УДК 004.272.2

O. N. Karasik[1], A. A. Prihozhy[2]
[1]ISsoft Solutions (part of Coherent Solutions)
[2]Belarusian National Technical University

THREAD-SAFE BITSET WITH FAST EXTRACT MIN OPERATION

Even in today's ever-changing world where every modern PC, game console, mobile device, or TV is equipped with GBs of RAM and CPUs have multiple cores, fast, thread-safe, space-efficient data structures remain an active field of research. Bitsets (binary array of individually accessible bits) have many applications in various industry domains like operating systems, database design, searching and allocating resources. The existing implementations of bitsets are mainly focused on compression and encoding of bits to reduce memory footprint, disk storage consumption and speed up bulk bitwise operations (primarily in cases of search and database design), or on low- and non-concurrent scenarios for tracking and allocating resources, or on the usage of bitset as a set of unique integers to insert, remove and test their presence. This paper proposes the implementation of fast, thread-safe bitset designed for high-concurrent scenarios like reservation and tracking of resources. High performance is achieved using an additional index array and a novel non-blocking synchronization mechanism. Experiments carried out on a server equipped with two Intel Xeon E5-2620 v4 processors have shown the speedup of 2 – 6 times compared to implementation which uses standard blocking synchronization mechanisms like mutexes and locks.

Keywords: bitset, concurrent data structure, find first set bit, extract min.

For citation: Karasik O. N., Prihozhy A. A. Thread-safe bitset with fast extract min operation. *Proceedings of BSTU, issue 3, Physics and Mathematics. Informatics*, 2025, no. 1 (290), pp. 62–69.
DOI: 10.52065/2520-6141-2025-290-10.

О. Н. Карасик[1], А. А. Прихожий[2]
[1]Иностранное производственное унитарное предприятие «Иссофт Солюшенз»
[2]Белорусский национальный технический университет

ПОТОКО-БЕЗОПАСНАЯ РЕАЛИЗАЦИЯ БИТОВОГО МНОЖЕСТВА С БЫСТРОЙ ОПЕРАЦИЕЙ ИЗВЛЕЧЕНИЯ МИНИМАЛЬНОГО ЧЛЕНА

Потоко-безопасные компактные структуры данных остаются активной областью исследований даже сегодня, когда каждый современный ПК, игровая консоль, мобильное устройство или телевизор оснащены гигабайтами оперативной памяти и многоядерными процессорами. Битовые множества (двоичный массив индивидуально-доступных битов) имеют применение в различных отраслях, таких как операционные системы, проектирование баз данных, поиск и распределение ресурсов. Существующие реализации битовых множеств сосредоточены в основном на сжатии и кодировании битов для сокращения потребляемой памяти и дискового пространства, а также на ускорении множественных битовых операций (в частности, в областях поиска и проектирования баз данных), или на сценариях с низко- и непараллельным отслеживанием и выделением ресурсов, или на использовании битовых множеств в качестве набора уникальных целых чисел с целью вставки, удаления и проверки их наличия. В данной статье предлагается потоко-безопасная реализация битового множества, предназначенная для использования в сценариях с участием огромного количества потоков, выполняющих внушительное количество параллельных операций, таких как резервирование и отслеживание ресурсов. Высокая производительность достигается за счет использования дополнительного массива индексов и нового механизма неблокирующей синхронизации. Эксперименты, проведенные на сервере, оснащенном двумя процессорами Intel Xeon E5-2620 v4, показали ускорение в 2–6 раз по сравнению с реализацией, использующей блокировку потоков.

Ключевые слова: битовое множество, потоко-безопастные структуры данных, поиск первого выставленного бита, извлечение минимального члена.

Для цитирования: Карасик О. Н., Прихожий А. А. Потоко-безопасная реализация битового множества с быстрой операцией извлечения минимального члена // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2025. № 1 (290). С. 62–69 (На англ.).
DOI: 10.52065/2520-6141-2025-290-10.

