Parallelizing Sequential Network Applications with Customized Lock-Free Data Structures

[1]Junchang Wang, [2]Kai Zhang, [3]Bei Hua

[1,2,3]School of Computer Science and Technology
University of Science and Technology of China (USTC)
Hefei, Anhui 230027, China

[1,2,3]Suzhou Institute for Advanced Study, USTC
Suzhou, Jiangsu 215123, China
{[1]wangjc, [2]kay21s}@mail.ustc.edu.cn, [3]bhua@ustc.edu.cn

***Abstract*—Parallelizing fine-grained network applications on general-purpose multi-core architectures is a big challenge, as it requires fast core-to-core synchronization that is not supported for now. This paper proposes a methodology for parallelizing legacy sequential network applications on multi-core architectures with only minor code rewriting. Single Program Multiple Data (SPMD) parallel programming style is used to retain the control flow of sequential code, and then data structures potentially involving data racing are replaced with customized lock-free data structures based on domain knowledge. We evaluate the methodology by parallelizing a sequential TCP SYN flood detection program on Intel Quad processors, and gain inspiring results.**

Keywords-Parallel, network applications, customized lock-free data structure, concurrent programming.

I. INTRODUCTION

Industry's wide shift to general-purpose multi-core architectures arouses great interests in parallelizing legacy sequential applications. Although coarse-grained applications where a thread runs over a few milliseconds or seconds to finish a task can be successfully parallelized, fine-grained network applications in which a packet must be processed within a microsecond are very difficult to be parallelized, as it requires fast core-to-core synchronization that general-purpose multi-core architectures do not support for now. For example, to achieve 1Gbps line-rate, each packet must be processed within 672ns on average; however one mutex operation alone may be over such time limit when contention is high.

This paper explores a method for parallelizing legacy sequential network applications on multi-core architectures with customized lock-free data structures. Network applications have inherent parallelism where packets belonging to different flows can be processed independently [16]. This feature can be exploited to facilitate the parallelization of sequential code by factoring data structures so that each process can operate on separate data set when sharing is not required. [1] [16] [24] adopt this method. However, data sharing in some applications is inevitable, e.g., TCP SYN flood detection system needs to count the number of half-open TCP connections to or from specific IP addresses, and HTTP performance monitoring system needs to calculate the average request-response time between the checking point and the target servers. In these cases, the event counting table shared by all cores becomes the performance-critical data structure, and a high efficient mechanism must be designed to solve the data synchronization problem.

Lock-free data structures implement concurrent objects without the use of mutual exclusion. They are promising candidates to support synchronization in multi-core architectures since the resulting algorithms are scalable and free of deadlock [6] [10]. However, a concurrent lock-free object involving thousands of lines of code is extremely hard for programmers to understand and reason about in general [5]. Moreover, the general-purpose lock-free data structure is not suitable to multi-Gbps applications due to high runtime overhead. We solve the problem with customized lock-free data structure, which retains the original data structure as much as possible to ease the parallelization, and only fields potentially involving data racing are protected by atomic instructions. As a case study, we parallelize a sequential TCP SYN flood detection program with only minor changes to a hash based event counting table. Performance of the parallelized system is more than doubled with six cores, and the processing speed remains above 1Gbps even under heavy attacks. This is a solid evidence for the effectiveness of our method.

The remainder of this paper is organized as follows. Section II discusses related work. Section III states the problem we try to solve with. Section IV presents the algorithm and its implementation. Section V proves the correctness of our lock-free data structure. Section VI presents and analyzes the performance evaluation results. Section VII concludes.

II. RELATED WORK

There is abound research on lock-free data structures. We only list those that have direct impacts on our design.

This work was supported by the Fundamental Research Funds for the Central Universities under Grant No. WK0110000007.

For general introduction to lock-free data structures, please refer to [3].

Lock-free linked lists have been studied in [15][17][19] and lock-free hash table in [15]. All of them are designed for general purpose, and most of the work is to guarantee the correctness of deleting operation by maintaining auxiliary nodes. Other issues, such as the ABA problem and lock free malloc, are discussed in [15][19] and [2][14], respectively.

