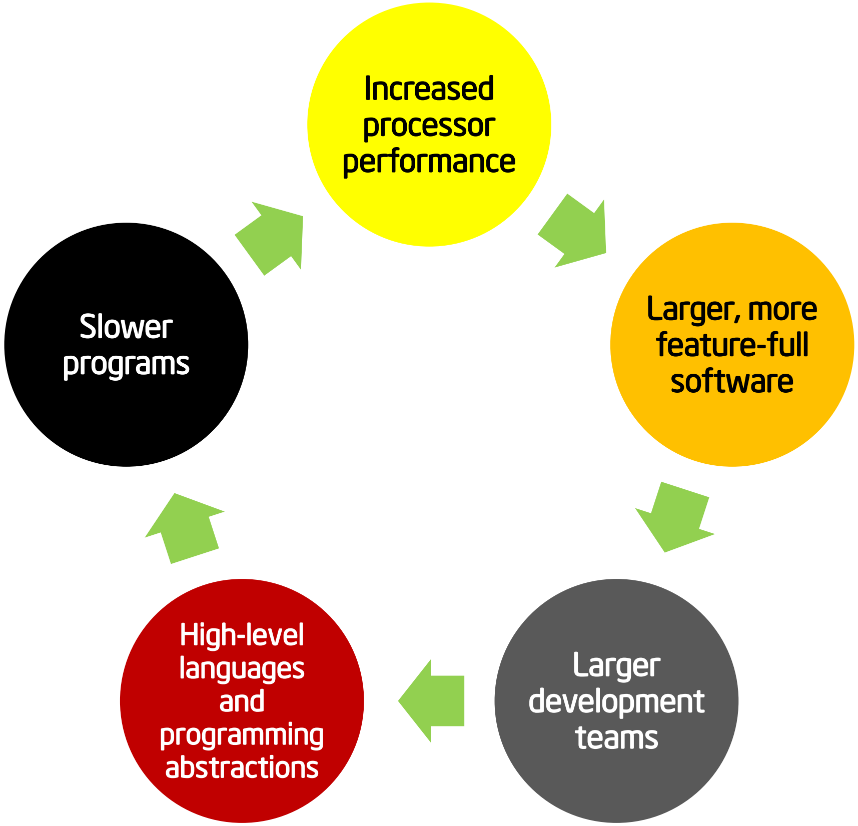
Performance Matters

Faster software

But also about software innovation

- Size
- Rich Functionality
- Improved Abstractions

Productivity and manageability


Increased processor performance
Larger, more feature-full software
Larger development teams
High-level languages and programming abstractions
Slower programs

cf. Jim Larus

Must Exploit All Dimensions of Parallelism to Achieve Highest Performance

Agenda

Next Generation Intel® Microarchitecture (Haswell)

