Bit Map Page Allocator: Searching Algorithm Implementations and Their Analysis

Fachry Azriel Fajdwani - 13525110
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: frielfajdwani@gmail.com , 13525110@std.stei.itb.ac.id

***Abstract*—Paging is a memory management approach that divides virtual memory into fixed-sized blocks called pages and are stored in physical memory so called page frames. The computer must track and search which page frames are not yet occupied. Bit map is proven to be the optimal data structure, while searching algorithm is left for optimization.**

***Keywords*—Paging, Bit map, Hierarchical Tree, Bitwise, Complexity Analysis, System Programming**

I. Introduction

Computer Organization and Architecture (IF1230) is a course in Informatics Engineering that teaches about the underlying components and frameworks that construct the computer as we know today. Out of many subjects there, paging is one of them.

Paging, simply put, is a memory management approach that divides virtual memory into fixed-sized blocks, unlike Segmentation that uses variable-sized blocks. Nowadays, computers utilize Paging exclusively, thus making Segmentation obsolete.

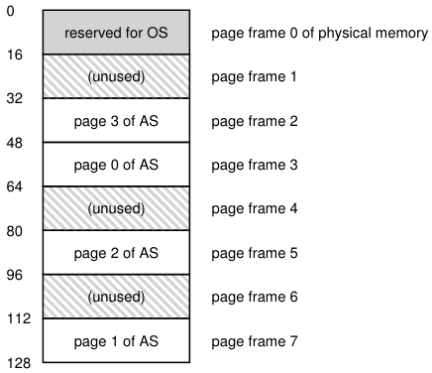


Fig. 1. Highly abstract slotted pages and page frames[3]

For a single 'page' to be stored in the memory, the computer must know which 'page frame' is not yet occupied. This requires an algorithm and data structure specifically designed to store such data. Bit map and its word-scanning searching algorithm is the most popular as of today. But just how efficient is it in regards to other algorithms? That question remains of great interest.

II. Preliminaries

A. Paging

Paging is a memory management approach that divides virtual memory into fixed-sized blocks called pages. In this approach, physical memories are called page frames. Virtual memory can be thought of as a block (page) that can be placed in a slot (page frame). Paging brings a lot of benefits: it prevents external fragmentation, and has amazing flexibility and simplicity.

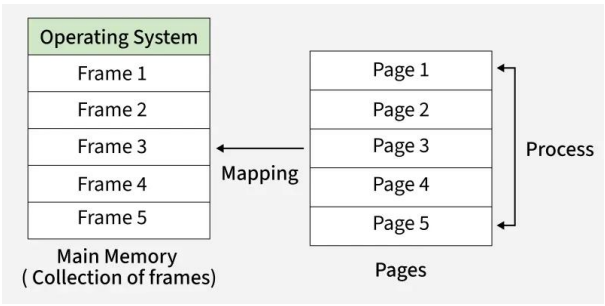


Fig. 2. Abstract representation of page frames and pages[5]

Before a page is allocated, the computer (in this case, the OS itself) must find which page frames are not occupied. This raise the need to track occupancy for such page frames. There are many data structures to choose for, but Bit map is by far the most popular and is proven optimal.

B. Bit map

Bit map (bit array or bit vector) is a data structure that store a compacted sequence of bits. It treats individual bits as flags/markers. Because a single data only uses 1 bit of space, bitmap can drastically reduce memory usage while still maintaining decent performance, compared to boolean or integer.

Index	0 1 2 3 4 5 6 . . . N
Bitmap	0 1 0 1 0 0 1 . . . 1

Fig. 3. Typical representation of a bit map[6]

In memory management, both volatile (requires electricity to save data) and non-volatile memory, each bit represents a block (or page) of memory. If the bit is 0, the block is free; if the bit is 1, the block is allocated.

C. *Boolean Algebra and Bitwise*

Boolean Algebra is an algebraic system that is used to manipulate logical values, that is true/1 and false/0, using specific logical operations and fulfills a set of axioms: identity, commutativity, distributivity, associativity, and complement.

Bitwise operators are symbols used in programming that manipulates primitive data types at the binary level. There are six bitwise operators: $\sim$ (NOT), $\wedge$ (AND), $\vee$ (OR), $\oplus$ (XOR) , $\ll$ (SHIFT LEFT), and $\gg$ (SHIFT RIGHT).

In low-level programming, boolean algebra is used to optimize control flow logic and create highly-performant bitwise-dependent algorithm. Such algorithm manipulates primitive data structure, like char, integer, and boolean to achieve specific goals. Bitwise manipulation suits really well with Bitmap data structure.