A great deal of work applies various parallelism method on sequential network applications [1][16][22][23]. Those work, however, commonly rewrite the control flow of sequential code to utilize multicore processors, making the development and validation require a lot of work.

III. PROBLEM STATEMENT

A. *Sequential TCP SYN Flood Detection Algorithm*

TCP SYN flood attack attempts to exhaust the victim's resource by initializing a large number of TCP connections. We slightly modify the port scan program from Libnids [11] to build up a sequential TCP SYN flood detection engine, and its key data structure is shown in Fig. 1. It is a hash based event counting table that counts the number of new TCP requests to each host during a time interval. Input to the hash function is the *destination* IP address of a TCP SYN request, and a linked list (horizontal) is used to resolve hash collision. Each element of the linked list contains an array (vertical) that records all the source ends (identified by *<source IP address, source port>* pair) initializing TCP connections to the host, a *n_packets* field that counts the number of initiators, and a *time_stamp* field that records the latest time the element is accessed.

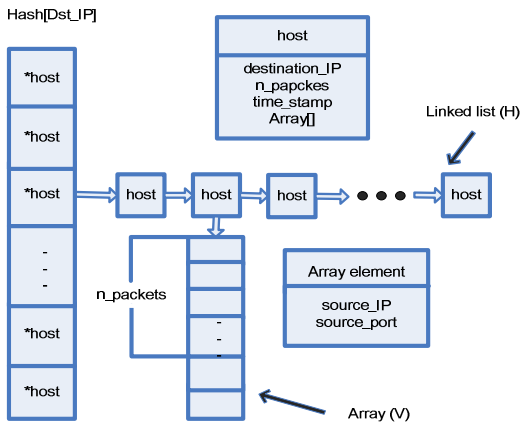

Fig. 1. Data structure used to detect TCP SYN flood attacks

Fig. 2 illustrates the work flow of the algorithm. When a TCP SYN request arrives, a hash function is performed on its *destination IP address*, and the corresponding linked list is searched to find an exact match. If a host node matches the *destination IP address*, its array is looked up with the *<source IP address, source port>* of the TCP SYN request. If the source end is already in the array, the algorithm returns without doing anything; otherwise, *n_packets* is increased by one and the pair is written in *array[n_packets]*. When *n_packets* goes up to a predefined threshold indicating that a super destination (probably the victim) is found, an alert is generated and the *array* is cleaned up by setting *n_packets* to zero. Each linked list has fixed number of elements, and the oldest element (with the minimal time_stamp) is updated with new host information when no match is found in the linked list.

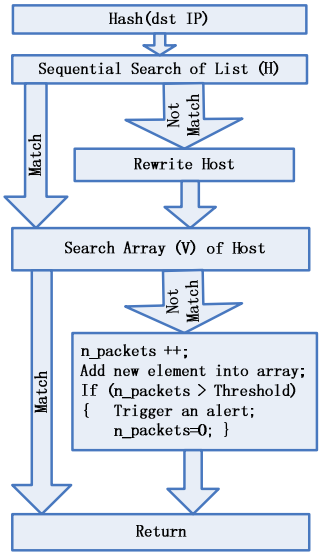

Fig. 2. Work flow of the TCP SYN flood detection engine

B. *Parallelizing Method*

We parallelize the TCP SYN flood detection engine in a parallel runtime system built in [16]. All CPU cores are organized in an N-way-2-stage pipeline fashion, where one core in the first stage receives and dispatches packets to other cores in the second stage by calculating a symmetric packet-classifying-hash function on a pair of *<source IP address, destination IP address>*, and a fast FIFO is used to connect two neighboring cores in each pipeline.

To parallelize the sequential engine in the parallel runtime system, a naive way is to use a mutex to lock a linked list before searching through it, and then each core can run the same code in parallel. However, the mutex-locked linked list may become a performance bottleneck when a large number of attacks target at a victim host. Since the linked list and each host node can be accessed in parallel if no update is needed (left branches in Fig. 2), a lock-free linked list is suitable for such fine-grained concurrency.

