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

Konstantina Mitropoulou, Vasileios Porpodas[1], Xiaochun Zhang and Timothy M. Jones
Computer Laboratory
University of Cambridge, UK
firstname.lastname@cl.cam.ac.uk

ABSTRACT

Designing high-performance software queues for fast inter-core communication is challenging, but critical for maximising software parallelism. State-of-the-art single-producer / single-consumer queues for streaming applications contain multiple sections, requiring the producer and consumer to operate independently on different sections from each other. While these queues perform well for coarse-grained data transfers, they perform poorly in the fine-grained case.

This paper proposes Lynx, a novel SP/SC queue, specifically tuned for fine-grained communication. Lynx is built from the ground up, reducing the generated code on the critical-path to just two operations per enqueue and dequeue. To achieve this it relies on existing commodity processor hardware and operating system exception handling support to deal with infrequent queue maintenance operations. Lynx outperforms the state-of-the art by up to 1.57× in total 64-bit throughput reaching a peak throughput of 15.7GB/s on a common desktop system. Real applications using Lynx get a performance improvement of up to 1.4×.

CCS Concepts

•**Software and its engineering** → **Buffering;**

Keywords

Single-Producer / Single-Consumer Software Queue, Fine-grained Communication, Hardware Exceptions

1. INTRODUCTION

High-performance parallel applications rely on fast inter-core communication to share data between tasks. Existing commodity processors implement cache coherence protocols to maintain a consistent shared memory for this purpose. Software then builds upon this memory model with data structures that facilitate the transfer. For asynchronous communication of large volumes of data, software queues are the most common programming abstraction, which provide a first-in first-out (FIFO) buffer with simple enqueue and dequeue operations. However, building high-performance software queues has proven to be a major challenge and there has been significant work on improving their efficiency [8, 14, 15, 22, 24, 25].

Single-producer / single-consumer (SP/SC) queues are a subset of the generic multiple-producer / multiple-consumer (MP/MC) model. This more specialised type of queue is widely used to aid pipeline parallelism, where each stage of the pipeline produces data for the next. The performance of the SP/SC queue is critical in determining the amount and granularity of parallelism that can be extracted. Even the fastest state-of-the-art queues have a prohibitively high overhead for transferring small amounts of data. To amortise this cost, programmers typically only use a queue for coarse-grained communication, thus limiting the potential for parallelism. However, there are application domains that rely on extremely fast SP/SC queues for fine-grained data transfers, such as automatic parallelization [9, 20], software-based error detection [24, 25, 28] and fast line-rate network traffic monitoring [16].

SP/SC queues require numerous major innovations to achieve high performance. These involved lock-free implementations [7, 14]; minimising or completely avoiding frequent bidirectional inter-core communication (a.k.a. cache ping-pong [11]) of the queue control variables [8, 15, 16, 24, 25]; and avoiding cache thrashing using specialised non-temporal memory instructions [11]. However, our analysis of a state-of-the-art queue shows that there is still performance left on the table.

In this work we propose a new SP/SC queue design that provides extremely fast inter-core communication even at a very fine granularity. We show that current queues display poor fine-grained performance due to the execution of infrequently-required code for producer/consumer synchronisation, and for reaching the end of the queue. We then develop Lynx, a novel architecture that makes use of existing processor hardware and operating system support for exception handling to minimise enqueue and dequeue operation overheads. Moving code off the critical path realises performance benefits of up to 1.57× compared to a state-of-the-art queue.

In the following sections we first provide an overview of the existing queue designs (section 2), then show the performance bottleneck of the state-of-the-art (section 3). Section 4 provides a detailed description of Lynx, which we

[1]Currently at Intel, Santa Clara.

ICS '16, June 01-03, 2016, Istanbul, Turkey
 ISBN 978-1-4503-4361-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2925426.2926274

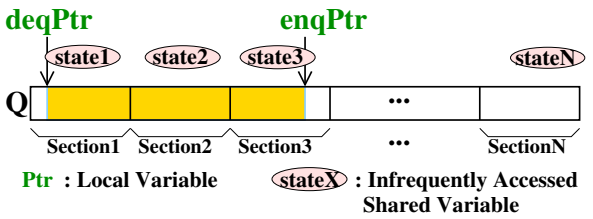


Figure 1: State-of-the-art lock-free multi-section queue.

evaluate in section 5. We then present related work (section 6) and section 7 concludes.

2. MULTI-SECTION QUEUE FOR STREAMING

We start with an overview of the current state-of-the-art queue implementation to provide insights into the overheads.

Multi-section lock-free SP/SC queues are designed for streaming, specialised compared to generic SP/SC queues so they are tuned for high throughput. The queue is divided into sections and only one thread (producer or consumer) is allowed to access each section at any time. Synchronisation occurs only at section boundaries. The multi-section queue is lock-free by design: both producer and consumer access the queue simultaneously without locking, provided that they do not access the same section [16]. This multi-section design solves many generic SP/SC performance problems, such as cache ping-pong and false sharing [11, 15, 16, 24].

Figure 1 shows an example and listing 1 gives the code. The queue shown uses *lazy synchronisation* [24, 25] for synchronising across sections (lines 5 to 10 and 15 to 20). Overall, an efficient implementation of a multi-section queue must address a number of challenges.

Queue Size.

In general, the larger the queue, the larger the sections and the smaller the amount of synchronisation required. However, once the queue is larger than the last level cache, performance drops because the data gets invalidated before being read and the dequeue thread must obtain it from main memory instead. Figure 2 shows the throughput for increasing queue sizes when using `mov` instructions to write data, and three other schemes (described in the following paragraphs). Our Intel Core i5-4570 evaluation system (more details in section 5), contains a 32KB first level cache so, counter-intuitively, there is a large performance boost once the queue is too large for the L1. This is because the threads evict their own data from their L1, meaning the other thread gets the cache line from the private L2, which is lower latency compared to another core's L1. Further, the last level cache is 6MB, so performance drops off when the queue is 8MB or larger because data is evicted to slow main memory.

Cache Thrashing.

When an application's working set fits into the cache, the queue should avoid evicting it which prevents later cache misses and performance loss. This can be achieved by replacing regular store instructions into the queue with non-temporal stores, as in the Liberty queues [11]. These instructions write directly to memory, bypassing the caches altogether and removing the problem. In x86-64 (SSE extensions) the non-temporal store instruction is the non-sequentially-consistent `movnti` [4].

```
1  void enqueue (queue_t q, long data) {
2    *q->enqPtr = data;
3    q->enqPtr = (q->enqPtr + 8) & ROTATE_MASK;
4    /* Synchronisation */
5    if ((q->enqPtr & SECTION_MASK) == 0) {
6      while (q->enqPtr == q->deqLocalPtr) {
7        q->deqLocalPtr = q->deqSharedPtr;
8      }
9      q->enqSharedPtr = q->enqPtr;
10   }
11 }
12
13 long dequeue(queue_t q) {
14   /* Synchronisation */
15   if ((q->deqPtr & SECTION_MASK) == 0) {
16     q->deqSharedPtr = q->deqPtr;
17     while (q->deqPtr == q->enqLocalPtr) {
18       q->enqLocalPtr = q->enqSharedPtr;
19     }
20   }
21   long data = *((long *)q->deqPtr);
22   q->deqPtr = (q->deqPtr + 8) & ROTATE_MASK);
23   return data;
24 }
```

Listing 1: State-of-the-art multi-section queue [24, 25].

