Software Barrier Algorithms

INCOMPLETE
Shows Progress

Outline

- Hardware Background
- Software Barrier Algorithms Implemented
- Intel Sandy Bridge Architecture
- Experiments and Results
- Conclusions

Hardware Background (1/3)

- Correctness Related
  - Read/Write/Full fences required for correct execution
  - Used C/C++11 memory model to abstract the underlying hardware machine memory model
    - Portability: all compilers support the C/C++11 memory model (at least in their newest versions)
    - DETAILS FOR THE MEMORY MODEL OMITTED

Hardware Background (2/3)

- Performance Related
  - Atomic Primitives Used
    - Instead of locks I used the atomics.
    - std::atomic<int> i, declares an atomic integer variable i.
      - i.load(memory_order)/i.store(v,memory_order)/i.exchange(v,memory_order)
      - i.compare_exchange_weak(vold,vnew,memory_order)/...
  - False-Sharing
    - Even if threads use different data variables, the cache ping-pong effect can still arise if those variables lie inside the same cache-line.
    - HARDWARE PREFETCHING NOT YET DEALT WITH
  - Relaxed Model
    - The implementations use the Relaxed Ordering offered by the C/C++ memory model which map efficiently to the underlying memory model by providing fine-grained control over the amount of *ordering* and *synchronization* required around the memory accesses of different threads
    - What does this mean for performance?
      - See next slide

Hardware Background (3/3)

AArch64

C/C++11 Operation	AArch64 implementation
Load Relaxed:	LDR
Load Consume:	LDR + preserve dependencies until next kill_dependency OR LDAR
Load Acquire:	LDAR
Load Seq Cst:	LDAR
Store Relaxed:	STR
Store Release:	STLR
Store Seq Cst:	STLR
Cmpxchng Relaxed:	_loop: ldxr roldval, [rptr]; cmp roldval, rold; bne _exit; stxr rres, rnewval, [rptr]; cbz rres, _loop; _exit
Cmpxchng Acquire:	_loop: ldaxr roldval, [rptr]; cmp roldval, rold; beq _exit; stxr rres, rnewval, [rptr]; cbz rres, _loop; _exit
Cmpxchng Release:	_loop: ldxr roldval, [rptr]; cmp roldval, rold; bne _exit; stlxr rres, rnewval, [rptr]; cbz rres, _loop; _exit
Cmpxchng AcqRel:	_loop: ldaxr roldval, [rptr]; cmp roldval, rold; bne _exit; stlxr rres, rnewval, [rptr]; cbz rres, _loop; _exit
Cmpxchng SeqCst :	_loop: ldaxr roldval, [rptr]; cmp roldval, rold; bne _exit; stlxr rres, rnewval, [rptr]; cbz rres, _loop; _exit
Acquire Fence:	DMB LD
Release Fence:	DMB
AcqRel Fence:	DMB
SeqCst Fence:	DMB

C/C++11 Operation	ARM implementation
Load Relaxed:	ldr
Load Consume:	ldr + preserve dependencies until next `kill_dependency` *OR* ldr; teq; beq; isb *OR* ldr; dmb
Load Acquire:	ldr; teq; beq; isb *OR* ldr; dmb
Load Seq Cst:	ldr; dmb
Store Relaxed:	str
Store Release:	dmb; str
Store Seq Cst:	dmb; str; dmb
Cmpxchg Relaxed (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop
Cmpxchg Acquire (32 bit):	_loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg Release (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop;
Cmpxchg AcqRel (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Cmpxchg SeqCst (32 bit):	dmb; _loop: ldrex roldval, [rptr]; mov rres, 0; teq roldval, rold; strexeq rres, rnewval, [rptr]; teq rres, 0; bne _loop; isb
Acquire Fence:	dmb
Release Fence:	dmb
AcqRel Fence:	dmb
SeqCst Fence:	dmb

