A Lock-Free, Cache-Efficient Multi-Core Synchronization Mechanism for Line-Rate Network Traffic Monitoring

Patrick P. C. Lee
Dept of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
pclee@cse.cuhk.edu.hk

Tian Bu, Girish Chandranmenon
Bell Laboratories
Alcatel-Lucent
Murray Hill, NJ, USA
{*tbu, girishc*}*@alcatel-lucent.com*

***Abstract*—Line-rate data traffic monitoring in high-speed networks is essential for network management. To satisfy the line-rate requirement, one can leverage multi-core architectures to parallelize traffic monitoring so as to improve information processing capabilities over traditional uni-processor architectures. Nevertheless, realizing the full potential of multi-core architectures still needs substantial work, especially in the face of the ever-increasing volume and complexity of network traffic. This paper addresses the issue through the design of a lock-free, cache-efficient synchronization mechanism that serves as a basic building block for a general class of multi-threaded, multi-core traffic monitoring applications. We embed the synchronization mechanism into *MCRingBuffer*, a multi-core shared ring buffer that provides fast data accesses among threads running in different cores. MCRingBuffer allows concurrent lock-free data accesses and improves the cache locality of accessing the control variables that are used for thread synchronization. Through extensive evaluation on an Intel Xeon multi-core machine, we show that MCRingBuffer achieves a throughput gain of up to 5× over existing lock-free ring buffers. Finally, we present a parallel traffic monitoring prototype that is built upon MCRingBuffer, and demonstrate via trace-driven simulation how MCRingBuffer facilitates packet processing at line rate.**

I. Introduction

Monitoring the behavior of data traffic in communication networks is essential for a variety of network management applications such as accounting, resource provisioning, failure diagnosis, and intrusion detection. One crucial requirement of network traffic monitoring is to process packets at *line rate*, meaning that the packet processing speed must keep up with the bandwidth of the communication link where data traffic is monitored. However, as the volume of network traffic continuously surges, so does the link bandwidth, typically to Gigabit-per-second scales. Also, as networking applications are getting more diversified, data traffic contains more sophisticated information that further complicates the monitoring process. Conventional uni-processor architectures may no longer support line-rate traffic monitoring, and this motivates the need of more advanced architectures.

The emergence of commercial multi-core architectures (e.g., Intel Xeon and AMD Opteron) provides a potential solution to line-rate traffic monitoring, since we may now divide the packet processing into smaller subtasks and parallelize their executions with multiple threads running on different cores. The idea of parallelizing packet processing has been advocated in prior work [1], [7], [12], [18], [19], yet exploiting the full potential of multi-core architectures remains a challenging issue. One such challenge is to *minimize the cost of inter-core communication*, so that threads residing in different cores can efficiently exchange state information for complete analysis. This issue can be addressed in the application protocols (i.e., the upper layer) where protocol messages exchanged across threads need to be minimized. On the other hand, we are also interested in looking into the shared data structures (i.e., the lower layer) through which threads exchange information since they are necessary for most multi-threaded applications.

Every shared data structure for multi-threaded applications requires a synchronization mechanism to coordinate the correct data accesses among multiple threads. However, most synchronization mechanisms involve different kinds of overhead that could reverse the effectiveness of parallelism. For example, locks are commonly used to enforce mutual exclusion of resources shared by multiple threads. However, lock-based approaches are generally inefficient, as they serialize thread accesses and limit only one thread to access the entire shared resources at one time. In addition, thread synchronization involves operations on a number of control variables. However, if these variables are frequently accessed via main memory rather than cache, then the memory access overhead can slow down the synchronization process. Therefore, we need to carefully design an efficient synchronization mechanism, especially in the context of multi-core architectures.

In this paper, we propose MCRingBuffer, a shared ring buffer that embeds a lock-free, cache-efficient synchronization mechanism and hence speeds up the shared data accesses in multi-core architectures. MCRingBuffer eliminates the use of locks, and allows different threads to concurrently access the ring buffer while ensuring the correctness of

insertions and extractions of data elements. Also, MCRingBuffer improves the cache locality of accessing the control variables that are used for thread synchronization in multi-core architectures. Furthermore, we make MCRingBuffer generic in the sense that: (i) it is purely a software-based data structure that works on general-purpose CPUs, and (ii) its performance gain is independent of the data types of elements being transferred and the implementation of the multi-threaded applications.

We prove the correctness of MCRingBuffer under the single-producer/single-consumer framework. We then conduct evaluation on an Intel Xeon 5355 quad-core machine, and show that MCRingBuffer improves the throughput of conventional lock-free ring buffers by up to 5×. We also profile different ring buffers with the Intel VTune Performance Analyzer [3], and further confirm that MCRingBuffer significantly reduces cache misses in thread synchronization.

We next propose a parallel traffic monitoring system that is built upon MCRingBuffer. Through the simulation driven by real traces collected from an operational network, we show that MCRingBuffer significantly improves the packet processing throughput by up to 5.2× and 1.9× over the single-threaded case and the multi-threaded case with conventional lock-free ring buffers, respectively. This further confirms the applicability of MCRingBuffer in line-rate traffic monitoring.

The remainder of the paper proceeds as follows. Section II presents the design of MCRingBuffer. Section III presents a parallel traffic monitoring prototype built upon MCRingBuffer and its trace-driven simulation results. Section IV discusses the related work, and we conclude in Section V.

II. MCRingBuffer

We describe MCRingBuffer, a shared ring buffer tailored for multi-core architectures. We start by describing an existing concurrent lock-free ring buffer that we use as a building block for MCRingBuffer. Then we present MCRingBuffer, including its design, sketch proof of correctness, and evaluation results.

A. Background

Ring buffer design, also known as the *producer/consumer problem*, is a classical problem in many introductory operating systems textbooks. In the problem, we have a bounded buffer with a fixed number of *slots*. A producer inserts *elements* to the buffer only when the buffer is not full, while a consumer extracts elements from the buffer only when the buffer is not empty. In addition, the *first-in-first-out (FIFO)* property needs to be guaranteed, meaning that the elements extracted by the consumer appear in the same order as the elements inserted by the producer.

