TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Department of Computer Science
Database and Information Systems Group

Project Thesis:

Performance Evaluation of Different Open Source Implementations of Data Structures and Other Algorithms in the context of a DBMS Buffer Manager

---

by **Max Fabian Gilbert***

Day of release: January 31, 2019

*m_gilbert13@cs.uni-kl.de

Abstract

Needless to say, any database management system (**DBMS**) must be able to manage data. The data structures used to manage this data in a database (**DB**) have a great influence on various characteristics (e.g. performance) of a DBMS and therefore the use of certain data structures (e.g. B-tree indexes) and even some implementation details of them are very important decisions in DBMS design.

While (re-)implementing and refactoring the buffer management of my DBMS testbed *Zero*[1], I came across a number of less performance-critical data structures and algorithms that I wanted to replace. But since even these less performance-critical submodules can become performance-critical in certain situations, one should choose a reasonable implementation. Therefore, it is appropriate to use open source libraries that implement the required algorithms.

While the hash table for finding potentially buffered DB pages in the buffer pool is performance-critical, the free list of the buffer pool, which is used to manage free buffer frames, is not performance-critical. Some open source implementations of concurrent queues that can be used to manage the buffer pool's free list are evaluated in chapter 1.

For my master's thesis I am evaluating a number of page eviction strategies for the buffer manager. The most important performance metric of a page eviction strategy is the hit rate achieved under a given workload. But this performance metric does not change for different implementations of the RANDOM and LOOP page evictions strategies. Different implementations of these page eviction strategies change the overhead they cause during the already expensive buffer pool page misses. In chapter 2 I compare an enormous number of different pseudo-random number generators (**PRNG**s) available in open source libraries and in chapter 3 I compare some concurrent counters that can be used for the LOOP page displacement strategy.

[1] `https://github.com/iMax3060/zero`

This page intentionally left blank.

Contents

This page intentionally left blank.

1 Buffer Frame Free List

1.1 Purpose

Unless one relies on the operating system's inappropriate virtual memory management, any disk-based DBMS requires some sort of buffer manager that provides in-memory copies of the persistently stored DB pages for processing in the upper layers of the DBMS.

This feature is provided by the buffer pool management, which manages the currently used subset of DB pages—the working set—in buffer frames located in memory. In the general case of fixed-size pages, a buffer frame is a fixed-size portion of memory that can hold one DB page, and each of these frames has a frame index as an identifier.

During operation, the DB pages are fetched dynamically from the DB into buffer frames. As soon as a page is no longer needed in memory (i. e. it is no longer needed to process a running transaction), it can be evicted from the buffer pool to free up the buffer frame.

The buffer manager needs a kind of free list when assigning a buffer frame to a fetched DB page, because an unrestricted overwriting of data in buffer frames would lead to undefined behavior.

1.2 Compared Open Source Queue Implementations

To ease implementation of eviction strategies like CLOCK, a free list should use a FIFO data structure like a queue—typically implemented as a linked list. Therefore, the buffer frame that is released first is also (re)used first.

Almost every state-of-the-art DBMS supports multi-threading and thus there are usually multiple threads concurrently fetching pages into the buffer pool and evicting pages from the buffer pool. Accordingly, a buffer frame free list must support thread-safe functions to push and pop frame indexes in and out of the free list. Queues providing these thread-safe access functions are usually referred to as multi-producer (add frame indexes), multi-consumer (retrieve/remove frame indexes) queues (**MPMC** queues).

An approximate number of buffer indexes in the free list should also be provided by each free list implementation to support (batch-wise) eviction of pages as soon as there are only a few free buffer frames left. Thread-safe access to this number is desirable, but not mandatory.

1.2.1 Boost Lock-Free Queue with Variable Size

The popular *Boost C++ Libraries*[1] offer a **lock-free unbounded** MPMC queue[2] in the library `Boost.Lockfree`[3].

Like many other **non-blocking** thread-safe data structures, this MPMC queue uses atomic operations instead of locks or mutexes. To support dynamic queue growth and shrinking, this queue implementation also uses a free list for its own internal dynamic memory management.

This data structure does not provide the number of elements it contains, so an approximate number of buffer indexes in the free list must be managed externally.

1.2.2 Boost Lock-Free Queue with Fixed Size

This queue implementation is identical to the data structure in subsection 1.2.1, but does not use dynamic memory management internally—it is a **bounded** queue. As long as the required capacity of the queue—in our case the maximum number of buffer frames in the buffer pool—is known when the queue is allocated, this queue implementation can be used. This implementation uses a fixed size array instead of dynamically allocated nodes for its stored data—the indexes of the free buffer frames.

[1] `https://www.boost.org/`
[2] `https://bit.ly/2Q9w45H`
[3] `https://bit.ly/2FiPyip`

1.2.3 CDS Basket Lock-Free Queue

Among many other concurrent data structures, the *Concurrent Data Structures* C++ library[4] provides many different thread-safe queue implementations. The **unbounded** *basket lock-free queue*[5] is based on the algorithm proposed by M. Hoffman, O. Shalev and N. Shavit in [HSS07].

Internally, this queue does not use an absolute FIFO order. Instead, it places concurrently enqueued elements into one "basket" of elements. The elements within one basket are not ordered specifically, but the various "baskets" used over time are ordered by FIFO. Therefore, the dequeue operation just removes one of the elements in the oldest "basket". The queue's dynamic memory management uses a garbage collector to deallocate emptied "baskets".

1.2.4 CDS Flat-Combining Sequential Queue

The *Concurrent Data Structures* C++ library does also provides an **unbounded** thread-safe queue that uses flat combining[6]. The flat combining technique was proposed by D. Hendler, I. Incze, N. Shavit and M. Tzafrir in [Hen+10]. This technique can make any sequential data structure thread-safe—in case of the *flat-combining sequential queue*, the `std::queue`[7] of the *C++ Standard Library*[8] is used as the base data structure.

The flat combining technique uses thread-local publication lists to record operations performed by these threads. To combine these thread-local publication lists into the global sequential data structure, a **global lock** must be. The thread that acquired the global lock also combines the publication lists of all other threads, thus reducing the lock overhead. The elements of each operation executed during the combining are stored into the respective publication list together with the global combining pass number. A thread with a non-empty publication list that tries to acquire the global lock needs to wait till the combining thread updated its publication list.

[4] `https://github.com/khizmax/libcds`
[5] `https://bit.ly/2SGv2A6`
[6] `https://bit.ly/2ZEsrYO`
[7] `https://en.cppreference.com/w/cpp/container/queue`
[8] `https://en.cppreference.com/w/cpp`

1.2.5 CDS Michael & Scott Lock-Free Queue

Another **unbounded lock-free** queue implementation provided by the *Concurrent Data Structures* C++ library is based on the famous Michael & Scott queue algorithm[9] proposed by M. Michael and M. Scott in [MS96].

The Michael & Scott lock-free queue basically uses compare-and-swap (**CAS**) operations on the tail of the queue to synchronize enqueue operations of different threads. If a thread reads a NULL value as the next element after the end of the queue, it swaps that value atomically with the value enqueued by this thread. It then adjusts the tail pointer. If a thread does not read the NULL value there during the CAS operation, another thread has not already adjusted the tail pointer and this thread must retry its enqueue operation with the new tail pointer. The dequeue operation is implemented similarly. The memory already occupied by dequeued elements is deallocated using a garbage collector provided by the library.

1.2.6 CDS Variation of Michael & Scott Lock-Free Queue

The *Concurrent Data Structures* C++ library also provides an optimized variation of the Michael & Scott **unbounded lock-free** queue algorithm[10], which is based on the work of S. Doherty, L. Groves, V. Luchangco and M. Moir in [Doh+04].

This optimization of the Michael & Scott lock-free queue optimizes the dequeue operation so that the tail pointer is read only once.

1.2.7 CDS Michael & Scott Blocking Queue with Fine-Grained Locking

M. Michael and M. Scott have also proposed a blocking queue algorithm in [MS96]. This **unbounded blocking** queue implementation[11] is also provided by the *Concurrent Data Structures* C++ library.

This blocking queue algorithm uses one read and one write lock protecting the head and tail of the queue. Therefore, only one thread at a time can enqueue elements and only one thread at a time can dequeue

[9] `https://bit.ly/37onMwC`
[10] `https://bit.ly/2MG8dbM`
[11] `https://bit.ly/2SCeFo5`

elements. The deallocation of memory during dequeuing is done directly by the dequeuing thread instead of relying on a garbage collector.

1.2.8 CDS Ladan-Mozes & Shavit Optimistic Queue

The *Concurrent Data Structures* C++ library also provides an **unbounded optimistic** queue implementation[12] based on an algorithm proposed by E. Ladan-Mozes and N. Shavit in [LS04].

Instead of using expensive CAS operations on a singly linked list (as in the Michael & Scott lock-free queue), this algorithm uses a doubly linked list with the ability to detect and correct inconsistent enqueue and dequeue operations. Deallocation of memory is done using a garbage collector.

1.2.9 CDS Segmented Queue

The **unbounded** segmented queue implementation[13] of the *Concurrent Data Structures* C++ library is based on an algorithm proposed by Y. Afek, G. Korland and E. Yanovsky in [AKY10].

This thread-safe queue algorithm is very similar to the basket lock-free queue from subsection 1.2.3. It also uses a relaxed FIFO order, ordering segments containing multiple elements instead of individual elements. A thread enqueuing elements into the tail segment or dequeuing elements from the head segment randomly selects one of the slots within the segment. CAS operations are used to enqueue or dequeue elements from a slot atomically. If the CAS fails, another slot is chosen randomly. The size of each segment—which can be specified (8 was used for the performance evaluation in section 1.3)—determines the relaxation of the FIFO order. The deallocation of the emptied segments is performed by a garbage collector.

1.2.10 CDS Vyukov's MPMC Bounded Queue

The last thread-safe queue implementation[14] provided by the *Concurrent Data Structures* C++ library is **bounded** and was developed by D. Vyukov[15].

[12] `https://bit.ly/37mgQQM`
[13] `https://bit.ly/37mjXYR`
[14] `https://bit.ly/2ML9Y7M`
[15] `https://bit.ly/39n4PMF`

The queue in subsection 1.2.12 is his original implementation.

1.2.11 Folly MPMC Queue

Facebook's open source library *Folly*[16] provides a **bounded lock-free** queue implementation. An unbounded version is also provided, but due to the typically higher performance of bounded ones, it is not evaluated here.

Folly's MPMC queue uses a ticket dispenser system to give a thread access to one of the single-element queues used. Those ticket dispensers for the head and tail of the queue use atomic increment operations, which are supposed to be more robust against contention than CAS operations used in e.g. the Michael & Scott lock-free queue.

1.2.12 Dmitry Vyukov's Bounded MPMC Queue

This[17] is Vyukov's original implementation of his **bounded** thread-safe MPMC queue.