Introduction. Conceptually, bitset (aka bitmap) is a binary array, where every bit can be set or reset independently. Every bit in the array corresponds to an integer (bit index), thus forming a set of unique integers. Because of its space efficiency (a single 64-bit word can handle unique integers from 0 to 63) and ability to perform fast bulk bitwise operations, bitset has application in various areas like database indices [1–2], web search [3], event matching [4], graphs [5], sequence predictions [6], memory and registry allocators [7–9].

Common non-thread-safe implementations of bitset include operations to *set* (include in set), *reset* (remove from set) and *test* (check if in set) individual bits; bulk operations between two or more bitsets to find *intersection* (logical AND) and *union* (logical OR); operation to *find first* (find first set bit or smallest value in a set). In non-concurrent scenarios, operations can be combined to form new operations, like *reset first* (find first set bit and reset or remove smallest value from the set) operation which is implemented as a combination of *find first* and *reset* operations.

In thread-safe implementations the bitset interface is either a more concise and includes *set*, *reset* and *test* operations (without *find first* operation).

In both implementations, the *set*, *reset* and *test* operations have constant execution time, while *find first* operation in contrary, is usually implemented as a lineal search.

In this research we propose a fast, thread-safe bitset implementation that implements *set* and *reset* operations in constant time and *reset first* operation in non-lineal time.

Main part. ***Non-thread-safe bitset.*** In a scenario of uncompressed and unencoded bitset (assuming the size of the bitset is larger than a single word), a bitset can be implemented as an array B of words W, where individual bits are accessed by first locating an array element and then locating a bit from the element.

The number of words N required to hold the number of bits C can be calculated as:

$$N = \lceil C / W \rceil.$$

The array element index i_B and bit position p_B within the element of the bit X can be calculated as quotient and remainder:

$$i_B = X / W \text{ and } p_B = X \bmod W.$$

Knowing i_B and p_B we can implement *set* and *reset* operations using bitwise logic (logical *and*, *or*, *not* and *shifts*). However, to implement the *find first set bit* operations we need a function to find a position of least significant bit in a word, hereinafter we denote this function as *bitscan* (modern CPUs implemented this function as *bfs* opcode). Fig. 1 presents a possible pseudocode of these functions.

Thread-safe bitset. Thread-safe versions of *set* and *reset* operations can be implemented using interlocked or atomic logical operations, supported by most of the modern CPUs. The *test* and *find first* function do not require changes because they don't modify the *bitset* and consistency of memory reads is guaranteed by cache coherency protocols.

The *reset first* operation, however, no longer can be implemented as a combination of *find first* and *reset* operations because the bit index returned from *find first* operation might be reset by a concurrent actor before the execution of *reset* operation. Therefore, it leaves two possible options for implementation:

1. Separate *reset first* operation which will iterate over array B and try to reset bits.

2. Separate *find next* operation which will accept B element index to start search from.

We yield to the first option (Fig. 2).

```
set(
  B: array, X: int, W: int
) -> bool
  local i = X / W, p = X % W, V = B[i];
  B[i] = B[i] | (1 << p);
  return V & (1 << p) == 0;
end
```

a

```
reset(
  B: array, X: int, W: int
) -> bool
  local i = X / W, p = X % W, V = B[i];
  B[i] = B[i] & (~(1 << p));
  return V & (1 << p) != 0;
end
```

b

```
find_first(B: array, N: int, W: int) -> integer
  for (local i = 0; i < N; ++i)
    local p = bitscan(B[i])
    if p >= 0 then return i * W + p end
  end
  return -1;
end
```

c

Fig. 1. Pseudocode of implementations of set (*a*), reset (*b*) and find first (*c*) operations

```
atomic_set(
  B: array, X: int, W: int
) -> bool
  local i = X / W, p = X % W;
  local V = atomic_or(B[i], 1 << p);
  return V & (1 << p) == 0;
end
```

a

```
atomic_reset(
  B: array, X: int, W: int
) -> bool
  local i = X / W, p = X % W;
  local V = atomic_and(B[i], ~(1 << p));
  return V & (1 << p) != 0;
end
```

b

```
atomic_reset_first(B: array, N: int, W: int) -> int
  for (local i = 0; i < N; ++i)
    local p = bitscan(B[i]), X = i * W + p;
    if atomic_reset(B, X, W) then return B; end
  end
  return -1;
end
```

c

Fig. 2. Pseudocode of implementations of concurrent set (*a*), reset (*b*) and reset first (*c*) operations using interlocked or atomic operations