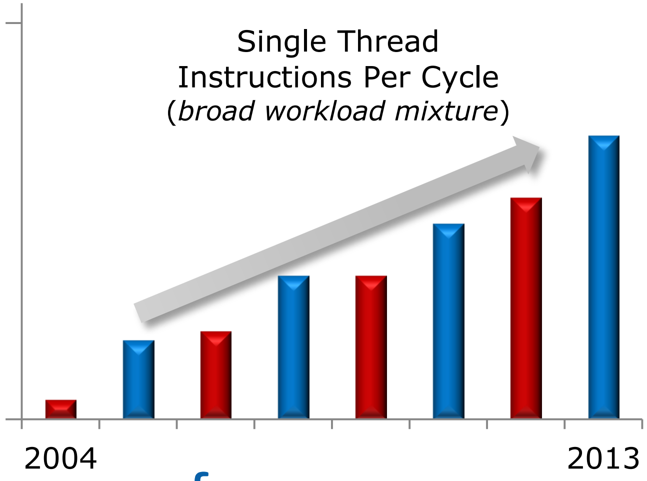
a. Instruction Level Parallelism

b. Data Level Parallelism

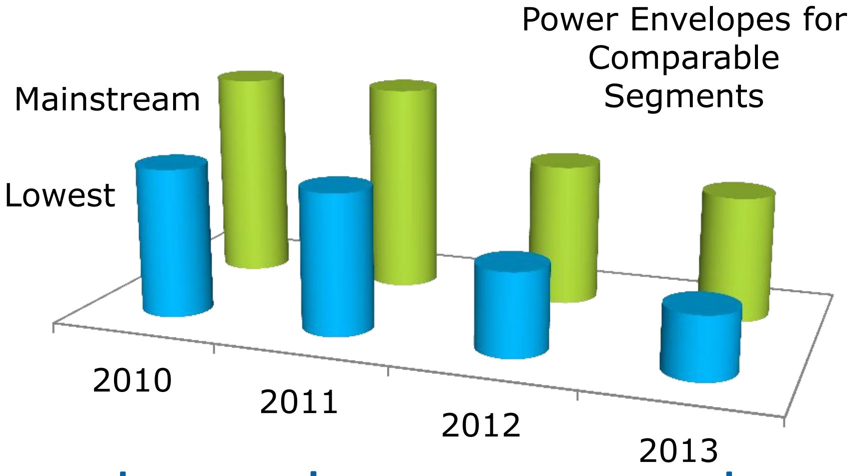
c. Thread Level Parallelism

A Matter of Time

EXPLOITING
PARALLELISM ACROSS INSTRUCTIONS

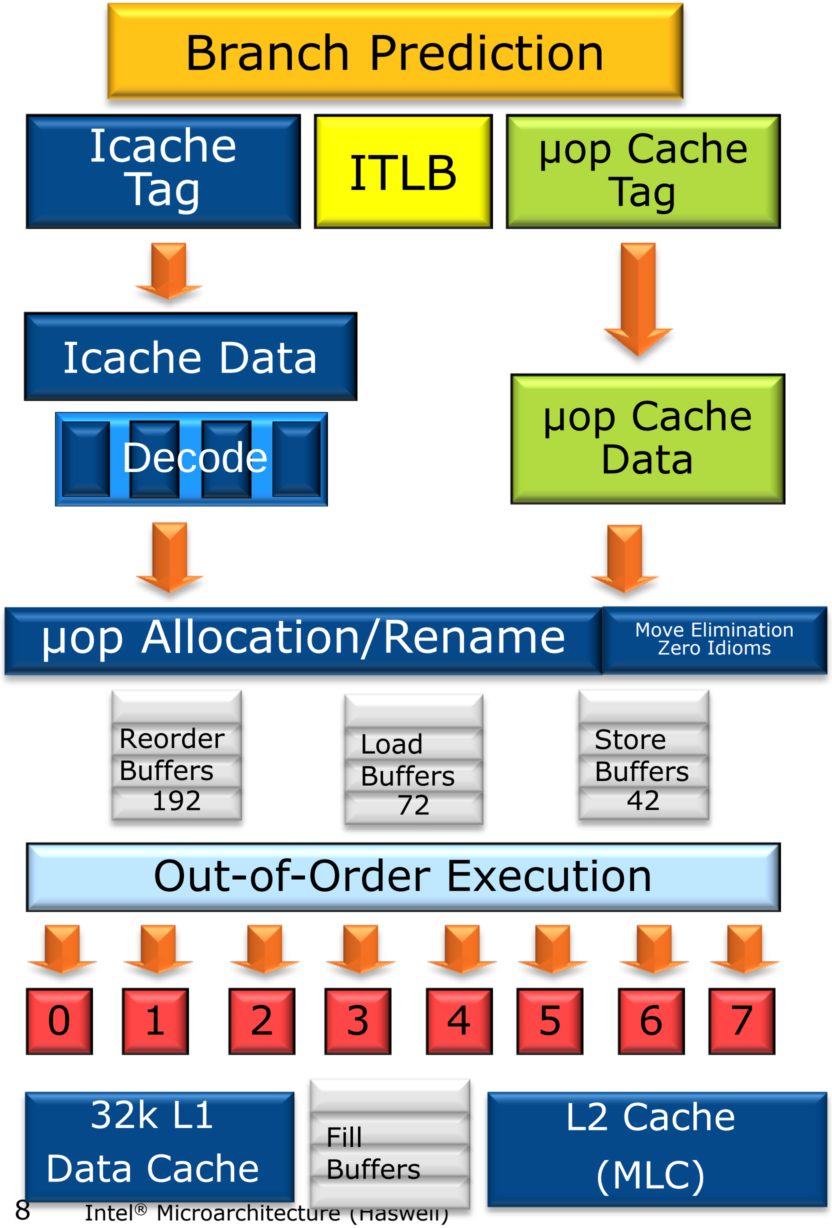


More performance per core



Flat or decreasing power envelopes

Haswell Core at a Glance



Next generation branch prediction

- Improves performance *and* saves wasted work

Improved front-end

- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

Deeper buffers

- Extract more instruction parallelism

More execution units, shorter latencies

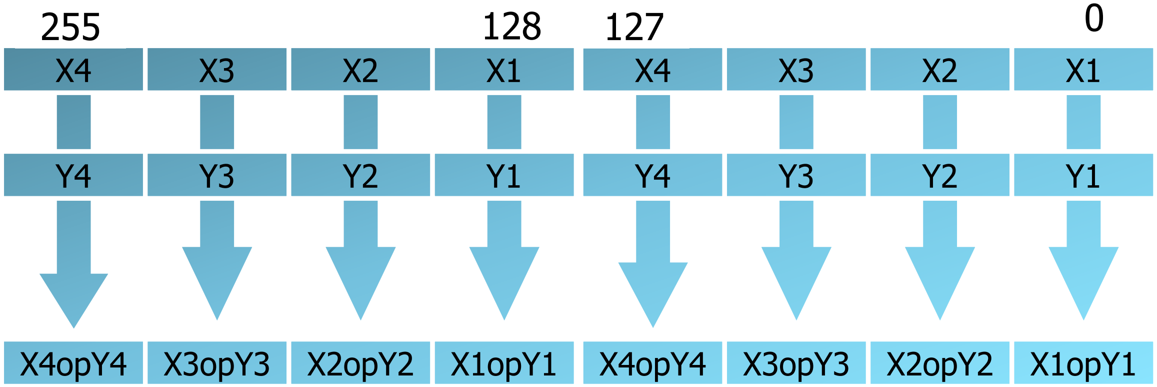
More load/store bandwidth

- 2x32 byte loads, 32 byte store / cycle to L1
- Better prefetching, better cache line split latency & throughput, double L2 bandwidth

No pipeline growth

- Same branch misprediction latency
- Same L1/L2 cache latency

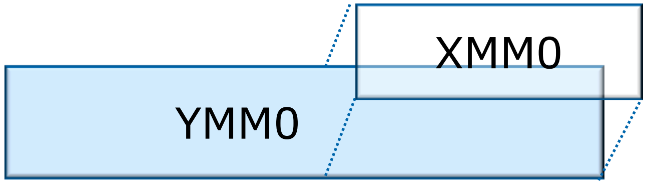
EXPLOITING PARALLELISM ACROSS DATA



Intel® Advanced Vector Extensions (Intel® AVX)

Intel® AVX (on Sandybridge)

- Extends all 16 XMM registers to 256 bits
- Intel® AVX instructions operate on
  - All 256 bits (FP only)
  - Lower 128 bits (SSE instructions)



256 bits(2011) 128 bits (1999)

Intel® AVX2 (on Haswell)

- Extends 128-bit integer vector instructions to 256 bit
- Enhanced vectorization with Gather, Shifts, and powerful permutes
  - 20+ new operations to the vector ISA
  - E.g., building block for sparse, indirect memory accesses
- FP Fused Multiply Add for FLOPS

Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge)

Bit Manipulation Instructions

New instructions for

- Arbitrary bit field manipulations
- Leading and trailing zero bits counts
- Trailing set bit manipulations
- Improved rotates and arbitrary precision multiplies

Group	Instructions
Bit Field Pack/Extract	BZHI, SHLX, SHRX, SARX, BEXTR
Variable Bit Length Stream Decode	LZCNT, TZCNT, BLSR, BLSMSK, BLSI, ANDN
Bit Gather/Scatter	PDEP, PEXT
Arbitrary Precision Arithmetic & Hashing	MULX, RORX

Speedup algorithms performing

- Bit-field extract & packing, bit-granular encoded data processing (compression algorithms, universal coding)
- Arbitrary precision multiplication, hashes