Vyukov's thread-safe queue implementation is very similar to Michael & Scott blocking queue with fine-grained locking from subsection 1.2.7, but instead of using mutexes as locks, his implementation uses atomic read-modify-write operations. This results in a cost of basically one CAS operation per enqueue/dequeue operation.

1.2.13 Gavin Lambert's MPMC Bounded Lock-Free Queue

This[18] is another version of Vyukov's thread-safe queue design from subsection 1.2.12 implemented by Gavin Lambert.

1.2.14 Matt Stump's Bounded MPMC Queue

This[19] is yet another version of Vyukov's thread-safe queue design from subsection 1.2.12 implemented by Matt Stump.

[16] `https://github.com/facebook/folly`
[17] `https://bit.ly/35a5lKL`
[18] `https://bit.ly/2F7jvlb`
[19] `https://github.com/mstump/queues`

1.2.15 Erik Rigtorp's Bounded MPMC Queue

The **bounded lock-free** queue[20] by Erik Rigtorp uses a ticket dispenser system similar to the MPMC queue of *Folly* from subsection 1.2.11.

1.2.16 TBB Concurrent Queue

The *Threading Building Blocks* library[21] is an open source library originally developed by Intel®. The first thread-safe queue implementation[22] of this library is **unbounded** and **non-blocking**.

Internally, this queue implementation uses multiple lock-based micro queues to allow concurrent enqueue/dequeue operations. Therefore, this thread-safe queue does not maintain the order of elements enqueued by different threads.

Due to the implementation with multiple single-producer, multi-consumer queues (**SPMC** queues), this queue implementation should only be used as a buffer frame free list if there is exactly one thread that is evicting pages from the buffer pool—and therefore, enqueuing buffer frame indexes of emptied buffer frames.

1.2.17 TBB Bounded Concurrent Dual Queue

The other thread-safe queue implementation[23] of the *Threading Building Blocks* library is **bounded** and **partially non-blocking**.

This queue implementation is almost identical to the other one of the *Threading Building Blocks* library, but allows the capacity limitation. An enqueuing operation must wait if the queue is already full according to the specified capacity.

[20] `https://github.com/rigtorp/MPMCQueue`
[21] `https://www.threadingbuildingblocks.org/`
[22] `https://software.intel.com/en-us/node/506200`
[23] `https://software.intel.com/en-us/node/506201`

1.3 Performance Evaluation

1.3.1 Microbenchmark

The used microbenchmark simulates a high contented free list. The number of working threads, the number of iterations (either fetching a page into a free buffer frame or evicting of a batch of pages) per thread, and the batch size of buffer frames to be freed at once can be varied. No complete buffer pool is simulated—there is only the free list with operations to enqueue and dequeue buffer frame indexes. Each working thread performs the following operations per iterations:

- If the free list is not empty:
  - Retrieve a buffer frame index from the free list.
  - Mark the retrieved buffer frame used.
- If the free list is empty:
  - While the free list is smaller than the batch eviction size:
    - Select a random buffer frame index using a fast random numbers generator.
    - If this buffer frame index is marked used:
      - Mark the selected buffer frame index unused.
      - Add the selected buffer frame index to the free list.

1.3.2 Used Versions of the Libraries and Queue Implementations

- *Boost C++ Libraries* 1.67
- *Concurrent Data Structures* C++ library 2.3.1[24]
- *Folly* `a8d1fd8`[25]
- Dmitry Vyukov's Bounded MPMC Queue as of September 2017[26]

[24] `https://bit.ly/39vysvB`
[25] `https://bit.ly/2sy4J4l`
[26] `https://bit.ly/2MHbXtG`

- Gavin Lambert's MPMC Bounded Lock-Free Queue `e409068`[27]
- Matt Stump's MPMC Queue `319c253`[28]
- Erik Rigtorp's MPMC Queue `553cf42`[29]
- Intel® Threading Building Blocks 2019 Update 9

1.3.3 Configuration of the Used System

- **CPU:** *Intel® Core™ i7-8700* @12 × 3.2 GHz from late 2017
- **Main Memory:** 2 × 8GB = 16GB of DDR4-SDRAM @2666 MHz
- **OS:** *Ubuntu 19.10*

1.3.4 Microbenchmark Results

Figure 1.1 shows the throughput of free list operations, measured with the microbenchmark. Each iteration of the microbenchmark from subsection 1.3.1—that performs either a pull or a push operation on the free list—is one operation on the evaluated free list queue. The operation throughput is total number of operations (of all working threads) performed on the free list per time.

The *CDS segmented queue* is the slowest free list queue implementation for any number of concurrent threads. The very similar *CDS basket lock-free queue* performs just as bad on 1 working thread and not much better on a higher number of threads.

The *CDS* queue implementations *CDS Ladan-Mozes & Shavit optimistic queue* , *CDS Michael & Scott lock-free queue* and *CDS variation of Michael & Scott lock-free queue* are also similar to each other and therefore they all perform very similar. Their performance for a low number of working threads is better than the one of the *CDS Basket Lock-Free Queue*, but for higher numbers of threads their performance is even worse.

The very simple *CDS flat-combining lock-free queue* performs not too bad for ≤2 threads, but the global lock limits the concurrency and therefore

[27] `https://bit.ly/2sy6QoN`
[28] `https://bit.ly/2ZBEMgk`
[29] `https://bit.ly/2F7cH7d`

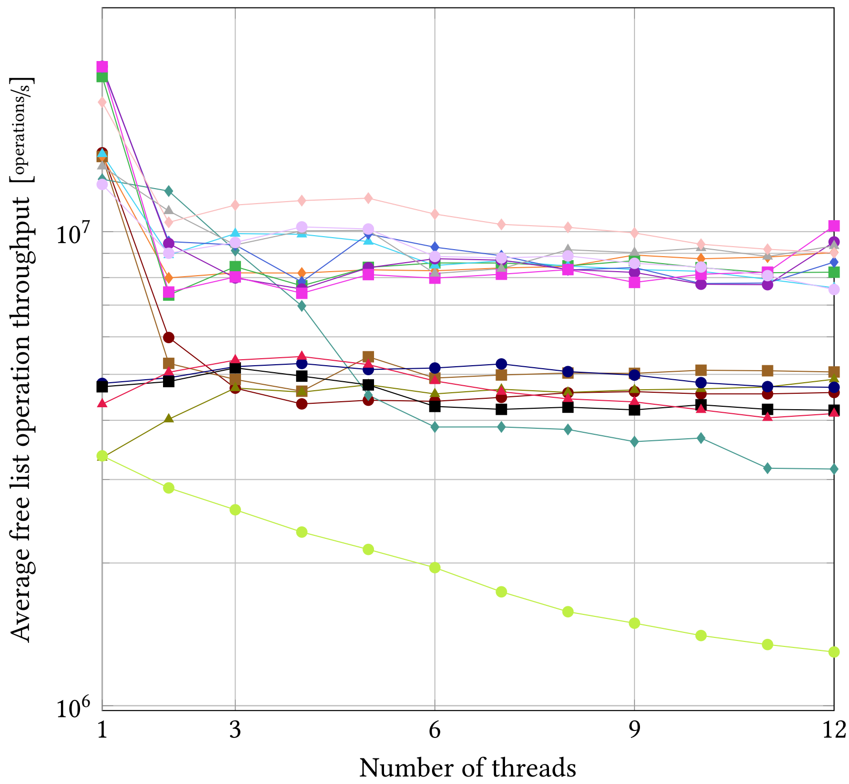

Figure 1.1: The operation throughput of the evaluated free list queue implementations

performance drops by a factor of >3 when ≥6 threads concurrently working on the free list queue.

The throughput of the two queue implementations from the *Boost C++ Libraries*—the *Boost lock-free queue with variable size* —●— and the *Boost lock-free queue with fixed size* —■—drops even earlier than the one of the *CDS flat-combining lock-free queue*. For 1 working thread, they perform good, but for >1 threads their performance is poor. The overhead due to the dynamic memory management of the unbounded version is negligible.

The three implementations of Vyukov's MPMC queue design—*CDS Vyukov's MPMC bounded queue* —■—, *Matt Stump's bounded MPMC queue* and *Gavin Lambert's MPMC bounded lock-free queue* —perform better than those already mentioned, but the original *Dmitry Vyukov's bounded MPMC queue* performs even better. All implementations of Vyukov's MPMC queue design are the best free list queues when there is only 1 working thread.

The two queue implementations which use a ticket dispenser system for synchronization—the *Folly MPMC queue* and *Erik Rigtorp's bounded MPMC queue* —perform as good as Vyukov's MPMC queue design.

The *CDS Michael & Scott blocking queue with fine-grained locking* and *TBB bounded concurrent dual queue* perform very similar as well.

The best free list queue for >2 threads concurrently working on the free list is the *TBB concurrent queue* .

1.4 Conclusion

The performance of the buffer pool free list is typically not critical to the overall performance of a DBMS—even a bad free list queue is unlikely to become a bottleneck for a DBMS. The buffer pool free list is usually called when a DB page is retrieved from secondary storage, which is a very expensive operation, even on an enterprise SSD.

Depending on further developments in NVRAM technology, memory devices of this class may not be able to fully replace DRAM in DB applications in the near future, but they could—and already do— replace existing secondary storage technologies such as SSDs. According to the specifications from [APD15], the limiting factor—at least in online transaction processing (**OLTP**) applications—will be the endurance of the memory cells—which is basically the number of writes per memory cell until it is worn out.

Following these general assumptions, high contention on the buffer pool free list is very unlikely and therefore the use of *Dmitry Vyukov's bounded MPMC queue* as buffer pool free list implementation can be recommended. But due to the superiority of the *TBB concurrent queue* during contention, the low drawback during non-concurrent operation, and the maturity of the library, this queue implementation can also be recommended.

2 RANDOM Page Eviction

2.1 Purpose

The buffer manager of a DBMS must evict pages from buffer frames if currently not buffered pages need to be fetched from the DB while there are no more free buffer frames. For this purpose, each buffer manager got a page eviction module—implementing one of the many page eviction algorithms developed since the 1960s.

According to Belady's classification in [Bel66], the RANDOM eviction algorithm is the most representative algorithm in his *Class 1* of page eviction algorithms. These *Class 1* page eviction algorithms do not use information about the usage of a buffered page but, but simply apply a static rule for the eviction decision. According to the more recent classification by Effelsberg and Härder in [EH84], the RANDOM eviction algorithm is the only algorithm in the class of algorithms that does neither use the age of a buffered page nor the references of it for the eviction decision.

The RANDOM strategy is the simplest possible page eviction strategy, resulting in a low overhead and poor hit rates.

2.2 Compared Pseudorandom Number Generators

The only operation performed by a RANDOM page eviction module to decide which page to evict from the buffer pool is the generation of a pseudorandom number in the range of buffer frame indexes. The DB page contained in the selected buffer frame is then evicted.