D. *Functions and Sets*

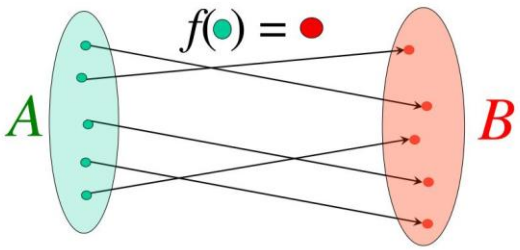


Fig. 4. Two sets that are being related by a function[7]

A set is an unordered collection of objects, called element or members of the set. A set is said to contain its elements. Derived from sets, a function is rule or relationship that pairs every input value with exactly one output value.

E. *Summation*

A summation is a consecutive addition of terms from a sequence. If the sequence is sequence is geometric, and the ratio is less than 1, that the summation can go to infinity while still being bounded. This has its own formula.

F. *Rooted N-ary Tree*

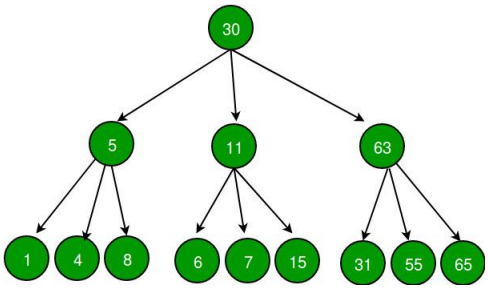


Fig. 5. A complete (or perfect) rooted n-ary tree[8]

A tree is a connected undirected graph with no simple circuits. A more specific type of tree is rooted N-ary tree, that is a tree whose internal vertex has no more than m children. Moreover, it is called 'complete' if every level is full, except perhaps the last level, and, more importantly, is compacted as far left as possible.

G. *Algorithmic Complexity*

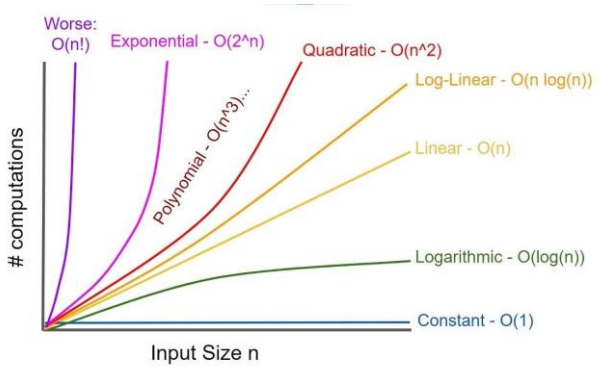


Fig. 6. General types of how a function can grow[9]

The computational complexity of an algorithm is a measure of the computer memory and time required to execute the algorithm. The complexity of an algorithm can be expressed in terms of the number of operations and space used by the algorithm when the input has a particular size. Big-$\mathcal{O}$ notation is used to estimate both time and space complexity, which provides an upper bound on the number of operations and space.

H. *CPU Characteristics*

Each CPU has their own characteristics that makes them unique and suitable for specific purposes. Out of many characteristics that constitute a CPU, there are 2 things to note for, those are CPU Word Size and Hardware Intrinsics.

Word size (word width or word length) is the natural unit of access in a computer, usually a group of 32 or 64 bits; corresponds to the size of a register in the architecture. This paper assumes the system is operated on 64 bits CPU.

Hardware Intrinsics are highly specialized, compiler-recognized functions that provide direct access to specific CPU instructions (like SIMD or AVX) for very specific tasks, such as vector math and cryptography. Many bitmap manipulations have their own specific CPU instructions.

III. Methodology

A. *Mathematical Representation*

Let $M = \{0, 1, 2, 3, \dots\}$ be a set that contains abstract identifiers (numbers) of page frames, programmatically represented as a fragmented bit map with size $|M| = NW$, for N fragments size and W CPU word size. Let $S = \{0,1\}$ be a set that contains all possible states a page frame can be.

Let $f: M \to S$. For all page frames $m \in M$, the function f will map $f(m) = 1$ if m is occupied/allocated; otherwise $f(m) = 0$.

B. Extraction using Bit Masking

Perhaps the most widely used technique in low-level programming, bit mask uses a specific bit pattern to isolate the wanted bits, and applying specific steps of bitwise operation to extract the information.

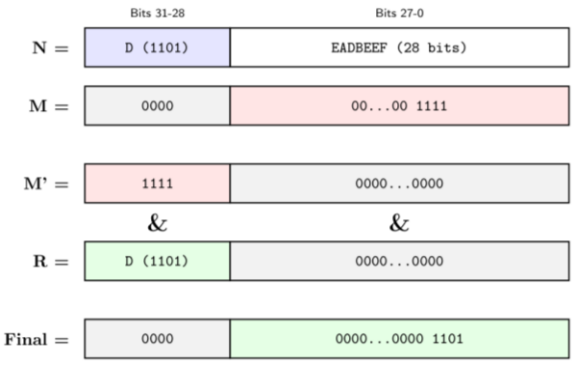


Fig. 7. Information extraction process using bit masking

- Create a bit mask. Here, $N = 0xDEADBEEF$ *and* $M = 0xF$. N is the target and M is the pattern; both numbers are 32-bit numbers.
- Utilize $\ll$ to move the mask to wanted bits. In this case, we want to extract bits starting from 3rd bit from most significant bit downward: $M' = M \ll 28$.
- Utilize & to extract the wanted bit: $R = N \,\&\, M'$.
- Finally, utilize $\gg$ to remove trailing zeros. This normalizes the result into a number that has a significant meaning: $R \gg 28$.

