A Lock-Free, Cache-Efficient Shared Ring Buffer for Multi-Core Architectures

Patrick P. C. Lee[1], Tian Bu[2], Girish Chandranmenon[2]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[2]Alcatel-Lucent, Bell Laboratories, USA
pclee@cse.cuhk.edu.hk, {tbu, girishc}@alcatel-lucent.com

ABSTRACT

We propose *MCRingBuffer*, a lock-free, cache-efficient shared ring buffer that provides fast data accesses among threads running in multi-core architectures. MCRingBuffer seeks to reduce the cost of inter-core communication by allowing concurrent lock-free data accesses and improving the cache locality of accessing control variables used for thread synchronization. Evaluation on an Intel Xeon multi-core machine shows that MCRingBuffer achieves a throughput gain of up to 4.9× over existing concurrent lock-free ring buffers. A motivating application of MCRingBuffer is parallel network traffic monitoring, in which MCRingBuffer facilitates multi-core architectures to process packets at line rate.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Synchronization*

General Terms

Design, Experimentation, Performance

1. INTRODUCTION

We propose *MCRingBuffer*, a lock-free, cache-efficient ring buffer that speeds up the shared data accesses in multi-threaded, multi-core traffic monitoring systems. MCRingBuffer minimizes the memory access overhead of thread synchronization by improving the cache locality of accessing the control variables that reference the buffer slots. Note that MCRingBuffer is a software-based solution that does not use any hardware synchronization primitives, and it works on general-purpose CPUs. Also, its performance gain is independent of the data types of the elements being transferred and the implementation of the multi-threaded applications.

One motivating application of MCRingBuffer is data traffic monitoring in high-speed networks. Multi-core architectures provide a potential solution to line-rate traffic monitoring by parallelizing the executions of packet processing. MCRingBuffer seeks to exploit the full potential of multi-core architectures, as it minimizes the inter-core communication cost so that threads residing in different cores can efficiently exchange information for network data analysis.

ANCS'09, October 19-20, 2009, Princeton, New Jersey, USA.

2. MCRINGBUFFER DESIGN

We focus on the single-producer/single-consumer model. Lamport [2] considers a concurrent lock-free ring buffer, which we call *BasicRingBuffer*, that does not require any hardware synchronization primitive (e.g., compare-and-swap). The correctness of BasicRingBuffer assumes that reading and writing the control variables (which are of integer type) are indivisible, atomic operations. Such an assumption generally holds for modern hardware architectures.

Other lock-free ring buffers [1, 3] improve BasicRingBuffer by comparing control variables directly with the buffer slots that hold data elements. However, this data/control coupling requires that the ring buffer must define a null data element that cannot be used by applications, thereby introducing an additional constraint when the ring buffer is to be used for generic data types.

We propose MCRingBuffer, a shared ring buffer that supports concurrent lock-free accesses. MCRingBuffer is built upon BasicRingBuffer, with a key objective to improve the cache locality of accessing control variables. Figure 1 shows the skeleton of MCRingBuffer, including the placement of control variables as well as the pseudo-code of the insert and extract procedures executed by the producer and the consumer, respectively. MCRingBuffer comprises two major design features: (i) cache-line protection, and (ii) batch updates of control variables.

Cache-line protection. When a variable has been accessed, it is placed in cache. To avoid *false sharing* (i.e., two threads each access different variables in the same cache line), we place the control variables so that the local, non-shared variables of different threads do not reside in the same cache line. With cache-line protection, we can also minimize the accesses to shared control variables. When the producer (consumer) is about to insert (extract) an element, it first checks the local control variable `localRead` (`localWrite`) residing in its own cache line to decide whether the buffer is *potentially* full (empty). If so, then the producer (consumer) further checks `read` (`write`) to decide whether the buffer is *actually* full (empty). The producer (consumer) will insert (extract) elements whenever the buffer is neither potentially nor actually full (empty). The intuition is that when `read` (`write`) is reloaded from main memory, it may have been incremented multiple times by the consumer (producer) to