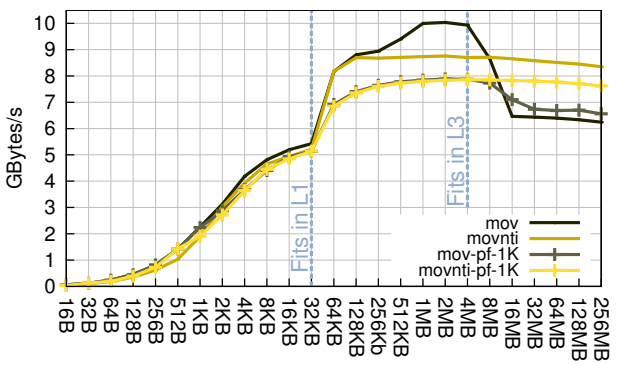


Figure 2: Throughput exploration for a state-of-the-art 2-section queue on an Intel Core i5-4570 using common `mov` instructions and non-temporal `movnti`, both with and without software prefetching.

The impact of `movnti` instructions is shown in figure 2. Its performance is largely unaffected by the cache size once the queue is too large for the L1 because the data to the dequeue thread is fetched straight from main memory. On the other hand, its performance is limited by the memory bandwidth available, and so is slower than a normal `mov` when the bandwidth is saturated but the queue fits in the last level cache (i.e., from 128KB to 4MB in this experiment). This trend is also shown in figure 9 (section 5.2). Overall it is up to the programmer to determine the correct instruction to use given their program's characteristics.

Prefetching.

Liberty queues [11] propose using software prefetch instructions in the dequeue function code. Figure 2 shows the throughput of dequeue with a software prefetching distance of 1024 bytes, labelled mov-pf-1K and movnti-pf-1K. Prefetching leads to 10% lower peak performance compared to `mov` alone, but slightly better performance for queue sizes

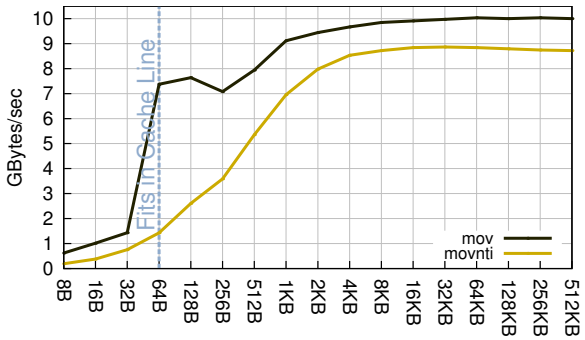


Figure 3: Throughput exploration as we increase the section size of a 1MB queue from 8B (128K sections) to 512KB (2 sections) on an Intel Core i5-4570.

larger than 8MB. The throughput of `movnti` with prefetching is about 10% lower than using `movnti` alone because the prefetch instructions simply add to the already-saturated off-chip memory bandwidth. Again, the use of software prefetch is orthogonal to the queue design and left for the programmer to determine.

Section Size.

The size of each section has an impact on the amount of data each thread can push or pop until hitting the section "owned" by the other thread. Larger sections mean that synchronisation is infrequent, but are less efficient at dealing with bursty queue usage from either thread. With larger sections, there is less room between enqueue and dequeue to absorb bursts.

Figure 3 shows the throughput of a 1MB queue as we increase the section size. For sections smaller than a cache-line (64 bytes in this case), performance is dominated by false sharing between enqueue and dequeue threads as they may both access the same cache line, even though they are in different sections. For larger sections the queue's performance is influenced by the overhead of synchronisation, but this becomes negligible for section sizes of 8KB and higher. When using `movnti`, performance is dominated by the fence instructions required at the end of each section that maintain correctness (i.e., between the spin-loop and the instruction that signals the other thread, in listing 1 line 9), so its performance increases steadily with the section size.

Software implementation.

An efficient implementation must ensure that global variables frequently modified by each individual thread, but rarely read by the other, end up in different cache lines, in order to avoid false sharing. Therefore the `enqPtr` and `deqPtr` (listing 1) should be declared in the code with sufficient padding, large enough to guarantee that they map to different cache lines.

3. ANALYSIS OF REMAINING OVERHEADS

A multi-section lock-free queue with infrequent accesses to the queue's shared synchronisation control variables is the state-of-the-art design. The performance bottleneck of this queue is no longer false sharing or inter-core communication of the control variables for synchronisation, but rather the boilerplate code for the enqueue and dequeue operations which is responsible for checking whether the thread has reached the end of the current section and moving the thread's pointer back to the beginning of the queue once it reaches the end. This boilerplate code becomes a significant overhead in the context of frequent fine-grained queue transfers. This section studies these overheads, motivating the need for a new queue implementation.

```
1 lea rax, [rdx+8]          ;Increment pointer
2 mov QWORD PTR [rdx], rcx  ;Store to queue
3 mov rdx, rax              ;Compiler's copy
4 and rdx, ROTATE_MASK      ;Rotate pointer
5 test eax, SECTION_MASK    ;End of section
6 jne .L2                   ;Skip sync code
```

Listing 2: Multi-section critical path in x86-64 assembly.

3.1 Critical Path Code

Listing 2 shows the most frequently-executed code for an enqueue in the multi-section queue (dequeue is very similar). This code was generated by the GCC-4.8.2 compiler [1] and is x86-64 assembly (Intel's dialect where the output is the leftmost operand). The rest of the code (not shown here) is 10 instructions long and has non-trivial control-flow (that includes the spin-loop).

The first two instructions are fundamental to the operation of enqueue (`mov` and `lea`, lines 1 and 2). They increment the queue pointer to the next location and store the data into the queue (into the old location—note the destination `rax` in line 1 but source `rdx` in line 2). The following `and` instruction (line 4) performs rotation of the index, once it reaches the end of the queue. This can be done in one instruction with an AND mask because the queue size is a power of 2 and is aligned in memory at a multiple of the queue size. Next, `test` (line 5) checks whether the thread is at the end of the section and sets the flag for the conditional jump that follows. The end of section is critical for the queue as it is the point where synchronisation happens. The `jne` (line 6) will fall through if the index is at the end of the section to execute the synchronisation code (not shown). This happens rarely, given that the section is large, so synchronisation is not on the critical path. In the common case, where the pointer is not at the end of the section, the `jne` will skip synchronisation and continue executing the application code after the enqueue.

The instruction in line 3 (`mov rdx, rax`) is a copy for performance optimisation (created by both GCC [1] and LLVM [3]). It allows the instructions in lines 1 and 2 to execute in parallel, as well as those in lines 4 and 5. Without this copy there would be one fewer instruction, but only lines 1 and 2 could execute in parallel, meaning less instruction-level parallelism (and they would have to execute in a different order: `mov`, `lea`, `test`, `jne`, `and`).

3.2 Performance Enhancements

Examination of the enqueue instructions shows that the last three (rotating the pointer, checking for the end of a section, and skipping the synchronisation code) are there to perform infrequent actions. If we had alternative mechanisms to perform these tasks only when needed, we would reduce the enqueue instructions down to the absolute mini-

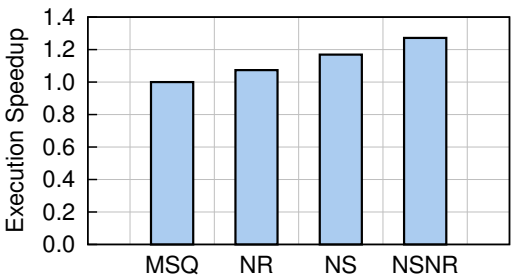


Figure 4: Removing index rotation (NR), synchronisation (NS) and both (NRNS) from the enqueue function in a 256MB queue on an Intel Core i5-4570.