C. Two's complement

In most computer architectures, signed numbers (numbers that have a sign, so it can be positive or negative) are represented using 2's complement. There are two interpretations for negative numbers using this system: (1) its most significant bit has negative weight and (2) it is the process of taking complement and an increment.

Formally, 2's complement base-2 number is defined as base-2 number system whose most significant bit is multiplied with negative values. Here, V is a function that turn 2's complement base-2 number x_2 with W-bits to its base-10 representation x_{10}.

$$V(x) = -x_{W-1}2^{W-1} + \sum_{i=0}^{W-2} -x_i 2^i$$

A more practical definition is given by the latter interpretation. Let $\bar{x}$ be the complement of a number such that $x + \bar{x} = 0$. In base-2, such equation would result in a number that have all their bits set (i.e. all bits set to 1). Thus, for base-2 number with W-bits

$$x + \bar{x} = 2^W - 1$$

By taking modulo of 2^W on both side and isolating $\bar{x}$, we get

$$-x \equiv \bar{x} + 1 \; (mod\; 2^W)$$

We have shown that indeed $-x = \bar{x} + 1$ in base-2 W-bit numbers, and programmatically written as $-x \leftarrow \sim x + 1$.

D. Rightmost Set Bit

Sometimes, like in this particular case, it is useful to find the rightmost set bit, that is the very right bit that has the value of 1. One of its usages (as we are about to find out) is to get the index of an element in a bit map. There are two approaches besides loop with constant time complexity: (1) bitwise manipulation and (2) hardware intrinsic.

Approach (1) is perhaps the most elegant use of bitwise. Let x and its complement $\bar{x}$ be represented as base-2 2's complement numbers with 3 parts: A, M, and B.

$$x = A\ 1\ B$$

Where B is the trailing zeros, 1 (the middle) is thus the rightmost bit, and A is whatever left besides them. Now, if we take the complement $\bar{x}$

$$\bar{x} = \overline{(A\ 1\ B)} = \bar{A}\ 0\ \bar{B}$$

And then take the successor, it becomes

$$\bar{x} + 1 = \bar{A}\ 1\ B$$

It is obvious that $-\bar{x} + 1 = -x$. Now, if we take the bitwise $\wedge$ of x and $-x$, it will become

$$x \wedge -x = (A\ 1\ B) \wedge (\bar{A}\ 1\ B) = 0\ldots0\ 1\ 0\ldots0$$

The middle and the B's stays while the A's cancel out. Remember, by definition B is just a set of 0's. This leaves us with a single set bit, the rightmost set bit. Because only a single bit is set, it is guaranteed to be a power of 2, say 2^k. And to get the position of the bit, we can just take the logarithm base-2.

$$POS = log_2(x \wedge -x) = log_2(2^k) = k$$

Chef kiss! This algorithm runs magnitude faster than traditional looping, but there is an even better way. Instead of manually constructing the formula, we can just utilize what the CPU already has.

Out of many hardware intrinsics that modern CPU packs with, there are 2 that are particularly useful for bitwise manipulation. Those are `__builtin_clz` (count leading zeros)

and `__builtin_ctz` (count trailing zeros). Both run in constant time complexity and are a lot faster than the previous algorithm.

E. Searching Algorithm

There are 3 widely known searching algorithm utilized in bit map for memory management: linear scanning, word-scanning, and hierarchical bit map. Set M is represented as static array of unsigned W-bit integers with N elements, and follows $|M| = NW$.

1) Linear Scanning

In this algorithm, occupancies are checked one-by-one, bit-by-bit in a transversal manner. For-loop is used to traverse elements and bits. This is the most straightforward, naïve bit searching algorithm.

Algorithm 1 Searching Algorithm *Linear Scanning* in Bit map

```
1: procedure SearchNaive(M, N, W)
2:     for i ← 0 to N − 1 do
3:         for j ← 0 to W − 1 do
4:             if (M[i] ∧ (1 ≪ j)) = 0 then
5:                 return j + (W × i)
6:             end if
7:         end for
8:     end for
9:     return −1
10: end procedure
```

Fig. 8. Linear scanning pseudocode

Here, bit masking technique is utilized. Mask $M = 1$ (written in base-10) is shifted step-by-step, to extract the j-th bit from the fragmented bit map, forming $M[i] \wedge (1 \ll j)$.

