Lynx: Using OS and Hardware Support for Fast Fine-Grained Inter-Core Communication

Konstantina Mitropoulou, Vasileios Porpodas, Xiaochun Zhang and Timothy M. Jones

Computer Laboratory

ICS 2016, Istanbul
Outline

• Background:
  • Lamport’s queue
  • Multi-section queue

• Lynx queue

• Performance evaluation
Lamport’s Queue Bottlenecks

- Frequent thread synchronisation
- Cache ping-pong
Lamport’s Queue Bottlenecks

while (next_enqueue_ptr == dequeue_ptr) {
      
}
Lamport’s Queue Bottlenecks

while(next_enqueue_ptr == dequeue_ptr) {
    ;
}

Performance degradation due to:

- Frequent thread synchronisation
- Cache ping-pong
Lamport’s Queue Bottlenecks

while (next_enqueue_ptr == dequeue_ptr) {
    ;
}

Performance degradation due to:

- Frequent thread synchronisation
Lamport’s Queue Bottlenecks

Performance degradation due to:
- Frequent thread synchronisation
- Cache ping-pong

while (next_enqueue_ptr == dequeue_ptr) {
    
}
Cache Ping-Pong

while (next_enqueue_ptr == dequeue_ptr) {
  ;
}
Cache Ping-Pong

while (next_enqueue_ptr == dequeue_ptr) {
    
    • Queue pointers ping-pong across cache hierarchy

}
Cache Ping-Pong

while (next_dequeue_ptr == enqueue_ptr) {
    
    • Queue pointers ping-pong across cache hierarchy
Multi-Section Queue (MSQ): state-of-the-art

- Each section is exclusively used by one thread

| section 1 | section 2 |
Multi-Section Queue (MSQ): state-of-the-art

- Each section is exclusively used by one thread
Multi-Section Queue (MSQ): state-of-the-art

- Enqueue thread cannot access section 1 because dequeue thread still uses it
Multi-Section Queue (MSQ): state-of-the-art

- Enqueue thread cannot access section 1 because dequeue thread still uses it
- Enqueue thread waits (spins) at the end of section 2
Multi-Section Queue (MSQ): state-of-the-art

- Dequeue thread reached the end of section 1
Multi-Section Queue (MSQ): state-of-the-art

- Dequeue thread reached the end of section 1
- Enqueue thread enters section 1
Multi-Section Queue (MSQ): state-of-the-art

Performance optimisations:

- Infrequent boundary checks (less frequent synchronisation)
- Reduced cache ping-pong
Multi-Section Queue (MSQ): state-of-the-art

Performance optimisations:

- Infrequent boundary checks (less frequent synchronisation)
Multi-Section Queue (MSQ): state-of-the-art

Performance optimisations:

- Infrequent boundary checks (less frequent synchronisation)
- Reduced cache ping-pong
MSQ Control-Flow Graph and Internals

enqueue function

1 -> 2 -> 3 -> 4 -> 5 -> 6

1 -> 2 -> 3 -> 4 -> 5

dequeue function

1 -> 2 -> 3 -> 4 -> 5

http://www.cl.cam.ac.uk/~km647/
MSQ Control-Flow Graph and Internals

enqueue function

enqueue
MSQ Control-Flow Graph and Internals

enqueue function

enqueue

synchronisation code

enqueue
enqueue function

enqueue

checks if next section is free

synchronisation code

enqueue
MSQ Control-Flow Graph and Internals

enqueue function

enqueue

spin loop

checks if next section is free

synchronisation code

enqueue function
MSQ Control-Flow Graph and Internals

enqueue function

enqueue

spin loop

update local variables

synchronisation code

checks if next section is free
enqueue function

enqueue

spin loop

update local variables

update shared variable

checks if next section is free

synchronisation code
MSQ Control-Flow Graph and Internals

enqueue function

enqueue

spin loop

update local variables

update shared variable

synchronisation code

join basic-block

checks if next section is free
MSQ Control-Flow Graph and Internals

dequeue_ptr

enqueue_ptr

section 1

section 2

synchronisation code
enqueue function

enqueue

synchronisation code

1

2

lea rax, [rdx+8]