There are many different classes of pseudorandom number generators (**PRNG**). Some of them provide pseudorandom numbers of high randomness—suitable for cryptographic applications—others require only few CPU cycles and almost no memory to generate a random number.

Due to the enormous number of PRNGs described in literature, an exhaustive comparison of PRNGs for use in RANDOM page eviction is not possible in this context. Therefore only a selection of PRNGs—mostly from the *C++ Standard Library*[1] and the *Boost Random Number Library*[2] (part of the *Boost C++ Libraries*[3])—was selected for this evaluation.

2.2.1 Linear Congruential Generator (LCG) – 1958

The *linear congruential generator*—a generalization of the earlier proposed *Lehmer generator*—is a family of PRNGs proposed independently by W. E. Thomson in [Tho58] and by A. Rotenberg in [Rot60].

A LCG is defined by the following recurrence relation X:

$$X_{n+1} = (a \cdot X_n + c) \bmod m \quad n \geq 0$$

In this definition, $a \in (0..m)$ is the multiplier, $c \in [0..m)$ is the increment, $m \in (0..\infty)$ is the modulus and $X_0 \in [0..m)$ is the seed.

The following members of the LCG family of PRNGs, which do not belong to specializations defined in subsections, were compared:

- **rand**: $a = 0x41C64E6D$, $c = 0x3039$, $m = 2^{31}$ if using *GNU C Library*[4]
- **rand48**: $a = 0x5DEECE66D$, $c = 0xB$, $m = 2^{48}$
- **Kreutzer1986**: Buffers 97 random numbers of a LCG with $a = 0x556$, $c = 0x24D69$, $m = 0xAE529$ and returns them shuffled according to an algorithm proposed by Carter Bays and S. D. Durham in [BD76]. This was proposed by Wolfgang Kreutzer in [Kre86].

2.2.1.1 Lehmer Generator – 1949

The *Lehmer generator* (also known as *multiplicative congruential generator*) is the earliest family of PRNGs of “usable” quality, proposed by Derrick H. Lehmer in [Leh51] in 1949.

[1] `https://en.cppreference.com/w/cpp`
[2] `https://www.boost.org/doc/libs/release/doc/html/boost_random.html`
[3] `https://www.boost.org/`
[4] `https://www.gnu.org/software/libc/`

It is a specialization of the later proposed LCG with $c = 0$.
The following members of the MCG family of PRNGs were compared:

- **MCG128**: $a = 0x1168C7BF168D765C661FD0407A968ADD$, $m = 2^{64} - 1$, 128 bit state
- **MCG128Fast**: *MCG128* with $a = 0xDA942042E4DD58B5$
- **RANECU**: Combination of two Lehmer generators ($a_1 = 0x9C4E$, $m_1 = 0x7FFFFFAB$, $a_2 = 0x9EF4$, $m_2 = 0x7FFFFF07$) where the output is $o_1 - o_2$ if $o_2 < o_1$ or $o_1 - o_2 + 0x7FFFFFAA$ (unsigned 32 bit output) else for o_1, o_2 random numbers generated by the two Lehmer generators. This was proposed by Pierre L'Ecuyer in [LEc88] and modified by F. James in [Jam90].

2.2.1.2 Park-Miller Generator – 1988

The *Park-Miller generator* (now known as MINSTD) is a set of parameters for the *Lehmer generator* proposed by Stephen K. Mark and Keith W. Miller in [PM88]. After the criticism from George Marsaglia and Stephen Sullivan they proposed a modified set of parameters in [PMS93].

In their initial proposal the parameters were $a = 16807$ and $m = 2^{31} - 1$. In their later proposal they used $a = 48271$ instead.

The following Park-Miller generators were compared:

- **MINSTD0**: $a = 0x41A7$, $m = 2^{31} - 1$
- **MINSTD**: $a = 0xBC8F$, $m = 2^{31} - 1$
- **KnuthB**: Buffers 256 random numbers of *MINSTD0* and returns them shuffled according to an algorithm proposed by Carter Bays and S. D. Durham in [BD76]. This was proposed by Donald E. Knuth in [Knu81].

2.2.1.3 MIXMAX Generator – 1991

The *MIXMAX generator* is a *matrix linear congruential generator* proposed by G. K. Savvidy and N. G. Ter-Arutyunyan-Savvidy in [ST91].

Unlike an LCG, a matrix LCG uses a $N{\times}N$-matrix of multipliers A instead of a multiplier a:

$$a'_i = \begin{cases} \left(\sum_{j=1}^{N} A_{ij} \cdot a_j\right) \bmod m + s \cdot a_2 & \text{if } i = 3 \\ \left(\sum_{j=1}^{N} A_{ij} \cdot a_j\right) \bmod m & \text{else} \end{cases}$$

In this definition, $s \in \mathbb{Z}$ is a small "magic" integer, $m \in (0..\infty)$ is the modulus and the initial N-dimensional vector a is the seed.

The following members of the MIXMAX family of PRNGs was compared:

- **MixMax2.0**: $N = 17$, $s = 0$, $m = 2^{36} + 1$

2.2.1.4 Permuted Congruential Generator (PCG) – 2014

The *permuted congruential generator* is a modified *linear congruential generator* proposed by Melissa E. O'Neill in [ONe14].

In contrast to a typical LCG, the PCG state has twice the width of its output, the modulus m is $m = 2^k$ for $k \in \mathbb{N}$ and the output is generated by a state-defined bitwise rotation of the state.

The following members of the PCG family of PRNGs were compared:

- **PCG32**: $a = 0x5851F42D4C957F2D$, $c = 0x14057B7EF767814F$, $m = 2^{31} - 1$, 64 bit state
- **PCG32Unique**: *PCG32* where c is based on a memory address
- **PCG32Fast**: *PCG32* where $c = 0$
- **PCG32K2**: 2-dimensionally equidistributed version of *PCG32*
- **PCG32K2Fast**: 2-dim. equidistributed version of *PCG32Fast*
- **PCG32K64**: 64-dim. equidistributed version of *PCG32*
- **PCG32K64Fast**: 64-dim. equidistributed version of *PCG32Fast*
- **PCG32K1024**: 1024-dim. equidistributed version of *PCG32*
- **PCG32K1024Fast**: 1024-dim. equidistributed version of *PCG32Fast*
- **PCG32K16384**: 16384-dim. equidistributed version of *PCG32*
- **PCG32K16384Fast**: 16384-dim. equidistributed version of *PCG32Fast*

2.2.2 Lagged Fibonacci Generator (LFG) – 1958

The *lagged Fibonacci generator* is a family of PRNGs—based on the generalization of the Fibonacci sequence—proposed (but never published) by G. J. Mitchell and D. P. Moore in 1958.

A LFG is defined by the following recurrence relation X:

$$X_n = \left(X_{n-j} + X_{n-k}\right) \mod m, n \geq j \wedge n \geq k$$

In this definition $j = 24$ and $k = 55$ are the lags of the original proposal and $\left(X_0, ..., X_{\max(j,k)}\right)$ is the seed to be seeded e.g. based on another random number generator.

The following (floating-point) members of the LFG family of PRNGs, which do not belong to the specializations defined in subsections, were compared:

- **LaggedFibonacci607**: $j = 607$, $k = 273$, $m = 1$
- **LaggedFibonacci1279**: $j = 1279$, $k = 418$, $m = 1$
- **LaggedFibonacci2281**: $j = 2281$, $k = 1252$, $m = 1$
- **LaggedFibonacci3217**: $j = 3217$, $k = 576$, $m = 1$
- **LaggedFibonacci4423**: $j = 4423$, $k = 2098$, $m = 1$
- **LaggedFibonacci9689**: $j = 9689$, $k = 5502$, $m = 1$
- **LaggedFibonacci19937**: $j = 19937$, $k = 9842$, $m = 1$
- **LaggedFibonacci23209**: $j = 23209$, $k = 13470$, $m = 1$
- **LaggedFibonacci44497**: $j = 44497$, $k = 21034$, $m = 1$
- **RANMAR**: $X_n = \begin{cases} X_{n-97} - X_{n-33} & \text{if } X_{n-97} \geq X_{n-33} \\ X_{n-97} - X_{n-33} + 1 & \text{else} \end{cases} \mod 1$
  combined with a simple arithmetic sequence as proposed by G. Marsaglia et al. in [MZT90] and modified by F. James in [Jam90]

Many parameters (lags) used were suggested by R. P. Brent in [Bre92].

2.2.2.1 Subtract-With-Borrow (SWB) – 1991

The *subtract-with-borrow* generator is a modification of the *lagged Fibonacci generator* proposed by George Marsaglia and Arif Zaman in [MZ91].

A SWB generator is defined by the following iterating function f:

$$f\left(x_1, ..., x_j, c\right) = \begin{cases} \left(x_{j+1-k} - x_1 - c, 0\right) & \text{if } x_{j+1-k} - x_1 - c \geq 0 \\ \left(x_{j+1-k} - x_1 - c + b, 1\right) & \text{if } x_{j+1-k} - x_1 - c < 0 \end{cases}$$

In this definition $X_n = f\left(X_n\right)$ is the generated sequence. The lags j, k and the base b must be chosen appropriately with $j > k$ and the initial seed vector $\left(x_1, ..., x_j, c\right)$ must be set e.g. based on another random number generator.

The following members of the SWB family of PRNGs were compared:

- **Ranlux24Base**: $j = 24$, $k = 10$, $b = 2^{24} - 1$
- **Ranlux24**: *Ranlux24Base* discarding 200 per 223 generated numbers
- **Ranlux3**: *Ranlux24Base* discarding 199 per 223 generated numbers
- **Ranlux4**: *Ranlux24Base* discarding 365 per 389 generated numbers
- **Ranlux48Base**: $j = 12$, $k = 5$, $b = 2^{48} - 1$
- **Ranlux48**: *Ranlux48Base* discarding 378 per 389 generated numbers
- **Ranlux64_3**: $j = 10$, $k = 24$, $b = 2^{48} - 1$ discarding 199 per 223 generated numbers
- **Ranlux64_4**: $j = 10$, $k = 24$, $b = 2^{48} - 1$ discarding 365 per 389 generated numbers
- **Ranlux3_01**: floating-point version of *Ranlux3*
- **Ranlux4_01**: floating-point version of *Ranlux4*
- **Ranlux64_3_01**: floating-point version of *Ranlux64_3*
- **Ranlux64_4_01**: floating-point version of *Ranlux64_4*

The *Ranlux* family was proposed by M. Lüscher in [Lüs94].

2.2.3 Linear Feedback Shift Register (LFSR) – 1965

The *linear feedback shift register* PRNG is a family of PRNGs proposed by Robert C. Tausworthe in [Tau65].

LSFR PRNGs operate on a bit sequence $a = \{a_k\}$, defined as follows:

$$a_k = c_1 \cdot a_{k-1} + c_2 \cdot a_{k-2} + \ldots + c_n \cdot a_{k-n} \mod 2$$

The parameters $c_i \in \{0, 1\}$ with $1 \leq i \leq n$ are fixed and n is the bit width of the state.