It is obvious the time complexity grows as the size of the bit map grows, while keeping the space complexity constant.

TABLE I. LINEAR SCANNING ALGORITHMIC COMPLEXITY

Time Complexity	$O(\|M\|) = O(NW)$
Space Complexity	$O(1)$

2) Word-scanning

Instead of scanning for set bit using bit masking and loop, we will introduce a hardware intrinsic specifically design for this task, `__builtin_ctzll`, 64-bit version of `__builtin_ctz`. We will denote the function as $ctz(x)$.

Algorithm 2 Searching Algorithm *Word-Scanning* in Bit map

```
1: procedure SearchWordScanning(M, N, W)
2:     for i ← 0 to N − 1 do
3:         if M[i] ≠ 2^W − 1 then
4:             return ctz(∼ M[i]) + (W × i)
5:         end if
6:     end for
7:     return −1
8: end procedure
```

Fig. 9. Word-scanning pseudocode

The function $ctz(x)$ counts how many zeros are there before the very first or rightmost set bit. What we want, instead, is to find the rightmost **unset** bit. Fortunately, it is as simple as counting the trailing zeros of the inverse of the fragmented bit map, written $ctz(\sim M[i])$.

Space complexity remains the same, while time complexity becomes a lot efficient.

TABLE II. WORD-SCANNING ALGORITHMIC COMPLEXITY

Time Complexity	$O\left(\frac{\|M\|}{W}\right) = O(N)$
Space Complexity	$O(1)$

3) Hierarchical Tree Scan

We will go further. Let us redefine M as a complete W-ary tree with depth $D = log_W(|M|)$. Parent nodes are summaries of their children, and are only set if all of their children are set. There are $|M| = W^D$ leaf nodes (or elements). This setup makes it possible to eliminate the outer, linear loop that the two previous algorithms have, and only take at most $O(D)$ operations.

Algorithm 3 Searching Algorithm *Hierarchical Bit map* with arbitrary depth

```
1: procedure SearchHierarchical(T, D, W)
2:     if T[0][0] = 2^W − 1 then
3:         return −1
4:     end if
5:     idx ← 0
6:     for k ← 0 to D − 1 do
7:         fragment ← T[k][idx]
8:         pos ← ctz(∼ fragment)
9:         idx ← (idx × W) + pos
10:    end for
11:    return idx
12: end procedure
```

Fig. 10. Hierarchical tree scan pseudocode

Interpreting parent nodes as summaries of their children makes it possible to completely eliminate horizontal linear transversal of the entire bit map. Instead, this approach traverses the whole data vertically. The complexity plummenvoriets drastically from $O(N)$ to $O(D) = O(log_W(|M|))$.

The whole algorithm itself still has $O(1)$ space complexity. But fundamentally, the data structure changed. Hierarchical bit map consumes more memory than traditional bit map.

Let $|M|$ be the size of the elements. Then, we have $|M|/W$ parent nodes. Those parent nodes have parents themselves, so another $|M|/W^2$ nodes. This keeps going, adding $|M|/W^x$ nodes until we reach the root node. We can formally define this as

$$T = \frac{|M|}{W} + \frac{|M|}{W^2} + \frac{|M|}{W^3} + \cdots + 1 = \sum_{i=1}^{x} \frac{|M|}{W^i}$$

As x approaches infinity, the sum then becomes

$$T_\infty = \frac{|M|}{1-\frac{1}{W}} = \frac{NW}{1-\frac{1}{W}} = \frac{N}{W-1}$$

So, there is indeed an overhead, resulting in about $1/(W-1)$ more memory usage. This overhead is introduced in data structure space complexity, the initial setup, rather than the space complexity for the algorithm itself. **Auxiliary space complexity** is what dictates the space complexity of the algorithm without the initial setup.

TABLE III. HIERARCHICAL TREE SCAN ALGORITHMIC COMPLEXITY

Time Complexity	$O(log_W(\|M\|))$
Space Complexity	$O(1)$ Auxiliary
	$O\left(\frac{N}{W-1}\right)$ Setup

F. Experimental Setup and Evaluation Methodology

1) Environment

Definitions for CPU, host, subsystem/virtual kernel, programming language, and compiler with all the options, including the standard and flag used for compilation.

TABLE IV. ENVIRONMENTAL CONDITIONS

Components	Value
CPU Model	Intel Core Ultra 7 258V
CPU Clock Frequency	2.2–4.8 GHz
CPU L1/L2/L3 Cache	832KB/14MB/12MB
CPU Architecture	x86_64 / AMD64
Host OS	Windows 11 Home 25H2
Host Subsystem	WSL 2 Ubuntu 22.04 LTS
Compiler/Language	GCC 11/C Language
Compiler Standard	gnu17 (c17 extension)
Compiler Flag[a]	`-O2 -fno-unroll-loops -fno-tree-vectorize`

[a] The compiler flag is chosen that way to minimize unwanted optimizations that may tamper with the actual result, while keeping some optimizations intact.