However, a general lock-free linked list cannot meet our performance goal due to its time-consuming deletion operation that may take more than 3,000 cycles (section VI.B). Fortunately, the sequential engine has no explicit delete operation, where clean up of *array* is done by setting *n_packets* to zero and the deletion of oldest host node is

done by in-place update. Therefore we may design a customized lock-free data structure that is as close as possible to the data structure in Fig. 1 so that: 1) code change is kept to the minimum; and 2) high-performance is achieved.

IV. Algorithm Design and Implementation

This section presents the design and implementation of the parallel lock-free TCP SYN flood detection engine.

A. Customized Lock-Free Linked List

By studying the work flow in Fig. 2, we identify the following two features that make the parallelization efficient.

a) The data structure is only used to count the events during a time interval, and is initialized whenever a new sampling interval starts. Therefore, the memory space taken by the data structure can be estimated and pre-allocated to avoid dynamic memory management. Such pre-allocation makes the hash table free of lock issues introduced by malloc [2].

b) With pre-allocated memory space, no insertion and deletion is needed in the algorithm. Since the major and toughest work of a general purpose lock-free data structure is to guarantee the correctness of its deletion operation, which usually involves maintaining auxiliary nodes and incurs high overhead in execution. Free of deletion means that we may customize a simple yet high-effcient lock-free data structure that is specific to this application.

Data racing may happen when more than one core try to operate on the same host node, either for rewriting an oldest node or updating some fields of a matched node. For example, if engine A and B happen to get a TCP SYN request bound for the same target *Host T* at the same time, they check the host T to get the same value of *n_packets*. If both senders are new to the array, both A and B increase *n_packets* by one and write information into *array[n_packets]*, which leads to contention. Similar situation may happen when two engines try to rewrite a single old node.

We design a customized lock-free linked list by making minor modification to the data structure of *Host*. The idea is to add a *tag* field in *Host* to track the update: the *tag* is increased whenever an update is made to the *Host* node, and all the updates must be executed atomically with a Compare-And-Swap (CAS) instruction. With such modification, concurrent updates to the same *Host* node are serialized, and data contention is avoided. In the above example, if A successfully increases *n_packets* by one, B fails to update the field and rolls back.

One limitation is that a CAS instruction can only protect a 64-bit data write. By carefully analyzing the update operation in Fig. 2, we figure out that only *n_packets* and *destination_IP* should be protected to make the event counting and node replacing automatically. The *time_stamp* is used to keep track of the oldest node for rewriting, and concurrent updates to this field do not harm the application, so we do not protect this variable. The data structure that must be updated atomically is depicted in Fig. 3.

```
typedef struct {
    unsigned int    addr;       /* 32-bits */
    unsigned short  n_packets;  /* 16-bits  */
    unsigned short  tag;        /* 16-bits  */
} _anchor;
```

Fig. 3. Data structure of *anchor* for avoiding contentions.

Fig. 4 illustrates the work flow of the parallel lock-free TCP SYN flood detection engine. Two anchor points and corresponding roll-backs are introduced into the two search loops, where a search will restart if instruction CAS fails at the end of the search loop. Since the two roll-back points collide, they are merged into one. Except for the two CAS instructions and the two roll-backs and anchor points in the search loops, original work flow remains the same.

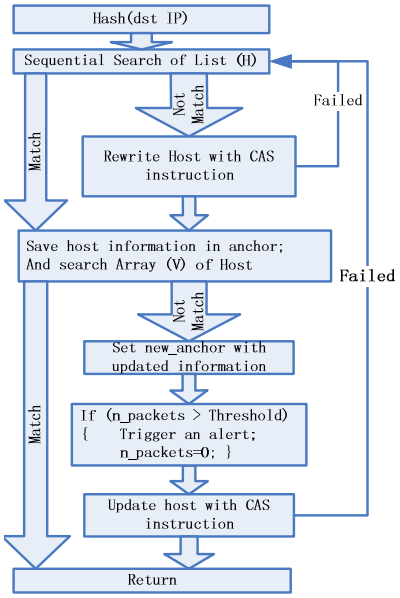

Fig. 4. Work flow of parallel TCP SYN flood detection engine

B. Implementation Details

This subsection discusses the algorithm implementation details, and focuses on the 64-bit X86-64 CAS instruction implementation. The data structure of anchor is shown in Fig. 3, and the pseudo-code of parallel TCP SYN flood detection engine is shown in Fig. 5.