Based on this state, the random number y_k is generated as follows:

$$y_k = \sum_{t=1}^{L} 2^{-t} \cdot a_{qk+r-t}$$

Here $L \leq n$ represents the bit width of the random number output, q is the number of bit between two successive y_k in a_k ($q \geq L$) and r is a random integer in the state interval $[0..2^n - 1]$.

In [LEc96], Pierre L'Ecuyer proposed a specific PRNG as the combination of three LSFR generators using bitwise XOR operations.

The following LFSR PRNGs were compared:

- **Taus88**: $n_1 = 32,\quad c_1 = 2^{32} - 2,\quad L_1 = 32,\quad q_1 = 12,\quad r_1 = 18$
  $n_2 = 32,\quad c_2 = 2^{32} - 2,\quad L_2 = 32,\quad q_2 = 4,\quad r_2 = 27$
  $n_3 = 32,\quad c_3 = 2^{32} - 2,\quad L_3 = 32,\quad q_3 = 17,\quad r_3 = 25$
- **Hurd160**: LFSR with 32 5 bit shift registers by W. J. Hurd in [Hur74]
- **Hurd288**: LFSR with 32 9 bit shift registers by W. J. Hurd in [Hur74]

2.2.3.1 Mersenne Twister (MT) – 1998

The *Mersenne Twister*—a twisted *generalized feedback shift register* (**GFSR**) operating on a state matrix—was proposed by M. Matsumoto and T. Nishimura in [MN98]. It is by far the most widely used general-purpose PRNG.

A more detailed description of the design and internals of the MT would unfortunately go beyond the scope of this thesis.

The following MTs—all in the original proposal of MT—were compared:

- **MT19937**: Mersenne prime is $2^{19937} - 1$

- **MT19937-64**: Mersenne prime is $2^{19937} - 1$, 64 bit version
- **MT11213B**: Mersenne prime is $2^{11213} - 1$

2.2.3.2 Xorshift – 2003

The *xorshift*—a subtype of the *linear feedback shift register*, which was implemented purely using fast bitwise XOR and shift operations—was proposed by George Marsaglia in [Mar03].

The following implementation of a 32 bit xorshift was given in [Mar03]:

```
uint32_t xorshift32() {
    static uint32_t state = 2463534242;
    state ^= (state << 13);
    state = (state >> 17);
    return (state ^= (state << 5));
}
```

The initial `state`—hard-coded in this example to 2463534242—should be randomly seeded in any real use case.

The following xorshift generators were compared:

- **xorshift32**: 32 bit xorshift
- **xorshift64***: 64 bit xorshift with truncated output
- **xorwow**: 128 bit xorshift combined with a Weyl sequence
- **xorshift128+**: 128 bit xorshift with 64 bit shifts ([Vig17])

2.2.3.3 Well Equidistributed Long-Period Linear (WELL) – 2006

The *well equidistributed long-period linear* generators—a family of PRNGs in the form of GFSR and MT generators—was proposed by François Panneton et al. in [PLM06].

The WELL algorithm is as follows:

$z_0 \longleftarrow \left(m_p \wedge v_{i,r-1}\right) \oplus \left(\widetilde{m}_p \wedge v_{i,r-2}\right)$
$z_1 \longleftarrow T_0 \cdot v_{i,0} \oplus T_1 \cdot v_{i,m_1}$
$z_2 \longleftarrow T_2 \cdot v_{i,m_2} \oplus T_3 \cdot v_{i,m_3}$
$z_3 \longleftarrow z_1 \oplus z_2$

$z_4 \longleftarrow T_4 \cdot z_0 \oplus T_5 \cdot z_1 \oplus T_6 \cdot z_2 \oplus T_7 \cdot z_3$
$v_{i+1,r-1} \longleftarrow v_{i,r-2} \wedge m_p$
for $j \longleftarrow r-2, 2$ **do**
  $v_{i+1,j} \longleftarrow v_{i,j-1}$
end for
$v_{i+1,1} \longleftarrow z_3$
$v_{i+1,0} \longleftarrow z_4$
return $y_i = v_{i,0}$

In the algorithm w is the bit-width of the random numbers output by the WELL algorithm, $r \in (0..\infty)$ and $p \in [0..w)$ are unique integers and $m_p \in (0..r)$ are bitmasks. The bit-width of the elements of the r-dimensional state vector x_i is w and the last p bits of the last element of this vector are 0. Possible values for the transformation $w \times w$-matrices $T_0, ..., T_7$ and further limitations to the parameters are given in [PLM06].

Shin Harase suggested a tempering method in [Har09] to make some WELL generators maximally equidistributed.

The following WELL generators were compared:

- **WELL512**: $w = 32$, $r = 16$, $p = 0$, $m_1 = 13$, $m_2 = 9$, $m_3 = 5$
- **WELL521**: $w = 32$, $r = 17$, $p = 23$, $m_1 = 13$, $m_2 = 11$, $m_3 = 10$
- **WELL607**: $w = 32$, $r = 19$, $p = 1$, $m_1 = 16$, $m_2 = 15$, $m_3 = 14$
- **WELL800**: $w = 32$, $r = 25$, $p = 0$, $m_1 = 14$, $m_2 = 18$, $m_3 = 17$
- **WELL1024**: $w = 32$, $r = 32$, $p = 0$, $m_1 = 3$, $m_2 = 24$, $m_3 = 10$
- **WELL19937**: $w = 32$, $r = 624$, $p = 31$, $m_1 = 70$, $m_2 = 179$, $m_3 = 449$
- **WELL21701**: $w = 32$, $r = 679$, $p = 27$, $m_1 = 151$, $m_2 = 327$, $m_3 = 84$
- **WELL23209**: $w = 32$, $r = 726$, $p = 23$, $m_1 = 667$, $m_2 = 43$, $m_3 = 462$
- **WELL44497**: $w = 32$, $r = 1391$, $p = 15$, $m_1 = 23$, $m_2 = 481$, $m_3 = 229$
- **WELL800-ME**: *WELL800* with $y_i \longleftarrow v_{i,0} \oplus (v_{i,19} \wedge 0x4880)$
- **WELL19937-ME**: *WELL19937* with $y_i \longleftarrow v_{i,0} \oplus (v_{i,180} \wedge 0x4118000)$
- **WELL21701-ME**: *WELL21701* with $y_i \longleftarrow v_{i,0} \oplus (v_{i,328} \wedge 0x1002)$
- **WELL23209-ME**: *WELL23209* with $y_i \longleftarrow v_{i,0} \oplus (v_{i,44} \wedge 0x5100000)$
- **WELL44497-ME**: *WELL44497* with $y_i \longleftarrow v_{i,0} \oplus (v_{i,482} \wedge 0x48000000)$

2.2.3.4 Xoshiro – 2018

The *xoshiro*—a *linear feedback shift register* generator implemented using XOR, shift and rotate operations—was—along with xoroshiro—proposed by David Blackman and Sebastiano Vigna in [BV18].

The following (slightly modified) implementation of a 32 bit xoshiro with a 128 bit state was given in [BV18]:

```
void xoshiro128() {
    static uint32_t s0_ = 0x01d353e5f3993bb1;
    static uint32_t s1_ = 0xf7381bed96327640;
    static uint32_t s2_ = 0xfdfcaa91110765b5;
    static uint32_t s3_ = 0x0;
    const uint64_t t = s1_ << a;
    s2_ ^= s0_;
    s3_ ^= s1_;
    s1_ ^= s2_;
    s0_ ^= s3_;
    s2_ ^= t;
    s3_ = (s3_ << b) | (s3_ >> (32 - b));
}
```

The initial `s0_`, `s1_`, `s2_` and `s3_`, which are the state—hard-coded in this example to 0x01d353e5f3993bb1, 0xf7381bed96327640, 0xfdfcaa91110765b5 and 0x0—should be randomly seeded in any real use case. For the 32 bit case, the authors suggested shift and rotate values to be $a = 9$ and $b = 11$.

It is easy to see in the implementation that xoshiro does not define the generation of a pseudorandom number from its state. The authors proposed four scramblers to be used with xoshiro (and xoroshiro) where the two more advanced ones try to eliminate linear artifacts from the state.

+ scrambler The simple + scrambler returns just the sum of two of the state words (e.g. `return s0_ + s3_`).

*** scrambler** The not less simple * scrambler returns just the product of one of the state words with a fixed, odd multiplier (e.g. `return s1_ + mult`).

++ scrambler The ++ scrambler first adds two of the state words, rotates the sum by `r` positions to the left and returns the sum of this rotated sum and the first of the two state words used in the first sum (e.g. `return ((s0_+s3_) << r) | ((s0_+s3_) >> (32-r)) + s0_`). The authors propose `r = 7` in the 32 bit case.

**** scrambler** The ** scrambler first multiplies one of the state words by a fixed, odd multiplier `s`, rotates the product by `r` positions to the left and returns the product of this rotated product and another fixed, odd multiplier `t` (e.g. `return ((s1_*s) << r) | ((s1_*s) >> (32-r)) * t`). The authors propose `s = 5`, `r = 7` and `t = 9` in the 32 bit case.

The following xoshiro generators were compared:

- **xoshiro128+32**: 32 bit *xoshiro* with 128 bit state and + scrambler
- **xoshiro128**32**: 32 bit *xoshiro* with 128 bit state and ** scrambler

2.2.3.5 Xoroshiro – 2018

The *xoroshiro*—another *linear feedback shift register* generator implemented using XOR, shift and rotate operations—was proposed by David Blackman and Sebastiano Vigna in [BV18].

The following (slightly modified) implementation of a 64 bit xoroshiro with a 128 bit state was given in [BV18]:

```
void xoroshiro128() {
    static uint64_t s0_ = 0xc1f651c67c62c6e0;
    static uint64_t s1_ = 0x30d89576f866ac9f;
    const uint64_t t = s0_ ^ s1_;
    s0_ = ((s0_ << a) | (s0_ >> (64 - a)))
        ^ t ^ (t << b);
    s1_ = (t << c) | (t >> (64 - c));
}
```

The initial `s0_` and `s1_` which are the state—hard-coded in this example to 0xc1f651c67c62c6e0 and 0x30d89576f866ac9f—should be randomly seeded in any real use case. For the 64 bit case, the authors proposed shift and rotate values to be a = 24, b = 16 and c = 37.

To generate pseudo-random numbers from the states of a xoroshiro generator, the scramblers +, *, ++ and **, which are also used for xoshiro, are used. The details of these scramblers are described in the subsection 2.2.3.4.

The following xoroshiro generators were compared:

- **xoroshiro128+32**: 64 bit *xoroshiro* with 128 bit state and + scrambler
- **xoroshiro64+32**: 32 bit *xoroshiro* with 64 bit state and + scrambler
- **xoroshiro64*32**: 32 bit *xoroshiro* with 64 bit state and * scrambler
- **xoroshiro64**32**: 32 bit *xoroshiro* with 64 bit state and ** scrambler

2.2.4 Inversive Congruential Generator (ICG) – 1986

The *inversive congruential generator* is a family of PRNGs proposed by Jürgen Eichenauer and Jürgen Lehn in [EL86].