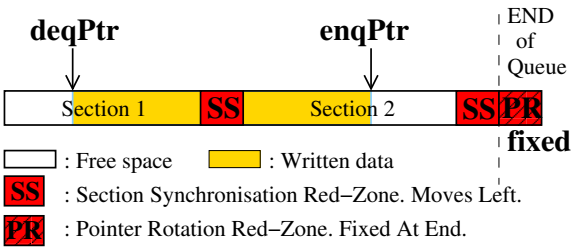


Figure 5: Memory layout in Lynx.

mum: just the store and the increment of the index.

To quantify the benefits of removing these overheads we measured the performance of a hypothetical queue with these infrequently-required instructions removed from the code[1]. We measured the execution time of the baseline state-of-the-art queue [24] ("MSQ") and three overhead-removing optimisations. It is important to note that for these experiments we simply removed the corresponding assembly instructions from the code. Therefore, they do not account for any optimisations that the compiler could perform on the code with these enhancements.

Removing Pointer Rotation Overhead (NR).

The first optimisation is to remove the pointer rotation (listing 2 line 4). This is only required once for each traversal through the queue (that is in the order of once per hundreds of thousands of executions), once the pointer reaches the end. At all other times it has no effect on the pointer variable. The NR bar in figure 4 shows the performance of enqueue when removing this instruction, indicating that there is 7% performance improvement available over the original queue.

Removing Synchronisation Overhead (NS).

Avoiding the synchronisation overhead is critical for performance. It not only means that we can remove two instructions from the critical path (listing 2 lines 5 and 6), but it also means that we can remove the body of the synchronisation code. This acts as an optimisation barrier for both the compiler and the architecture as it comprises of 9 assembly instructions in 4 basic blocks (including a spin-loop). According to Jablin et al. [11], the compiler will not perform efficient code movement across the spin-loop as it has no guarantee that the spin-loop will halt. Another example of a simple compiler optimisation that is not applied due to this code is loop unrolling. Figure 4 shows the performance of removing this code from enqueue (the NS bar), which is approximately 17% faster than the full queue.

Removing Both Overheads (NSNR).

Removing both synchronisation and pointer rotation overheads leads to even better performance. The NSNR bar in figure 4 indicates that the performance improvements of both optimisations aid each other, meaning there is 27% speed-up available if we can remove these instructions.

[1]For this experiment the queue size was set to 256MB, the dequeue function was disabled and the workload was a loop pushing the 64-bit loop index into the queue until it gets full.

3.3 Summary

Analysis of the enqueue operation shows that there are 4 instructions that are only required infrequently, for rotating the queue pointer and checking for the end of a section. The results in figure 4 show that removing these infrequently-required instructions can lead to speed-ups of 27%. The next section shows how we can build a queue that does this.

4. LYNX

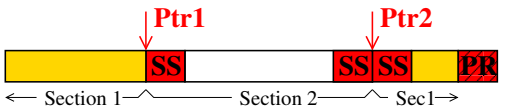
Lynx is a radically different queue design that reduces the overheads of enqueue and dequeue actions to a minimum by removing instructions that perform infrequently-required operations. The novelty resides in a combination of hardware and operating system support, using memory access violations to deal with uncommon events in a specialised signal handler. We first explain Lynx's memory layout, then show the C code to access the queue, and finally describe how signals are handled to synchronise the threads using the queue and to move from the end of the queue back to the beginning. Lynx is architecture and operating system independent, but we evaluate it in section 5 on x86-64 systems running Linux.

4.1 Memory Layout

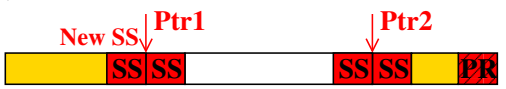
An overview of the memory layout in Lynx is shown in figure 5. The queue is aligned on a page boundary and is split into sections by *red-zones* (red-filled boxes). Each red-zone is one page-size long (usually 4KB) and is marked as non-readable and non-writable (e.g., using `mprotect()` on Unix-like systems or `VirtualProtect()` on Windows). Therefore, whenever a thread attempts to access data in a red-zone, the processor triggers an interrupt and an operating system exception is raised.

There are two types of red-zone. Section synchronisation red-zones (SSRZs) deal with thread synchronisation for each section and a pointer rotation red-zone (PRRZ) enables threads to move back to the start of the queue once they reach the end. Both are explained in more detail in subsequent sections. Figure 5 shows a queue with two sections, hence it has two SSRZs and one PRRZ.

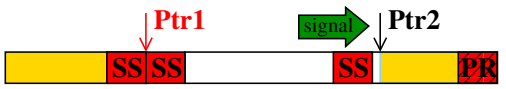
Red-zones are core components of Lynx. Their purpose is to allow the removal of non-essential code from the enqueue and dequeue operations, replacing instructions in the critical path of an application with those in a specialised signal handler which is called infrequently. This means that code to



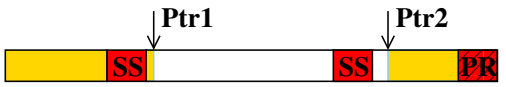
(a) Ptr1 hits SSRZ; Ptr2 in spin-loop before section1.



(b) New SSRZ created.



(c) At signal, Ptr2 deletes old SSRZ, enters section1.



(d) Ptr1 deletes old SSRZ and enters section2.

Figure 6: Moving a Section Synch. Red-Zone (SSRZ).

move from the end of the queue to the start, to find the next queue section, to wait until the next section becomes free, and to signal the other thread that the old section is available can all be moved out of enqueue and dequeue functions and placed in the signal handler.

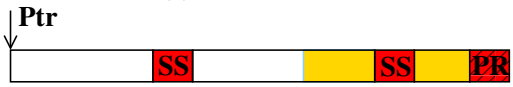
4.1.1 *Section Synchronisation Red-Zones (SSRZs)*

The first type of red-zone is used for synchronising threads between sections. These are to ensure that a thread only enters a queue section when the other thread has left, ensuring that a maximum of one thread occupies any section at any time. The SSRZs are not fixed, but move through the queue towards the front as the threads move between sections. Once they get to the beginning of the queue, they wrap around to the back and move towards the front again. Moving these red-zones means that instructions do not get trapped in a red-zone and the signal handler can operate independently; once it is finished, the queue access instruction will be re-executed (at the same address as before) and will succeed so execution can continue.

Figure 6 shows the details of how red-zone movement and synchronisation are performed. In figure 6a a thread accesses the queue (in this case to perform an enqueue operation) and uses an address in the SSRZ at the end of section 1 (Ptr1). This causes a hardware interrupt from an access violation, since the red-zone is non-readable and non-writable, and, in response, the OS kernel calls the queue signal handler. The first thing the signal handler does is to create a new red-zone immediately to the left of the current one (figure 6b). The thread then signals to the other thread that it has left section 1, meaning that the other thread (Ptr2) can safely access section 1 (to read from the queue), shown in figure 6c. The first thread spin-waits for section 2 to become free, which in this example happens immediately, then it deletes the original red-zone and leaves the signal handler (figure 6d). The original instruction is then re-executed and, since it is now accessing an address inside section 2, rather than an SSRZ, it will succeed and the application can continue.

The SSRZs never meet each other because they always move in the same direction, always move when they are hit,

(a) Ptr hits PRRZ.



(b) Ptr updated to point to the beginning of the queue.

Figure 7: Rotating the pointer when the Pointer Rotation Red-Zone (PRRZ) is hit.

and they move by the same amount when a thread reaches them. When a red-zone gets moved towards its predecessor, its predecessor has already been moved by the same thread, meaning that the SSRZs are always a fixed distance from their neighbours ± one page-size. Additionally, we size the queue so that each section is at least two pages long (including an SSRZ), so that there is always room to move the SSRZs without overlapping another red-zone.