IBM Power / PowerPC

C/C++11 Operation	Power implementation
Load Relaxed:	ld
Load Consume:	ld + preserve dependencies until next `kill_dependency` *OR* ld; cmp; bc; isync
Load Acquire:	ld; cmp; bc; isync
Load Seq Cst:	hwsync; ld; cmp; bc; isync
Store Relaxed:	st
Store Release:	lwsync; st
Store Seq Cst:	hwsync; st
Cmpxchg Relaxed (32 bit):	_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:
Cmpxchg Acquire (32 bit):	_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:
Cmpxchg Release (32 bit):	lwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:
Cmpxchg AcqRel (32 bit):	lwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit
Cmpxchg SeqCst (32 bit):	hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit
Acquire Fence:	lwsync
Release Fence:	lwsync
AcqRel Fence:	lwsync
SeqCst Fence:	hwsync

Itanium

C/C++11 Operation	IA64 implementation
Load Relaxed:	ld.acq
Load Consume:	ld.acq
Load Acquire:	ld.acq
Load Seq_Cst:	ld.acq
Store Relaxed:	st.rel
Store Release:	st.rel
Store Seq Cst:	st.rel; mf
Cmpxchg Acquire:	cmpxchg.acq
Cmpxchg Release:	cmpxchg.rel
Cmpxchg AcqRel:	cmpxchg.rel; mf
Cmpxchg SeqCst:	cmpxchg.rel; mf
Consume Fence:	<ignore>
Acquire Fence:	<ignore>
Release Fence:	<ignore>
Acq_Rel Fence:	<ignore>
Seq_Cst Fence:	mf

x86 (including x86-64)

C/C++11 Operation	x86 implementation
Load Relaxed:	MOV (from memory)
Load Consume:	MOV (from memory)
Load Acquire:	MOV (from memory)
Load Seq_Cst:	MOV (from memory)
Store Relaxed:	MOV (into memory)
Store Release:	MOV (into memory)
Store Seq Cst:	(LOCK) XCHG // alternative: MOV (into memory),MFENCE
Consume Fence:	<ignore>
Acquire Fence:	<ignore>
Release Fence:	<ignore>
Acq_Rel Fence:	<ignore>
Seq_Cst Fence:	MFENCE

Software Barrier Algorithms Implemented

- Centralized Sense Reversing Barrier
- Static Tree Barrier
  - Arrival Tree
  - Departure Tree
- Static Tree Barrier with Global Departure
  - Arrival Tree
  - Global atomic boolean flag for departure
    - Like the departure phase of the centralized sense-reversing barrier algorithm
- Dissemination Barrier
  - NOT YET