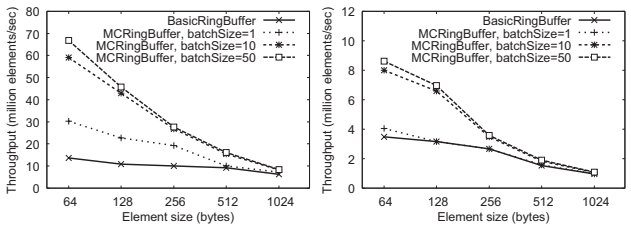
```
/* Variable definitions */
 1: char cachePad0[CACHE_LINE];
 2: /*shared control variables*/
 3: volatile int read;
 4: volatile int write;
 5: char cachePad1[CACHE_LINE - 2 * sizeof(int)];
 6: /*consumer's local control variables*/
 7: int localWrite;
 8: int nextRead;
 9: int rBatch;
10: char cachePad2[CACHE_LINE - 3 * sizeof(int)];
11: /*producer's local control variables*/
12: int localRead;
13: int nextWrite;
14: int wBatch;
15: char cachePad3[CACHE_LINE - 3 * sizeof(int)];
16: /*constants*/
17: int max;
18: int blockOnEmpty;
19: int batchSize;
20: char cachePad4[CACHE_LINE - 3 * sizeof(int)];
21: T* element;

function Insert(T element)
 1: int afterNextWrite = NEXT(nextWrite);
 2: if afterNextWrite == localRead then
 3:    while afterNextWrite == read do
 4:       /*busy waiting*/
 5:    end while
 6:    localRead = read;
 7: end if
 8: buffer[nextWrite] = element;
 9: nextWrite = afterNextWrite;
10: wBatch++;
11: if wBatch ≥ batchSize then
12:    write = nextWrite;
13:    wBatch = 0;
14: end if

function Extract(T* element)
 1: if nextRead == localWrite then
 2:    while nextRead == write do
 3:       if blockOnEmpty == 0 then
 4:          return -1; /*no element is read*/
 5:       end if
 6:       /*busy waiting*/
 7:    end while
 8:    localWrite = write;
 9: end if
10: *element = buffer[nextRead];
11: nextRead = NEXT(nextRead);
12: rBatch++;
13: if rBatch ≥ batchSize then
14:    read = nextRead;
15:    rBatch = 0;
16: end if
17: return 0; /*an element is read*/
```

Figure 1: MCRingBuffer.

refer to a few buffer slots ahead. Thus, the producer (consumer) only needs to access `read` (`write`) after more than one insert (extract) operation. Thus, *MCRingBuffer reduces the frequency of reading the shared control variables from main memory.*

Batch updates of control variables. In BasicRingBuffer, the shared variable `read` (`write`) is updated after every extract (insert) operation. Here, we apply batch updates on the shared control variables `read` and `write` so as to have them modified less frequently. We divide a ring buffer into blocks, each of which contains `batchSize` slots. We advance `read` (`write`) to the next block only after a `batchSize` number of elements have been extracted (inserted). Thus, *MCRingBuffer reduces the frequency of writing the shared control variables to main memory.* Note that our batch update scheme is applied to control variables and is transparent to how the insert and extract operations on the data elements are scheduled.

The batch update scheme assumes that data elements are constantly available so that the control variables can be updated. This is justified for the ring buffers that share packet information for high-speed networks that contain a high volume of packets.



(a) Sibling cores (b) Non-sibling cores

Figure 2: Throughput vs. element size.

3. EVALUATION

We evaluate BasicRingBuffer and MCRingBuffer on an Intel Xeon 5355 quad-core Linux machine with 2.66 GHz CPU and 32 GB RAM. The CPU comprises two replicas of dual-core modules, each with a pair of cores and a shared second-level (L2) cache. We call a pair of cores *sibling cores* if they reside in the same module, or *non-sibling cores* otherwise. The ring buffers are written in C++ and compiled using GCC 4.1.2 with the `-O2` option.

In our evaluation, the producer thread inserts 10 M elements, and the consumer thread extracts the inserted elements in order. We measure the *throughput*, i.e., the number of pairs of insert/extract operations performed per second. Each data point is averaged over 30 trials.

Figures 2 shows the throughput of BasicRingBuffer and different MCRingBuffer variants versus the data element size, where each ring buffer has capacity 2,000 elements. Overall, MCRingBuffer achieves higher throughput than BasicRingBuffer. For example, when the element size is 64 bytes, the throughput of MCRingBuffer with `batchSize` = 50 is 4.9× and 2.5× over BasicRingBuffer for sibling and non-sibling cores, respectively.

4. CONCLUSIONS

We present MCRingBuffer, a lock-free, cache-efficient ring buffer that achieves efficient thread synchronization in multi-core architectures. MCRingBuffer improves cache locality of accessing control variables via cache-line protection and batch updates of control variables.

5. REFERENCES

[1] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism - A Cache-Optimized Concurrent Lock-Free Queue. In *PPoPP*, 2008.

[2] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. on Software Engineering*, 3(2), Mar 1977.

[3] J. Wang, H. Cheng, B. Hua, and X. Tang. Practice of Parallelizing Network Applications on Multi-core Architectures. In *ISC*, 2009.