Replace Complex and Slow Instruction Sequences

EXPLOITING
PARALLELISM ACROSS THREADS

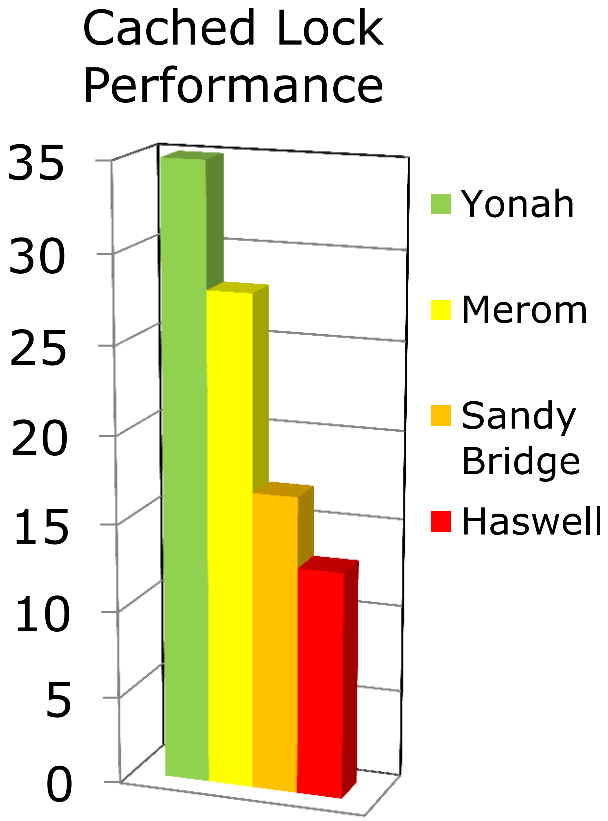
Synchronization Improvements

Improving existing primitives

- Faster LOCK-prefixed instructions
- A focus in recent generations

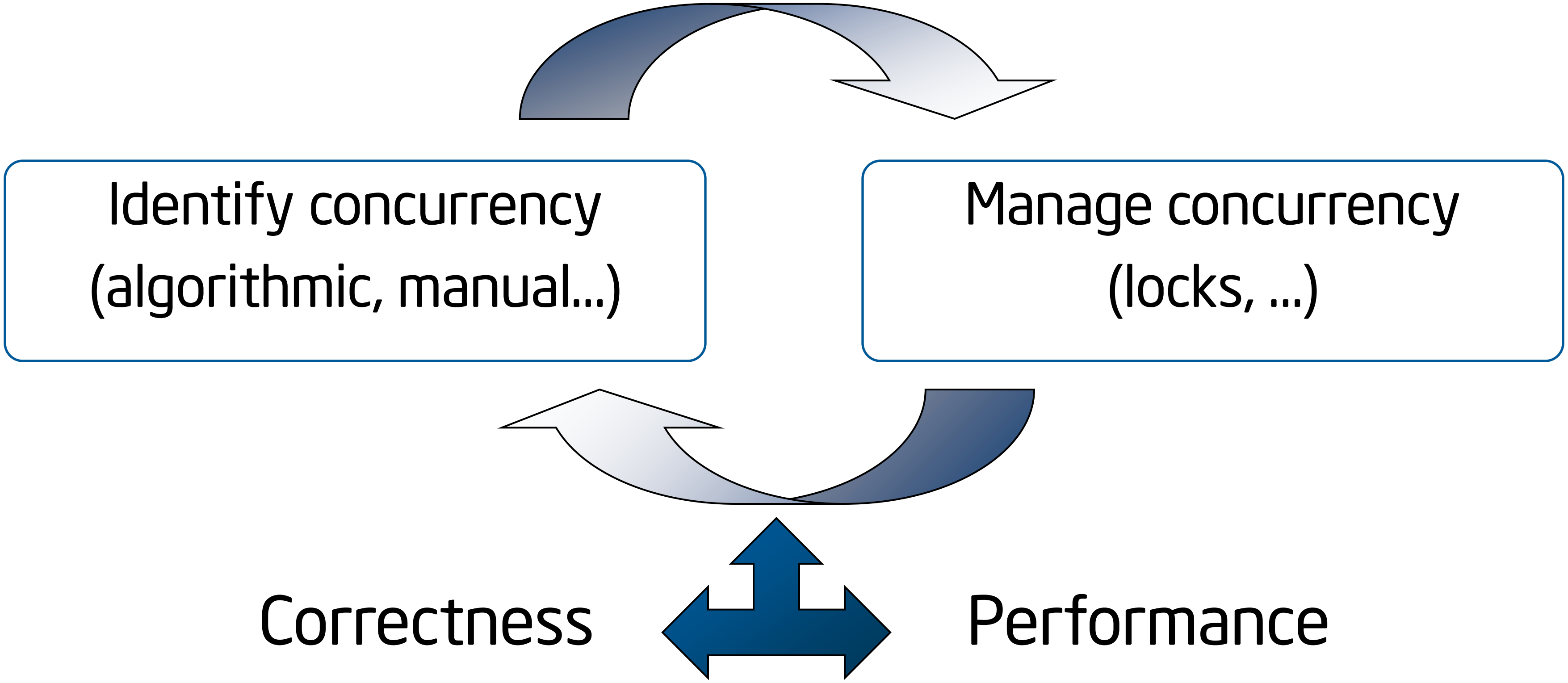
But locks still limit concurrency

- Lock-protected critical sections
- Needed for threading correctness



What Can We Do About Exposing Concurrency?

Difficulty of Software Development



Identify concurrency
(algorithmic, manual…)

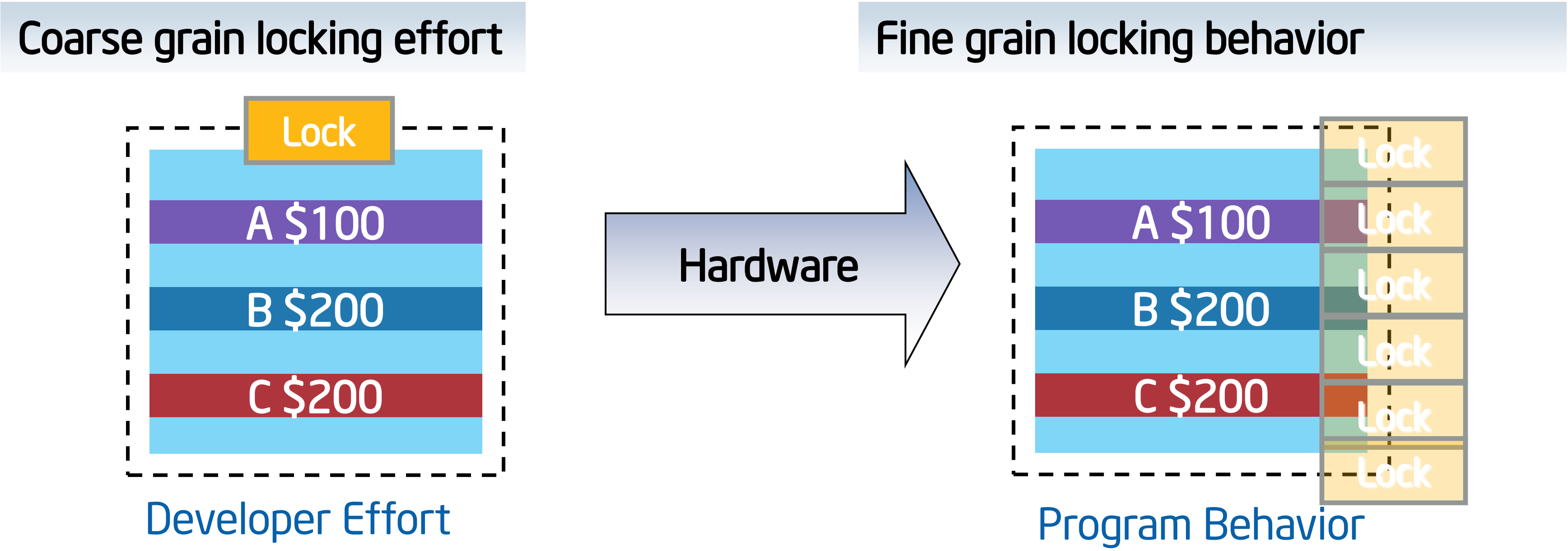
Manage concurrency
(locks, …)