Non-thread-safe bitset with index. In non-concurrent scenarios the lookup speed of *find first* operation can be significantly improved by introducing an additional binary array *I* (index array) such that every bit of every element of the array represents a state of the corresponding element of array *B* (bit's index of *I* corresponds to element index of *B*) in a way that if the bit in *I* is set, then at least one bit is set in the corresponding element *B* and vice versa. The number of words *M* required to index the number of bits *C* can be calculated as:

$$M = \lceil C / W^2 \rceil.$$

Total number of words required to hold both *I* and *B* arrays is a sum of *M* and *N*.

The *I* array element index i_I and bit position p_I within the element of the bit *X* can be calculated as:

$$i_I = X / W^2 \text{ and } p_I = (X / W) \bmod W.$$

The array *I* need to be constantly synchronised with the state of the array *B*. This synchronisation is done by *set* and *reset* operations (Fig. 3).

Inclusion of an index array improves the lookup speed at the cost of:

1. Slow *set* and *reset* operations. Operations must verify and update *I* when necessary.
2. More memory to store arrays *I* and *B*.
3. No obvious way to implement concurrent versions of *set* and *reset* operations without a mutex. The atomic operations work on a single memory location while *set* and *reset* operations need to update both arrays.

```
index_set(
  I: array, B: array, X: int, W: int
) -> bool
  local iB = X / W, pB = X % W, V = B[i];
  if V == 0 then
    local iI = X / (W * W), pI = (X / W) % W;
    I[iI] = I[iI] | (1 << pI)
  end
  B[iB] = B[iB] | (1 << pB);
  return V & (1 << pB) == 0;
end
```

a

```
index_reset(
  I: array, B: array, X: int, W: int
) -> bool
  local iB = X / W, pB = X % W, V = B[i];
  if V & ~(1 << pB) == 0 then
    local iI = X / (W * W), pI = (X / W) % W;
    I[iI] = I[iI] & ~(1 << pI)
  end
  B[i] = B[i] & (~(1 << pB));
  return V & (1 << pB) != 0;
end
```

b

```
index_find_first(I: array, B: array, N: int, M: int, W: int) -> int
  for (local iI = 0; iI < M; ++iI)
    local pI = bitscan(I[iI])
    if pI >= 0 then
      local iB = iI * W + pI, pB = bitscan(B[iB]);
      return iB * W + pB;
    end
  end
  return -1;
end
```

c

Fig. 3. Pseudocode of implementations of indexed set (*a*), reset (*b*), and find first (*c*) operations

Bitfield. The implementation of thread-safe bitset with fast extract min operation (hereinafter “bitfield”) is based on the *non-thread-safe bitset with index*, demonstrated in previous section, with one notable exception – index array I uses two bits instead of one to represent a state of an element from a bit array B.

Index implementation. The use of two bits to represent the state has the following effects on the index array I and related calculations:

1. The number of words M required to keep the same number of bits is doubled.

2. The calculation of I array element i_I and bit position within the element p_I must be adjusted to smaller number of states per element.

Because index array element now represents states of $W \div 2$ bit array elements (capacity), the values of M, i_I and p_I are calculated as following:

$$M = \lceil C / (W \cdot (W/2)) \rceil;$$

$$i_I = B / (W \cdot (W/2));$$

$$p_I = ((B / W) \bmod (W/2)) \cdot 2.$$

The “$(W/2)$” part in all the equations corresponds to the capacity of index array element and the “×2” part in the p_I bit position calculation is required to align the position to 2 bit boundry to ensure states don’t intesect.

Every state in the element can assume one of four possible values: 00, 01, 10 and 11, where values 00, 10 corresponds to the “reset” state and values 01, 11 to the “set” state (Table 1).

Table 1

All possible values of state within the index array element and their meaning

State	Meaning
00	The bit array element has no set bits (“reset” state)
10	
01	The bit array element has at least one set bit (“set” state)
11	

As in the *non-thread-safe bitset with index* implementation the states are updated by *set* and *reset* operations.