Intel Sandy Bridge Architecture

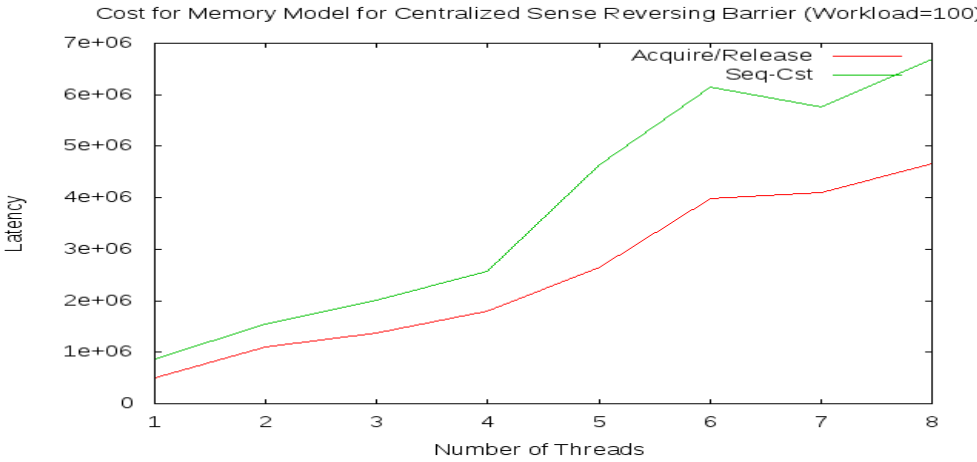
- OMITTED

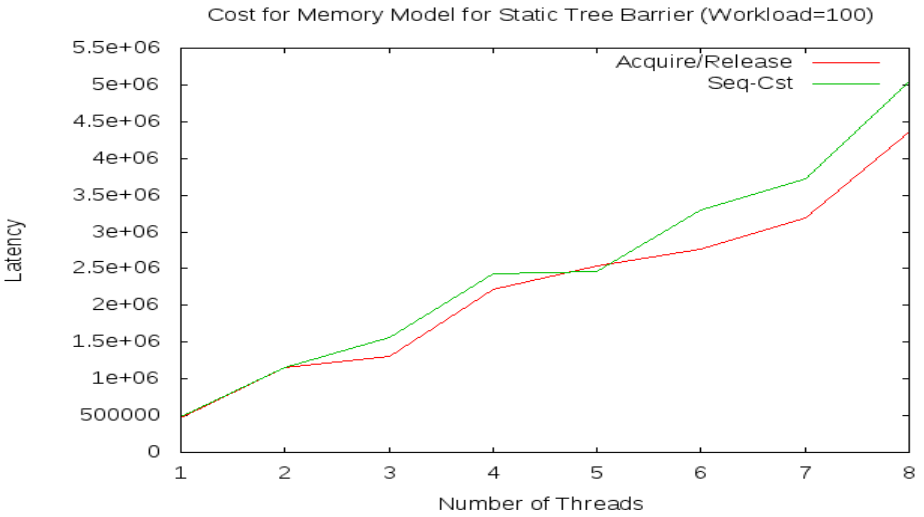
Experiments and Results (1/5)

- Thread mapping:
  - First fill first thread inside each core from left to right
  - Then fill the second thread inside each core from left to right
- Experimental Process:
  - 10.000 barrier episodes. Each thread makes a random workload of 100 counter increments between episodes.
  - Repeat 30 times and measure (lower,mean,upper) *latency* with a confidence level of 99.9%

Experiments and Results (2/5)

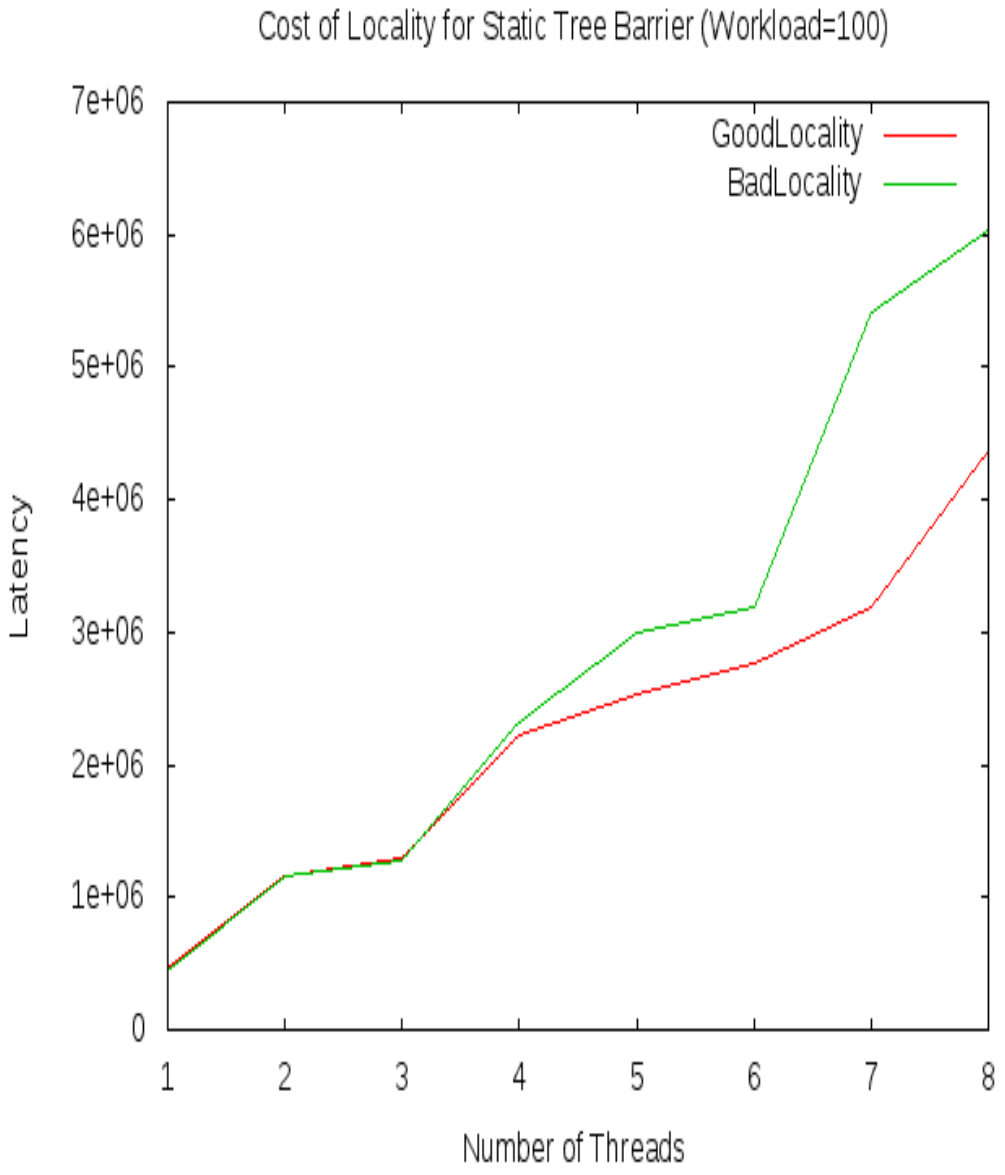
- I first studied the effect of the relaxed memory model over the sequentially consistent one.
  - The cost of **mfence** for Intel
  - Relaxed is better!