Correctness

Performance

Hard to Write Fast and Correct Multi-Threaded Code

What We Want…



Developer uses coarse grain lock

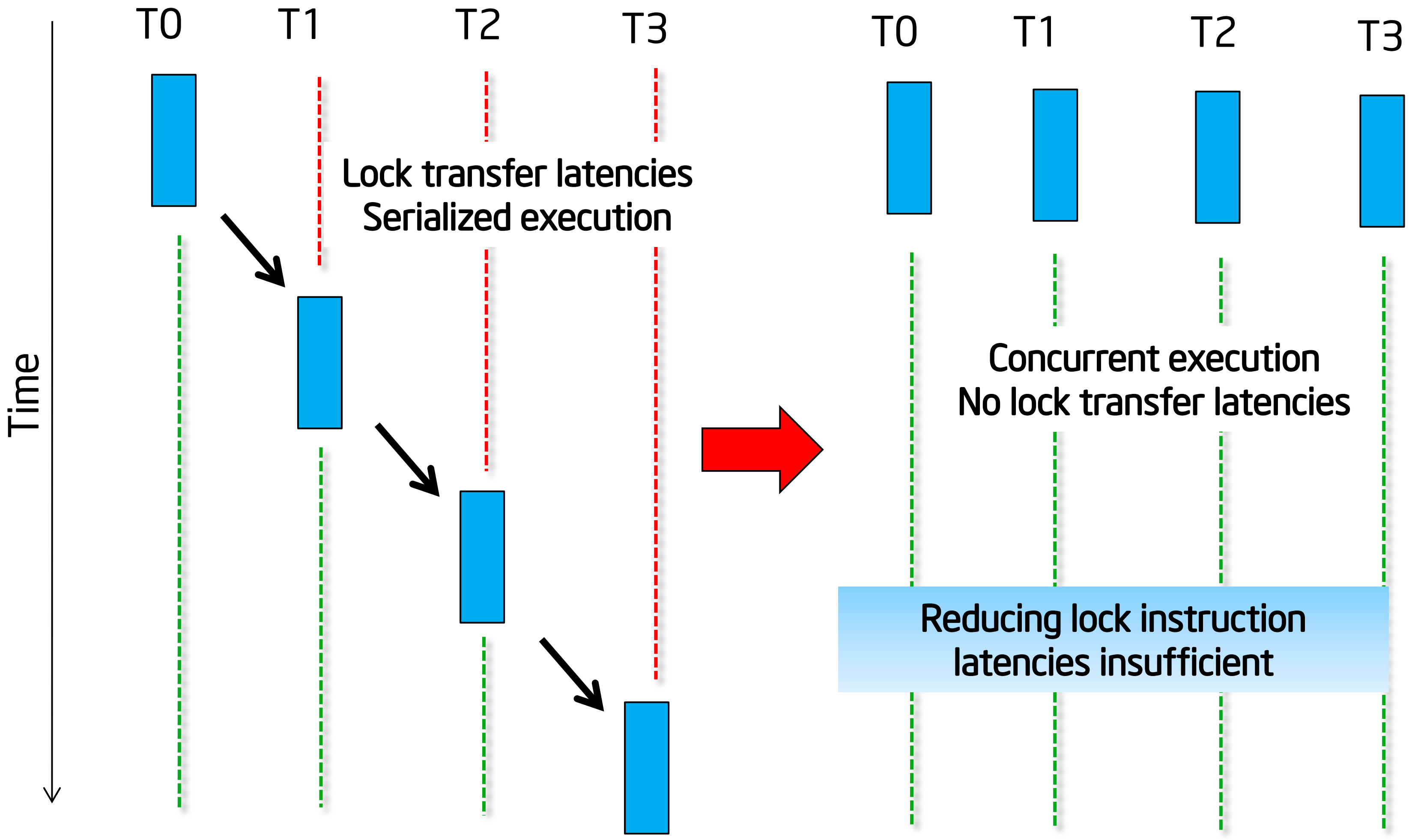
Hardware elides the lock to expose concurrency

– Multiple threads don't serialize on the lock

– Hardware automatically detects **real** data conflicts

Lock Elision: Fine Grain Behavior at Coarse Grain Effort

Benefit of Lock Elision



Lock transfer latencies
Serialized execution

Concurrent execution
No lock transfer latencies

Reducing lock instruction
latencies insufficient

Exposes Concurrency & Eliminates Unnecessary Communication

Transactional Synchronization

Hardware support to enable lock elision

- Focus on lock granularity optimizations
- Fine grain performance at coarse grain effort

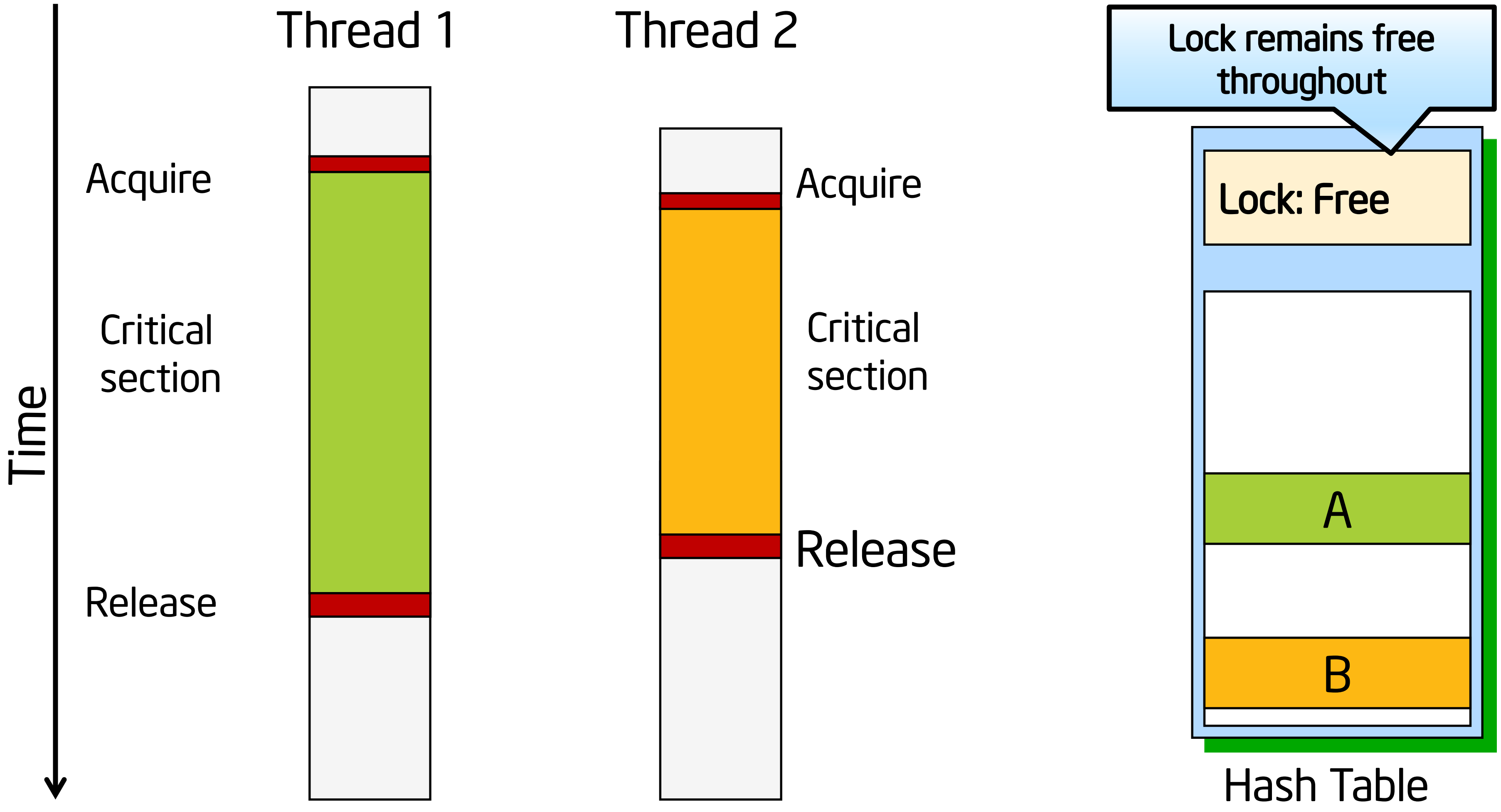
Intel® TSX: Instruction set extensions for IA‡