An ICG is defined by the following recurrence relation X:

$$X_{n+1} = \begin{cases} \left(a \cdot X_n^{-1} + b\right) \bmod p & \text{if } X_n \geq 1 \\ b & \text{else} \end{cases}$$

In this definition $a \in \mathbb{N}$ is the multiplier, $b \in \mathbb{N}$ is the increment, p is the prime modulus and $X_0 \in [0..p)$ is the seed. X_n^{-1} is the multiplicative inverse of X_n in the finite field $GF(p)$.

The following member of the ICG family of PRNGs was compared:

- **Hellekalek1995**: $a = 0x238E$, $b = 0x7DCD313A$, $p = 0x7FFFFFFF$ as proposed by Peter Hellekalek in [Hel95]

2.2.5 Ranshi – 1995

The *ranshi* algorithm is a PRNG proposed by F. Gutbrod in [Gut95].

The idea behind the algorithm is a physical system consisting of a number of black balls, each of which has a position and a spin (state of the PRNG). A red ball—also having a spin and a position—colliding with the black balls is used to generate pseudorandom numbers.

2.2.6 Gjrand – 2005

The *gjrand* algorithm is based on a random, invertible mapping[5] of addition, XOR and rotate operations. It was proposed by David Blackman[6].

The following member of the gjrand family of PRNGs was compared:

- **gjrand32**: 32 bit PRNG with 128 bit state, parameters from the author

2.2.7 A Small Noncryptographic PRNG (JSF) – 2007

The *JSF* algorithm is based on a reversible, non-linear function in which all internal state bits affect one another using addition, XOR, rotate and conditional branch operations. It was proposed by Bob Jenkins[7].

The following implementation (with `b_ = c_ = d_` properly seeded) of a 32 bit JSF was used:

```
uint32_t jsf32() {
    static uint32_t a_ = 0xf1ea5eed;
    static uint32_t b_ = 0xcafe5eed00000001;
    static uint32_t c_ = 0xcafe5eed00000001;
    static uint32_t d_ = 0xcafe5eed00000001;
    uint32_t e = a_ - ((b_ << p)
                     | (b_ >> (32 - p)));
    a_ = b_ ^ ((c_ << q) | (c_ >> (32 - q)));
    b_ = c_ + (r ? ((d_ << r)
                 | (d_ >> (32 - r))) : d_);
    c_ = d_ + e;
    d_ = e + a_;
    return d_;
}
```

The following members of the JSF family of PRNGs were compared:

- **JSF32n**: `p = 27`, `q = 17`, `r = 0`
- **JSF32r**: `p = 23`, `q = 16`, `r = 11`

[5] http://www.pcg-random.org/posts/random-invertible-mapping-statistics.html
[6] http://gjrand.sourceforge.net/
[7] http://burtleburtle.net/bob/rand/smallprng.html

2.2.8 SFC – 2010

The *SFC* algorithm is based on a random, invertible mapping[8] of addition, XOR, shift and rotate operations. It was proposed by Chris Doty-Humphrey as part of his PractRand[9] statistical test and PRNG library.

The following implementation (with `a_`, `b_` and `c_` properly seeded) of a 32 bit SFC was used:

```
uint32_t sfc32() {
    static uint32_t a_ = 0xcafef00dbeef5eed;
    static uint32_t b_ = 0xcafef00dbeef5eed;
    static uint32_t c_ = 0xcafef00dbeef5eed;
    static uint32_t d_ = 0x1;
    uint32_t t = a_ + b_ + d_++;
    a_ = b_ ^ (b_ >> q);
    b_ = c_ + (c_ << r);
    c_ = (c_ << p) | (c_ >> (64 - p)) + t;
    return t;
}
```

The following member of the SFC family of PRNGs was compared:

- **SFC32**: `p = 21`, `q = 9`, `r = 3`

2.2.9 Counter-Based Random Number Generator (CBRNG) – 2011

The *counter-based random number generator* is a family of PRNGs proposed by J. Salmon et al. in [Sal+11].

The state of a CBRNG is a simple integer counter but the output mapping is done using a complex function—usually a cryptographic block cipher.

2.2.9.1 ARC4 – 1997

The *ARC4* is a PRNG first implemented in OpenBSD 2.1[10] in 1997 for the `arc4random` function.

[8] http://www.pcg-random.org/posts/random-invertible-mapping-statistics.html
[9] http://pracrand.sourceforge.net/
[10] https://man.openbsd.org/arc4random

It generates pseudorandom numbers from the keystream of the RC4 stream cipher, released by Ronald L. Rivest in 1987. ARC4 is not exactly a CBRNG since it uses a second state which is not a counter, but the PRNG is closely related to the other CBRNGs since it uses just a stream cipher to generate pseudorandom numbers.

2.2.9.2 ChaCha – 2008

ChaCha is a stream cipher proposed by Daniel J. Bernstein in [Ber08]. It is used as PRNG by encoding the state of the PRNG—a simple integer counter—using the ChaCha stream cipher.

The following PRNGs based on the family of ChaCha stream ciphers were compared:

- **ChaCha4**: Based on ChaCha 4-round cipher
- **ChaCha5**: Based on ChaCha 5-round cipher
- **ChaCha6**: Based on ChaCha 6-round cipher
- **ChaCha8**: Based on ChaCha 8-round cipher
- **ChaCha20**: Based on ChaCha 20-round cipher

2.2.9.3 Advanced Randomization System (ARS) – 2011

The *advanced randomization system* is a *counter-based random number generator* where the state—a simple integer counter—is mapped to the random output using a simplified AES block cipher.

The following ARS generator was compared:

- **ARS4x32**: 7 rounds, operating on four 32 bit integers

2.2.9.4 Threefry – 2011

The *Threefry* is a *counter-based random number generator* where the state is mapped to the random output using a simplified Threefish block cipher.

The following Threefry generators have been compared:

- **Threefry2x32**: 20 rounds, operating on two 32 bit integers

- **Threefry4x32**: 20 rounds, operating on four 32 bit integers
- **Threefry2x64**: 20 rounds, operating on two 64 bit integers
- **Threefry4x64**: 20 rounds, operating on four 64 bit integers

2.2.9.5 Philox – 2011

The *Philox* is a *counter-based random number generator* where the state is mapped to the random output using a custom block cipher.

The following Philox generators were compared:

- **Philox2x32**: 10 rounds, operating on two 32 bit integers
- **Philox4x32**: 10 rounds, operating on four 32 bit integers
- **Philox2x64**: 10 rounds, operating on two 64 bit integers
- **Philox4x64**: 10 rounds, operating on four 64 bit integers

2.2.9.6 Advanced Encryption Standard (AES) – 2011

The *Advanced Encryption Standard* PRNG is a *counter-based random number generator* where the state is mapped to the random output using the AES block cipher.

The following AES generator was compared:

- **AES4x32**: 10 rounds, operating on four 32 bit integers

2.2.10 SplitMix – 2014

SplitMix is a PRNG similar to the *counter-based random number generators* that was proposed by Guy Steele et al. in [SLF14]. It is derived from the PRNG *DotMix* which was proposed by Charles Leiserson et al. in [LSS12].

While the state of CBRNGs is advanced by adding 1—it is a simple counter—the state of *SplitMix* is advanced by adding a fixed γ. Instead of using a complex hash function for the generation of a pseudorandom integer from the state, *SplitMix* uses the finalization mix of the MurmurHash3[11] hash function. This is sufficient as long as γ is not a simple value like 1, even or some other problematic value.

[11] `https://bit.ly/2tI7IqW`

The following implementation (with `state` and `gamma` properly seeded) of 32 bit *SplitMix* was used:

```
uint32_t splitmix32() {
    static uint64_t state = 0xbad0ff1ced15ea5e;
    static uint64_t gamma = 0x9e3779b97f4a7c15
                          | 1;
    uint64_t seed = state;
    state += gamma;
    seed ^= seed >> v;
    seed *= m5;
    seed ^= seed >> w;
    seed *= m6;
    return result_type(seed >> 32);
}
```

The `| 1` after the seed of `gamma` takes care of even `gamma`s, which would degrade the quality of the generated pseudorandom numbers.

The *SplitMix* PRNG used for the evaluation—**SplitMix32**—uses the following parameters: `m5 = 0x62a9d9ed799705f5`, `m6 = 0xcb24d0a5c88c35b3`, `v = 33` and `w = 28`.

2.2.11 Combinations of different PRNGs

The following combined PRNGs were compared:

- **DualRand**: LCG with $a = 0x10405$, $c = 0x3035$, $m = 2^{32} - 1$ XORed with a LFSR approximated by $X_n = X_{n-1 \bmod 64} \oplus X_{n-33 \bmod 64} \bmod 2$ on a 128 bit state
- **TripleRand**: *DualRand* XORed with *Hurd288*

2.2.12 Biased Uniform Integer Distribution

The PRNGs listed above return pseudorandom integers in the range of 0 to $2^{32} - 1$ or any other arbitrary range. But for the use in a RANDOM page eviction algorithm, the numbers need to be in the range of the buffer frame indices.

This requires an algorithm that transforms pseudorandom numbers uniformly distributed in a given range to pseudorandom numbers in the desired range, keeping the uniform distribution. A blog post[12] by Melissa E. O'Neill revealed that this transformation is the bottleneck of fast random number generation when using the `std::uniform_int_distribution` from the *C++ Standard Library*.

Therefore, the algorithm that turned out to be the fastest in her comparison was used when evaluating PRNGs for RANDOM page eviction. In contrast to the algorithm built into the *C++ Standard Library*, this algorithm returns pseudorandom numbers in a biased uniform distribution when the range of the PRNG used is not a multiple of the range of the buffer frame indices. For example, if the PRNG returns numbers uniformly distributed in the integer interval [1..6] and if the buffer frame indices are in the interval [1..4], this algorithm returns 1 and 2 with a probability of $1/3$ and 3 and 4 with a probability of $1/6$. But as long as the range of buffer frame indices is much smaller than the range of the used PRNG, the bias is less severe.

For a PRNG that returns random numbers in the range of 0 to $2^{32} - 1$, the algorithm is as follows:

```
uint32_t biased_int_dist(uint32_t ranNum,
                         uint32_t rangeMin,
                         uint32_t rangeMax) {
    uint64_t r = uint64_t(rangeMax - rangeMin);
    uint64_t m = uint64_t(ranNum) * (r + 1);
    return uint32_t(rangeMin + (m >> 32);
}
```

The actual implementation used—available on GitHub[13]—works with PRNGs returning both integers in any range as well as floating point numbers. It uses metaprogramming to utilize compile-time calculation wherever possible.