Note that pushed data is never over-written when moving the SSRZ because the operation only alters the access permission bits of the memory page. When moving a red-zone, the enqueue thread will place the new one over the top of the data it has just written into the queue. However, when the dequeue thread hits that red-zone, it will move it again and, since the access address remains the same, once the signal handler has finished it will be able to read the data out of the queue. The only difference is that this page of memory it reads from will be in a later section of the queue to the one it was written into, but this has no effect on the queue's operation.

4.1.2 *Pointer Rotation Red-Zone (PRRZ)*

As previously mentioned in section 3.2, pointer rotation is another performance overhead for enqueue and dequeue that is on the critical path. Even in a highly optimised queue with a power-of-two size and aligned start address, it is still one instruction in the critical path of execution that does not alter the pointer for the vast majority of accesses. Lynx optimises this away through an additional red-zone just after the end of the queue. Unlike the SSRZs, this red-zone is fixed and is unique, serving only to rotate the access pointer back to the start of the queue when it reaches the end.

Figure 7 shows how this occurs. In figure 7a the enqueue thread accesses the queue just past the end, hitting in the PRRZ. The hardware interrupt again results in the OS calling the signal handler, which alters the address that the instruction is trying to access back to the start of the queue. Section 4.3 describes how this is performed. The result is shown in figure 7b which is the state of the queue once the signal handler finishes. The thread now proceeds to re-execute its access, and this time it will succeed because it will write at the start of the queue in the continuation of section 1.

4.2 User Code

C code for the enqueue and dequeue functions is shown in listing 3 using inline assembly for the x86-64 architecture. Inline assembly code is required for reading and writing to the queue so that we have control over the instruction and

```
1  void enqueue (queue_t q, long data) {
2    asm("movq %0, (%1, %2)"
3        :                       /* no output */
4        : "r" (data),           /* input %0  */
5          "r" (q->enqBase),     /* input %1  */
6          "r" (q->enqIdx)       /* input %2  */
7        : "1" );                /* clobber   */
8    q->enqIdx += sizeof(long);
9  }
10
11 long dequeue(queue_t q) {
12   long data;
13   asm("movq (%1, %2), %0"
14       : "=&r" (data)          /* output    */
15       : "r" (q->deqBase),     /* input %1 */
16         "r" (q->deqIdx)       /* input %2 */
17       : "1" );                /* clobber   */
18   deqIdx += sizeof(long);
19   return data;
20 }
```

Listing 3: Implementation of Lynx.

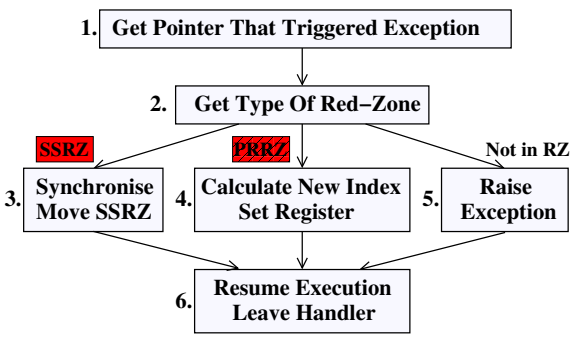


Figure 8: Overview of the operation of Lynx's handler.

operands used, which is vital to enable the PRRZ to function correctly. Section 4.3.2 describes this in more detail. An implementation of a subset of Lynx without the PRRZ pointer rotation functionality would not require assembly coding and could be implemented purely in C.

Once compiled, the final assembly code for Lynx's enqueue and dequeue functions consists of just two x86-64 instructions each: a memory instruction (for storing or loading from the queue) and an addition that increments the pointer. As with state-of-the-art queues, the compiler fully inlines this code. However, in contrast to the state-of-the-art, there is no complicated control flow (and no spinning loop) which allows the code to get heavily optimised by the compiler.

4.3 Lynx's Exception Handler

The Lynx exception handler is called whenever a thread attempts to access one of the queue's red-zones. As explained in sections 4.1.1 and 4.1.2, the handler's actions depend on the type of red-zone being accessed. An overview of the handler is shown in figure 8.

Step 1. The first task is to get the address which caused the exception, so that it can take appropriate actions based on the type of red-zone being accessed (if any). This is available through the arguments to the signal handler (specifically the `si_addr` field in the POSIX `siginfo_t` structure).

Step 2. Using the address, the type of red-zone can be determined and acted upon.

```
1  enqSync() {
2    newRedzone = getNewRedzoneLeft(currPtr);
3    configRedZone (ON, newRedzone);
4    redzone = newRedzone;
5    prevSectionState = ENQ_DONE;
6    /* Spin-loop */
7    while (nextSectionState != DEQ_DONE) ;
8    configRedZone (OFF, currPtr);
9    *nextSectionState = ENQ_WRITES;
10 }
11
12 deqSync() {
13   newRedzone = getNewRedzoneLeft(currPtr);
14   configRedZone (ON, newRedzone);
15   redzone = newRedzone;
16   if (prevSectionState != ENQ_EXITED)
17     prevSectionState = DEQ_DONE;
18   /* Spin-loop */
19   while (nextSectionState != ENQ_DONE
20       && nextSectionState != ENQ_EXITED) ;
21   configRedZone (OFF, currPtr);
22   nextSectionState = DEQ_READS;
23 }
```

Listing 4: Lynx handler's synchronisation code.

Step 3. If the address is in the SSRZ then the red-zone must be moved and the threads synchronised, as explained in section 4.3.1.

Step 4. If the address is in the PRRZ then the thread's state needs to be altered so that it re-executes the access at the beginning of the queue. This is described in section 4.3.2, and requires altering the instruction's source registers.

Step 5. Otherwise the exception did not come from access the queue, but is part of the actual program, so it gets re-raised.

Step 6. Once the handler has dealt with an exception in a red-zone, the thread is free to leave and execution continues by replaying the instruction that caused the fault.

4.3.1 SSRZ Movement and Synchronisation

As section 4.1.1 explained, accessing an SSRZ means that the handler must move the red-zone towards the start of the queue and synchronise the threads. Listing 4 shows the C code for these actions. The functions to enqueue and dequeue are almost the same, except the dequeue operation has to deal with the enqueue thread exiting before reaching a red-zone in line 16, so for brevity we only walk through the `enqSync()` function.

Each section has its own state variable, and these are used to synchronise the threads. The first step is to get the address of a new red-zone on the left of the current one (listing 4 line 2). The new red-zone is configured (access permission bits set, line 3) then the previous section's state is updated (line 5). This allows the dequeue thread to enter the previous section, if it is ready to, and so avoids deadlock. The enqueue thread then enters a spin-loop, waiting for the next section to become available (line 7), which it will when the other thread is no longer accessing it (state set to DEQ_DONE). Keeping the thread spinning avoids repeatedly calling the signal handler while waiting which is important for performance; handling an exception has a significant overhead because the operating system has to be

involved. Finally, in lines 8 and 9, the current red-zone is disabled and the next section's state changed to indicate the enqueue thread accessing it.

In both enqueue and dequeue functions, deadlock is avoided by unblocking the other thread before trying to move into the next section. In addition, by creating the new red-zone before setting the previous section's state, we ensure that the two threads can both be at the end of the same section, but will be hitting different red-zones.

To guarantee correctness, the compiler should not reorder the memory operations in the handler's synchronisation code. We do this by inserting compiler memory reordering barriers between the critical instructions: `asm volatile(" ::: "memory")` in GCC. Under the Total Store Order (TSO) memory model of the x86 architectures [4], no memory barrier instructions are required in either the enqueue()/dequeue() or the handler. Even though memory operations may execute out-of-order on the actual hardware, TSO guarantees that loads will see the values of earlier stores across cores. The handler requires fences for `movnti` instructions (these are not TSO) and architectures with relaxed consistency models. These fences guarantee that the status variables get updated after all queue data has been updated. Supporting targets with more relaxed memory models does not introduce any performance overheads as all the additional barrier instructions are in the handler, not in the critical path of execution.