2) Implementation

The algorithms are implemented in C17 with GNU's own extension. Library usage is kept minimum. This section also includes test cases constraints, test cases generation, and measurement specification.

TABLE V. CONSTRAINTS

Variables	Value
N (word fragments count)	$2^8 - 2^{24}$, is guaranteed to be a power of 2
W (word size)	8, 16, 32, 64

TABLE VI. DATA STRUCTURES

Variables	Value
M (traditional bit map)	array of W-bit integers
W, N	integers
M (hierarchical bit map)	flattened tree; array of W-bit integers

TABLE VII. MEASUREMENT SPECIFICATIONS

Measurements	Details
Total Time	`clock_gettime()` function with flag `CLOCK_MONOTONIC` set from header `<time.h>` is used to keep track of the absolute and pure execution time with nanoseconds precision. Will be repeated a total of 1000 times to reduce noise.
Total Clock Cycle	`__rdtsc()` function from header `<x86intrin.h>` is used to keep track of the absolute clock cycle using the provided clock cycle counter. Will be repeated a total of 1000 times to reduce noise.
Memory Overhead (Setup)	No specific function. Instead, we can simply track the size of bytes being allocated by `malloc()` or `calloc()`. Then, we calculate the ratio. Will be repeated a total of 1000 times to reduce noise.

3) Measurement Evaluations

Firstly, page frames will be allocated contiguously until a saturation rate of 65% is achieved. Then, the remaining 35% memory will be allocated randomly using `rand()` from the header `<stdlib.h>`. This way, the condition mimics the condition of a random fragmentation and bottom-filled memory in real world use case, especially in heavy duty and high-performance computing environment

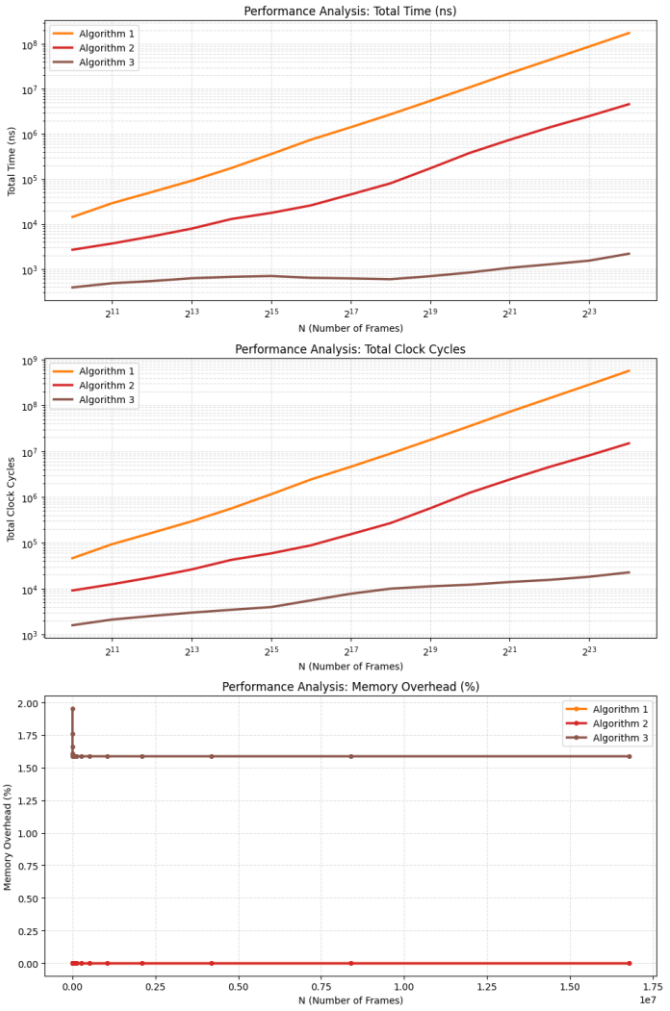
TABLE VIII. MEASUREMENT EVALUATIONS

Measurements	Details
Total Time	Time-bound measurement in nanoseconds that shows how long it takes for an algorithm to complete the task. The most straight-forward complexity measurement. The graph is a log scale graph.
Total Clock Cycle	Time-bound measurement that shows how many clock cycles it takes for an algorithm to complete the task. It is hardware-agnostic and is the absolute measurement of

	complexity. The graph is a log scale graph.
Memory Overhead (Setup)	Non-time-bound measurement that show how much additional memory is being used by an algorithm to complete the task. The graph will be represented as is.