[12] `http://www.pcg-random.org/posts/bounded-rands.html`
[13] `https://bit.ly/37RQQNx`

2.3 Performance Evaluation

Benchmarks of an exemplary DBMS using all the compared RANDOM page eviction implementations revealed that there is no statistically significant difference in the hit rates achieved with these different RANDOM page eviction implementations. The only performance difference between the different RANDOM page eviction implementations is therefore the overhead that results from the generation of pseudorandom numbers. For this reason, a microbenchmark only measuring the execution time of the PRNGs is appropriate.

The only variable of the evaluation is the PRNG used—the alternatives presented in the previous section are evaluated.

Another potential variable is the number of threats generating pseudorandom numbers. But the behavior of the PRNGs when used concurrently is usually not specified and therefore all the PRNGs would require synchronization when used by multiple evicting threads. However, since the quality of the generated pseudo-random numbers is not important here, it is assumed that each evicting thread uses its own thread-local instance of the used PRNG to choose candidates for eviction. And these thread-local instances scale perfectly as long as there are hardware threads available and therefore an evaluation on one thread is sufficient.

Different algorithms to generate the pseudorandom integers uniformly distributed over a given range could also be compared. But a quick comparison of the custom algorithm presented in subsection 2.2.12 with the ones provided by the *C++ Standard Library*[14] and by the *Boost Random Number Library*[15] showed that the one[16] used is never slower than the competition.

The smallest (> 0) and largest integers returned by the PRNGs—representing the smallest and largest buffer pool indexes—do not significantly affect the performance of RANDOM page eviction. Therefore, this integer interval [1..53467] is a constant in this evaluation.

[14] `https://bit.ly/39Xuiwn`
[15] `https://bit.ly/37JKHTs`
[16] The classic modulo algorithm was used for **rand**, **xorwow** and **xorshift128+**.

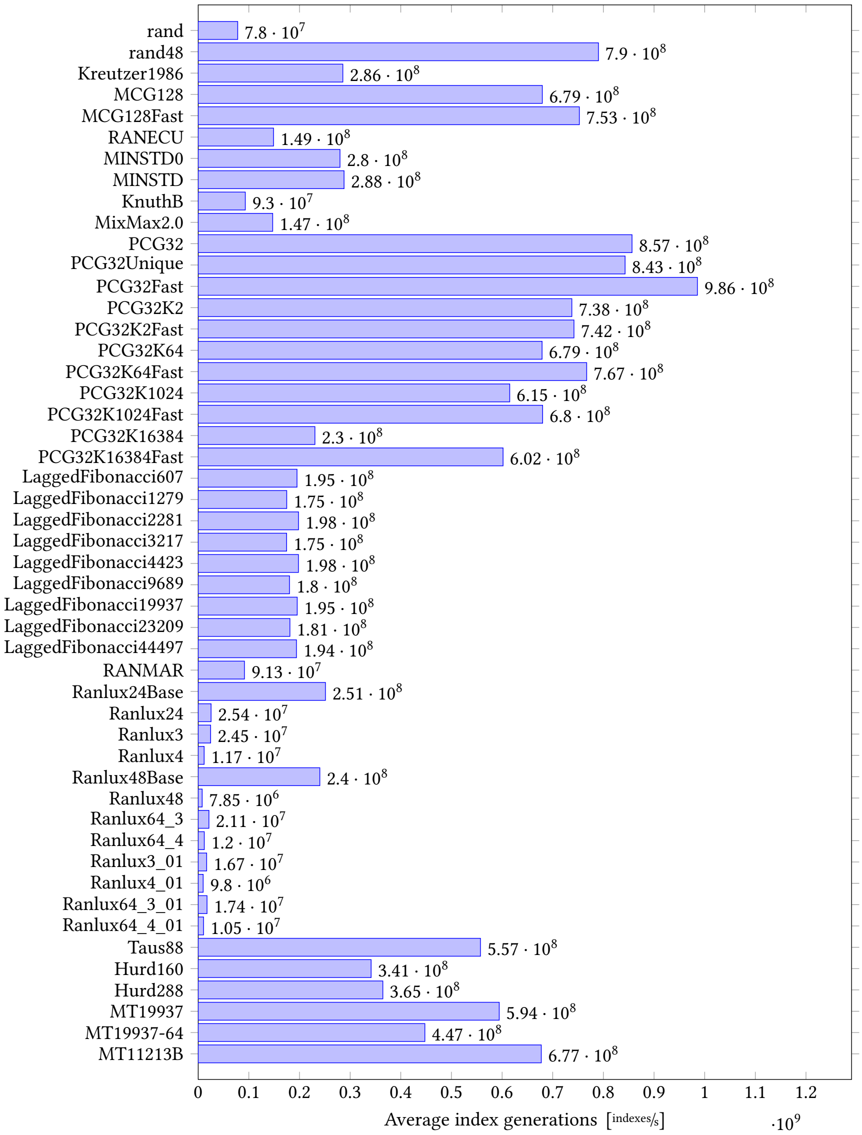

Figure 2.1: The index generation throughput of the evaluated RANDOM implementations (1 of 2)

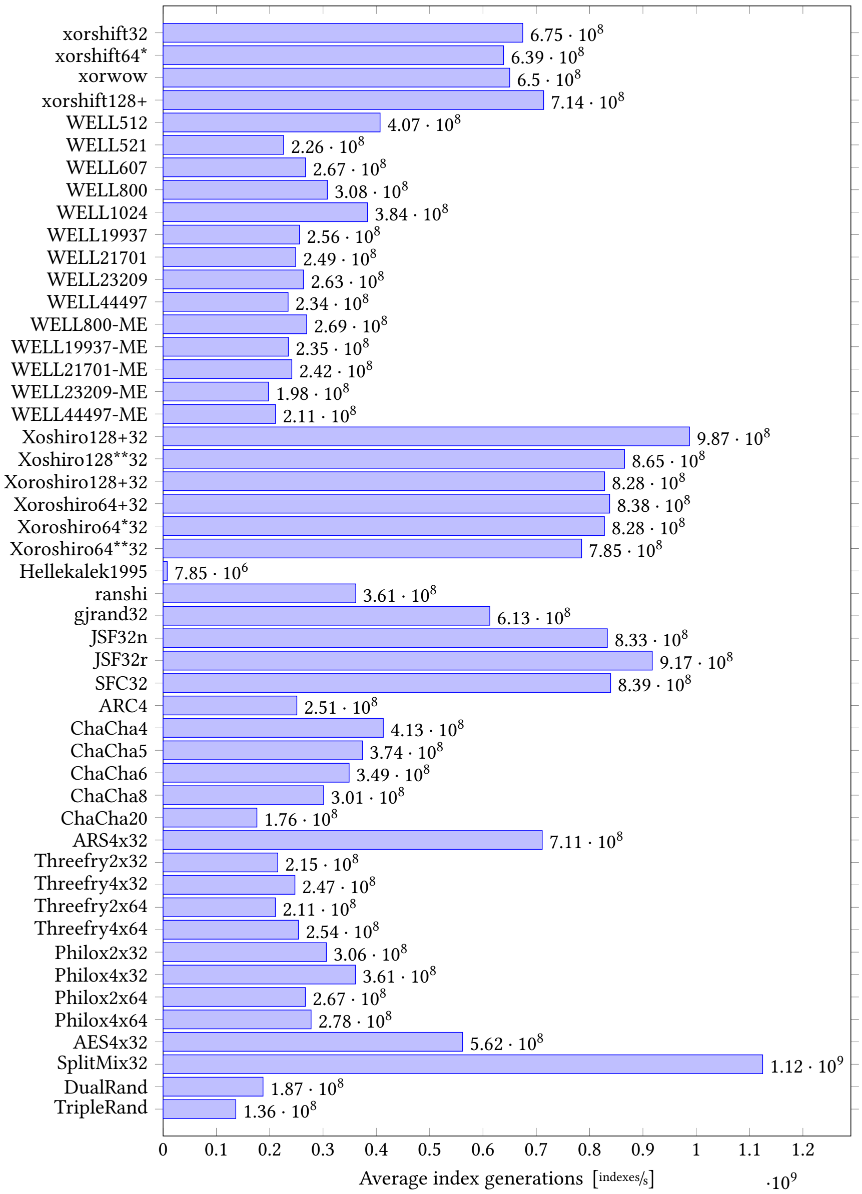

Figure 2.2: The index generation throughput of the evaluated RANDOM implementations (2 of 2)

2.3.1 Microbenchmark

The microbenchmark used for the performance evaluation of the RANDOM page eviction algorithms instantiates the evaluated PRNG with a seed—generated with `std::random_device`[17]—, calculates a given number (5 000 000) of pseudorandom integers in the interval [1..53467] using the PRNG and measures the wall time elapsed.

2.3.2 Configuration of the Used System

- **CPU:** *Intel® Core™ i7-8700* @12 × 3.2 GHz from late 2017
- **Main Memory:** 2 × 8GB = 16GB of DDR4-SDRAM @2666 MHz
- **OS:** *Ubuntu 19.10*
- **Compiler:** *GCC 9.2.1* with `-O3` flag

2.3.3 Benchmark Results

The figures 2.1 and 2.2 show the index generation throughput of the evaluated RANDOM page eviction implementations.

The *ICG* **Hellekalek1995** and the *SWB*s of the **Ranlux** family (not the non-discarding "base" ones) are the slowest PRNGs in the evaluation. The XOR-based *LSFR* PRNGs and the *LCG*s of the **PCG** and **MCG** families are among the fastest PRNGs. The very recent **SplitMix32** PRNG, described in subsection 2.2.10, is by far the fastest algorithm in the competition with an average of 1 124 474 870 indexes/s on the used system.

2.4 Conclusion

Like any other DBMS component evaluated for this thesis, the performance (the overhead of the eviction candidate selection, not the achieved hit rate) of the RANDOM eviction algorithm is not critical in most cases. However, the hit rate achieved with the chosen eviction strategy is a significant performance factor of a DB system (**DBS**).

[17] `https://bit.ly/306x2TE`

When it comes to the selection of a page eviction strategy, the RANDOM eviction strategy is the worst but simplest option. Its simplicity makes the RANDOM page eviction strategy a good choice if the main memory is expected to be (almost) always large enough to hold the entire working set of a DBS. In this case, the RANDOM eviction strategy would not perform (much) worse than any other "good" page eviction strategy.

If the RANDOM page eviction strategy is chosen for the use in a buffer pool manager of a DBMS, there is basically no reason not to use the fastest PRNG available in the particular programming language. The fastest PRNG in this comparison—*SplitMix*—is part of the Java Development Kit and good implementations are also available in Haskell and C++. The somewhat slower XOR-based *LSFR* PRNGs are available for many programming languages, and the mid-range PRNG **MT19937** is the default PRNG of most of the programming languages. Most of the fast general-purpose PRNGs can be easily implemented in any programming language typically used in the development of a DBMS, and therefore the lack of such a PRNG in a particular programming language can quickly be compensated.