3

mov QWORD PTR [rdx], rcx

4

mov rdx, rax

5

and rdx, ROTATE MASK

6
test eax, SECTION_MASK

jne .L2

http://www.cl.cam.ac.uk/~km647/
enqueue function

1. enqueue
2. lea rax, [rdx+8]
3. mov QWORD PTR [rdx], rcx
4. mov rdx, rax
5. and rdx, ROTATE MASK
6. test eax, SECTION_MASK
7. jne .L2

synchronisation code

slide 13 of 30
http://www.cl.cam.ac.uk/~km647/
enqueue function

1
lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2

2

3
store

4
incr pointer

5

6
slide 13 of 30
http://www.cl.cam.ac.uk/~km647/
enqueue function

1. lea rax, [rdx+8]
2. mov QWORD PTR [rdx], rcx
3. mov rdx, rax
4. and rdx, ROTATE MASK
5. test eax, SECTION_MASK
6. jne .L2

enqueue

synchronisation code

incr pointer
store
compiler’s copy
incr pointer
store
compiler’s copy
enqueue function

synchronisation code

enqueue

lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2

increment pointer

store

compiler’s copy

rotate pointer

MSQ Control-Flow Graph and Internals

slide 13 of 30

http://www.cl.cam.ac.uk/~km647/
enqueue function

1. enqueue
2. lea rax, [rdx+8]
3. mov QWORD PTR [rdx], rcx
4. mov rdx, rax
5. and rdx, ROTATE MASK
6. test eax, SECTION_MASK
7. jne .L2

synchronisation code

- incr pointer
- store
- compiler’s copy
- rotate pointer
- end of section

end of section
rotate pointer
compiler’s copy
store
incr pointer
enqueue function

1

enqueue

2
synchronisation code

lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2

3

4

5

6

skip sync code
end of section
rotate pointer
compiler’s copy
store
incr pointer
Optimal Queue

Optimal queue features:
- infinite size
Optimal Queue

Optimal queue features:

- infinite size
- 2 instructions overhead
  1. pointer increment
  2. store into the queue
Lynx: Just 2 instructions overhead

Lynx removes part of enqueue (boundary checks) and all the synchronisation overhead off the critical path
Lynx(1): H/W triggered Synchronisation

enqueue function

enqueue

synchronisation code

lea rax, [rdx+8]

mov QWORD PTR [rdx], rcx

mov rdx, rax

and rdx, ROTATE MASK

test eax, SECTION_MASK

jne .L2

http://www.cl.cam.ac.uk/~km647/
Lynx(1): H/W triggered Synchronisation

enqueue function

1. lea rax, [rdx+8]
2. mov QWORD PTR [rdx], rcx
3. mov rdx, rax
4. and rdx, ROTATE MASK
5. test eax, SECTION_MASK
6. jne .L2

enqueue
Lynx(1): H/W triggered Synchronisation

enqueue function

enqueue

synchronisation code

lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2

http://www.cl.cam.ac.uk/~km647/
Lynx(1): H/W triggered Synchronisation

- A red zone is a non-read and non-write part of memory
- SSRZ: Section Synchronisation Red-Zone
Lynx(1): H/W triggered Synchronisation

- A red zone is a non-read and non-write part of memory
Lynx(1): H/W triggered Synchronisation

section 1 section 2

SSRZ SSRZ

- SSRZ: Section Synchronisation Red-Zone
Lynx(1): H/W triggered Synchronisation

- Whether the SIGSEGV is from the queue or the system
- Which thread raised the exception
- If the thread is in section 1 or 2
- If the next section is free

enqueue_ptr

section 1

SSRZ

section 2

SSRZ

dequeue_ptr
Lynx(1): H/W triggered Synchronisation

- dequeue_ptr
- enqueue_ptr
- section 1
- section 2
- SSRZ
- SSRZ

- whether the SIGSEGV is from the queue or the system
- which thread raised the exception
- if the thread is in section 1 or 2
- if the next section is free
Lynx(1): H/W triggered Synchronisation

- Whether the SIGSEGV is from the queue or the system
- Which thread raised the exception
- If the thread is in section 1 or 2
- If the next section is free
Lynx(1): H/W triggered Synchronisation

Lynx’s handler checks:

- whether the SIGSEGV is from the queue or the system
- which thread raised the exception
- if the thread is in section 1 or 2
- if the next section is free
Lynx(1): H/W triggered Synchronisation

Lynx’s handler checks:

• whether the SIG_SEGV is from the queue or the system
Lynx(1): H/W triggered Synchronisation

Lynx’s handler checks:

- whether the SIG_SEGV is from the queue or the system
- which thread raised the exception
Lynx(1): H/W triggered Synchronisation

Lynx’s handler checks:

- whether the SIG_SEGV is from the queue or the system
- which thread raised the exception
- if the thread is in section 1 or 2
Lynx(1): H/W triggered Synchronisation

Lynx’s handler checks:

- whether the SIG_SEGV is from the queue or the system
- which thread raised the exception
- if the thread is in section 1 or 2
- if the next section is free
Lynx(1): H/W triggered Synchronisation