Figure 1 illustrates a high-level block diagram of a multi-core architecture on which a ring buffer is deployed. Suppose that a single pair of the producer and consumer threads

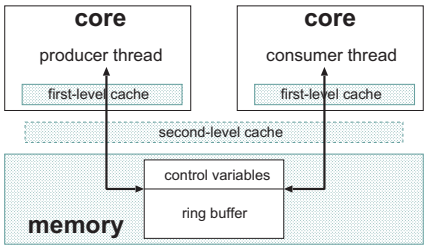


Figure 1. Block diagram of a multi-core architecture on which a ring buffer is deployed. Note that two cores may share a second-level (L2) cache if they reside in the same CPU module (see Section II-D).

```
function Insert(T element)
 1: lock(lock);
 2: if buffer is full then
 3:     unlock(lock);
 4:     return INSERT_FAILED;
 5: end if
 6: write element to next available buffer slot;
 7: unlock(lock);
 8: return INSERT_SUCCESS;

function Extract(T* element)
 1: lock(lock);
 2: if buffer is empty then
 3:     unlock(lock);
 4:     return EXTRACT_FAILED;
 5: end if
 6: read element from next available buffer slot;
 7: unlock(lock);
 8: return EXTRACT_SUCCESS;
```

Figure 2. LockRingBuffer, a lock-based ring buffer.

have information to exchange. Then both threads share the information through the ring buffer located in the shared main memory, and operate on a set of *control variables* associated with the ring buffer for thread synchronization. To reduce memory accesses, which are generally expensive operations, variables that have just been accessed may be *cached* to speed up subsequent accesses to themselves.

1) Lock-based Ring Buffers: One simple way to implement a ring buffer is to use a *lock-based approach*, which defines a critical section for the entire ring buffer and allows only one thread (i.e., either the producer or the consumer) to access the buffer at one time. A thread needs to first acquire a lock before accessing the buffer, and it releases the lock after it is done accessing the buffer. Figure 2 shows the pseudo-code of a lock-based ring buffer, which we call *LockRingBuffer*. The lock-based approach is generally inefficient since it prevents the producer and consumer threads from simultaneously accessing the ring buffer even though they may access different buffer slots.

Instead of using the lock-based approach, we implement MCRingBuffer as a lock-free ring buffer so that the producer and the consumer can simultaneously access the ring buffer provided that they do not touch the same buffer slot. MCRingBuffer is built upon the work in [10], which we

```
function Insert(T element)
 1: if NEXT(write) == read then
 2:     return INSERT_FAILED;
 3: end if
 4: buffer[write] = element;
 5: write = NEXT(write);
 6: return INSERT_SUCCESS;

function Extract(T* element)
 1: if read == write then
 2:     return EXTRACT_FAILED;
 3: end if
 4: *element = buffer[read];
 5: read = NEXT(read);
 6: return EXTRACT_SUCCESS;
```

Figure 3. BasicRingBuffer [10].

```
function Insert(T element)
 1: if buffer[write] != NULL then
 2:     return INSERT_FAILED;
 3: end if
 4: buffer[write] = element;
 5: write = NEXT(write);
 6: return INSERT_SUCCESS;

function Extract(T* element)
 1: if buffer[read] == NULL then
 2:     return EXTRACT_FAILED;
 3: end if
 4: *element = buffer[read];
 5: buffer[read] = NULL;
 6: read = NEXT(read);
 7: return EXTRACT_SUCCESS;
```

Figure 4. FastForward [2].

describe as follows. We also describe other lock-free ring buffers proposed in the literature.

2) Lock-Free Ring Buffers: Suppose that the single-producer/single-consumer model is considered, meaning that a ring buffer is only accessed by two threads, i.e., the producer and the consumer. Then Lamport [10] considers a concurrent lock-free ring buffer that does not require any hardware synchronization primitives (e.g., compare-and-swaps and load-locked/store-conditionals). We call this ring buffer *BasicRingBuffer* and Figure 3 shows its implementation. BasicRingBuffer uses two control variables `read` and `write` to refer to the buffer slots that the consumer and the producer will extract from and insert to, respectively. Let `max` be the capacity of the ring buffer, and `NEXT()` be the function that advances `read` or `write` to the next buffer slot and wraps around the buffer if needed. The ring buffer is said to be full if `NEXT(write) == read`, and empty if `read == write`. The operations on `read` and `write` allow the producer and the consumer to concurrently access different buffer slots as long as the buffer is neither full nor empty.

Using the same single-producer/single-consumer model, FastForward [2], whose pseudo-code is shown in Figure 4, improves BasicRingBuffer by comparing the control variables directly with the buffer slots that hold data elements. The buffer slots use a special null value to denote whether they are empty slots. However, with this data/control coupling approach, 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[1].

Wang et al. [18] improve FastForward by aggregating the insert and extract operations on a per-cache-line basis to make the producer and the consumer work on different cache lines. However, the mechanism is useful only when the element size is less than that of a cache line. Also, [18] still uses the data/control coupling approach, which limits the data types of elements that can be transferred.

Many lock-free FIFO ring buffers have been proposed for the more complicated multiple-producer/multiple-consumer model (e.g., [8], [15], [17]), yet they are built on hardware synchronization primitives (e.g., compare-and-swaps and load-locked/store-conditionals). Given that such primitives are generally computationally expensive [2], we focus on the single-producer/single-consumer model without compromising our goal of achieving line-rate traffic monitoring.

B. Design

MCRingBuffer is built upon BasicRingBuffer to support concurrent lock-free accesses, and enhances BasicRingBuffer with a key objective to improve the cache locality of thread synchronization. In particular, MCRingBuffer is a software-based data structure without any hardware synchronization primitives. MCRingBuffer de-couples the data and control operations as in BasicRingBuffer, so that it can transfer data elements of general data types. Also, the producer and consumer threads do not need to specifically schedule their insert and extract operations to maximize the performance gain of MCRingBuffer.

We make several assumptions for MCRingBuffer, while these assumptions are inherited from BasicRingBuffer. First, the single-producer/single-consumer model is used. Also, reading and writing the shared control variables `read` and `write`, which are of integer type, are indivisible, atomic operations. In general, atomic operations on 32-bit integers are guaranteed in modern 32/64-bit CPUs such as Intel (see [5], Chapter 8). Furthermore, the memory model satisfies *sequential consistency* [11], in which the order of operations executed by all threads preserves the order of operations specified by each individual thread. Note that sequential consistency is guaranteed in the Intel architecture (see [5],

[1]The authors of [2] suggest to transfer pointers to an external storage over the ring buffer, so that the `NULL` pointer can be used to indicate empty slots. However, transferring a pointer implies two memory accesses, including inserting (extracting) the pointer to (from) the ring buffer, and inserting (extracting) the element payload referenced by the pointer to/from the external storage. This incurs one more memory access than transferring the element payload over the ring buffer directly.