Set and reset operations implementation. The pseudocode of *set* (bitfield_set) and *reset* (bitfield_reset) operations is presented in Fig. 4. Below is a breakdown of both operations:

1. Both operations start with the calculation of arrays indices and bits positions (lines 04–07).

2. Then they perform atomic modifications (atomic_or in case of *set* and atomic_and in case of *reset*) of B array element. If no bits were set or reset, they stop execution and return false (lines 09–13).

3. Next, if the first bit was set (in case of *set* operation) or last bit was reset (in case of *reset* operation) they modify the state of the I array’s element by executing atomic_xor operation with 01 (on *set*) and 11 (on *reset*) values (lines 15–17). All combinations of set and reset operations are presented in (Table 2).

```
01: bitfield_set(
02:  I: array, B: array, X: int, W: int
03: ) -> bool
04:  local iB = X / W;
05:  local pB = X % W;
06:  local iI = X / (W * (W / 2));
07:  local pI = ((X / W) % (W / 2)) * 2
08:
09:  local T = 1 << pB;
10:  local V = atomic_or(B[i], T);
11:  if V & (~T) != 0 then
12:    return false
13:  end
14:
15:  if V == 0 then
16:    atomic_xor(I[iI], 01 << pI)
17:  end
18:  return true;
19: end
```

a

```
01: bitfield_reset(
02:   I: array, A: array, X: int, W: int
03: ) -> bool
04:   local iB = X / W;
05:   local pB = X % W;
06:   local iI = X / (W * (W / 2));
07:   local pI = ((X / W) % (W / 2)) * 2
08:
09:   local T = 1 << pB, R = ~T;
10:   local V = atomic_and(B[iB], R);
11:   if V & T == 0 then
12:     return false
13:   end
14:
15:   if V & R == 0 then
16:     atomic_xor(I[iI], 11 << pI)
17:  end
18:  return true;
19: end
```

b

Fig. 4. Pseudocode of implementations of bitfield set (*a*) and reset (*b*) operations

Table 2

All combinations of state values after the execution of set or reset operations

State	XOR	Result
00	01 (set)	01 (set)
10	01 (set)	11 (set)
01	11 (reset)	10 (reset)
11	11 (reset)	00 (reset)

It is obvious from the pseudocode that modifications of arrays B and I are separated in time. The reasons both arrays stay in sync are:

1. The *B* array is atomically modified first.

2. The modified bit is indeed set or reset (if not, the return).

3. The modified bit is the first set bit (in case of *set* operation) or the last reset bit (in case of *reset* operation).

4. The *I* array is modified only if all the above points are true. This excludes possibility of two threads modifying the *I* array with two same operations without having the third around them (for instance, there is no way to have *set*, *set* or *reset*, *reset* modifications without having the third *reset* or *set* operation).

5. The I array is atomically modified by bitwise XOR operation, with special values 01 (in case of set) and 11 (in case of reset) such that the reset state always becomes the set and vice-versa (Table 2).

6. The XOR operation is commutative and associative. Therefore, because of #4, no matter how many modifications (and in what order) are done to the I array, eventually the result will depend on the order of modifications done to the B array.

Table 3 includes a set of examples of concurrent *set* operations interrupted by concurrent *reset* operations, and Fig. 5 illustrates cases 1, 2 and 3 from it.

Experimental environment. All experiments were run on a rack server equipped with two Intel Xeon E5-2620v4 CPU (8 cores, 16 hardware threads) and 32 GB of RAM. All cores have private L1 and L2 cache of 32 KB and 256 KB respectively. All CPUs have 20 MB L3 shared cache.

All experiments were implemented using C++ language. The source code was compiled by GNU GCC compiler v14.1.0 with O3 optimization level.

Experimental benchmark. Benchmark emulates a highly concurrent scenario of resource allocation / release. The code instantiates 31 "noise" threads and 1 "allocator" thread and assigns them to individual hardware threads. Every "noise thread" works in a tight loop and on every iteration sets or resets a random bit (emulating concurrent allocation and release of the resources). The "allocator" thread on every iteration executes *reset first* operation (replicating continuous resource allocation procedure). The benchmark software measures how long it takes to perform a single *reset first* operation.