Compared with the two sequential for-loop (lines 7-11, 25-28) based search algorithm, the lock-free algorithm has two nested outer do-until loops. The inner do-until loop (lines 5-21) is for rolling back if updating the oldest host node fails. The second part of the outer do-until loop (lines 4-37) is for rolling back if updating the *n_packets* field

fails. Two roll-back points are merged at line 6, the linked list entry point, which makes the roll-back simple without any other book-keeping efforts.

```
1:  int detect_ddos(ip_head * iph) {
2:      _anchor  anchor, new_anchor;
3:      hash_val = hash(iph->dst_addr);
4:      do {
5:          do {
6:              this_host=hashhost[hash_val];      //! Merged roll-back
    point
7:              for(i=0; this_host->addr != iph->dst_addr; i++)
8:              {   // Sequentially search linked list;
9:                  // Find out the oldest host or matched.
10:                 …
11:             }
12:             if( mached )
13:                 break;
14:             else  {  // Edit the oldest node.
15:               anchor.addr = this_host->addr;
16:               new_anchor.addr = anchor.addr;
17:               anchor.n_packets = this_host->n_packets;
18:               new_anchor.n_packets = 0;
19:               new_anchor.tag = anchor.tag + 1;
20:             }
21:         } until CAS(&this_host->addr, anchor, new_anchor);
22:
23:         anchor.tag = this_host->tag;
24:         anchor.n_packets = this_host->n_packets;
25:         for (i=0; i<anchor.n_packets; i++) {
26:           // Sequentially search array, and  find out matched
27:              …
28:         }
29:         if( matched )
30:             goto no_op;
31:         new_anchor.n_packets = anchor.n_packets + 1;
32:         new_anchor.tag = anchor.tag + 1;
33:         if(new_anchor.n_packets > threashhold) {
34:             Alert;
35:             new_anchor.n_packets = 0;
36:         }
37:     } until CAS(&this_host->addr, anchor, new_anchor);
38:
39:  this_host->packets[new_anchor.n_packets].addr = iph->src_addr;
40:  no_op:
41:     return 1;
42: }
```

Fig. 5. Pseudo-code of the parallel TCP SYN flood detection engine

In summary, the parallel algorithm is designed on the basis of existing data structure and operations. The goal is to keep the code change as minimal as possible by sticking to the original control flow and data logic as much as possible. Parallelization is carried out efficiently by exploiting some identified features (Section IV.A). First, the ABA problem is avoided by (1) memory pre-allocation by which no malloc related lock is introduced and no garbage collection is needed; 2) field *tag* has a unique value for each update. An in-place node update employs the atomic CAS instruction. If it fails, the control flow will roll back to the beginning of the linked list and restart the search all over.

V. PROOF OF CORRECTNESS

To prove the correctness of presented parallel algorithm, we recall that the memory space taken by this data structure can be estimated and pre-allocated to avoid dynamic memory management (Section IV.A), and as a result, no insertion and deletion is needed in the algorithm. By correctness of the parallel algorithm, we mean that for a given group of incoming packets, the parallelized algorithm gets the same result as the sequential one does.