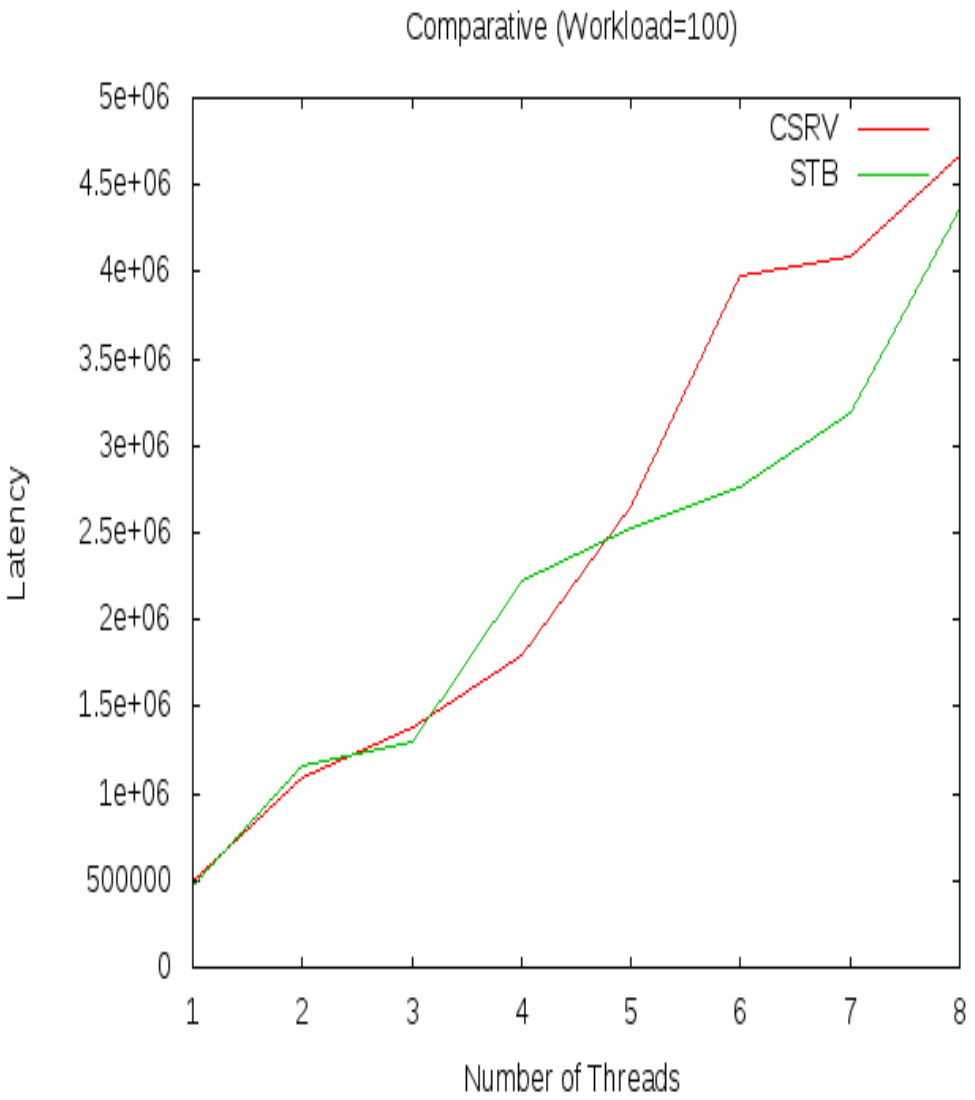
Experiments and Results (3/5)

- Then I studied the effect of *locality* for the static tree barrier
  - Locality matters!