Experimental results. In the experiments, we compare the "bitfield" with the "bitset with mutex" (hereinafter "Impl. #1") and "indexed bitset with multiple mutexes" (hereinafter "Impl. #2") implementations.

The "Impl.#1" implementation uses a "std::bitset" from C++ standard library to store the data and "std::mutex" to protect it from concurrent access. The lock is acquired every time "noise" threads sets or resets the bit and when "allocator" threads attempt to reset the bit. The search for a first set bit isn't protected by lock.

The "Impl.#2" implementation uses an "std::array" of 64 bit unsigned longs to store bits (*B* array), an "std::array" of 64-bit unsigned longs to store index (*I* array) and an "std::array" of "std::mutex" (one for each index array element) to protect access to index and bits. The "noise" thread acquires a lock to corresponding index right before executing set or reset operation to ensure consistency between bit array and index array. The "allocator" thread acquires corresponding lock right before resetting the bit. The search for first set bit isn't protected by the lock.

We performed a set of experiments on different number of bits for all three implementations. In each implementation, the reset first operation was repeated at least 1,000,000 times. The error % was less then 2.5% for all implementations and experiments except the last one where error% reached 7% for "bitfield" implementation. The results of the experiments are presented in the Table 4.

Table 3

Example of multiple, concurrent "set" and "reset" operations setting and resetting the bit in the same element of bit array *B*, which results in the modification of index array *I*. The S_B, and S_I denote modification of *B* and *I* array during "set" operation. The R_B and R_I represent the corresponding modifications done by "reset"

Number	Operation sequence	Index state		Result
		00 (reset)	10 (reset)	
1	$S_B \rightarrow S_I \rightarrow R_B \rightarrow R_I$	00 ^ 01 ^ 11 = 10	10 ^ 01 ^ 11 = 00	Reset
2	$S_B \rightarrow R_B \rightarrow R_I \rightarrow S_I$	00 ^ 11 ^ 01 = 10	10 ^ 11 ^ 01 = 00	Reset
3	$S_B \rightarrow R_B \rightarrow S_I \rightarrow R_I$	00 ^ 01 ^ 11 = 10	00 ^ 01 ^ 11 = 10	Reset
4	$S_B \rightarrow R_B \rightarrow S_B \rightarrow R_I \rightarrow S_I \rightarrow S_I$	00 ^ 11 ^ 01 ^ 01 = 11	10 ^ 11 ^ 01 ^ 01 = 01	Set
5	$S_B \rightarrow R_B \rightarrow S_B \rightarrow S_I \rightarrow S_I \rightarrow R_I$	00 ^ 01 ^ 01 ^ 11 = 11	10 ^ 01 ^ 01 ^ 11 = 01	Set
6	$S_B \rightarrow R_B \rightarrow S_B \rightarrow S_I \rightarrow R_I \rightarrow S_I$	00 ^ 01 ^ 11 ^ 01 = 11	10 ^ 01 ^ 11 = 00	Set