A. Safety

For safety, we mean the parallelized TCP SYN flood detection algorithm does not introduce false positives or fail to report an alert. It is safe because it satisfies the following properties:

- Fig. 4 illustrates that, compared with the sequential version (Fig. 2), the parallelized algorithm mainly adds two anchor points and corresponding roll-backs, with other work flow remaining the same. The two back edges in Fig. 4 will roll back execution only if the commits fail. It is easy to see when only one thread is executing the algorithm, roll-backs will never occur, making the parallelized algorithm shown in Fig. 4 identical to the sequential version shown in Fig. 2.
- If roll-backs are incurred due to commit failure. The algorithm will discard completed work and roll back to the starting points without (1) saving status, (2) changing global status or (3) dropping this packet. The algorithm will restart from scratch. Since a symmetric packet-classifying-hash function is used to distribute packets belonging to a TCP flow to a single CPU core (Section III.B), there is no out-of-order problem between packets in a single connection. Roll-backs introduce latency, but they do not break data flow of processes, making the parallel algorithm get results as a sequential one.

In summary, the parallelized algorithm does not introduce false positives or fails to report an alert no matter there is competition between different processes or not.

B. Linearizability

The parallelized TCP SYN flood detection algorithm is linearizable because there are two linearization points (line 21 and 37 in Fig. 5) , and the program is considered to "take effect" [21] at linearization points. As shown in the previous subsections (Section IV.A), critical resources, including IP address and the number of packets, are encapsulated in *anchor* and all the updates must be executed automatically with CAS instructions in linearization points. These variables always reflect the state of the algorithm; they never enter a transient state in which the state of the algorithm can be mistaken.

C. Liveness

An algorithm is non-blocking if the suspension of one or more threads will not stop the potential progress of the remaining threads, and as a result the threads competing for

shared resources do not have their execution indefinitely postponed, guaranteeing the algorithm to complete within finite time.

The parallel TCP SYN flood detection algorithm loops only if the condition in line 21 (Fig. 5) fails or the condition in line 37 (Fig. 5) fails.

- The condition in line 21 fails only if *anchor* of this host is written by an intervening process after executing line 17. The data structure *anchor* shown in Fig. 3 always record information of a potential victim server, and when modified it shows a different server has been selected as the new victim server by other processes. Therefore, if the condition in line 21 fails more than once, then other processes must have succeeded in recording detected attacks.
- The condition in line 37 fails only if *anchor* of this host is written by an intervening process after executing line 32. This time, the data structure *anchor* records potential attackers, and when modified potential attackers have been identified and recorded by other processes. Therefore, if the condition in line 37 fails more than once, then other processes must have succeeded in recording potential attackers or even triggered an alert.

The above analysis also shows that the parallelized algorithm is *lock-free* because a process loops beyond a finite number of times only if another process completes an operation on the algorithm, making there a guaranteed system-wide progress.

VI. Experiments and Performance Analysis

A. Evaluation Platform

A Dell PowerEdge server with two Intel Core 2 Xeon E5530 processors is used in the experiments. Xeon E5530 runs at 2.4GHz with 8MB L3 cache. The server runs a 64-bit Linux 2.6.29 kernel and codes are compiled by the GCC 4.1.2 with –O2 option.

B. Performance of A General Lock-free Linked List

We measure the performance of a classical general-purpose lock-free linked list [14], whose implementation uses the algorithms described in [15] and SMR (Safe Memory Reclamation) [12] for memory management. On Xeon E5530, the average CPU cycles taken by insertion and deletion of the linked list are collected with different number of threads. Each thread firstly inserts N nodes into the list, and then deletes them by itself. The experimental results are shown in Table 1.

The experimental results show that even if there are only two threads, a deletion may take as high as more than 2000 CPU cycles or 869ns on Xeon E5530 processor. Obviously general-purpose lock-free linked list cannot meet the 1Gbps line-rate requirement in the worst case.

Table 1. Performance of general-purpose lock-free linked list

N=100,000	Operation	Threads number		
		1	*2*	*3*
With SMR	***Insert***	743	1753	2058
	Delete	1110	2234	2749
No SMR	***Insert***	566	1737	2685
	Delete	1004	2215	3051

C. Performance of TCP SYN Flood Detection Engine

For higher speed (>1Gbps) testing, we read the trace files into memory to simulate higher speeds.