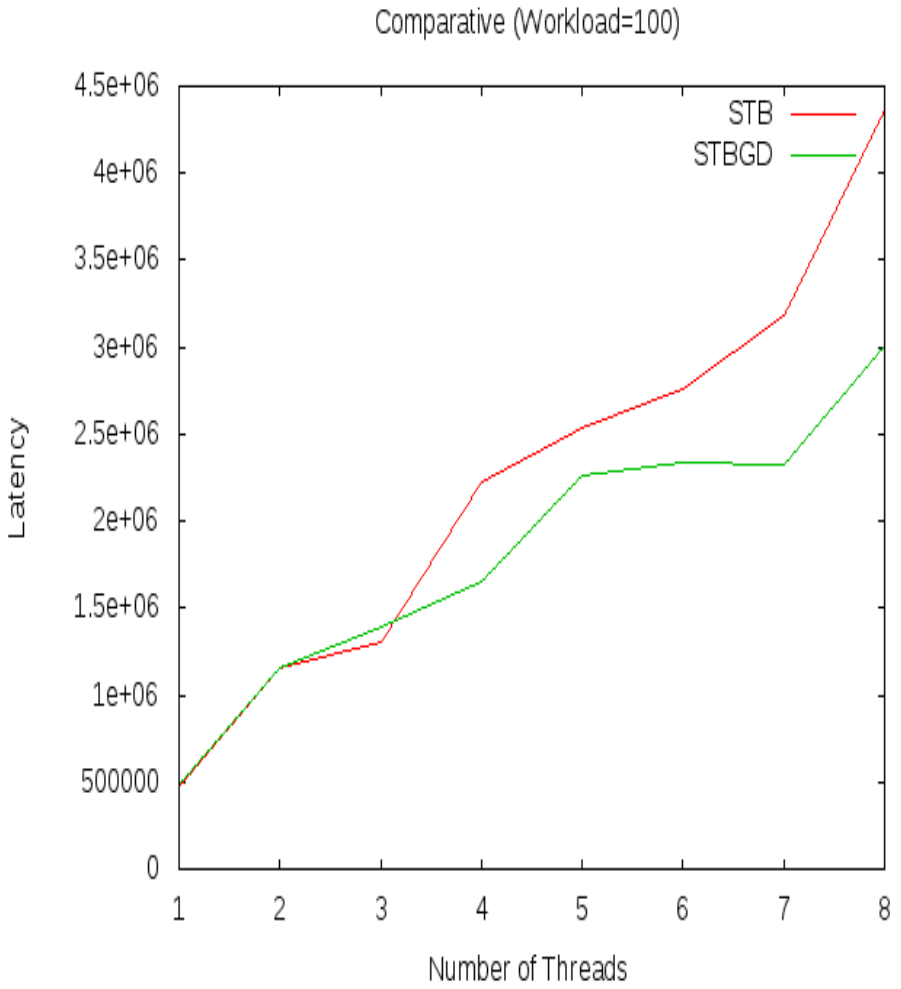
Experiments and Results (4/5)

- How do the Centralized Sense Reversing Barrier and Static Tree Barrier compare against?
  - Best versions
    - Relaxed+good-locality
    - Static Tree Barrier is better!



Experiments and Results (5/5)

- Surely the static tree barrier benefits from the *parallel* arrival phase versus the sequential bottleneck of the *shared counter* in the centralized sense reversing tree.
- But what about the departure phase?
  - I compare the previous static tree barrier with a static tree barrier whose global departure phase is implemented using a global sense flag
  - The version with the global departure flag is better.
    - The *ring interconnect* has to do something with it?
  - Further improvement over the centralized sense reversing barrier!



Conclusions

- OMITTED