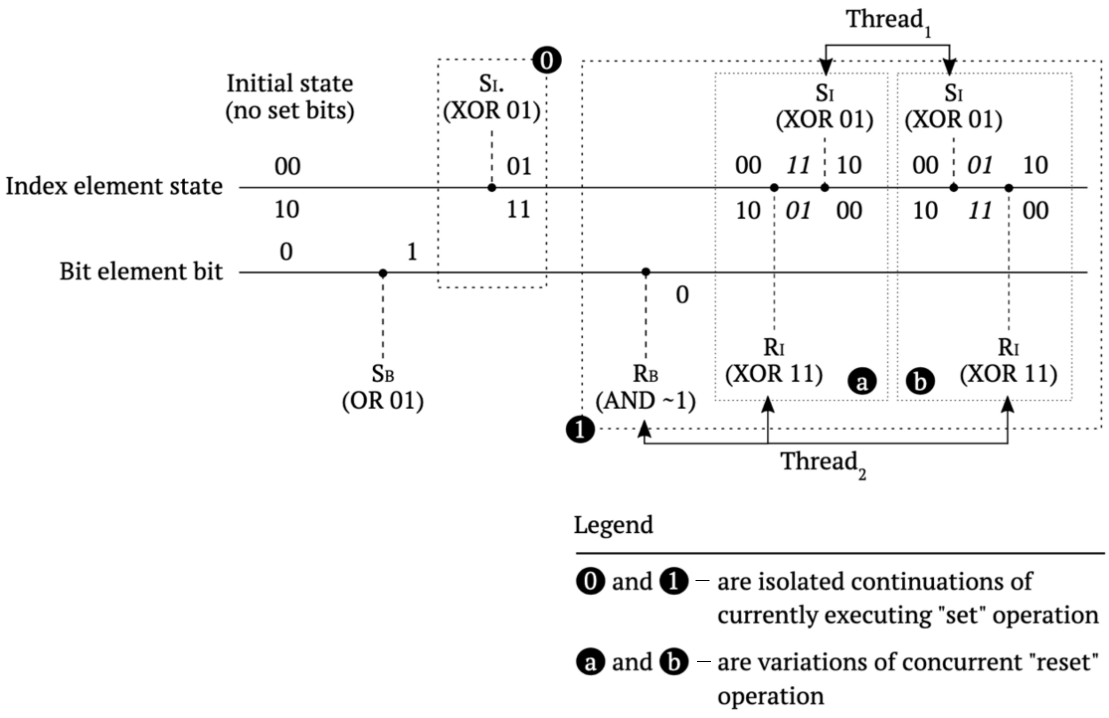

Fig. 5. An example of a situation when a set operation executed by Thread$_1$ is interrupted by a concurrent reset operation by Thread$_2$. The markers 0 and 1 present different chains of events:
0 – the set operation completed before the reset operation started;
1a – the set operation modified index after the reset operation completed;
1b – the set operation modified index after the reset operation modified the bit but before it modified the index. In all scenarios, the result is reset bit and reset index

The experimental results demonstrate that in a highly concurrent scenarios (500 – 10,000) the "Bitfield" surpasses both implementation in 2–6 times (compared to "Impl.#2"). Because both "Bitfield" and "Impl.#2" are based on the "indexed bitset" ideas, the difference in the execution time is attributed to the concurrency control implementation. It is also seen, that when the number of bits reaches 50,000, the "noise" threads start to fail to put pressure on the "allocator" thread and both "Bitfield" and "Impl.#2" experience a significant speedup.

Table 4
Execution time and comparison of experimental implementations on different number of bits

Bit count	Bitfield (ns)	*Impl.#1 (ns)*	*Impl.#2 (ns)*
500	1,337.38	3,113.35	2,996.29
1000	1,268.66	3,288.17	2,976.85
5000	456.53	5,195.32	2,930.34
10000	419.76	5,026.04	2,634.46
50000	207.28	9,030.44	892.79
100000	341.48	11,149.87	703.26

Potential application. The proposed "bitfield" implementation of a thread-safe bitset can be applied in different problems where multiple threads compete for a shared pool of resources, for instance, in scenario of static, shared memory pool (here, threads temporary allocate blocks of memory from the pool using *reset first* operation); or when multiple threads calculate work items of uneven size like in [10–15] (here, "bitfield" can represent all work items to calculate and every thread, initially, will try to calculate all work items which are multiple of the thread's index and when done, will use *reset first* operation to "steal" work items from other threads). These improvements can result in both speedup and increased energy efficiency of the applications [16].

Future work. The current "bitfield" implementation works with individual bits. The future work can be focused on implementation of additional methods to set or reset multiple bits. This can enable scenarios where a thread needs to allocate/reserve multiple resources, without sacrificing speed of *reset first* operation. In addition, with more modifications, this new implementation might be even extended to support allocation/reservation of multiple consequent resources.

Conclusion. In this paper, we introduced "bitfield" a thread-safe implementation of bitset with fast and lock free "extract min" operation, which can be used in allocation, reservation or work distribution scenarios. The experiments demonstrated that "bitfield" outperforms the "indexed bitset with multiple mutexes" implementation in 2–6 times and outperforms "bitset with lock" implementation in 2–60 times in resource allocation/reservation scenarios.