4.3.2 Pointer Rotation in PRRZ

The sole job of the PRRZ is to alter the thread's state so that it accesses the start of the queue again, instead of continuing past the end. This requires the signal handler to identify the registers used by the instruction to create the memory address, determine the values they need to access the start of the queue, and then update them.

As shown in listing 3, we use inline assembly to specify the instructions that read and write to the queue. Using inline assembly means that we have control over the exact instruction that is used and the format of the memory access calculation. The compiler is allowed to choose the actual registers that contain the operands so that it can perform register allocation as usually. Using the POSIX `sigaction` API, the handler can get a pointer to the instruction which it can then parse to identify the source operands. Once the registers involved in the computation are identified, their values at the point of the exception can be retrieved through the `ucontext_t` structure that is given as the third argument to the signal handler.

As a concrete example, an x86 memory instruction calculates its address using equation 1.

$$Addr = SegReg + BaseReg + (IdxReg * Scale) + Offset \quad (1)$$

Our inline assembly instruction uses only the *BaseReg* and *IdxReg*, which are linked to queue variables, setting *Scale* to 1 and *Offset* to 0. We allow the *IdxReg* to increment whenever the queue is accessed, as shown in listing 3 line 8, and do not alter it in the signal handler. The *BaseReg* comes from another queue variable that we modify when accessing the PRRZ to set the address in equation 1 back to the start of the queue. In practice, this means we use equation 2.

$$Val = QueueStartAddr - IdxReg \quad (2)$$

The result of this is that *IdxReg* can take any value, even addresses that are beyond the boundaries of the queue, but the effective address calculation performed by the processor will always create an address within the queue. This works even when the value of *IdxReg* overflows. We perform a similar calculation and transformation for other architectures.

Using inline assembly is required for correctness for two reasons: 1) By using it we have a dedicated *BaseReg* for our own use, and we are free to update it within the handler without modifying the semantics of the surrounding code. If inline assembly is not used, then the compiler may optimise the code to use one register for both the loop iteration variable and *IdxReg*(or *BaseReg*), meaning that if we alter *IdxReg* (or *BaseReg*) within the signal handler, we also change the semantics of the code. 2) The inline assembly acts as an instruction re-ordering barrier in GCC, prohibiting dangerous re-ordering like in this case: `obj->elem = x; enqueue(obj);`

To actually update the *BaseReg* from within the signal handler, we cannot use a simple `mov` instruction because any changes to registers are reverted once the handler finishes. Instead we alter the relevant entry in the `ucontext_t` structure which defines the values of the registers that will be restored once the signal has been dealt with. We also update the variables that *BaseReg* is linked to (`q->enqBase` and `q->deqBase` as listed in listing 1 lines 4 and 15) so that the code will work even when compiled without optimisation (-O0), because in this case the value is read straight from memory before being used, and is not kept in a register.

4.4 Reporting Program Exceptions Correctly

It is crucial for the queue to be completely transparent for all exceptions that are unrelated to the workings of the queue. For example, if the program has a bug and dereferences a pointer to invalid memory, this should not be confused with the exceptions triggered by Lynx's enqueue or dequeue actions. We can effectively distinguish between the two by setting the handler to only catch segmentation faults (`SIGSEGV`) and by re-raising these faults if they are not related to queue operations (i.e., if the address that triggers the exception is not in a red-zone).

4.5 Summary

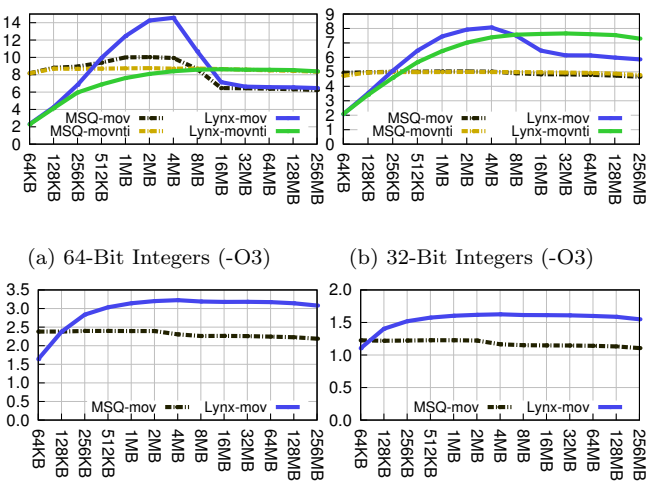
We have presented Lynx, a novel queue that uses hardware virtual memory permission checks and OS signal support to deal with infrequently-occurring queue actions. We augment the queue with two types of red-zone and configure them so that the threads are not allowed access. This allows us to read and write to the queue using only two machine instructions, placing all other code in a signal handler that is executed whenever a red-zone is touched.

5. RESULTS

We evaluate the throughput of Lynx for different data types and compare it to the state-of-the-art Multi-Section Queue (MSQ). We then show case studies for the use of Lynx in real applications in section 5.6.

5.1 Experimental Setup

We evaluated Lynx on a number of machines ranging from architectures used in embedded systems (like the Intel Bay-Trail-based J1900), up to those used in servers (like the Xeon and the Opteron). They are listed in table 1. Unless otherwise stated, our analysis was performed on the Intel Core



(a) 64-Bit Integers (-O3)

(b) 32-Bit Integers (-O3)

(c) 16-Bit Integers (-O3)

(d) 8-Bit Integers (-O3)

Figure 9: Throughput GBytes/s (y axis) of Lynx and the state-of-the-art Multi-Section Queue for different queue sizes (x axis) on an Intel Core i5-4570.

i5-4570. We pinned the enqueue and dequeue threads to distinct cores in all experiments (avoiding core sharing in processors supporting hyper-threading), choosing cores that shared the highest level of cache, i.e., an L2 cache if shared, otherwise L3. All experiments used a 2-section queue (for both Lynx and the state-of-the-art queue).

The throughput experiment's code consists of two threads moving 8GB of data through the queue. One thread pushes values into the queue and sums them, while the other thread removes them from the queue and also sums them. We ran the throughput experiments 3 times for warm-up and then took the average over the following 10 runs. We compiled with the system's GCC (shown in table 1) with `-O3`. The binaries in section 5.5 were generated with `-O3 -funroll-loops`.

5.2 Throughput Tests

We measured the throughput of Lynx for various queue sizes and compared it against the state-of-the-art [11, 15, 16, 24] (as in listing 1). We tested four data widths: 64-bit, 32-bit, 16-bit and 8-bit, shown in figure 9. We only show the `mov` results for the latter two because there is no non-temporal move instruction for these data widths[2].

Once the queue reaches a certain size, Lynx outperforms the state-of-the-art queue. The results show that Lynx (Lynx-mov) outperforms the state-of-the-art queue (MSQ-mov) for a range of queue sizes, depending on the data width. For the 64-bit test, Lynx is better for any size larger than 512KB. As the data width decreases, Lynx becomes better sooner, with the 32-bit and 16-bit cases starting at 256KB, and the 16-bit case at 128KB. The reason is that the narrower the data width, the more data of this type can fit in each queue section, therefore the larger the effective section size. For example, for the 8-bit data type (figure 9d), a 128KB queue size is effectively 8 times larger than that of

[2]A `movnti` with a wider data type could be used, but exploring this is not in the scope of this paper.