- The dequeue thread still uses the first section.
- The enqueue thread waits at the end of the second section and adds a new red zone.
- The new red zone is part of the synchronisation and is temporarily added.
Lynx(1): H/W triggered Synchronisation

- The dequeue thread has finished with the first section
- The enqueue thread removes the second red zone and it enters the first section
Lynx(2): H/W triggered Pointer Rotation

enqueue function

enqueue

synchronisation code

lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2
Lynx(2): H/W triggered Pointer Rotation

enqueue function

enqueue

synchronisation code

lea rax, [rdx+8]
mov QWORD PTR [rdx], rcx
mov rdx, rax
and rdx, ROTATE MASK
test eax, SECTION_MASK
jne .L2
Lynx(2): H/W triggered Pointer Rotation

- SSRZ: Section Synchronisation Red-Zone
Lynx(2): H/W triggered Pointer Rotation

- SSRZ: Section Synchronisation Red-Zone
- PRRZ: Pointer Rotation Red-Zone
Lynx(2): H/W triggered Pointer Rotation

- SSRZ: Section Synchronisation Red-Zone
- PRRZ: Pointer Rotation Red-Zone
Lynx(2): H/W triggered Pointer Rotation

- SSRZ: Section Synchronisation Red-Zone
- PRRZ: Pointer Rotation Red-Zone
Lynx(2): H/W triggered Pointer Rotation

- SSRZ: Section Synchronisation Red-Zone
- PRRZ: Pointer Rotation Red-Zone
Lynx(2): H/W triggered Pointer Rotation

Two types of red-zones:

- SSRZ (Synchronisation Red-Zone)
- PRRZ (Pointer Rotation Red-Zone)
Lynx(2): H/W triggered Pointer Rotation

Two types of red-zones:

1. moving red-zone: SSRZ (Section Synchronisation Red-Zone)

SSRZ

SSRZ PRRZ

enqueue_ptr

decqueue_ptr

section 1

section 2
Two types of red-zones:

1. moving red-zone: SSRZ (Section Synchronisation Red-Zone)
2. fixed red-zone: PRRZ (Pointer Rotation Red-Zone)
Experimental Setup

- Implementation in C++ with inline assembly
Experimental Setup

- Implementation in C++ with inline assembly
- Evaluation on a wide range of machines: from embedded SOCs to server CPUs
Experimental Setup

- Implementation in C++ with inline assembly
- Evaluation on a wide range of machines: from embedded SOCs to server CPUs
- Throughput experiments for a wide range of queue sizes
Experimental Setup

- Implementation in C++ with inline assembly
- Evaluation on a wide range of machines: from embedded SOCs to server CPUs
- Throughput experiments for a wide range of queue sizes
- Absolute throughput performance in GB/s
Throughput (GB/s) on Intel core-i5

Throughput for 64bit Memory Instr. (Core-i5 4570)
Breakdown of Lynx Overheads

The image shows a breakdown of Lynx Overheads across different queue sizes. The graph displays the percentage execution time for various categories such as real, kernel, sync, handler, and other. The queue sizes range from 64KB to 256MB, and the execution time is represented on the y-axis as a percentage.

For example, at 64KB queue size, the execution time is distributed as follows:
- Real: 10%
- Kernel: 5%
- Sync: 30%
- Handler: 20%
- Other: 35%

The graph visually represents how different queue sizes affect the execution time across these categories.
Throughput (GB/s) on Various Machines

Throughput for 64bit Memory Instr. (Xeon E5-2667v2)

Throughput for 64bit Memory Instr. (Opteron 6376)

Throughput for 64bit Memory Instr. (Core-i3 2367M)

Throughput for 64bit Memory Instr. (Celeron J1900)
Real World Applications on Intel Xeon

- The best queue configuration with Lynx is better than the best with MSQ
Conclusion

• Proposed Lynx: a lock-free SP/SC software queue with just 2 instructions overhead
Conclusion

• Proposed Lynx: a lock-free SP/SC software queue with just 2 instructions overhead
• Relies on existing commodity H/W and O/S support for memory protection
Conclusion

- Proposed Lynx: a lock-free SP/SC software queue with just 2 instructions overhead
- Relies on existing commodity H/W and O/S support for memory protection
- The overhead of synchronisation and boundary checking is moved to the exception handler
Conclusion

- Proposed Lynx: a lock-free SP/SC software queue with just 2 instructions overhead
- Relies on existing commodity H/W and O/S support for memory protection
- The overhead of synchronisation and boundary checking is moved to the exception handler
- Throughput increases by up to 57%
Source Code

https://www.cl.cam.ac.uk/~km647/papers/lynx/lynxQ.tar.bz2

or

https://www.repository.cam.ac.uk/handle/1810/254651