References

1. Chambi S., Lemire D., Kaser O., Godin R. Better bitmap performance with Roaring bitmaps. *Software: Practice and Experience*, 2015, vol. 46, no. 5, pp. 709–719. DOI: 10.1002/spe.2325.

2. Corrales F., Chiu D., Sawin J. Variable length compression for bitmap indices: *Database and Expert Systems Applications: 22nd International Conference*, 2011, vol. 2, pp. 381–395.

3. Goodwin B., Hopcroft M., Luu D., Clemmer A., Curmei M., Elnikety S., He Y. BitFunnel: Revisiting Signatures for Search. *SIGIR'17*: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017, pp. 605–614. DOI: 10.1145/3077136.3080789.

4. Liang W., Shi W., Liao Z., Qian S., Zheng Z., Cao J., Xue G. BOP: A Bitset-based Optimization Paradigm for Content-based Event Matching Algorithms (S). *SEKE 2023*: *The 35th International Conference on Software Engineering and Knowledge Engineering*, 2023, pp. 487–492. DOI: 10.18293/SEKE2023-142.

5. VanCompernolle M., Barford L., Harris F. Maximum Clique Solver Using Bitsets on GPUs. *Information Technology: New Generations*, 2016, pp. 327–337.

6. Gueniche T., Fournier-Viger P., Tseng V. S. Compact Prediction Tree: A Lossless Model for Accurate Sequence Prediction. *Advanced Data Mining and Applications*, 2013, pp. 177–188.

7. McIlroy R., Dickman P., Sventek J. Efficient dynamic heap allocation of scratch-pad memory: *ISMM'08*: *Proceedings of the 7th international symposium on Memory management*, 2008, pp. 31–40.

8. Terci G. S., Abdulhalik E., Bayrakci A. A., Boz B. BitEA: BitVertex Evolutionary Algorithm to Enhance Performance for Register Allocation, *IEEE Access*, 2024. vol. 12, pp. 115497–115514. DOI: 10.1109/ACCESS.2024.34465.96.

9. Karasik O. N., Prihozhy A. A. Cooperative multi-thread scheduler for solving large-size tasks on multi-core systems. *Big Data and Advanced Analytics VI*. Minsk, 2020, pp. 202–212.

10. Karasik O. N., Prihozhy A. A. Blocked algorithm of finding all-pairs shortest paths in graphs divided into weakly connected clusters. *System analysis and applied information science*, 2024, no. 2, pp. 4–10. DOI: 10.21122/2309-4923-2024-2-4-10.

11. Prihozhy A. A., Karasik O. N. New blocked all-pairs shortest paths algorithms operating on blocks of unequal sizes. *System analysis and applied information science*, 2023, no. 4, pp. 4–13.

12. Prihozhy A. A., Karasik O. N. Advanced heterogeneous block-parallel all-pairs shortest path algorithm. *Trudy BSTU* [Proceedings of BSTU], issue 3, Physics and Mathematics. Informatics, 2023, no. 1 (266), pp. 77–83 (In Russian).

13. Prihozhy A. A. Generation of shortest path search dataflow networks of actors for parallel multicore implementation. *Informatics*, 2023, vol. 20, no. 2, pp. 65–84.

14. Prihozhy A. A. Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms. *System analysis and applied information science*, 2021, no. 3, pp. 40–50.

15. Karasik O. N., Prihozhy A. A. Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation. *System analysis and applied information science*, 2022, no. 3, pp. 57–65.

16. Prihozhy A. A., Karasik O. N. Influence of shortest path algorithms on energy consumption of multicore processors. *System analysis and applied information science*, 2023, no. 2, pp. 4–12.

Список литературы

1. Better bitmap performance with Roaring bitmaps / S. Chambi [et al.]. Software: Practice and Experience. 2015. Vol. 46, no. 5. P. 709–719. DOI: 10.1002/spe.2325.