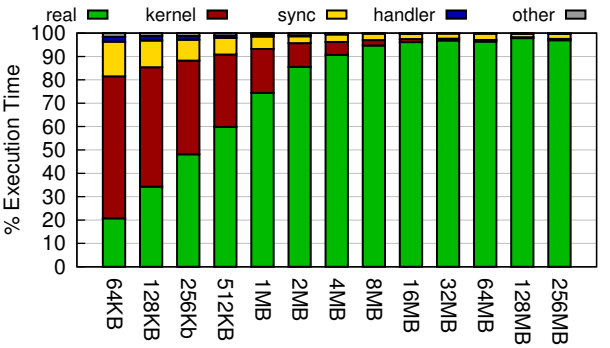


Figure 10: Breakdown of Lynx's cycles for the throughput test on the Intel Core-i5 system and 64-bit integers.

the 64-bit case (equal to 1MB), and, as shown in figure 9a, Lynx is better than the state-of-the-art for that size. The maximum throughput speedup of Lynx versus MSQ is 1.44× for 64-bit, 1.6× for 32-bit, and 1.4× for 16- and 8-bit data.

The throughput of the MSQ-movnti and Lynx-movnti for 64-bit data (figure 9a) is lower than the peak MSQ-mov and Lynx-mov by a large margin (8.5GB/s versus 14.6GB/s). This is because the movnti experiments' throughput is limited by the throughput of the system's main DRAM memory, since all stores bypass the caches completely. The evaluation system was equipped with dual-channel 1600MT/s DDR3 modules, with a maximum total interface throughput of approximately 2×12.8GB/s = 25.6GB/s. The 8.5GB/s queue throughput involves both storing and loading from memory, thus causing a 2×8.5GB/s = 17GB/s write+read memory load, which is very close to the theoretical maximum. This also explains the more intuitive results of the 32-bit tests (figure 9b where the maximum throughput with movnti is at 7.8GB/s versus 5GB/s for the standard mov.

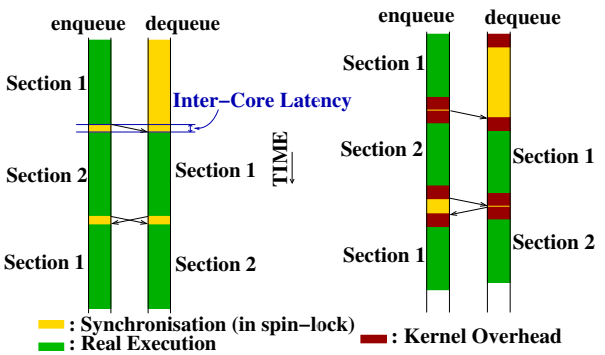
Accessing data through the cache (MSQ-mov and Lynx-mov) has higher memory bandwidth compared to bypassing the caches altogether. This is the reason why the throughput of Lynx-mov (figure 9a) is significantly higher compared to Lynx-movnti or MSQ-movnti for the bandwidth-stagnated 64-bit case.

5.3 Breakdown of Lynx Overheads

Figure 10 breaks down the execution time of Lynx for increasing queue sizes, showing that the overheads decrease rapidly, becoming negligible for large queues. Data was collected while running the throughput test for 64-bit integers using *perf*, which uses sampling to perform its measurements. The kernel time spent servicing the red-zone interrupts (kernel) is a significant fraction of the overall execution time for small queues. It is approximately 60% for a 64KB queue, but decreases as the queue gets larger at a rate of about 12% per queue size increment, until it gets to about 10% for a 2MB queue. This is expected as the kernel is involved in section synchronisation and index rotation, which together occur at least three times per walk of the queue. For small sizes, traversing the queue is fast, so the kernel is entered frequently. For larger queues, there are more entries to push and pop within each section, so the SSRZ and PRRZ are not hit as often and the kernel is less regularly called.

Processor Name	u-arch	nm	TDP	GHz	Cores	L1 Cache	L2 Cache	L3 Cache	Mem MT/s	Linux	GCC
Intel Xeon E5-2667 v2	Sandy Bridge	32	130W	3.30	2×8	6×32KB	6×256KB	15MB (S)	4×1600	3.13.0	4.8.4
AMD Opteron 6376	Piledriver	32	115W	2.3	2×16	16×16KB	8×2MB (S)	2×8MB (S)	4×1600	3.13.0	4.8.4
Intel Core i5-4570	Haswell	22	84W	3.20	1×4	4×32KB	4×256KB	6MB (S)	2×1600	3.10.17	4.8.3
Intel Core i3-2367M	Sandy Bridge	32	17W	1.40	1×2	2×32KB	2×256KB	3MB (S)	1×1333	3.9.3	4.8.2
Intel Celeron J1900	BayTrail-D	22	10W	2.42	1×4	4×24KB	2×1MB (S)	-	1×1333	3.16.0	4.9.2

Table 1: Description of systems that we evaluated. The shared cache is marked with an (S).



(a) Multi-Section Queue (b) Lynx

Figure 11: The synchronisation overhead of queues.

The kernel overhead is strongly correlated to the synchronisation overhead (sync). This is because when one thread is in the process of servicing an interrupt (in kernel mode) the other thread is likely in a spin-loop, waiting for the state variable to change from the other thread, allowing it to proceed to the next section. This is illustrated in figure 11 where the threads execute from a cold start (dequeue has nothing to read in the beginning). In Lynx (figure 11b), even though the real execution is faster (shorter green boxes), there is the additional kernel overhead (dark red) that leads to more time spent in spin-loops (yellow). In MSQ (figure 11a), on the other hand, synchronisation is direct and faster. The cycles spent in the spin-loops ranges from 15% for a queue size of 64KB down to less than 2% for queue sizes 2MB or higher.

Finally there are miscellaneous remaining overheads. The code of the handler itself (handler) is the code that parses the instruction and calculates the value of the index. Its overhead is negligible even for small queue sizes. The remaining overheads (other) refer to time spent in other boilerplate code, e.g., libc and libpthread. Again, these overheads are negligible for all queue sizes.

5.4 Performance Impact of Compiler Optimisations

Lynx, due to its minimal instruction count and lack of control flow in enqueue and dequeue operations, allows the compiler to highly optimise it within its surrounding code. Figure 12 shows the throughput impact of increasing the compiler optimisation level (using GCC-4.8.3), starting from no optimisation (-O0) all the way to -O3 and then forcing loop unrolling (-O3 -funroll-loops). At -O0, the compiler will not cache values in registers. Instead, before each instruction all its inputs are loaded from memory and after its execution the outputs are stored back into memory. However with higher optimisation levels Lynx is significantly better.

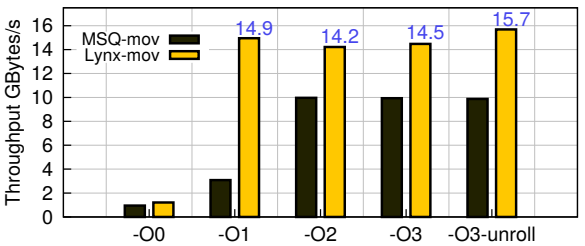


Figure 12: Throughput under several compiler optimisation levels on Intel Core-i5 for 64-bit integers (4MB queue).