Chapter 8). For other processor architectures with weaker consistency models, we need to include memory barriers to ensure proper ordering of memory accesses by multiple cores. From this point forward, we focus on the Intel architecture as our deployment platform.

Figure 5 shows the software-based implementation 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, which seeks to minimize the frequency of *reading* the shared control variables from main memory, and (ii) batch updates of control variables, which seeks to minimize the frequency of *writing* the shared control variables to main memory. We explain both features in detail as follows.

1) Cache-line Protection: When a variable has been accessed, it is placed in cache so as to reduce the latency of the subsequent accesses of the same variable. A requirement of efficient thread synchronization on multi-core architectures is to avoid *false sharing* [4], meaning that two threads each access different variables residing in the same *cache line*[2], the basic block of a cache. When a thread modifies a variable in cache, it invalidates the whole cache line as being modified. When another thread accesses a different variable on the same cache line, a cache miss will be triggered and the whole cache line will be reloaded from main memory, even though the variable being accessed remains unchanged.

Our goal is to carefully place the control variables so that the local, non-shared variables of different threads do not reside in the same cache line. We classify the control variables of MCRingBuffer into four groups: (i) shared variables `read` and `write`, which are accessed by both the producer and consumer threads, (ii) local variables for the consumer, (iii) local variables for the producer, and (iv) constant variables. We separate the variable groups with padding bytes whose length is large enough to put each variable group in a different cache line. This approach is known as *cache line protection*, in which modifying values within a variable group does not cause false sharing to other variable groups.

Note that the addition of padding bytes itself is not sufficient. It is important to re-design the synchronization operations so as to take advantage of cache-line protection. Instead of directly checking with shared control variables `read` and `write` for synchronization, the producer and the consumer can check with non-shared, local control variables that reside in their own cache lines. We let `nextWrite` and `nextRead` be the local variables of the producer and the consumer that point to the next buffer slot to be inserted to and extracted from, respectively, and let `afterNextWrite` denote `NEXT(nextWrite)`. Also, we let `localRead` and `localWrite` be the local variables that the producer and the consumer "guess" for the current values of `read` and `write`, respectively. When the pro-

[2]For example, the size of a cache line in Intel Xeon 5300 series is 64 bytes.

```
/* 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 variables*/
 7: int localWrite;
 8: int nextRead;
 9: int rBatch;
10: char cachePad2[CACHE_LINE - 3 * sizeof(int)];
11: /*producer's local 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 batchSize;
19: char cachePad4[CACHE_LINE - 2 * sizeof(int)];
20: T* element;

/* Insert: called by the producer */
function Insert(T element)
 1: int afterNextWrite = NEXT(nextWrite);
 2: if afterNextWrite == localRead then
 3:    if afterNextWrite == read then
 4:       return INSERT_FAILED;
 5:    end if
 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
15: return INSERT_SUCCESS;

/* Extract: called by the consumer */
function Extract(T* element)
 1: if nextRead == localWrite then
 2:    if nextRead == write then
 3:       return EXTRACT_FAILED;
 4:    end if
 5:    localWrite = write;
 6: end if
 7: *element = buffer[nextRead];
 8: nextRead = NEXT(nextRead);
 9: rBatch++;
10: if rBatch ≥ batchSize then
11:    read = nextRead;
12:    rBatch = 0;
13: end if
14: return EXTRACT_SUCCESS;
```

Figure 5. MCRingBuffer.

ducer (consumer) is about to insert (extract) an element, it first checks the condition `afterNextWrite` == `localRead` (`nextRead` == `localWrite`) to decide whether the buffer is *potentially* full (empty). If the buffer is potentially full (empty), then the producer (consumer) further checks the condition `afterNextWrite` == `read` (`nextRead` == `write`) to decide whether the buffer is *actually* full (empty), and updates `localRead` (`localWrite`) with the latest value of `read` (`write`). The producer (consumer) will insert (extract) elements whenever the buffer is neither potentially nor actually full (empty).

Here, the intuition is that when the producer (consumer) sets `localRead` = `read` (`localWrite` = `write`) and reloads `read` (`write`) from main memory, the shared variable `read` (`write`) may have been incremented multiple times by the consumer (producer) to refer to a few buffer slots ahead. Thus, the producer (consumer) only needs to access `read` (`write`) after more than one insert (extract) operation. This is in contrast to BasicRingBuffer, in which the producer (consumer) needs to reload `read` (`write`) from main memory every time the variable is incremented to refer to the next buffer slot. Therefore, MCRingBuffer reduces the frequency for the producer and consumer threads to read the shared control variables from main memory.

2) Batch Updates of Control Variables: In BasicRingBuffer, the shared variable `read` (`write`) is updated after every extract (insert) operation. When the producer (consumer) thread modifies `write` (`read`), the consumer (producer) is forced to reload from memory this shared variable when it checks for the empty (full) condition. The frequent updates of the shared variables cause the memory reload operations to occur frequently as well.

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. The variable `batchSize` is a tunable parameter determined by applications. We advance `read` (`write`) to the next block only after a `batchSize` number of elements have been extracted (inserted). Specifically, we put non-shared variables `wBatch` and `rBatch` in the local cache lines for the producer and the consumer, respectively. The value of `wBatch` (`rBatch`) is incremented by one for every element being inserted (extracted). If `wBatch` (`rBatch`) equals `batchSize`, the shared variable `read` (`write`) will then be updated. Our goal here is to minimize the frequency of writing the shared control variables to main memory.

If the arrival rate of elements is too small, then the variables `read` and `write` will not advance since they do so only after a `batchSize` number of elements have been processed. In such cases, elements may not be inserted (extracted) even the buffer is not full (empty). Therefore, we assume that the batch update scheme is applied to the scenario where elements are constantly available so as to update the control variables. Nevertheless, this is justified in many practical environments. For example, if we monitor a high volume of packets in a high-speed network, then we can apply our batch update scheme for the ring buffers that share the packet information (see Section III).

```
function Insert(T element)
 1: wait until write - read < max;
 2: buffer[write mod max] = element;
 3: write = write + 1;

function Extract(T* element)
 1: wait until read < write;
 2: *element = buffer[read mod max];
 3: read = read + 1;
```

Figure 6. The variant of BasicRingBuffer, where `read` and `write` are monotonically increasing.