- Transactionally execute lock-protected critical sections
- Execute without acquiring lock → expose hidden concurrency
- Hardware manages transactional updates – All or None
  - Other threads can't observe intermediate transactional updates
  - If lock elision cannot succeed, restart execution & acquire lock

Intel® TSX Exposes Concurrency Through Lock Elision

‡Intel® Architecture Instruction Set Extensions Programming Reference (http://software.intel.com/file/41604)
Intel® Transactional Synchronization Extensions (Intel® TSX)

A Canonical Intel® TSX Execution



No Serialization and No Communication if No Data Conflicts

Intel® TSX Interfaces for Lock Elision

Hardware Lock Elision (HLE) – XACQUIRE/XRELEASE

- Software uses legacy compatible hints to identify critical section. Hints ignored on hardware without TSX
- Hardware support to execute transactionally without acquiring lock
- Abort causes a re-execution without elision
- Hardware manages all architectural state

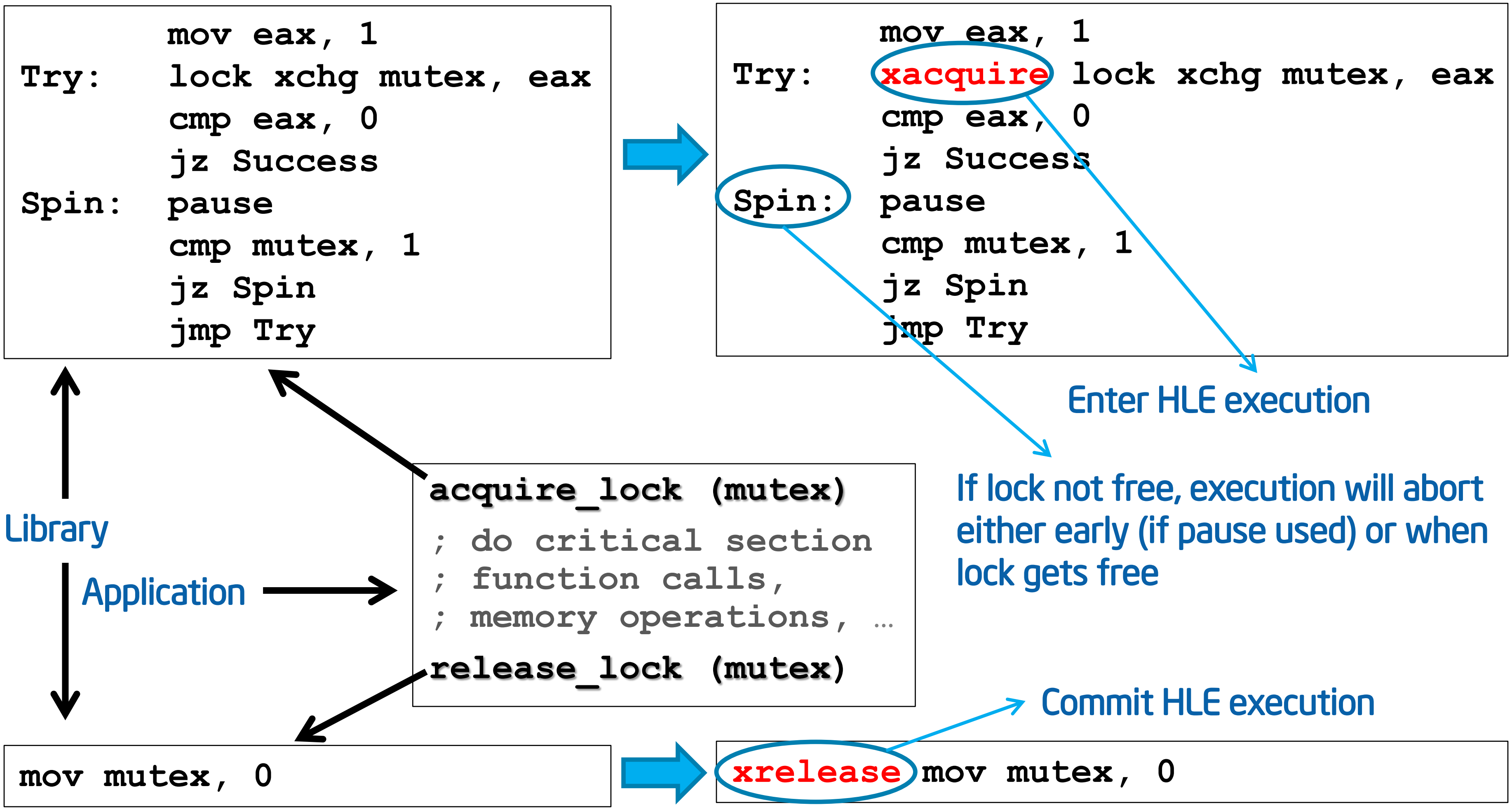
Restricted Transactional Memory (RTM) – XBEGIN/XEND

- Software uses new instructions to specify critical sections
- Similar to HLE but flexible interface for software to do lock elision
- Abort transfers control to target specified by XBEGIN operand
- Abort information returned in a general purpose register (EAX)

XTEST and XABORT – Additional instructions

Flexible and Easy To Use

Intel® TSX Interface: HLE



```
        mov eax, 1
Try:    lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

```
        mov eax, 1
Try:    xacquire lock xchg mutex, eax
        cmp eax, 0
        jz Success
Spin:   pause
        cmp mutex, 1
        jz Spin
        jmp Try
```

Enter HLE execution

If lock not free, execution will abort either early (if pause used) or when lock gets free

Library

Application