Table 2. Characteristics of trace files

FILE	#Packets	#Len	#Alert	#RB	#RB Rate
File-1	3,181,141	243	13,558	181	1%
File-2	3,393,924	307	99,139	4,957	5%
File-3	9,143,457	84	1,717,272	301,628	18%
File-4	5,165,018	84	464,532	4,053	9%

We use four trace files whose characteristics are described in Table 2. File-1 comes from the 1998-1999 DARPA intrusion detection evaluation at MIT Lincoln Lab [4], and File-2 comes from the Defcon 9 [20]. File-3 and File-4 are generated by packet generators. Column *#Packets* is the total number of packets contained in the trace file, whose average size is in *#Len.*. *#Alert.* is the total number of alerts triggered. *#RB*. is the number of rollbacks, and *#RB. Rate* is the number of rollbacks per Alert.

Table 3 compares the performance of two algorithms implemented with mutex and lock-free linked list. Six cores are used and organized in a 5-way-2-stage fashion, where one core receives packets and dispatches them onto the five cores in the second stage. Column *#Lock* shows the performance of mutex-based algorithm, and column *#lock-free* shows the performance of lock-free algorithm. Performance is measured both in Mpps and Gbps. Column *Imp.* shows the performance (measured in Mpps) improvement of lock-free approach upon the mutex approach.

Table 3. Performance of two algorithms

FILE	#Lock		#lock-Free		Imp.
	#Mpps	*#Gbps*	*#Mpps*	*#Gbps*	
File-1	3.13	6.08	3.23	6.28	3%
File-2	2.63	4.08	3.18	4.94	21%
File-3	2.76	1.85	4.38	2.94	59%
File-4	1.34	0.90	2.31	1.55	72%
Ave.	2.47	3.22	3.28	3.93	33%

Because File-1 doesn't have many TCP SYN flood attacks, the rollback rate is low, and the performance improvement of lock-free algorithm is not obvious. File-3 and File-4 each has a large number of TCP SYN flood attacks that cause higher rollback rates, and the lock-free

engine achieves much higher performance improvement. On average, the lock-free engine achieves 33% performance improvement.

File-4 is generated as an extreme case where all the packets target the same *host* link list. In this case, the lock-free engine achieves 72% performance improvement, exhibiting robustness under heavy attacks. It is worth noting that only lock-free approach can achieve 1Gbps line rate under heavy attacks (File-4). Fig. 6 tests the scalability of lock-free engine with different number of cores. In this experiment, File-2 is used as the input file. We can see from the figure that when the number of cores increases, the system performance also increases.

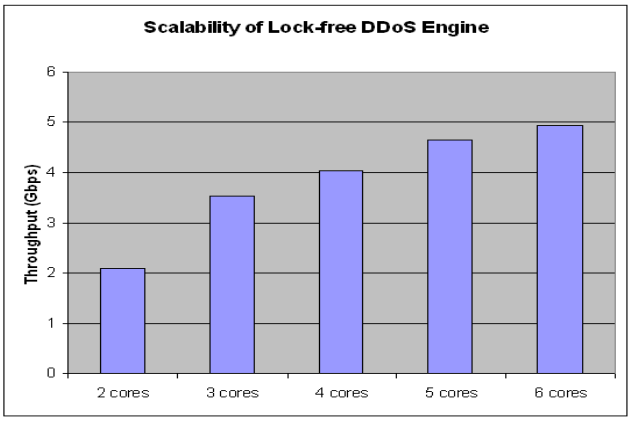

Fig. 6. Scalability of Lock-free Engine

VII. CONCLUSIONS

This paper presents a method to parallelize sequential network applications on multi-core architectures with minimal code modification. The main idea is to stick to the sequential control flow as much as possible, and only apply lock-free data structures at selective points to make sequential code run in parallel. Whenever it is possible, a customized lock-free data structure rather than a general-purpose one should be used to simplify the code rewriting and reduce the execution overhead.