IV. Results

A. *Word Size 64 (64-bit system)*

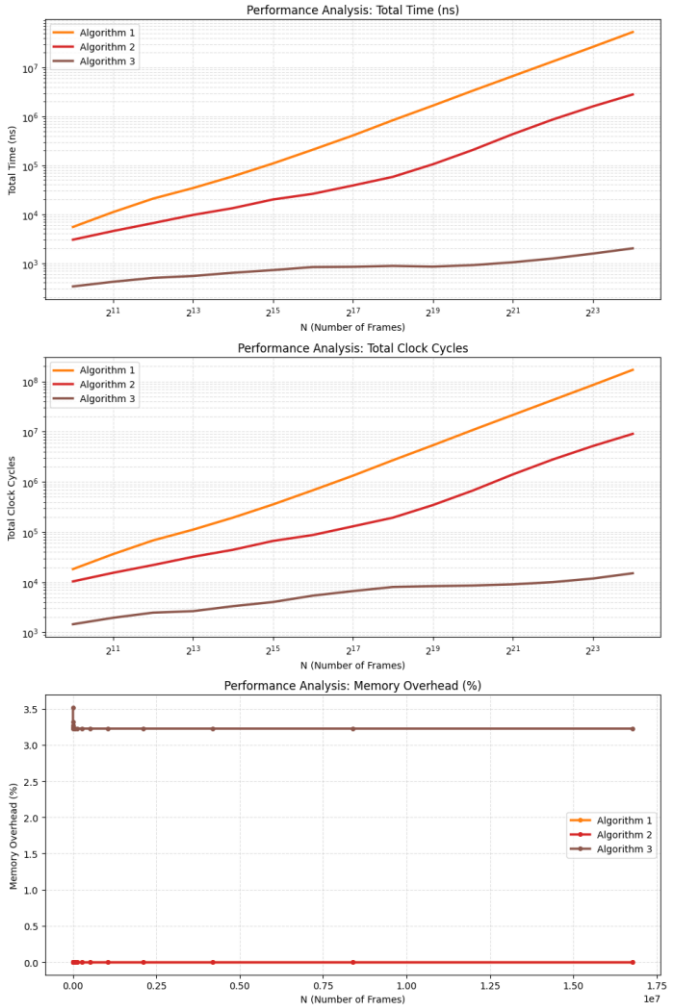


Aspects	Value
Algorithm 1 Time Range	5,469.19–302,122,249.62 ns
Algorithm 2 Time Range	1,266.91–7,498,565.58 ns
Algorithm 3 Time Range	250.06–3,170.94 ns
Algorithm 1 Slope	0.9763
Algorithm 2 Slope	0.7801
Algorithm 3 Slope	0.1577
Algorithm 3 Overhead	1.63%

Fig. 11. Line graph, time ranges, and slopes for W=64

Summary: It is obvious that hierarchical bit map searching algorithm run faster by a significant margin compared to the previous two algorithm. Algorithm 2 scored a decent performance. It absolutely shows that linear transversal algorithm is not effective in heavy duty or high-performance computing environment.