2. Corrales F., Chiu D., Sawin J. Variable length compression for bitmap indices // Database and Expert Systems Applications: 22nd International Conference. 2011. Vol. 2. P. 381–395.

3. BitFunnel: Revisiting Signatures for Search / B. Coodwin [et al.] // SIGIR'17: Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval. 2017. P. 605–614. DOI: 10.1145/3077136.3080789.

4. BOP: A Bitset-based Optimization Paradigm for Content-based Event Matching Algorithms (S) / W. Liang [et al.] // SEKE 2023: The 35th International Conference on Software Engineering and Knowledge Engineering. 2023. P. 487–492. DOI: 10.18293/SEKE2023-142.

5. VanCompernolle M., Barford L., Harris F. Maximum Clique Solver Using Bitsets on GPUs // Information Technology: New Generations. 2016. P. 327–337.

6. Gueniche T., Fournier-Viger P., Tseng V. S. Compact Prediction Tree: A Lossless Model for Accurate Sequence Prediction // Advanced Data Mining and Applications. 2013. P. 177–188.

7. McIlroy R., Dickman P., Sventek J. Efficient dynamic heap allocation of scratch-pad memory // ISMM'08: Proceedings of the 7th international symposium on Memory management. 2008. P. 31–40.

8. BitEA: BitVertex Evolutionary Algorithm to Enhance Performance for Register Allocation / G. S. Terci [et al.] // IEEE Access. 2024. Vol. 12. P. 115497–115514. DOI: 10.1109/ACCESS.2024.3446596.

9. Karasik O. N., Prihozhy A. A. Cooperative multi-thread scheduler for solving large-size tasks on multi-core systems // Big Data and Advanced Analytics VI. Minsk, 2020. P. 202–212.

10. Karasik O. N., Prihozhy A. A. Blocked algorithm of finding all-pairs shortest paths in graphs divided into weakly connected clusters // System analysis and applied information science. 2024. No. 2. P. 4–10. DOI: 10.21122/2309-4923-2024-2-4-10.

11. Prihozhy A. A., Karasik O. N. New blocked all-pairs shortest paths algorithms operating on blocks of unequal sizes // System analysis and applied information science. 2023. No. 4. P. 4–13.

12. Prihozhy A. A., Karasik O. N. Advanced heterogeneous block-parallel all-pairs shortest path algorithm. Труды БГТУ. Сер. 3, Физико-математические науки и информатика. 2023. № 1 (266). С. 77–83.

13. Prihozhy A. A. Generation of shortest path search dataflow networks of actors for parallel multicore implementation // Informatics. 2023. Vol. 20, no. 2. P. 65–84.

14. Prihozhy A. A. Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms // System analysis and applied information science. 2021. No. 3. P. 40–50.

15. Karasik O. N., Prihozhy A. A. Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation // System analysis and applied information science. 2022. No. 3. P. 57–65.

16. Prihozhy A. A., Karasik O. N. Influence of shortest path algorithms on energy consumption of multicore processors // System analysis and applied information science. 2023. No. 2. P. 4–12.

Information about the authors

Karasik Oleg Nikolayevich − PhD (Engineering), Lead Engineer, ISsoft Solutions (5 Chapaeva str., 220034, Minsk, Republic of Belarus). E-mail: karasik.oleg.nikolaevich@gmail.com

Prihozhy Anatoly Alexievich − DSc (Engineering), Professor, Professor, the Department of Computer and System Software. Belarusian National Technical University (65 Nezavisimosti Ave., 220013, Minsk, Republic of Belarus). E-mail: prihozhy@bntu.by

Информация об авторах

Карасик Олег Николаевич − кандидат технических наук, ведущий инженер. Иностранное производственное унитарное предприятие «Иссофт Солюшенз» (ул. Чапаева 5, 220034, г. Минск, Республика Беларусь). E-mail: karasik.oleg.nikolaevich@gmail.com

Прихожий Анатолий Алексеевич − доктор технических наук, профессор, профессор кафедры программного обеспечения информационных систем и технологий. Белорусский национальный технический университет (пр. Независимости 65, 220013, г. Минск, Республика Беларусь). E-mail: prihozhy@bntu.by

Received 15.11.2024