To ensure that no elements are permanently stalled in the ring buffer, we can have the producer periodically inject unused elements to drive the control variables, where such elements will be discarded by the consumer. Note that the unused elements are defined by applications and are transparent to the underlying MCRingBuffer.

In addition to minimizing the updates of the shared control variables, a side benefit of our batch update scheme is to avoid false sharing of data elements. Note that we have the producer and consumer threads access different blocks, each of which has size given by the product of `batchSize` and the slot size. If the block size is large enough, then it is highly probable that the producer and consumer threads operate on data elements that are separated by more than one cache line (unless they work near the boundary of two blocks). This property follows the same line as in the batched producer-consumer model [4], where elements are inserted or extracted in batch, and in the temporal slipping algorithm [2]. However, both approaches in [2], [4] require the producer and consumer threads to specifically schedule the insert and extract operations. In contrast, our batch update scheme fundamentally achieves this property through the updates of control variables, and the throughput improvement is made independent of how the insert and extract operations on the data elements are scheduled.

C. Proof of Correctness

We now provide a sketch of proof for the correctness of MCRingBuffer. By correctness, we mean: (i) the producer and the consumer insert and extract an element only when the ring buffer is neither full nor empty, respectively, and (ii) the elements that the consumer extracts from the ring buffer are in the same order as they are inserted by the producer.

The correctness of MCRingBuffer is built upon that of BasicRingBuffer [10]. A formal proof based on formal axioms for the correctness of BasicRingBuffer is shown in [10], while a simplified sketch of proof is also given in [9]. To understand the correctness of BasicRingBuffer,

```
function Insert(T element)
 1: wait until nextWrite - localRead < max OR
    nextWrite - read < max;
 2: set localRead = read if nextWrite - localRead = max;
 3: buffer[nextWrite mod max] = element;
 4: nextWrite = nextWrite + 1;
 5: update wBatch, write as in Figure 5;

function Extract(T* element)
 1: wait until nextRead < localWrite OR
    nextRead < write;
 2: set localWrite = write if nextRead = localWrite;
 3: *element = buffer[nextRead mod max];
 4: nextRead = nextRead + 1;
 5: update rBatch, read as in Figure 5;
```

Figure 7. The variant of MCRingBuffer, where `read` and `write` are monotonically increasing.

we first consider an identical variant of BasicRingBuffer shown in Figure 6, where `NEXT(x)` is given by `x + 1`. This implies that the control variables `read` and `write` are always monotonically increasing. Given that reading and writing of `read` and `write` are atomic operations, [9], [10] show that if the conditions in line 1 of the insert and extract procedures of the BasicRingBuffer variant (Figure 6) hold, then the buffer is neither full nor empty when the specified buffer slot is accessed in line 2, respectively. We apply this intuition for MCRingBuffer. Specifically, we want to show that the producer and consumer threads access the buffer only if they satisfy the equivalent conditions that ensure the buffer is neither full nor empty, respectively.

We consider an identical variant of MCRingBuffer as shown in Figure 7, in which we replace `NEXT(x)` with `x + 1` so as to make all control variables monotonically increasing. We now prove the following theorems.

Theorem 1. *In MCRingBuffer, the producer inserts an element to the buffer only when the buffer is not full, and the consumer extracts an element from the buffer only when the buffer is not empty.*

Proof: We first consider the producer. In the MCRingBuffer variant, the producer inserts an element to the buffer only when at least one of the following conditions holds:

$$\texttt{nextWrite - localRead} < \texttt{max} \quad (1)$$

$$\texttt{nextWrite - read} < \texttt{max} \quad (2)$$

Note that `nextWrite` and `localRead` are only modified by the producer itself, so the producer always obtains their correct values.

The operations of Insert() in MCRingBuffer ensure that the condition $\texttt{localRead} \leq \texttt{read}$ holds. Note that `read` is modified by the consumer, and hence the producer may not obtain the latest value of `read`. More precisely, the producer may load `read` from memory right before the consumer updates the variable, and the producer checks for the full condition using the older value of `read`. However, since `read` is monotonically increasing, the value of `read` obtained by the producer is always no greater than the latest value of `read`. This implies that $\texttt{localRead} \leq \texttt{read}$ still holds.

In addition, the operations of Extract() always ensure that $\texttt{read} \leq \texttt{nextRead}$, and this implies $\texttt{localRead} \leq \texttt{nextRead}$. When the producer inserts an element to the buffer (i.e., when at least one of the conditions (1) and (2) holds), we ensure that the condition $\texttt{nextWrite - nextRead} < \texttt{max}$ must hold.

Similar to the above arguments, we can show that when the consumer extracts an element from the buffer, the condition $\texttt{nextRead} < \texttt{nextWrite}$ must hold.

Note that `nextWrite` and `nextRead` refer to the buffer slots accessed by the producer and the consumer, respectively. In essence, the variant of MCRingBuffer applies the same full and empty conditions as does the variant of BasicRingBuffer. Based on the correctness of BasicRingBuffer [10], we can infer that the theorem holds. ■

Theorem 2. *In MCRingBuffer, the elements extracted by the consumer appear in the same order as they are inserted by the producer.*

Proof: Consider the variant of MCRingBuffer. Note that `nextWrite` and `nextRead` are monotonically increasing and are incremented by one each time an element is inserted and extracted, respectively. Also, by Theorem 1, the producer and the consumer access the buffer only when the buffer is neither full nor empty, respectively. The theorem follows. ■

D. Evaluation

We now evaluate various lock-based and lock-free ring buffers, including: (i) LockRingBuffer (see Figure 2), (ii) BasicRingBuffer [10] (see Figure 3), (iii) FastForward [2] (see Figure 4), and (iv) MCRingBuffer with different values of `batchSize` (see Figure 5).

For MCRingBuffer, we test different values of `batchSize` to see how each design component affects the performance. With $\texttt{batchSize} = 1$, we show the improvement of applying cache-line protection alone, while with $\texttt{batchSize} > 1$, we show the additional improvement of applying the batch updates of control variables.