B. *Word Size 32 (32-bit system)*

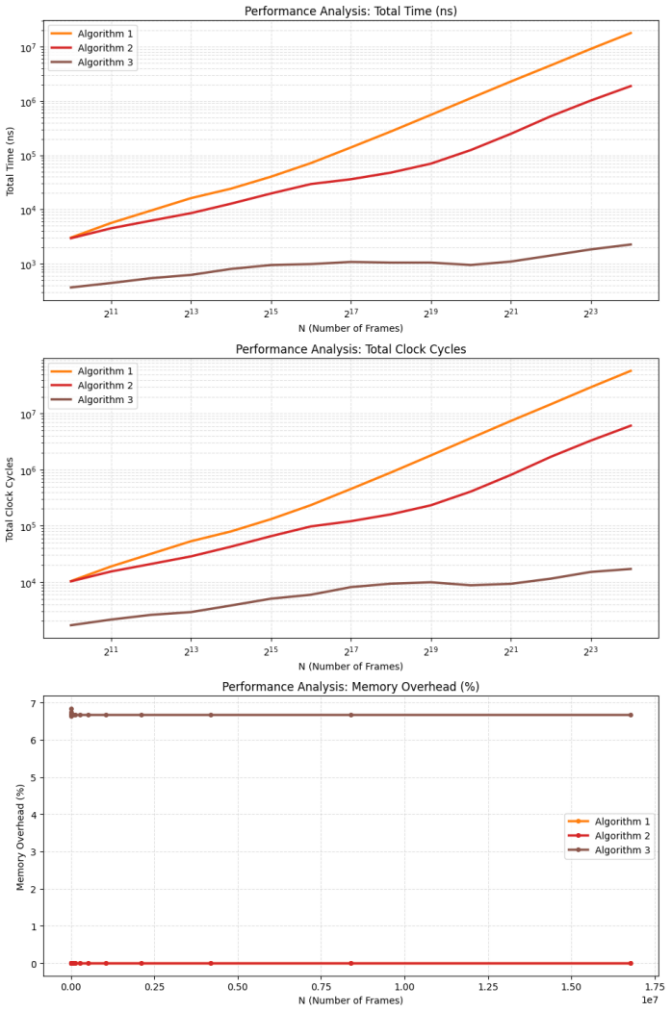


Aspects	Value
Algorithm 1 Time Range	2,197.10–90,993,814.05 ns
Algorithm 2 Time Range	1,515.67–4,373,206.82 ns
Algorithm 3 Time Range	249.44–2,601.18 ns
Algorithm 1 Slope	0.9458
Algorithm 2 Slope	0.6948
Algorithm 3 Slope	0.1683
Algorithm 3 Overhead	3.25%

Fig. 12. Line graph, time ranges, and slopes for W=32

Summary: Algorithm 3 stays relatively fast as shown in the graph having the lowest grow rate. Interestingly, while its slope is decreasing by a wide amount compared to 64-bit word size, algorithm 2 starting to show struggle for small N values. The gap between algorithm 1 and 2 noticeably shrinks.

C. *Word Size 16 (16-bit system)*

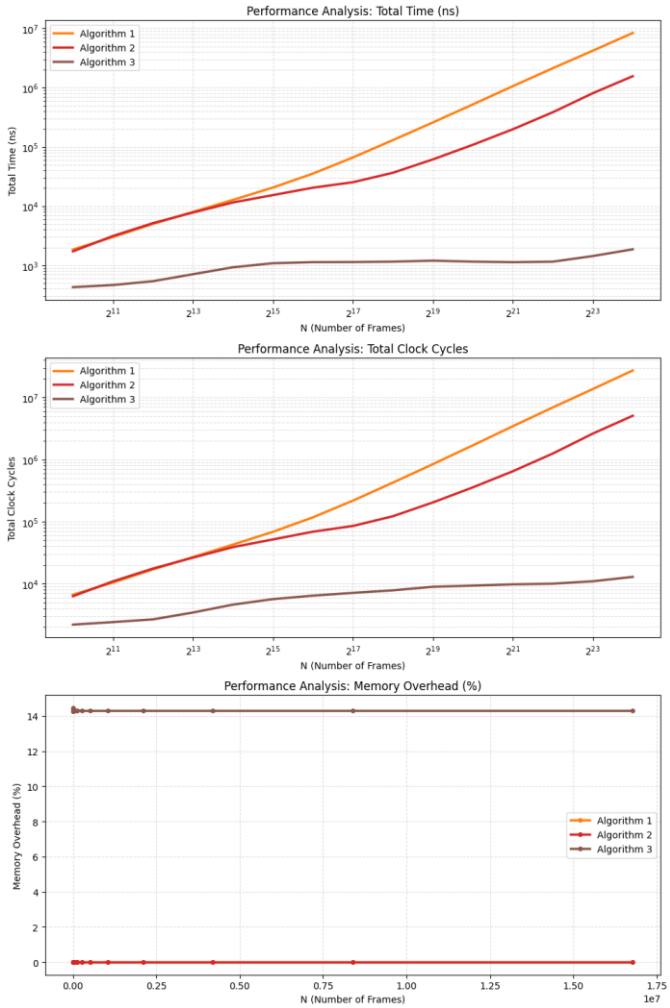


Aspects	Value
Algorithm 1 Time Range	1,310.67–30,154,020.10 ns
Algorithm 2 Time Range	1,619.31–3,013,872.56 ns
Algorithm 3 Time Range	290.18–2,681.80 ns
Algorithm 1 Slope	0.8946
Algorithm 2 Slope	0.6404
Algorithm 3 Slope	0.1661
Algorithm 3 Overhead	6.68%

Fig. 13. Line graph, time ranges, and slopes for W=16

Summary: The gap shrinks even further. For small N values, there is virtually no difference between algorithm 1 and algorithm 2. Furthermore, the upper bound for algorithm 2 shrinks significantly slower than algorithm 1 does. Algorithm 1 is also showing signs of deviance from the traditional sense of linearity, as shown by its slope.

D. *Word Size 8 (8-bit system)*



Aspects	Value
Algorithm 1 Time Range	906.34–14,367,938.12 ns
Algorithm 2 Time Range	1,027.20–2,592,842.15 ns
Algorithm 3 Time Range	366.43–2,361.34 ns
Algorithm 1 Slope	0.8684
Algorithm 2 Slope	0.6645
Algorithm 3 Slope	0.1357
Algorithm 3 Overhead	14,3%

Fig. 14. Line graph, time ranges, and slopes for W=8

Summary: It really shows the true nature of word scanning using hardware intrinsic: high penalty for considerably small word size. For very small word size, the difference is negligible. Shockingly, for $N = 2^8$, algorithm 1 run 100ns faster.