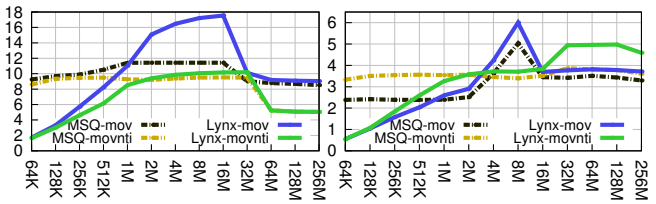
There is a mismatch between the two queue implementations at -O1, with Lynx seeing a dramatic rise in throughput, whereas the MSQ achieves only a modest increase. The MSQ code when compiled with -O1 contains more instructions (some of them even access memory). Adding strict aliasing and partial redundancy elimination to -O1 (by default only enabled in -O2) brings the performance of MSQ to the expected levels. On the other hand, the code of Lynx is only 2 instructions long, contains no control-flow instructions and therefore it can be optimised very efficiently even with fewer optimisation passes.

While MSQ achieves the same throughput from -O2 onwards, Lynx improves when compiled with the unrolling flag, reaching a peak throughput of 15.69 GB/s, 5% faster than the previous best at -O1. The compiler's unrolling pass successfully unrolls Lynx's code 8 times, but it fails to do so for the MSQ due to its larger code size and complexity.

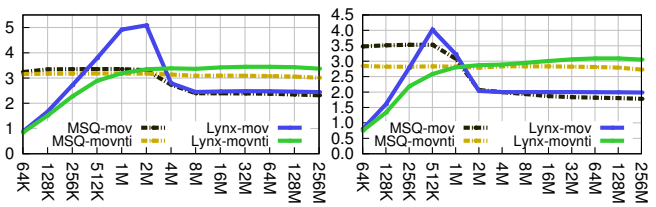
5.5 Evaluation on Various Machines

We evaluated Lynx on several systems in a large range of the power and performance spectrum. We evaluated low-power architectures (Intel Celeron J1900 Bay Trail-D 10W), low-power laptop processors (like the Intel Core i3 M series), common desktop processors (Intel Core i5) and powerful server components (Intel Xeon and AMD Opteron). A description of the systems can be found in table 1. The 64bit throughput for these machines is shown in figure 13 and the 32bit throughput in figure 14.

In all systems Lynx-mov starts to outperform the state-of-the-art queue (MSQ-mov) once the queue size is big enough to amortise the cost of frequent exception handling. This point is at a queue size of either 512KB or 1MB for the 64bit test and usually earlier (starting from 256KB for the Core-i3 and Celeron-J1900) for the 32-bit test. The results for the `movnti` versions of the queues are similar but Lynx-movnti usually overtakes MSQ-movnti for larger queue sizes in the 64-bit test, usually 4MB, and the performance difference is significantly smaller compared to the regular `mov`. For the 32-bit results, just like in figure 9b for the Core i5, the queue's performance is less constrained by the mem-

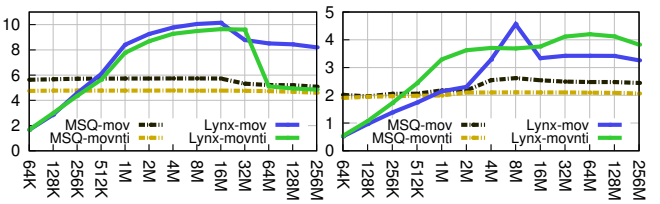


(a) Intel Xeon E5-2667 v2 (b) AMD Opteron 6376

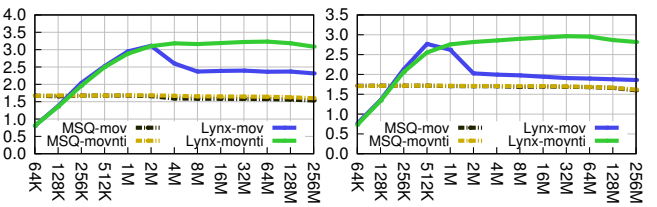


(c) Intel Core-i3 2367M (d) Intel Celeron J1900

Figure 13: Throughput GB/s (y axis) of 64-bit data for different queue sizes (x axis) and machines.



(a) Intel Xeon E5-2667 v2 (b) AMD Opteron 6376



(c) Intel Core-i3 2367M (d) Intel Celeron J1900

Figure 14: Throughput GB/s (y axis) of 32-bit data for different queue sizes (x axis) and machines.

ory throughput of either caches or DRAM. Lynx, with its lightweight enqueue and dequeue operations achieves significantly higher bandwidth for both mov and movnti versions with speedups of 1.9× for Core-i3 `mov` and 2.0× for Xeon and Opteron with `movnti`.

5.6 Case Studies

This section evaluates Lynx in actual applications to show its speedups over state-of-the-art transfer to real-world settings.

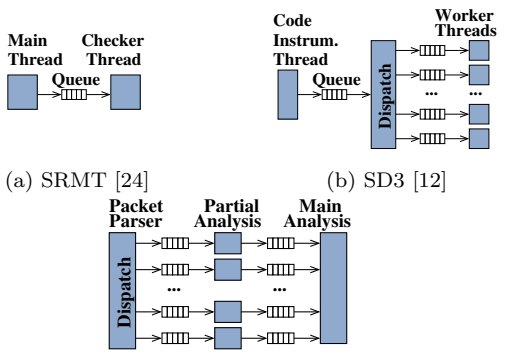
SRMT Fault Tolerance.

SRMT [24, 25] is a technique for detecting transient errors (i.e., bit-flips) with software support, by running the application on one thread (main thread) and a modified copy of the code on another thread (checker thread). The main thread sends all data it reads / writes from / to memory to the checker thread via a software queue. The checker thread compares this data against the values that are produced locally. If they differ, then a fault has occurred.

We implemented SRMT using both the state-of-the-art MSQ and Lynx and used benchmarks from the NAS NPB-2.3 [2] suite (BT, CG, EP, IS, LU, MG and SP) as inputs. FT is missing due to compilation error in SRMT's implementation. We used the queue size that performed best on average for each queue (256KB for MSQ and 2MB for Lynx), the best performing instruction (`mov` for MSQ and `movnti` for Lynx) and prefetch instructions for both. We measured the execution time and standard deviation on the Core-i5 system (table 1). Figure 16a shows that the performance with Lynx is improved up to 1.4× at an average of 1.12× (geometric mean). For some benchmarks, the Lynx queue size is not ideal. Choosing the best queue size per benchmark means Lynx always out-performs MSQ.

SD3 Data-Dependence Profiling Tool.

This is a state-of-the-art tool for fast dynamic data-dependence profiling [12]. At a high level view, the instrumented code of a benchmark sends information about data accesses to the worker threads, which perform the data-dependence analysis. To speed up the execution we use an intermediate thread, the dispatcher thread, to minimise the burden on the instrumented thread. This enhancement improves both queues equally.



(a) SRMT [24] (b) SD3 [12]

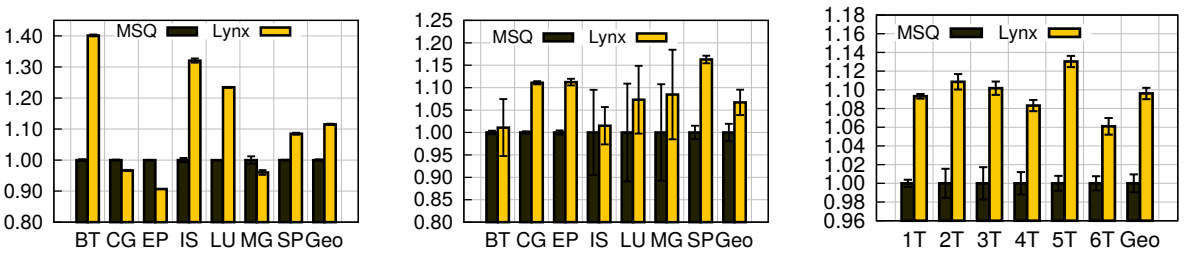
(c) Network Monitoring[16]

Figure 15: Software structure of case studies.