3 LOOP Page Eviction

3.1 Purpose

The LOOP page eviction algorithm is the simplest form of the RANDOM page eviction strategy where the eviction candidates are selected round-robin based on their buffer frame index.

The LOOP eviction strategy uses a "pseudorandom" number generator—basically a CBRNG as in subsection 2.2.9, but instead of using a complex mapping function, the identity function modulo the number of buffer pool frames is used to generate the pseudorandom numbers—which generates an ordered sequence of buffer indices.

3.2 Compared Counter Implementations

Six different counter implementations were evaluated for the use in a LOOP page eviction algorithm. They can be grouped as follows:

- blocking counters
  - mutex counter
  - spinlock counter
- non-blocking counters
  - modulo counter
  - lock-free counter
- local counters
  - local counter
  - local modulo counter

The counters return values from 1 to `blockCount` − 1—the range of the buffer frame indices.

3.2.1 Mutex Counter

The *mutex counter* is a blocking counter which uses one global counter—used to select candidates for page eviction—synchronized with a mutex called via the `std::mutex` interface.

```
inline uint32_t mutexCounter() {
    static std::mutex indexLock;
    static uint32_t lastIndex = 0;
    std::lock_guard<std::mutex> guard(indexLock);
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    return lastIndex;
}
```

3.2.2 Spinlock Counter

The *spinlock counter* is also a blocking counter that uses one global counter but it is protected by a spinlock implemented using a `std::atomic_flag`.

```
inline uint32_t spinlockCounter() {
    static std::atomic_flag indexLock =
        ATOMIC_FLAG_INIT;
    static uint32_t lastIndex = 0;
    uint32_t newIndex;
    while (indexLock.test_and_set(
                std::memory_order_acquire)) {}
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    newIndex = lastIndex;
    indexLock.clear(std::memory_order_release);
    return newIndex;
}
```

3.2.3 Modulo Counter

The *modulo counter* is a non-blocking counter that uses atomic increment operations on a global counter (of type `std::atomic<uint64_t>`). The modulo is calculated thread-local and therefore the value of the global counter is strictly increasing.

```
inline uint32_t moduloCounter() {
    static std::atomic<uint64_t> lastIndex = 0;
    static uint32_t moduloDivisor = blockCount
                                  - 1;
    return (lastIndex++ % moduloDivisor) + 1;
}
```

3.2.4 Local Counter

The *local counter* is—obviously—a local counter, where each thread that performs page evictions uses its own circular counter.

```
inline uint32_t localCounter() {
    static thread_local uint32_t lastIndex = 0;
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    return lastIndex;
}
```

3.2.5 Local Modulo Counter

The *local modulo counter* is a local counter where the circular counting is not achieved with a branch operation, but with a possibly cheaper modulo operation calculated during each call.

```
inline uint32_t localModuloCounter() {
    static thread_local uint64_t lastIndex = 0;
    static uint32_t moduloDivisor = blockCount
                                    - 1;
    return (lastIndex++ % moduloDivisor) + 1;
}
```

3.2.6 Lock-Free Counter

The *lock-free counter* is a non-blocking counter which uses atomic increment operations on a global counter (of type `std::atomic<uint32_t>`) and algorithm-specific synchronization to achieve circular counting using branch operations.

```
inline uint32_t lockFreeCounter() {
    static std::atomic<uint32_t> newIndex(1);
    uint32_t pickedIndex = newIndex;
    if (pickedIndex < blockCount) {
        newIndex++;
        return pickedIndex;
    } else {
        return newIndex = 1;
    }
}
```

3.3 Performance Evaluation

Since the eviction decisions of LOOP page evictioners are largely implementation-independent, the achieved hit rates in an exemplary DBMS are not required as performance benchmark for this evaluation. The only difference in performance between the different LOOP page eviction algorithms is the overhead resulting from the concurrent counting. Therefore, a microbenchmark measuring only the execution time of the counting is appropriate.

The variables of the evaluation are the concurrent counter implementation—the alternatives presented in the previous section are evaluated—

and the number of threads concurrently incrementing the counter. The smallest (> 0) and largest integers returned by the counters—representing the smallest and largest buffer pool indexes—do not significantly affect the performance of the LOOP page eviction, but when the largest buffer pool index is a power of 2, the modulo operations of the *modulo counter* and *local modulo counter* are significantly faster. Therefore this integer interval [1..53467]—where 53467 is not a power of 2 because the evaluation should be as general as possible—is a constant in this evaluation.

3.3.1 Microbenchmark

The microbenchmark used for the performance evaluation of the LOOP page eviction algorithms forks a certain number of worker threads—each of them calls the evaluated concurrent counter a certain number of times (1 000 000)—and measures the wall time that has elapsed until all the worker threads finished their operation and joined.

3.3.2 Configuration of the Used System

- **CPU:** *Intel® Core™ i7-8700* @12 × 3.2 GHz from late 2017
- **Main Memory:** 2 × 8GB = 16GB of DDR4-SDRAM @2666 MHz
- **OS:** *Ubuntu 19.10*
- **Compiler:** *GCC 9.2.1* with `-O3` flag

3.3.3 Microbenchmark Results

Figure 3.1 shows the LOOP index generation throughput, which is total number of indexes generated (on all working threads) per time.

The *blocking counters—mutex counter* and *spinlock counter*—are the slowest counters when there are multiple working threads. The locking overhead and—for a higher number of working threads—the lock contention makes these concurrent counters slower than the *non-blocking counters* by up to more than one order of magnitude. The more optimized mutex, which uses low-overhead busy-waiting (spinlock behavior) in situations of low contention and higher-overhead queuing and context switches in situations of higher contention, can better utilize the available physical

CPU cores by suspending waiting threads—thus eliminating unnecessary uses of hyper-threading.

The *non-blocking counters—modulo counter* and *lock-free counter* —perform almost identical to each other. In situations without contention, the *modulo counter* is identical to the *local modulo counter*, but the *lock-free counter* suffers from the overhead due to the conditional branch operation. But most of the times, a modern CPU can correctly predict the targets of these branches. But when there are multiple working threads counting, the atomic operations on the counter variable are the bottleneck slowing down the *non-blocking counters* compared to *local counters*.

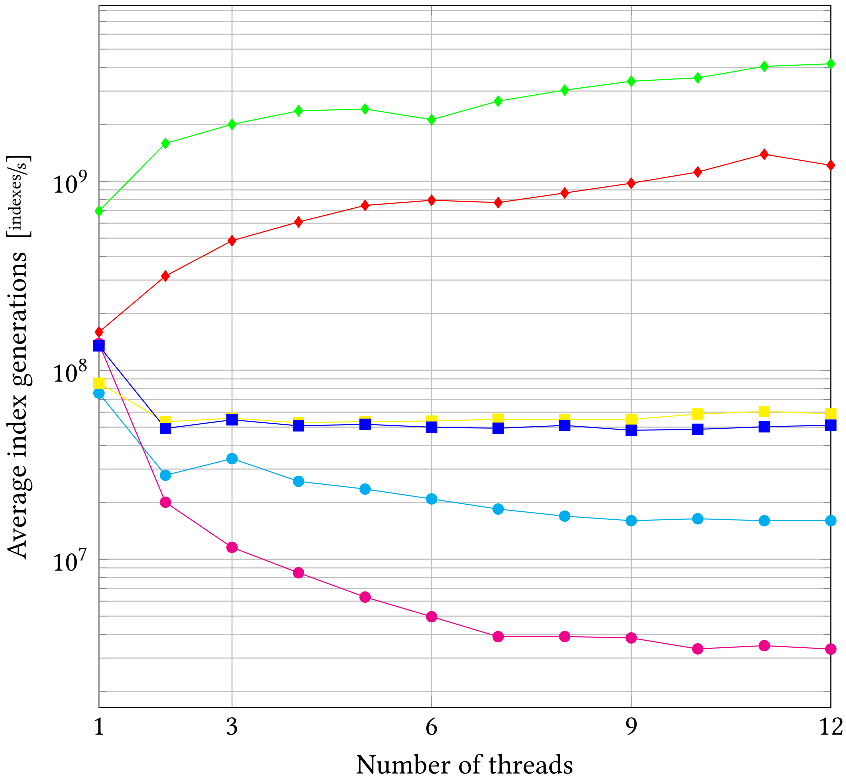

Figure 3.1: The throughput of index generations of the evaluated LOOP implementations

Obviously, the *local counters*—*local counter* and *local modulo counter* —are the fastest counters because they completely omit any synchronization between the counting threads. While the other counters count based on a global counter, these ones count based on local counters, resulting in a thread-wise round-robin selection of eviction candidates instead of a global round-robin selection of eviction candidates. The independence of the threads makes these counters very scalable as long as there are hardware threads available. This results in a performance advantage of up to almost two orders of magnitude compared over the *non-blocking counters*. The significantly higher performance of the *local counter* compared to the *local modulo counter* is the result of the slow modulo operation when the divisor is not a power of two. The branch target of the conditional branch in the *local counter* is predicted correctly most of the times, making its overhead negligible.

3.4 Conclusion

Each LOOP page eviction algorithm is a good replacement for any other RANDOM page eviction algorithm. Tests in an OLTP database showed that the hit rate of the LOOP strategy is not worse than that of any other RANDOM page eviction strategy, and the overheads of the LOOP page eviction strategies—especially of the *local counters*—are lower than the ones of any other RANDOM page eviction strategy. This makes the LOOP page evictioners superior to the other RANDOM page evictioners in OLTP applications (represented by the TPC-C benchmark). Since the use of local counters instead of a global counter does not change the hit rate in the DBS, the *local counter*—the LOOP page eviction algorithm with the lowest overhead—can be recommended.

Bibliography

[AKY10] Yehuda Afek, Guy Korland, and Eitan Yanovsky. “Quasi-Linearizability: Relaxed Consistency for Improved Concurrency”. In: *Principles of Distributed Systems.* Lecture Notes in Computer Science (6490 2010): *14th International Conference, OPODIS 2010 Tozeur, Tunisia, December 14-17, 2010 Proceedings.* Ed. by Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 395–410. DOI: `10.1007/978-3-642-17653-1_29`. URL: `https://bit.ly/2shYTDH` (visited on Jan. 24, 2019).

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. “Let’s Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’15. Melbourne, Victoria, Australia: ACM, May 2015, pp. 707–722. ISBN: 978-1-4503-2758-9. DOI: `10.1145/2723372.2749441`. URL: `https://bit.ly/2F9omlS` (visited on Dec. 31, 2019).

[BD76] Carter Bays and S. D. Durham. “Improving a Poor Random Number Generator”. In: *ACM Transactions on Mathematical Software* 2.1 (Mar. 1976), pp. 59–64. ISSN: 0098-3500. DOI: `10.1145/355666.355670`.

[Bel66] Laszlo A. Belady. “A Study of Replacement Algorithms for Virtual-Storage Computer”. In: *IBM Systems Journal* 5 (2 June 1966), pp. 78–101. ISSN: 0018-8670. DOI: `10.1147/sj.52.0078`.

[Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20.* Jan. 28, 2008. URL: `https://bit.ly/2raJ7KB` (visited on Dec. 20, 2019).

[BV18] David Blackman and Sebastiano Vigna. "Scrambled Linear Pseudorandom Number Generators". In: *Computing Research Repository* (2018). eprint: 1805.01407.

[Bre92] Richard P. Brent. "Uniform Random Number Generators for Supercomputers". In: (1992): *Proceedings Fifth Australian Supercomputer Conference, 1949.* URL: https://bit.ly/2Ph1y9K (visited on Dec. 12, 2019).

[Doh+04] Simon Doherty et al. "Formal Verification of a Practical Lock-Free Queue Algorithm". In: *Formal Techniques for Networked and Distributed Systems – FORTE 2004.* Lecture Notes in Computer Science (3235 2004): *24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004 Proceedings.* Ed. by David de Frutos-Escrig and Manuel Núñez. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 97–114. DOI: 10.1007/978-3-540-30232-2_7.

[EH84] Wolfgang Effelsberg and Theo Härder. "Principles of Database Buffer Management". In: *ACM Transactions on Database Systems* 9 (4 Dec. 1984). Ed. by Robert W. Taylor, pp. 560–595. ISSN: 0362-5915. DOI: 10.1145/1994.2022.

[EL86] Jürgen Eichenauer and Jürgen Lehn. "A Non-Linear Congruential Pseudo Random Number Generator". In: *Statistische Hefte* 27.1 (Dec. 1986), pp. 315–326. ISSN: 1613-9798. DOI: 10.1007/BF02932576.

[Gut95] F. Gutbrod. "A Fast Random Number Generator for the Intel Paragon Supercomputer". In: *Computer Physics Communications* 87.3 (June 1995). Ed. by P. G. Burke et al., pp. 291–306. ISSN: 0010-4655. DOI: 10.1016/0010-4655(95)00005-Z.

[Har09] Shin Harase. "Maximally Equidistributed Pseudorandom Number Generators via Linear Output Transformations". In: *Mathematics and Computers in Simulation* 79.5 (Jan. 2009). Ed. by R. Beauwens, pp. 1512–1519. ISSN: 0378-4754. DOI: 10.1016/j.matcom.2008.06.006.

[Hel95] Peter Hellekalek. "Inversive Pseudorandom Number Generators: Concepts, Results and Links". In: (Dec. 1995): *Proceedings of the 1995 Winter Simulation Conference.* Ed. by C. Alexopoulos et al., pp. 255–262. DOI: 10.1109/WSC.1995.478732.

[Hen+10] Danny Hendler et al. "Flat Combining and the Synchronization-Parallelism Tradeoff". In: (2010): *SPAA '10: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures.* Ed. by Friedhelm Meyer auf der Heide and Cynthia Phillips, pp. 355–364. DOI: 10.1145/1810479.1810540. URL: https://bit.ly/2YFIMfm (visited on Jan. 24, 2019).

[HSS07] Moshe Hoffman, Ori Shalev, and Nir Shavit. "The Baskets Queue". In: *Principles of Distributed Systems.* Lecture Notes in Computer Science (4878 2007): *11th International Conference, OPODIS 2007 Guadeloupe, French West Indies, December 17-20, 2007 Proceedings.* Ed. by Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 401–414. DOI: 10.1007/978-3-540-77096-1_29. URL: https://bit.ly/2EoTfmd (visited on Jan. 24, 2019).

[Hur74] William. J. Hurd. "Efficient Generation of Statistically Good Pseudonoise by Linearly Interconnected Shift Registers". In: *IEEE Transactions on Computers* C-23.2 (Feb. 1974), pp. 146–152. ISSN: 2326-3814. DOI: 10.1109/T-C.1974.223877.

[Jam90] F. James. "A Review of Pseudorandom Number Generators". In: *Computer Physics Communications* 60.3 (Oct. 1990). Ed. by P. G. Burke et al., pp. 329–344. ISSN: 0010-4655. DOI: 10.1016/0010-4655(90)90032-V.

[Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Boston, MA, USA: Addison-Wesley, 1981. ISBN: 0-201-03822-6.

[Kre86] Wolfgang Kreutzer. *System Simulation Programming Styles and Languages.* Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-12914-0.

[LEc88] P. L'Ecuyer. "Efficient and Portable Combined Random Number Generators". In: *Communications of the ACM* 31.6 (June 1988). Ed. by Peter J. Denning, pp. 742–751. ISSN: 0001-0782. DOI: 10.1145/62959.62969. URL: https://bit.ly/34GQTd6 (visited on Dec. 19, 2019).

[LEc96] Pierre L'Ecuyer. "Maximally Equidistributed Combined Tausworthe Generators". In: *Mathematics of Computation* 65.213 (Jan. 1996), pp. 203–213. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-96-00696-5. URL: https://bit.ly/2PHy9EC (visited on Dec. 13, 2019).

[LS04] Edya Ladan-Mozes and Nir Shavit. "An Optimistic Approach to Lock-Free FIFO Queues". In: *Distributed Computing*. Lecture Notes in Computer Science (3274 2004): *18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004 Proceedings.* Ed. by Rachid Guerraoui. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 117–131. DOI: 10.1007/978-3-540-30186-8_9. URL: https://bit.ly/34dyfJy (visited on Jan. 24, 2019).

[Leh51] Derrick H. Lehmer. "Mathematical Methods in Large-scale Computing Units". In: The Annals of the Computation Laboratory of the Harvard University XXVI (1951): *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, 1949.* Ed. by Howard H. Aiken, pp. 141–146. URL: https://bit.ly/2Pf8c0i (visited on Dec. 11, 2019).

[LSS12] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. "Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms". In: *SIGPLAN Notices* 47.8 (Aug. 2012). Ed. by Andy Gill, pp. 193–204. ISSN: 0362-1340. DOI: 10.1145/2370036.2145841. URL: https://bit.ly/2uxe90D (visited on Jan. 9, 2020).

[Lüs94] Martin Lüscher. "A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations". In: *Computer Physics Communications* 79 (1 Feb. 1994), pp. 100–110. ISSN: 0010-4655. DOI: 10.1016/0010-4655(94)90232-1.

[Mar03] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software, Articles* 8.14 (July 4, 2003), pp. 1–6. ISSN: 1548-7660. DOI: 10.18637/jss.v008.i14.

[MZ91] George Marsaglia and Arif Zaman. "A New Class of Random Number Generators". In: *The Annals of Applied Probability* 1.3 (Aug. 1991). Ed. by J. Michael Steele, pp. 462–480. ISSN: 2168-8737. DOI: 10.1214/aoap/1177005878.

[MZT90] George Marsaglia, Arif Zaman, and Wai Wan Tsang. "Toward a Universal Random Number Generator". In: *Statistics & Probability Letters* 9.1 (Jan. 1990). Ed. by R. A. Johnson, pp. 35–39. ISSN: 0167-7152. DOI: 10.1016/0167-7152(90)90092-L.

[MN98] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8 (1 Jan. 1998): *Special Issue on Uniform Random Number Generation.* Ed. by Philip Heidelberger, Raymond Coutre, and Pierre L'Ecuyer, pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: https://bit.ly/2qQD0uU (visited on Jan. 24, 2019).

[MS96] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: (1996): *PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing.* Ed. by James E. Burns and Yoram Moses, pp. 267–275. DOI: 10.1145/248052.24810629. URL: https://bit.ly/34cP9rz (visited on Jan. 24, 2019).

[ONe14] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.* technical report. Harvey Mudd College, Sept. 5, 2014. URL: https://bit.ly/2E9jpcr (visited on Dec. 12, 2019).

[PLM06] François Panneton, Pierre L'Ecuyer, and Makoto Matsumoto. "Improved Long-period Generators Based on Linear Recurrences Modulo 2". In: *ACM Transactions on Mathematical Soft-*

ware 32.1 (Mar. 2006). Ed. by Ian Gladwell, pp. 1–16. ISSN: 0098-3500. DOI: 10.1145/1132973.1132974.

[PM88] Stephen K. Park and Keith W. Miller. "Random Number Generators: Good Ones Are Hard To Find". In: *Communications of the ACM* 31 (10 Oct. 1988), pp. 1192–1201. ISSN: 0001-0782. DOI: 10.1145/63039.63042. URL: https://bit.ly/2LKj6cf (visited on Jan. 30, 2019).

[PMS93] Stephen K. Park, Keith W. Miller, and Paul K. Stockmeyer. "Technical Correspondence: Response". In: *Communications of the ACM* 36 (7 July 1993): *Special Issue on Computer Augmented Environments: Back to the Real World*, pp. 108–110. ISSN: 0001-0782. DOI: 10.1145/159544.376068. URL: https://bit.ly/2PJ1PkT (visited on Jan. 30, 2019).

[Rot60] A. Rotenberg. "A New Pseudo-Random Number Generator". In: *Journal of the ACM* 7.1 (Jan. 1960). Ed. by Mario L. Juncosa, pp. 75–77. ISSN: 0004-5411. DOI: 10.1145/321008.321019.

[Sal+11] John K. Salmon et al. "Parallel Random Numbers: As Easy As 1, 2, 3". In: (2011): *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* DOI: 10.1145/2063384.2063405.

[ST91] G. K. Savvidy and N. G. Ter-Arutyunyan-Savvidy. "On the Monte Carlo Simulation of Physical Systems". In: *Journal of Computational Physics* 97.2 (Dec. 1991), pp. 566–572. ISSN: 0021-9991. DOI: 10.1016/0021-9991(91)90015-D.

[SLF14] Guy L. Steele, Doug Lea, and Christine H. Flood. "Fast Splittable Pseudorandom Number Generators". In: *SIGPLAN Notices* 49.10 (Oct. 2014). Ed. by Andy Gill, pp. 453–472. ISSN: 0362-1340. DOI: 10.1145/2714064.2660195. URL: https://bit.ly/2t1MB2V (visited on Jan. 9, 2020).

[Tau65] Robert C. Tausworthe. "Random Numbers Generated by Linear Recurrence Modulo Two". In: *Mathematics of Computation* 19.90 (Apr. 1965), pp. 201–209. ISSN: 1088-6842. DOI: 10.1090/S0025-5718-1965-0184406-1.

[Tho58] W. E. Thomson. "A Modified Congruence Method of Generating Pseudo-random Numbers". In: *The Computer Journal* 1.2 (Jan. 1958), pp. 83, 86. ISSN: 0010-4620. DOI: `10.1093/comjnl/1.2.83`.

[Vig17] Sebastiano Vigna. "Further Scramblings of Marsaglia's Xorshift Generators". In: *Journal of Computational and Applied Mathematics* 315 (May 1, 2017), pp. 175–181. ISSN: 0377-0427. DOI: `10.1016/j.cam.2016.11.006`.