Assumptions: Since the lock-free ring buffers considered in this paper assume the single-producer/single-consumer model, we focus on having two running threads: one producer and one consumer. Section III also describes how to allow shared data accesses among more than two threads by using multiple ring buffers. In particular, we assume that both the producer and consumer threads use busy-waiting to poll for the availability of a ring buffer, as the sleep-and-wait approach generally gives poor performance due to context switching. For LockRingBuffer, we use a spin lock instead of a sleep-and-wait lock.

Evaluation testbed: We conduct our evaluation on a machine equipped with two Intel Xeon 5355 64-bit quad-core 2.66 GHz CPUs (i.e., a total of eight cores) and 32 GB RAM. Our evaluation here uses only one of the CPUs, while in Section III we use both. Each CPU is composed of two replicas of dual-core modules [4], each with a pair of cores and a shared second-level (L2) cache. The L2 cache provides fast shared data accesses for the core pair in the same module. We call a pair of cores *sibling cores* if they reside in the same module and share an L2 cache. On the other hand, we call a pair of cores *non-sibling cores* if they belong to the same CPU, but reside in different modules and do not share an L2 cache[3]. We randomly pick a pair of sibling cores and a pair of non-sibling cores, and then evaluate the cases when both threads are bound to the selected core pairs. Note that consistent results are observed for different sibling and non-sibling core pairs.

Our evaluation machine runs Linux 2.6.18. The ring buffers are written in C++ and compiled using GCC 4.1.2 with the `-O2` option.

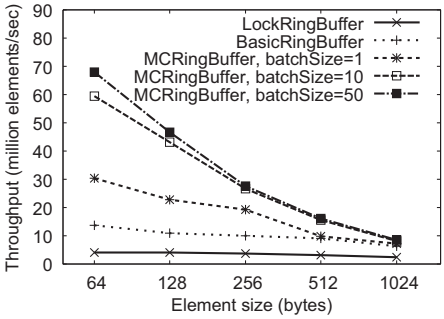
Evaluation metrics: For each ring buffer being evaluated, the producer thread inserts 10 M elements, and the consumer thread extracts the inserted elements in order. We are interested in two metrics: (i) the *throughput* (see Evaluations 1 to 3), defined as the number of pairs of insert/extract operations performed per second, and (ii) the *number of L2 cache misses* (see Evaluation 4), which denotes the number of times that a cache line needs to be reloaded from main memory. Unless otherwise stated, each data point is obtained from the average result over 30 trials.

Evaluation 1 (Throughput versus element size). We first evaluate the throughput versus the element size for LockRingBuffer, BasicRingBuffer and different MCRingBuffer variants. We fix the capacity of each ring buffer to be 2,000 elements. Figures 8(a) and 8(b) show the results when the producer and consumer threads reside in sibling cores and non-sibling cores, respectively.

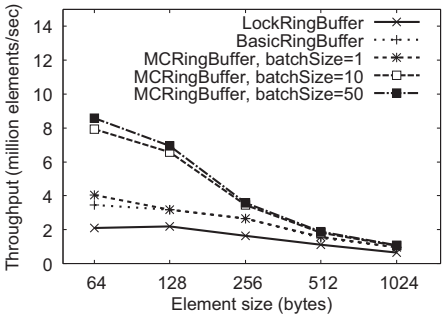
As expected, LockRingBuffer has the smallest throughput as compared to other lock-free ring buffers, which allow concurrent thread accesses. The throughput gains of BasicRingBuffer are 2.6−3.3× and 1.4−1.6× for sibling and non-sibling cores, respectively, and the gains of MCRingBuffer (with `batchSize` = 50) are 3.6−16.6× and 1.6−4.1× for sibling and non-sibling cores, respectively [4].

[3]In Intel Xeon 5300 series, non-sibling cores communicate through a system bus, although they might reside in the same chip but different modules. Some quad-core technologies such as AMD Opteron provide an L3 cache for the four cores on the same chip.

[4]Ideally, we expect that since lock-free ring buffers allow the concurrent accesses of both producer and consumer threads, they should achieve at least a 2× throughput gain over LockRingBuffer, which can only be accessed by one thread at a time. However, different factors may decrease the actual throughput gain. For example, the producer (consumer) is blocked when the buffer is full (empty). We plan to look into the possible factors in future work.



(a) Sibling cores



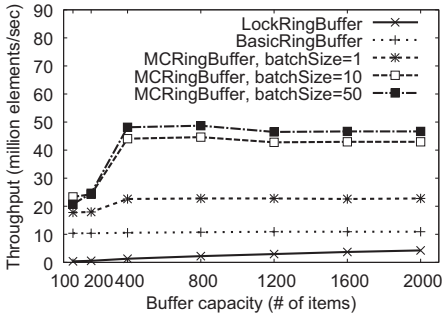
(b) Non-sibling cores

Figure 8. Evaluation 1: Throughput vs. element size.

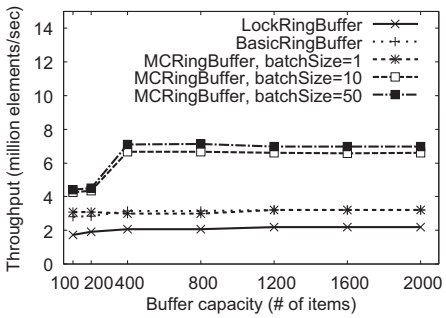
In general, MCRingBuffer provides higher throughput than BasicRingBuffer. In particular, the throughput gain of MCRingBuffer is more significant for smaller elements, since a larger proportion of time is spent on synchronizing the control variables and the design goal of MCRingBuffer is to minimize the synchronization overhead.

Let us focus on the case where the element size is 64 bytes. For sibling cores (Figure 8(a)), the throughput gains of MCRingBuffer with `batchSize` = 1 and `batchSize` = 50 are 2.2× and 4.9× over BasicRingBuffer, respectively. Thus, the cache-line protection alone (`batchSize` = 1) already improves the throughput over BasicRingBuffer, and this gain is further amplified with the batch updates of control variables. On the other hand, the throughput for non-sibling cores (Figure 8(b)) is less than that in sibling cores since elements must be sent through the system bus. Nevertheless, MCRingBuffer can increase the throughput with `batchSize` $>$ 1. For example, the throughput of MCRingBuffer with `batchSize` = 50 is 2.5× over BasicRingBuffer.