```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

```
mov mutex, 0
```

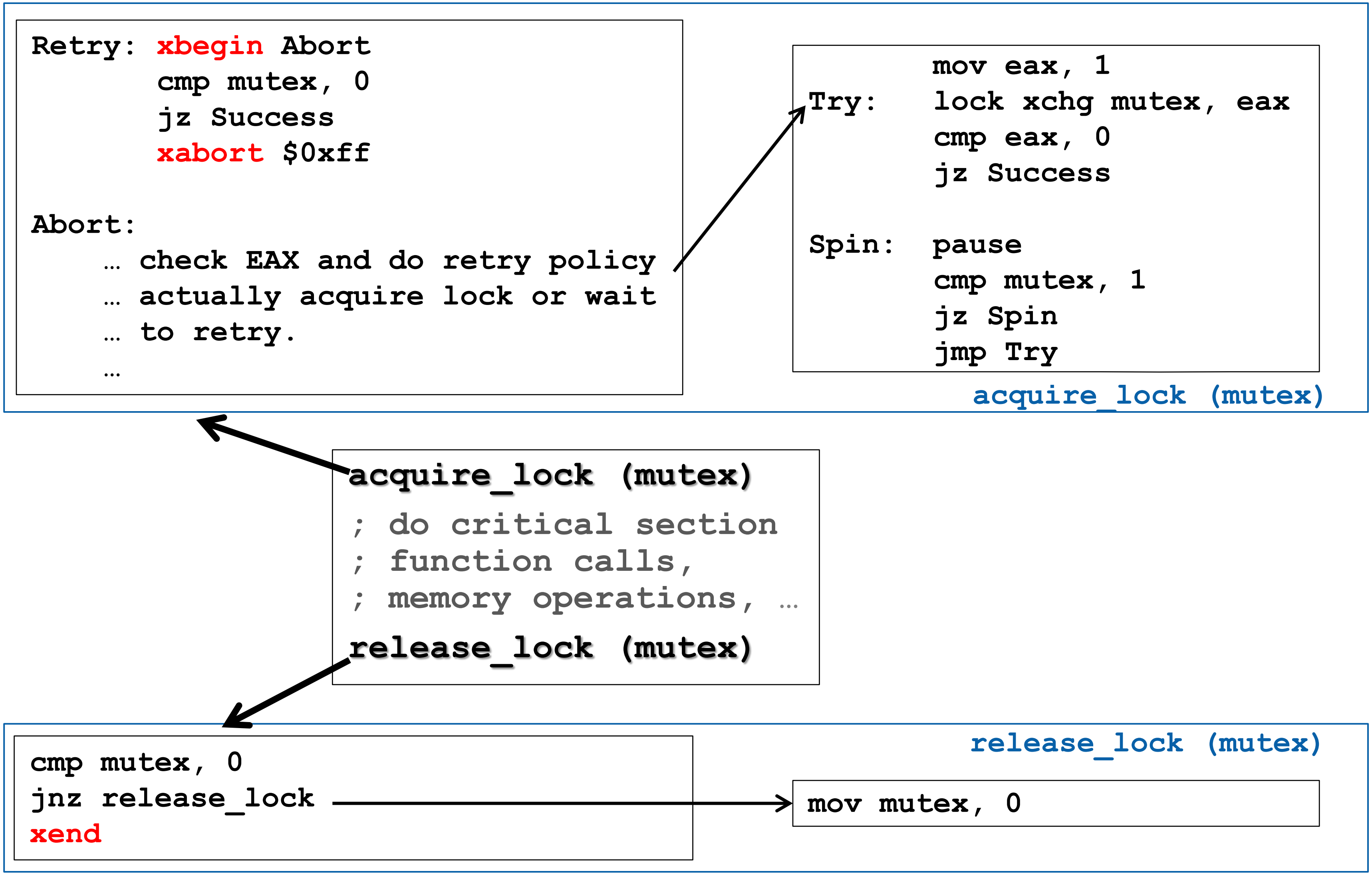
```
xrelease mov mutex, 0
```

Commit HLE execution

Legacy Compatible Enabling Within Libraries

Code example for illustration purpose only

Intel® TSX Interface: RTM



```
Retry: xbegin Abort
       cmp mutex, 0
       jz Success
       xabort $0xff

Abort:
    … check EAX and do retry policy
    … actually acquire lock or wait
    … to retry.
    …
```

```
       mov eax, 1
Try:   lock xchg mutex, eax
       cmp eax, 0
       jz Success

Spin:  pause
       cmp mutex, 1
       jz Spin
       jmp Try
```

acquire_lock (mutex)

```
acquire_lock (mutex)
; do critical section
; function calls,
; memory operations, …
release_lock (mutex)
```

release_lock (mutex)

```
cmp mutex, 0
jnz release_lock
xend
```

```
mov mutex, 0
```

Code example for illustration purpose only

Identify and Elide: HLE

Hardware support to elide multiple locks

- Hardware elision buffer manages actively elided locks
- XACQUIRE/XRELEASE allocate/free elision buffer entries
- Skips elision without aborting if no free entry available

Hardware treats XACQUIRE/XRELEASE as hints

- Functionally correct even if hints used improperly
- Hardware checks if locks meet requirements for elision
- May expose latent bugs and incorrect timing assumptions

Hardware Management of Elision Enables Ease of Use

Execute Transactionally

Hardware manages all transactional updates

- Other threads cannot observe any intermediate updates
- If transactional execution doesn't succeed, hardware restarts execution
- Hardware discards all intermediate updates prior to restart

Transactional abort

- Occurs when abort condition is detected
- Hardware discards all transactional updates

Transactional commit

- Hardware makes transactional updates visible instantaneously
- No cross-thread/core/socket coordination required

Software Does Not Worry About State Recovery

Execute Transactionally – Memory

Buffering memory writes

- Hardware uses L1 cache to buffer transactional writes
  - Writes not visible to other threads until after commit
  - Eviction of transactionally written line causes abort
- Buffering at cache line granularity

Sufficient buffering for typical critical sections

- Cache associativity can occasionally be a limit
- Software always provides fallback path in case of aborts

Hardware Manages All Transactional Writes

Detect Conflicts

Read and write addresses for conflict checking

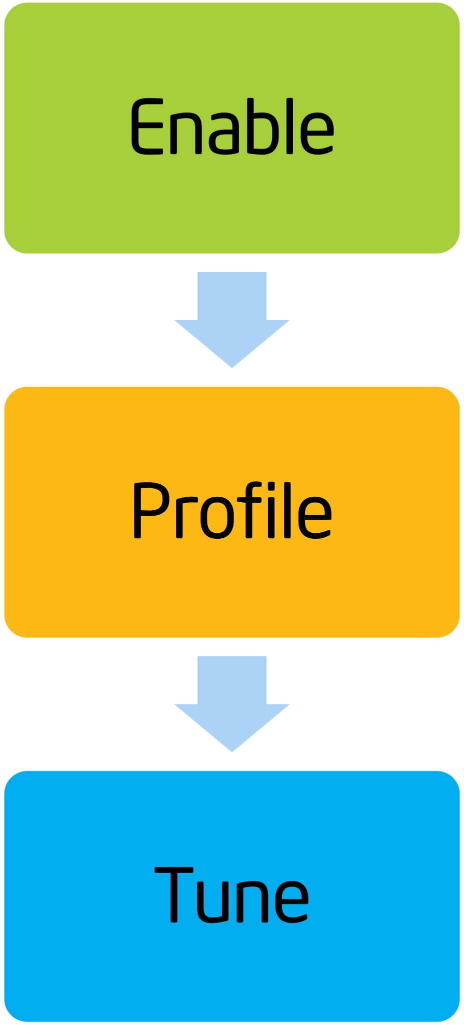
- Tracked at cache line granularity using physical address
- L1 cache tracks addresses written to in transactional region
- L1 cache tracks addresses read from in transactional region
  - Cache may evict address without loss of tracking