VIII. REFERENCES

[1] J. Giacomoni, T. Moseley, and M. Vachharajani, "Fastforward: A cache-optimized concurrent lock-free queue," in PPoPP'08, New York, NY, USA, February 2008. ACM Press, 2008.

[2] L. C, Borheham D, "malloc() Performance in a Multithreaded Linux Environment," in USENIX 2000 Annual Technical Conference: FREENIX Track, May 2000.

[3] K. Fraser and T. Harris, "Concurrent programming without locks," In ACM Transactions on Computer Systems, Vol. 25 (2), May 2007.

[4] J. W. Haines, R. P. Lippmann, D. J. Fried, E. Tran, S. Boswell, and M. A. Zissman, "1999 DARPA Intrusion Detection System Evaluation: Design and Procedures," Technical Report 1062, MIT Lincoln Laboratory, 2001.

[5] B. Clare, "Application-Level Abstractions for Lock-Free Data Sharing," Dec. 21, 2007, http://www.ddj.com/cpp/205200452?pgno=1.

[6] W. N. Scherer, D. Lea, M. L. Scott, "Scalable Synchronous Queues," CACM, Vol. 52 No. 5, Pages 100-111, May, 2009.

[7] M. Spear, L. Dalessandro, V. Marathe, and M. Scott, "A Comprehensive Strategy for Contention Management in Software Transactional Memory," in 14th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP 2009), Feb. 2009.

[8] T. Shpeisman, A. Adl-Tabatabai, R. Geva, Y. Ni, A. Welc, "Towards transactional memory semantics for C++," Symposium on Parallelism in Algorithms and Architectures (SPAA), August 2009.

[9] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, M. Valero, "Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server," in Proc. of ACM PPoPP'09, Feb., NC, 2009.

[10] A. Dokumentov, "Lock-free Interprocess Communication," June 15, 2006, http://www.ddj.com/cpp/189401457.

[11] Libnids, http://libnids.sourceforge.net .

[12] M. M. Michael. "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes," In Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, July 2002.

[13] M. M. Michael, "Scalable Lock-Free Dynamic Memory Allocation," ACM SIGPLAN Notices, pp.35-46, 2004.

[14] M, M. Michael. "ABA prevention using single-word instructions," Technical Report RC 230089, IBM T. J. Watson Research Center, January 2004.

[15] M. M. Michael. "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets," SPAA'02, Manitoba, Canada, August, 2002.

[16] J. Wang, H. Cheng, B. Hua, and X. Tang, "Practice of Parallelizing Network Applications on Multi-core Architectures," In proc. of ACM ICS'09, New York, June, 2009.

[17] J. D. Valois. "Lock-free linked lists using compare-and-swap," In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 214-222, 1995.

[18] S. Doherty, M. Herligy, V. Luchangco, and M. Moir, "Bring Practical Lock-Free Synchronization to 64-Bit Applications," In Proc. of ACM PODC'04, Newfoundland, Canada, July 25-28, 2004.

[19] T. L. Harris. "A pragmatic implementation of non-blocking linked-lists," In Proceedings of the 15th International Symposium on Distributed Computing, pages 300-314, 2001.

[20] Shmoo Group. Defcon 9 Capture the Flag Data, Sept. 2001

Concurrent building blocks, http://amino-cbbs.sourceforge.net.

[21] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," ACM Transactions on Programming Languages and Systems, 12(3):463—492, July 1990.

[22] G. Upadhyaya, V. S. Pai and S. P. Midkiff, "Expressing and Exploiting Concurrency in Networked Applications in Aspen," In Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 13--23, 2007.

[23] G. Vasiliadis, M. Polychronakis and S. Ioannidis, "MIDeA: a multi-parallel intrusion detection architecture," Proceedings of the 18th ACM conference on Computer and communications security, 297--308, 2011.

[24] K. Zhang, J. Wang, B. Hua and X. Tang, "Building High-Performance Application Protocol Parsers on Multi-core Architectures," IEEE 17th International Conference on Parallel and Distributed Systems, 188-195, 2011.