V. Discussion

The graphs generally follow what we have already predicted earlier, that is algorithm 3 run the fastest, followed by algorithm 2 with a massive gap, and lastly algorithm 3. For large word size values, the task is completed in as fast as 250ns, and as long as 3 million ns from algorithm 1! Compared to the upper bound for algorithm 3, that is **1000x slower**.

Starting at word size equals 32, the gap between algorithm 1 and algorithm 2 appear to shrink noticeably. Indeed, as we can see down the road, the gap is increasingly getting smaller. In fact, for word size equals 8, there is absolutely no difference between algorithm 1 and algorithm 2 for rather moderately small N values. The gap diminishes. There are many variables that influence the result, especially the obvious fact that these tests ran on 64-bit system, so it is not optimized for 16-bit and 8-bit integers manipulation. The 64-bit GCC 11 compiler with gnu17 standard only provides $ctz(x)$ as low as 32-bit integer. So, it mandatory for $W < 32$ integers to be casted to 32-bit integer like so: `#define ctz(x) __builtin_ctz((uint32_t)x)`. This introduces a rather large overhead.

Interestingly, the memory overheads are exactly like we predict. Because algorithm 3 uses a different data structure, there will be additional memory usage compared to algorithm 1 or 2. The overhead is exactly $1/(W-1)$, so it grows as W shrinks.

Word Size	Calculated Overhead	Actual Overhead
64	$\frac{1}{63} \approx 0.0158$	0.0163
32	$\frac{1}{31} \approx 0.0326$	0.0325
16	$\frac{1}{15} \approx 0.0667$	0.0668
8	$\frac{1}{7} \approx 0.1428$	0.143

Fig. 15. Algorithm 3 memory overhead calculations

VI. Conclusion

Algorithm 3, hierarchical bit map scan, is the absolute fastest algorithm out of the 3. It excels in all possible word size values, although requires $1/(W-1)$ more memory compared to algorithm 1 and algorithm 2. For $W = 8$, the overhead is about 14,28%, which may be a dealbreaker for 8-bit system like microcontrollers, where memory usage holds higher priority than raw performance.

For system with small word size, linear scan or word scan algorithm is better, because it delivers a good performance without introducing additional memory overhead. As the graph suggest, the gap between the two algorithm diminishes the smaller the word size.

Miscellaneous Links

[1] https://drive.google.com/drive/folders/19dNl1_mVMmejzlHHPpqMyug6lAU8RZhR?usp=sharing (Google Drive work folder, code included)

Acknowledgment

This paper is not possible without the help and support that my colleagues and lecturer, Prof. Dr. Ir. Rinaldi Munir, give. Their expertise in computer architectures and discrete mathematics really ease the process of research and understanding.

References

[1] R. Munir, "Materi Kuliah Matematika Diskrit," Informatika STEI-ITB, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/. [Accessed: June 17, 2026].

[2] K. H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed. New York: McGraw-Hill Education, 2019.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Madison, WI: Arpaci-Dusseau Books, 2018.

[4] R. M. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3rd ed. Boston: Pearson, 2016.

[5] GeeksforGeeks, "Paging in Operating System," Oct. 2023. [Online]. Available: https://www.geeksforgeeks.org/operating-systems/paging-in-operating-system/. [Accessed: June 17, 2026].

[6] K. M. A. Hasan, "An Introduction to Roaring Bitmaps for Software Engineers," *Medium*, Nov. 2022. [Online]. Available: https://machine-learning-made-simple.medium.com/an-introduction-to-roaring-bitmaps-for-software-engineers-dd98859dd29a. [Accessed: June 17, 2026].

[7] E. D. Demaine, "Lecture Notes on Sets and Functions," Massachusetts Institute of Technology, 2011. [Online]. Available: https://www.slideserve.com/ulric/sets-and-functions. [Accessed: June 17, 2026].

[8] GeeksforGeeks, "Generic Tree (N-ary Tree)," Jan. 2024. [Online]. Available: https://www.geeksforgeeks.org/dsa/what-is-generic-tree-or-n-ary-tree/. [Accessed: June 17, 2026].

[9] S. Subramani, "Understanding Algorithmic Complexity: A Comprehensive Guide," *LinkedIn Pulse*, Feb. 2025. [Online]. Available: https://www.linkedin.com/pulse/understanding-algorithmic-complexity-comprehensive-guide-subramani-y98zc/. [Accessed: June 17, 2026].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Tangerang, 17 Juni 2026

Fachry Azriel Fajdwani, 13525110