Evaluation 2 (Throughput versus buffer capacity). We now evaluate the throughput of LockRingBuffer, BasicRingBuffer, and different MCRingBuffer variants under various buffer capacities. We fix the element size to be 128 bytes. Figures 9(a) and 9(b) show the results. As the buffer capacity is small, the producer thread is likely to be blocked on insertion, and hence the throughput is smaller. However, the throughput becomes more or less the same



(a) Sibling cores



(b) Non-sibling cores

Figure 9. Throughput vs. buffer capacity.

when the buffer is large enough. For example, when the buffer capacity is at least 400 elements, the throughput gains of MCRingBuffer with `batchSize` = 50 are about 4.2× and 2.2× over BasicRingBuffer for sibling and non-sibling cores, respectively. Again, LockRingBuffer has the smallest throughput as compared to all lock-free ring buffers.

Evaluation 3 (Comparison with FastForward [2]). We now include FastForward in our evaluation. Our FastForward implementation is based on the pseudo-code in [2] (see Figure 4), in which each element is a variable pointer. Thus, we only focus on the case where all elements are 64-bit pointers (i.e., the size of each element is 8 bytes).

Here, we simply have the producer (consumer) insert (extract) elements as fast as possible. In other words, the producer calls the insert operation when an element is available, and the consumer calls the extract operation when it is ready to process the next element. This allows us to evaluate the robustness of all ring buffers toward a straightforward implementation of the producer and consumer threads. Thus, our FastForward implementation is a baseline version that does not specifically schedule the producer and consumer threads to work on different cache lines. Also, like BasicRingBuffer, our FastForward implementation does not explicitly enforce the cache locality of accessing control variables (e.g., no cache-line protection is used). As shown here and in Evaluation 4, the cache locality of thread synchronization, which is the key design emphasis of MCRingBuffer, is critical to the performance gain.

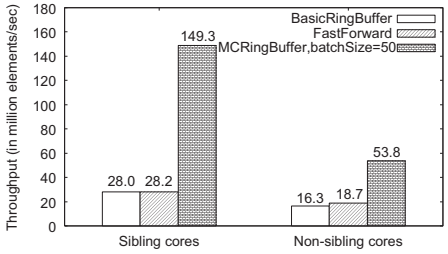


Figure 10. Evaluation 3: Comparison with FastForward, element size = 8 bytes, buffer capacity = 2,000 elements. MCRingBuffer has `batchSize` = 50.

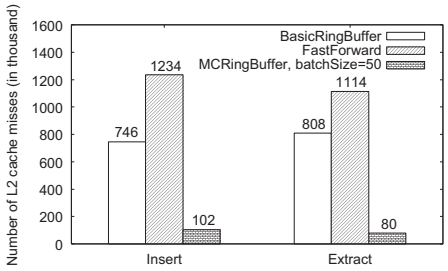


Figure 11. Evaluation 4: Number of L2 cache misses.

Here, we focus on evaluating the lock-free ring buffers. Figure 10 shows the throughput results where the element size is 8 bytes and the buffer capacity is 2,000 elements. The throughput of FastForward is similar to that of BasicRingBuffer for sibling cores, but is 15% higher for non-sibling cores. On the other hand, MCRingBuffer (with `batchSize` = 50) gives 5.3× and 2.9× throughput improvement over FastForward for sibling cores and non-sibling cores, respectively.

Evaluation 4 (Impact of L2 cache misses). We now analyze how cache misses adversely affect the throughput of a ring buffer. We profile our ring buffers using the Intel VTune Performance Analyzer [3], which provides a breakdown of performance metrics for the insert and extract operations. In particular, we are interested in the number of L2 cache misses[5], each of which implies a memory access operation. In general, it is desirable to have a small number of L2 cache misses, as memory accesses are significantly costly.

We focus on using sibling cores, as both of them share an L2 cache. We use the same configuration as in Evaluation 3, where the element size is 8 bytes and the buffer capacity is 2,000 elements. For MCRingBuffer, we use `batchSize` = 50. We profile each ring buffer over five trials and obtain the average results.

Figure 11 shows the total number of L2 cache misses obtained from VTune. MCRingBuffer (with `batchSize` = 50) incurs 88% and 92% fewer cache misses than BasicRingBuffer and FastForward, respectively.

[5]The corresponding event in VTune is called MEM_LOAD_RETIRED.L2_LINE_MISS.

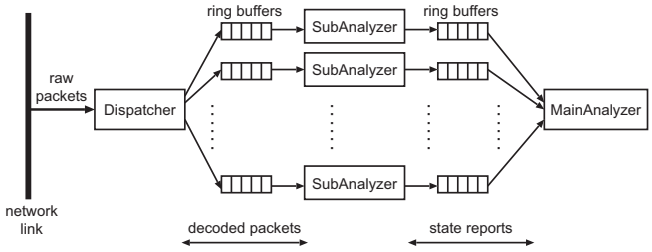


Figure 12. Parallel traffic monitoring system.

Summary. By organizing the layout of definitions of control variables carefully and accessing the control variables in a cache-efficient manner, MCRingBuffer significantly reduces cache misses and hence improves throughput over BasicRingBuffer and FastForward. The throughput improvement is seen for various element sizes and buffer capacities, regardless of whether the producer and consumer threads reside in sibling or non-sibling cores.

III. Parallel Traffic Monitoring

We now motivate the applicability of MCRingBuffer in parallel traffic monitoring. Our major goal is to speed up packet processing through minimizing the overhead of inter-core communication, while maintaining the correctness of the overall analysis as seen in the single-threaded version. We validate our goal via trace-driven simulation.

A. Design

Figure 12 illustrates one possible parallel traffic monitoring system that can use MCRingBuffer. The system divides the traffic monitoring into three stages: *dispatch stage*, *sub-analysis stage*, and *main analysis stage*, which are composed of a Dispatcher thread, a number of SubAnalyzer threads, and a MainAnalyzer thread, respectively. The details of each stage are described as follows.

1) Dispatch Stage: The dispatch stage consists of the *Dispatcher* thread, which decodes raw packets captured from the monitored network link and splits the decoded packets to the correct SubAnalyzer. Our current prototype focuses on the packet-header analysis (e.g., identifying portscan events), and hence each decoded packet includes its arrival timestamp and packet header. The decoded packet is then sent across one of the ring buffers. Note that we can also include packet payload for more complicated analysis.