Since SD3 has a total of 10 parallel threads (instrumentation, dispatch and 8 workers), we configured the queue size at 1MB such that they all fit in L3 cache (best performance). We pinned all threads on a single physical hyper-threaded 8-core Xeon (table 1) and made sure that the instrumentation and dispatch threads ran on different cores. The normalised performance and standard deviation of SD3 for the NAS benchmarks with `mov` instructions are shown in figure 16b. SD3 benefits from high-speed inter-core communication. The performance improvement with Lynx is up to 1.16× with a geometric mean of 1.07×.

Network Traffic Monitoring (NTM).

This tool is a fast parallel network traffic monitor to be used for line-rate network statistics on very high-speed networks. The design is based on the tool evaluated by Lee



(a) SRMT (b) SD3 (c) NTM Speedups

Figure 16: Normalised Speedup using Lynx over MSQ.

CLF SP/SC Queue	Sections	Drawbacks	Performance Enhancements
Lamport's [14, 7]	1	Cache ping-pong	Lock-free queue
FastForward [8]	1	Non-generic synchronisation	Write/Read slip
DBLS[24]	> 1	Number of synchronisation instructions	Lazy synchronisation
MCRingBuffer[15, 16]	> 1	Number of synchronisation instructions	Lazy synchronisation
Liberty[11]	> 1	Number of synchronisation instructions	Non-temporal SSE instructions (e.g. `movnti`), prefetching
HAQu [17]	1	Hardware modification	Extra Hardware
Lynx (proposed)	> 1	Handler's overhead on very small queue sizes	Synchronisation overhead is moved off the critical path

Table 2: Overview of features provided by CLF queues.

et al. [16] and is composed of several threads, as shown in figure 15c. The main thread parses IPv4 network packet headers, extracts information and dispatches them to sub-analyser threads. These gather partial statistics and provide the data to the main analyser which collects them together.

We evaluated NTM performance on the Xeon E5-2667 (table 1) varying the number of sub-analyser threads from 1 to 6. We used the queue size that leads to best performance for each queue (128KB for MSQ, 2MB for Lynx), `mov` for both. The speedups shown in figure 16c are normalised to MSQ. The geometric mean speedup of Lynx over MSQ is 1.09x.

6. RELATED WORK

Lamport proved that a concurrent lock-free (CLF) queue can be implemented in a SP/SC scenario [7, 14]. However, even though it is lock-free, this queue has very poor performance due to frequent cache ping-pong of queue control variables.

An improvement over Lamport's implementation is the FastForward CLF queue [8]. In this implementation the enqueue and dequeue indices are private to the enqueue and dequeue functions and their values are never passed to the other. To avoid the threads overtaking each other, the dequeue operation writes NULL values to the queue after reading data from it; enqueue checks that the queue value at the write index is NULL before overwriting it. False sharing (reading and writing to different queue elements in the same cache line) is avoided by enforcing a buffer between the writes and reads (referred to as slip).

Multiple sections were introduced by DBLS [24] and MCRingBuffer [15, 16] queues. Both queues keep enqueue and dequeue indices private and occasionally share these values, once for every section. Liberty queues [11] are similar to both DBLS and MCRingBuffer in design, but they introduce several implementation-specific performance improvements, including non-symmetric producer and consumer, prefetching, streaming instructions. Finally, [26] studies the deadlock problem of multi-section queues and proposes a solution to it.

In HAQu [17], the authors recognise that the high instruction overhead of existing software queues becomes critical in fine-grained communication. They propose a hardware-accelerated queue to decrease the number of instructions within enqueue/dequeue functions.

An overview of the attributes of the SP/SC CLF queues are shown in table 2. Several CLF queues have been studied since Lamport's research [13, 18, 19, 21, 22, 23]. These works, however, have focused on improving MP/MC queues, rather than improving the SP/SC case. These have higher overheads to avoid ABA problems [18] and to maintain linearizability [10].

Exploiting the hardware/OS memory protection systems for improving performance has been used in the past. A survey of algorithms that make use of memory protection techniques is presented in [6]. Typical uses include garbage collection, and overflow protections. A similar technique has been used under the fine-grained Mondrian memory protection [27] system to implement a zero-copy network stack. More recently, the memory protection system has also been used in the context of software transactional memories to achieve strong atomicity [5].

7. CONCLUSION

High performance single-producer / single-consumer software queues are fundamental building components of parallel software. Maximising their efficiency is crucial for extracting the maximum performance of parallel software. This paper has presented Lynx, a radically new software architecture for SP/SC queues, which reduces the critical path overhead of enqueue and dequeue operations down to a minimum. Evaluation of Lynx on various commodity hardware platforms shows throughput improvements of over 1.57× compared to the state-of-the-art and significant improvements in actual applications of up to 1.4×.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant reference EP/K026399/1. Additional data related to this publication is available in the data repository at https://www.repository.cam.ac.uk/handle/1810/254651.

8. REFERENCES

[1] GCC: Gnu Compiler Collection. *http://gcc.gnu.org*.

[2] NAS Parallel Benchmarks. *http://www.nas.nasa.gov/publications/npb.html*.

[3] The LLVM Compiler Infrastructure. *http://llvm.org*.

[4] Intel 64 and IA-32 Architectures Software Developer's Manual. 2015.

[5] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-shelf Memory Protection Hardware. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[6] A. W. Appel and K. Li. Virtual Memory Primitives for User Programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, 1991.

[7] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *Proceedings of the 3rd Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 1991.

[8] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-free Queue. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[9] M. Girkar and C. D. Polychronopoulos. Automatic Extraction of Functional Parallelism from Ordinary Programs. *Transactions on Parallel and Distributed Systems*, 3(2), 1992.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *Transactions on Programming Languages and Systems*, 12(3), 1990.

[11] T. B. Jablin, Y. Zhang, J. A. Jablin, J. Huang, H. Kim, and D. I. August. Liberty Queues for EPIC Architectures. In *Proceedings of EPIC Workshop*, 2010.

[12] M. Kim, H. Kim, and C.-K. Luk. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling. In *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO 43)*, 2010.

[13] E. Ladan-Mozes and N. Shavit. An Optimistic Approach to Lock-Free FIFO Queues. *Distributed Computing*, 20(5), 2007.

[14] L. Lamport. Specifying Concurrent Program Modules. *Transactions on Programming Languages and Systems*, 5(2), 1983.

[15] P. P. C. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Shared Ring Buffer for Multi-Core Architectures. In *Proceedings of the 5th Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

[16] P. P. C. Lee, T. Bu, and G. Chandranmenon. A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[17] S. Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-Accelerated Queueing for Fine-Grained Threading on a Chip Multiprocessor. In *17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[18] M. M. Michael and M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1), 1998.

[19] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using Elimination to Implement Scalable and Lock-Free FIFO Queues. In *Proceedings of the 17th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[20] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO 38)*, 2005.

[21] S. Prakash, Y. H. Lee, and T. Johnson. A Nonblocking Algorithm for Shared Queues Using Compare-and-Swap. *Transactions on Computers*, 43(5), 1994.

[22] W. N. Scherer, III, D. Lea, and M. L. Scott. Scalable Synchronous Queues. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[23] P. Tsigas and Y. Zhang. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proceedings of the 13th Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.

[24] C. Wang, H. s. Kim, Y. Wu, and V. Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.

[25] C. C. Wang and Y. Wu. Apparatus and Method for Redundant Software Thread Computation. 2010. US Patent 7,818,744.

[26] J. Wang, K. Zhang, X. Tang, and B. Hua. B-Queue: Efficient and Practical Queuing for Fast Core-to-Core Communication. *International Journal of Parallel Programming*, 41(1), 2012.

[27] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[28] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.