Data conflicts

- Occurs if at least one request is doing a write
- Detected at cache line granularity
- Detected using existing cache coherence protocol
- Abort when conflicting access detected

Hardware Automatically Detects Conflicting Accesses

Software



Architected for Enabling Ease

Extensive performance monitoring
and
profiling support

Easy to pin-point problem spots
Low touch changes

Software Enabling and Profiling

Doesn't need operating system changes to use

Compiler support through intrinsics and inline assembly

- Intel® Compiler (ICC) (v13.0), GCC (v4.8), Microsoft* VS2012

Various managed runtimes

- Enabling inside runtime, hidden from application developer

Changes can be localized to libraries

- Augment existing lock library to support Intel® TSX-based elision
- Dynamic linking ➔ no need to recompile (e.g., Linux GLIBC for pthreads (rtm-2.17))

Extensive support for performance monitoring and profiling

- Various TSX specific events and counting modes
- Extensions to Precise Event Based Sampling enables detailed abort profiling

Easy to Get Started with Intel® TSX

Software Considerations

Good coding practices will also help Intel® TSX

- Avoid false or inadvertent sharing
- Avoid timing based synchronization

Most common locks are already elision friendly

- Some locks need effort to make them elision friendly
- RTM provides improved flexibility

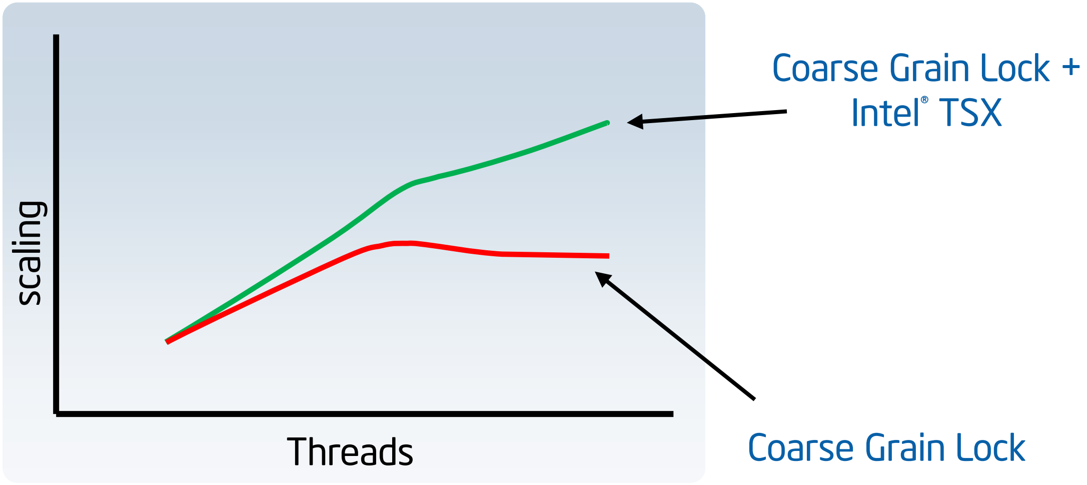
Not everything can or should use Intel® TSX