We partition packets into SubAnalyzers based on a stateless approach [14], [19], in which we hash the sum of the source and destination IP address values of each packet into the correct SubAnalyzer. This stateless approach mitigates the processing load on the Dispatcher while ensuring that all packets of the same address pair go to the same SubAnalyzer. If a network is heterogeneous enough to be interleaved with all possible address pairs, then we expect that the processing loads of all SubAnalyzers are fairly balanced.

2) Sub-Analysis Stage: The sub-analysis stage is composed of multiple SubAnalyzer threads. Since we always forward the packets of an address pair to the same SubAnalyzer, each SubAnalyzer can individually conduct the *local analysis* on this address pair without coordinating with other SubAnalyzers. For example, each SubAnalyzer can monitor the statistics of all associated 5-tuple flows or the number of connections between the pair.

Each SubAnalyzer resides in a unique core. It polls for an available packet in a busy-waiting loop, and processes any packet extracted from the associated ring buffer.

3) Main Analysis Stage: The main analysis stage is composed of the MainAnalyzer thread, which aggregates the states of all SubAnalyzers and conducts the *global analysis*. For example, the MainAnalyzer may count the number of connections and the amount of data originated from a single source address.

Each SubAnalyzer collects partial results for the global analysis, and notifies the collected results to the MainAnalyzer in the form of *state reports*. Note that a state report is sent only when it leads to a "change" in the global analysis result. For example, if we monitor the connection counts of every host, a SubAnalyzer sends state reports only when a host establishes or releases a connection; if we monitor the originated traffic of every host, a SubAnalyzer sends state reports for every M bytes sent by a host, where M is the scale of the units used in the traffic counts. In general, we expect that the number of state reports sent from SubAnalyzers to the MainAnalyzer is significantly less than the total number of packets being monitored.

The MainAnalyzer checks the ring buffers in a round-robin manner. It extracts and processes any state report whenever it is available in a ring buffer. Here, we assume that the state reports contain frequency counts (e.g., the number of portscans) so that the MainAnalyzer can aggregate the counts in the state reports originated from all SubAnalyzers.

4) Discussion: We point out that Figure 12 is only one possible design that demonstrates the benefits of parallelism in traffic monitoring. We note that the Dispatcher uses a stateless approach to dispatch packets, and that the MainAnalyzer only processes a small number of state reports summarized by the SubAnalyzers. On the other hand, all SubAnalyzers need to process all captured raw packets. Thus, we assume that the Sub-Analysis stage is the major performance bottleneck of the system. In this case, we seek to minimize its workload by increasing the number of SubAnalyzers (see Section III-B).

Also, we run the Dispatcher, a varying number of SubAnalyzers, and the MainAnalyzer as separate threads, each of which is deployed on a different core. Since we seek to utilize all available cores, it is unavoidable to have shared data accesses between non-sibling cores (see Section II-D). For simplicity, we let the kernel decide the core assignment for the threads.

B. Trace-driven Simulation

We now evaluate our parallel traffic monitoring system using trace-driven simulation and demonstrate its capability of processing high-speed traffic. In particular, we want to show that the performance gain of our system can be further enhanced with MCRingBuffer.

1) Evaluation Approach: To validate our parallel system, we focus on detecting *portscans* [6], [16]. While there are many ways of defining a portscan, we consider a portscan that is defined as a *failed connection attempt* [6], in which a source host sends a TCP SYN packet to a distinct destination IP/port pair and does not receive a SYN-ACK packet in response. We look into two types of portscans: *vertical portscans* and *horizontal portscans* [16]. In a vertical portscan, a source host scans a number of distinct ports on a single destination host, while in a horizontal portscan, a source host scans a number of distinct destination hosts on a particular port. Our objective is to identify the source hosts that launch more than a user-defined threshold number of portscan events. Here, we set the thresholds for both portscans to be 20.

Based on the above definitions of portscans, we can monitor vertical portscans in the sub-analysis stage, in which each SubAnalyzer monitors the number of failed connection attempts for each address pair. On the other hand, we monitor horizontal portscans in the main analysis stage. If each SubAnalyzer sees a source host send a SYN packet to, or receive a SYN-ACK packet from, a distinct destination IP/port pair, then it sends a state report to the MainAnalyzer, which can then count the number of failed connection attempts for the source host.

By monitoring both vertical and horizontal portscans, we validate the *correctness* of the local and global analyses in our system, respectively. Here, by correctness, we mean that our parallel system returns the same set of portscan events as in the single-threaded version where we run our packet collection and processing within the same thread. Therefore, improving the accuracy of our detection algorithm (i.e., minimizing the false positive and negative rates of portscans) is not our main emphasis. Instead, we are interested in the performance gain of our parallel system.

Our evaluation is based on a one-hour packet header trace file collected from an operational wireless data network. The trace contains about 82 M packets that account for a total of 8.1 GB. The trace provides us reference distributions of data traffic and address pairs in a real network setting. To analyze the performance of our system at the peak traffic rate, we load the entire trace file into memory, and have our system process the packet headers as fast as possible. Loading the trace file into memory enables us to eliminate the overhead due to disk read, and hence the performance bottleneck should lie within our system. We then measure the time required by our system to process all packet headers within the trace.

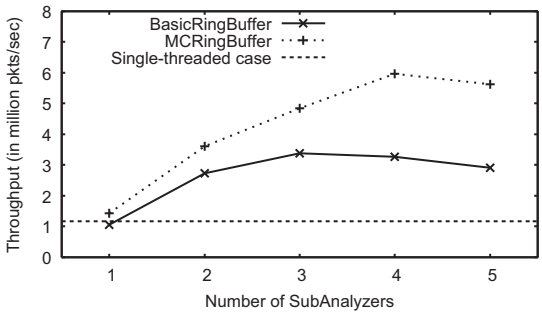


Figure 13. Packet processing rate of our parallel traffic monitoring system. We set aside one core to handle background kernel jobs. Hence, we test up to five SubAnalyzers, and use a total of up to seven cores for our system.

In essence, our evaluation is a *stress test*, since our system monitors a condensed version of one-hour traffic in a short time window. The number of portscans being identified is expected to be higher than in practice.

We evaluate our system on the Intel Xeon machine used in Section II-D. We consider BasicRingBuffer and MCRingBuffer, both of which have buffer capacity set to 2,000 elements. Specifically, for MCRingBuffer, we set `batchSize` = 50 for the ring buffers between the Dispatcher and each SubAnalyzer. In high-speed networks, we expect that the volume of monitored packets is large enough to drive the increments of the shared control variables `read` and `write` (see Section II-B). On the other hand, we set `batchSize` = 1 for the MCRingBuffer between each SubAnalyzer and the MainAnalyzer, since state reports arrive less frequently and we want to process them immediately.

Also, we implement a single-threaded version, which decodes packets and performs both local and global analyses in a single core without using any ring buffer. We validate that both the single-thread version and our parallel system generate the same set of portscans.

2) Evaluation Results: Figure 13 shows the throughput (i.e., packet processing rate) versus the number of SubAnalyzers, where each data point is averaged over 10 trials.

The case of using one SubAnalyzer can be viewed as *pipelining* the processing of each packet into the dispatch, sub-analysis, and main analysis stages. However, this pipelined version, when using BasicRingBuffer, has smaller throughput than the single-threaded version by 10%, mainly due to the inter-core communication overhead. On the other hand, the pipelined version with MCRingBuffer mitigates this overhead and improves the throughput over the single-threaded version by 23%.

In general, our parallel system (either with BasicRingBuffer or MCRingBuffer) outperforms the single-threaded version with an increasing number of SubAnalyzers since we parallelize traffic monitoring. Using MCRingBuffer, our system with MCRingBuffer achieves a maximum of 5.2× throughput gain over the single-threaded version. Further-

more, if we fix the number of SubAnalyzers being used, using MCRingBuffer has higher throughput than using BasicRingBuffer by 30−90%.

The throughput of our parallel system increases with the number of SubAnalyzers until a certain number (e.g., five) is reached. One possible reason is that when the Dispatcher and MainAnalyzer threads each access more ring buffers, they need to access more control variables that contend for the finite cache space, and hence more cache misses could be expected. Thus, parallelizing the packet processing with too many threads may degrade the overall performance. A similar observation is also found in [19].

In our evaluation, our system with MCRingBuffer can process the entire one-hour trace in less than 15 seconds (when the number of SubAnalyzers is between 3 and 5). Certainly, in practical environments, the throughput of our parallel system greatly depends on the complexity of the traffic analysis. Our goal here is to demonstrate how MCRingBuffer exploits the full potential of multi-core architectures in achieving line-rate traffic monitoring.

IV. Related Work

In Section II-A2, we review different lock-free ring buffers proposed in the literature. Here, we focus on reviewing existing parallel network traffic monitoring schemes.

Kruegel et al. [7] propose a parallel, stateful intrusion detection architecture that partitions and processes traffic on different PCs (called sensors). Foschini et al. [1] extend [7] with a parallel matching algorithm for stateful signatures. Weaver and Sommer [19] propose an intrusion detection system using a cluster of end hosts. Paxson et al. [12] sketch a network monitoring system tailored for multi-core architectures, and their main focus is on how to dispatch packets to the analysis threads on different cores. Wang et al. [18] apply the variant of FastForward [2] and show that multi-core-based processing can boost the packet processing speed to Gigabit levels. Similar work of multi-core-based traffic monitoring is also found in [13], [20], whose major focus is on the algorithmic design, such as flow classification [13] and load balancing [20]. In contrast, our work on MCRingBuffer mainly addresses a shared data structure that minimizes the cost of inter-core (or inter-thread) communication and applies to general multi-threaded, multi-core network traffic monitoring systems.

V. Conclusions and Future Work

We propose a basic primitive that enables line-rate network traffic monitoring. We present MCRingBuffer, a lock-free, cache-efficient ring buffer that achieves efficient thread synchronization in multi-core architectures. MCRingBuffer improves the cache locality of thread synchronization via: (i) cache-line protection and (ii) batch updates of control variables. Also, MCRingBuffer is a software-based data structure that supports generic data types of elements, and threads do not need to schedule their insert and extract operations to adapt themselves to MCRingBuffer. We prove the correctness of MCRingBuffer. We also show via simulation and code profiling that MCRingBuffer increases the throughput of inter-core data accesses over conventional lock-free ring buffers.

We then develop a parallel traffic monitoring prototype that embodies MCRingBuffer. Using trace-driven simulation, we show that MCRingBuffer boosts the speed of packet monitoring. This justifies how multi-core architectures can benefit from MCRingBuffer in achieving line-rate traffic monitoring.

In addition to network traffic monitoring, MCRingBuffer can be used by general multi-threaded applications that involve shared data accesses. In future work, we plan to explore more possible applications that can be built upon MCRingBuffer.

References

[1] L. Foschini, A. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A Parallel Architecture for Stateful, High-Speed Intrusion Detection. In *Proc. of ICISS*, Dec 2008.

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

[3] Intel Corporation. Intel VTune. http://software.intel.com/en-us/intel-vtune/.

[4] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, Mar 2009.

[5] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, Jun 2009.

[6] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *IEEE Symp. on Security and Privacy*, 2004.

[7] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *IEEE Symp. on Security and Privacy*, May 2002.

[8] E. Ladan-Mozes and N. Shavit. An Optimistic Approach to Lock-free FIFO Queues. *Distributed Computing*, 20:323–341, 2008.

[9] L. Lamport. Concurrent Reading and Writing. *Communications of the ACM*, 20(11), Nov 1977.

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

[11] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9), Sep 1979.

[12] V. Paxson, R. Sommer, and N. Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. In *IEEE Sarnoff*, 2007.

[13] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li. Towards High-performance Flow-level Packet Processing on Multi-core Network Processors. In *ACM/IEEE ANCS*, 2007.

[14] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Proc. of Computing Frontiers (CF)*, May 2005.

[15] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable Synchronous Queues. In *PPoPP*, 2008.

[16] S. Staniford, J. A. Hoagland, and J. M. McAlerney. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security*, 10:105–136, 2002.

[17] P. Tsigas and Y. Zhang. A Simple, Fast and Scalable Non-blocking Concurrent FIFO Queue for Shared Memory Multiprocessor Systems. In *Proc. of ACM SPAA*, 2001.

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

[19] N. Weaver and R. Sommer. Stress Testing Cluster Bro. In *DETER workshop*, 2007.

[20] Q. Wu and T. Wolf. On Runtime Management in Multi-Core Packet Processing Systems. In *ACM/IEEE ANCS*, 2008.