Intel® TSX is not a magic bullet

Watch for the Programmer Optimization Guide

Applying Intel® TSX

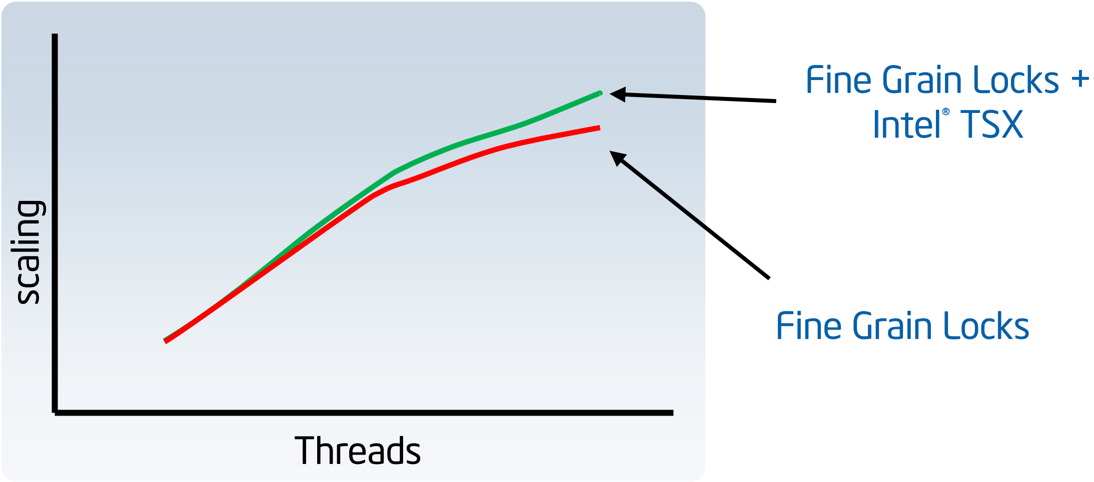
Application with Coarse Grain Lock



Coarse Grain Lock + Intel® TSX

Coarse Grain Lock

Application re-written with Finer Grain Locks



Fine Grain Locks + Intel® TSX

Fine Grain Locks

An example of secondary benefits of Intel® TSX

Fine Grain Behavior at Coarse Grain Effort

Enabling Simpler Algorithms

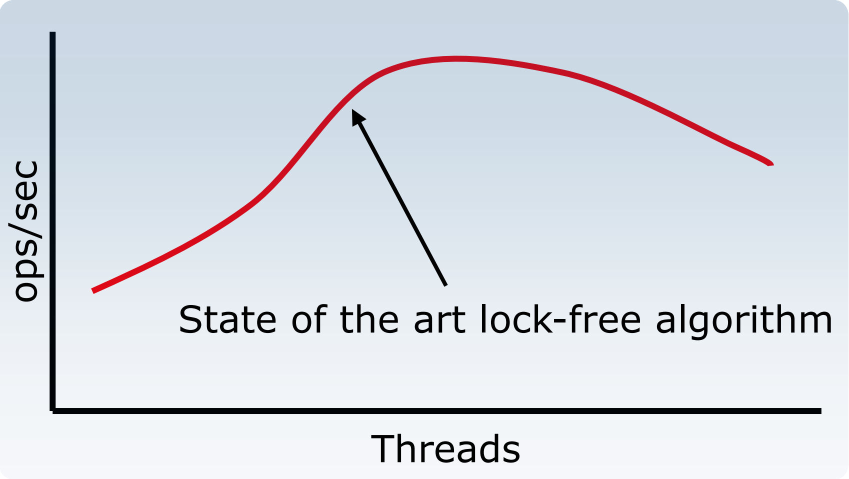
Lock-Free Algorithm

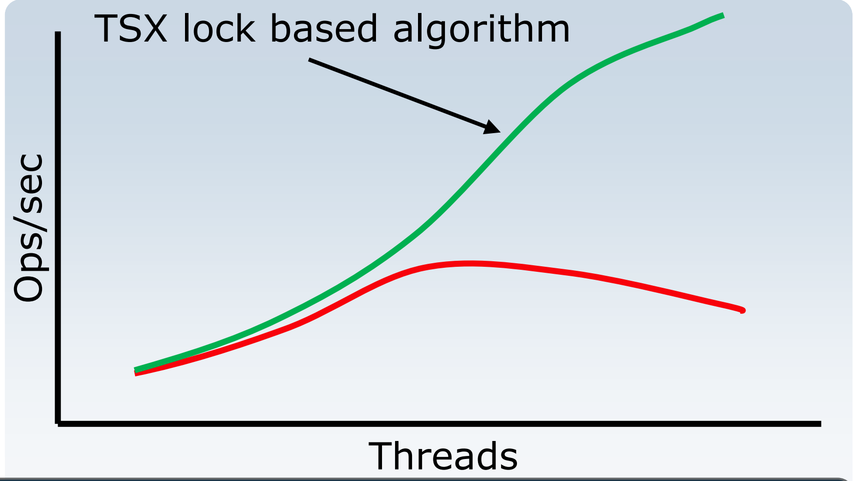
- Don’t use critical section locks
- Developer manages concurrency
- Very difficult to get correct & optimize
  - **Constrain data structure selection**
  - Highly contended atomic operations

Lock-Based + Intel® TSX

- Use critical section locks for ease
- Let hardware extract concurrency
- Enables algorithm simplification
  - **Flexible data structure selection**
  - Equivalent data structure lock-free algorithm very hard to verify

Real World Example





Intel® TSX Can Enable Simpler Scalable Algorithms

A MATTER OF TIME IN A PARALLEL WORLD…

Discussion of Time applies to recent hardware generations

How to Measure Time?

What is time in a modern processor?

Two key components captured in hardware

- Epoch
- Time Stamp Counter (TSC)
  - Invariant TSC: Incremented at a constant rate (in all ACPI P-, C-, and T- states)

Through a software API:

- GETTIMEOFDAY()
- Returns time as we normally understand it (Typically Epoch + TSC)

Directly through the Instruction Set:

- RDTSC{P}
- Returns a (monotonically increasing) value of TSC

Sounds Simple Enough

Three aspects to keep in mind

Accuracy

- How accurate is the crystal oscillator?
- Variance in crystals, EMI countermeasures, … introduce subtle differences
- Don’t want to compare time with something external to the system

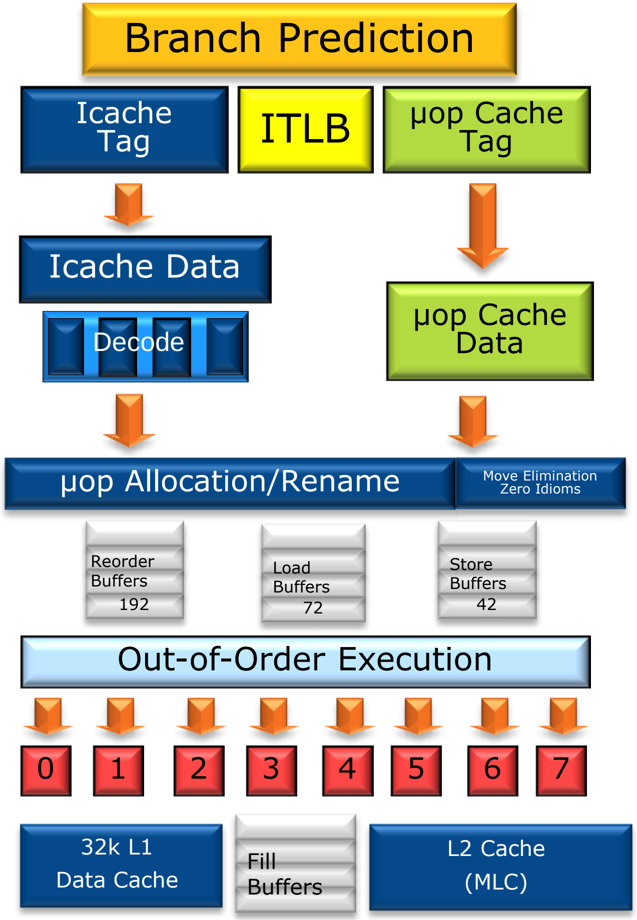
Resolution

- Very high resolution actually… more than software may care…

Timeliness

- When exactly are you reading time?
- This is what is going to trip developers the most

When is the TSC Actually Read?



Read at some point within a range

- Guaranteed to be monotonic
- But is not serializing

What about operations around RDTSC?

- Can proceed completely in parallel
- RDTSCP
  - Wait for older operations
  - But younger ones can go in parallel

Pipeline effects introduce variance

- Unreliable to measure small values
- Measure aggregate loops – not individual

SUMMARY

Summary

Haswell is the next Intel® "tock" microarchitecture

- Scalability across broad range of domains and workloads
- Per core performance for the vast majority of workloads
- Lower power for better performance and smaller envelopes

Developer-friendly features

- Fundamental performance and power improvements for legacy workloads
- New instructions addressing key developer requests
  - Intel® AVX2 with FMA and 256-bit integer vectors
  - Intel® Bit Manipulation Instructions
  - Intel® TSX for thread parallelism through lock elision

Resources

- ISA documentation for Haswell New Instructions
  - Intel® Architecture Instruction Set Extensions Programming Reference (PDF).
  - Intel®64 and IA-32 Architectures Software Developer Manuals.
- Software Developer Emulator (SDE)
  - Emulate new instructions before hardware is available
  - Intel® Software Development Emulator (Intel® SDE) (PDF)
- Intel® Architecture Code Analyzer
  - Code analysis for new instructions before hardware is available
  - Intel® Architecture Code Analyzer
- Intel® Compiler
  - Version 12.1 supports most Haswell New Instructions
  - Version 13.0 supports Intel® TSX
  - Intel® C++ Compiler
- Intel® VTune™ analyzer
  - New release will support Haswell PerfMON shortly after shipment

BACKUP

Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way

All caches use 64-byte lines

Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)