Mathematical modeling and experimental evaluation of the information content completeness at fixed memory volumes: application of compression algorithms*

Mykola Mitikov[1,†], Natalia Guk[1,†], Roman Voliansky[2,*,†], Mykhailo Mozhaiev[3,†], Andri Pranolo[4,†]

[1] *Oles Honchar Dnipro National University, Ukraine*
[2] *National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Beresteiskyi Ave., 37, Kyiv, 03056, Ukraine Scientific Research Institute of Forensic Sciences, Ukraine*
[3] *Scientific Research Institute of Forensic Sciences, Ukraine*
[4] *Universitas Ahmad Dahlan, Jl. Kapas No.9, Semaki, Kec. Umbulharjo, Kota Yogyakarta, Daerah Istimewa Yogyakarta, 55166, Indonesia*

Abstract
This paper analyses the problem of preserving the completeness of information content under conditions of fixed amounts of RAM. Modern software development practices, which are primarily focused on functionality, often neglect the optimal use of resources, leading to excessive memory consumption. A mathematical model and approach are proposed, based on the analysis of memory snapshots and the application of compression algorithms (in particular, LZ4) for byte[] arrays. This makes it possible to increase the amount of stored information without expanding the existing memory capacity. The paper also presents the results of an experimental evaluation of the effectiveness of the selected compression algorithms, confirming the feasibility of the proposed approach. Existing scientific publications focus on the study of memory leak problems and limit their attention to excessive memory usage due to the lack of a unified model for searching for excessive memory usage. Industrial systems that require hundreds of GB of RAM to operate and contain millions of objects in RAM are analysed. With such large amounts of data, there is a need for efficient memory usage. The research method is to analyse memory snapshots of highly loaded systems using software code developed on .NET technology and the ClrMD library. A memory snapshot reflects the state of the process under study at a given moment in time and contains all objects, threads, and operations being performed. The ClrMD library allows you to programmatically examine objects, their types, obtain field values, and build graphs of relationships between objects. Based on the results of the study, an optimisation was proposed that significantly reduces memory consumption. The scientific contribution of the research lies in the creation of a mathematically sound approach that contributes to a significant reduction in memory resource usage and optimisation of computational processes. The practical usefulness of the model is confirmed by the optimization results achieved thanks to the recommendations obtained, the reduction in hosting costs (which ensures greater economic efficiency in the deployment and use of software systems in industrial conditions), as well as an increase in the volume of processed data.

Keywords
mathematical model; optimisation; algorithm; performance; memory snapshot; array

1. Introduction

The development of information technology has led to the widespread use of software applications in many areas of modern life. The need to process ever-increasing amounts of information can be offset by increasing computing power [1] or optimising existing software implementations [2].

Optimising memory usage by a software application requires a deep understanding of how the system works and access to the development process, which also makes this method economically

ITTAP'2025: 5th International Workshop on Information Technologies: Theoretical and Applied Problems, October 22–24, 2025, Ternopil, Ukraine, Opole, Poland
[1*] Corresponding author.
[†] These authors contributed equally.

mitikov.m22@fpm.dnu.edu.ua (M. Mitikov); huk_n@fpm.dnu.edu.ua (N. Guk); volianskyi.roman@lll.kpi.ua (Roman Voliansky); mozhayev.misha89@gmail.com (M. Mozhaiev); andri.pranolo.id@ieee.org (A. Pranolo)
0009-0002-1297-5676 (M. Mitikov); 0000-0001-7937-1039 (N. Guk); 0000-0001-5674-7646 (Roman Voliansky); 0000-0003-1566-9260 (M. Mozhaiev); 0000-0002-3677-2788 (A. Pranolo)

unjustified when there is a more expedient scaling option available. Scaling computing power is beneficial for small sizes, but becomes less and less economically viable with each increase in the size of the virtual machine. Beyond pure scaling, efficiency gains are often achieved through mathematically grounded modeling and representation choices that preserve structure while reducing footprint [3].

Most existing publications [4], [5] focus on researching the problem of memory leaks, paying less attention to excessive memory usage. It is known that a memory leak is a condition where a program mistakenly cannot release unnecessary memory for reuse and therefore continues to allocate new blocks.

At the same time, questions remain open regarding the assessment of the need to use memory capacity and the completeness of information content. That is, to represent information, it is possible to use data in a compressed form with dense packing, or in an arbitrary form with a large number of voids. In both cases, the data is presented in the form of byte arrays, but different amounts of memory are used to store the same information. When using the compressed form, it is possible to store more information per unit of physical memory. This aligns with information-preserving transforms used in signal and image processing, where homomorphic filtering and description minimization reduce storage while retaining task-relevant content [6], [7]. This paper proposes a mathematical model and experimental verification of an approach to compressing byte objects (in particular, using the LZ4 algorithm) while preserving the accessibility of all information. This approach avoids costly scaling and at the same time increases system performance by reducing data storage overhead. Comparable optimization mindsets are widely reported across industrial electrical systems and drives, where tighter models and representations translate into measurable operational gains [8], [9], [10], [11], [12], [13].

2. Literature review

Nguyen's research [14] examines in detail the problems of memory bloat (increased memory usage) in managed systems with automatic memory management, while Lee [15] proposes solutions to reduce excessive memory usage and improve performance through allocator optimisation and object recycling. According to Xia [16], hardware-accelerated memory compression minimises the load on the central processor and maintains system stability. This work highlights the use of modern processor capabilities, namely specific intrinsics to accelerate compression algorithms. Orthogonally to hardware acceleration, structural modeling can also improve memory behavior by conserving semantics under transform, as in homomorphic object and filtering frameworks [3], [6]. The initial results of the article prove the feasibility of using compression in high-load systems, but also bypass data detection mechanisms for compression.

In their work, Tsai and Sanchez [17] propose the concept of object-level compression (Zippads) instead of cache lines, which improves the compression ratio and reduces memory traffic. The first compressed memory hierarchy specifically designed for object-oriented programs is presented, where the object is the natural unit of compression and redundancy is observed between objects of the same type. Thanks to the Zippads solution, which provides transparent compression of variable-size objects, it was possible to achieve a compression ratio 1.63 times higher and a performance gain of 17% compared to current technologies. At the representation level, minimizing description length for application-specific data similarly reduces memory traffic and storage costs while preserving accessibility for downstream processing [7].

Excessive memory usage is difficult to detect because it is not explicitly manifested – the memory leak criterion is not met [18]. Complementary dump-centric analyses have been shown effective for surfacing overuse patterns in production systems and motivating compression-friendly refactorings [18].

In [19] by Microsoft engineers (OSDI 2022), the RESIN system is presented, which illustrates the capabilities of dynamic memory snapshot analysis in high-load cloud environments and can be adapted not only for detecting memory leaks but also for monitoring excessive usage. It continuously monitors memory usage on thousands of servers, using the ‘bucketisation’ method to filter out normal fluctuations and detect anomalies. When a leak is suspected, the system takes real-time memory dumps from problematic processes, analyses them to find unfreed objects and stack calls that caused the problem, and then automatically applies temporary measures, such as restarting services or clearing resources, until developers fix the cause of the leak.

3. Problem statement

Develop a recommendation system for estimating the amount of RAM reduction when storing information in objects of type t = System. Byte (byte array) using compression algorithms. We target array-resident data because its task-level semantics can often be preserved under compact encodings, akin to homomorphic transforms in signal pathways [6], [7]. A byte array is used to represent file contents, send messages between systems, store information in binary form, and is the primary carrier of information in computer systems.

4. Mathematical model of a memory snapshot

A program memory snapshot can be represented as a structured array of bytes:

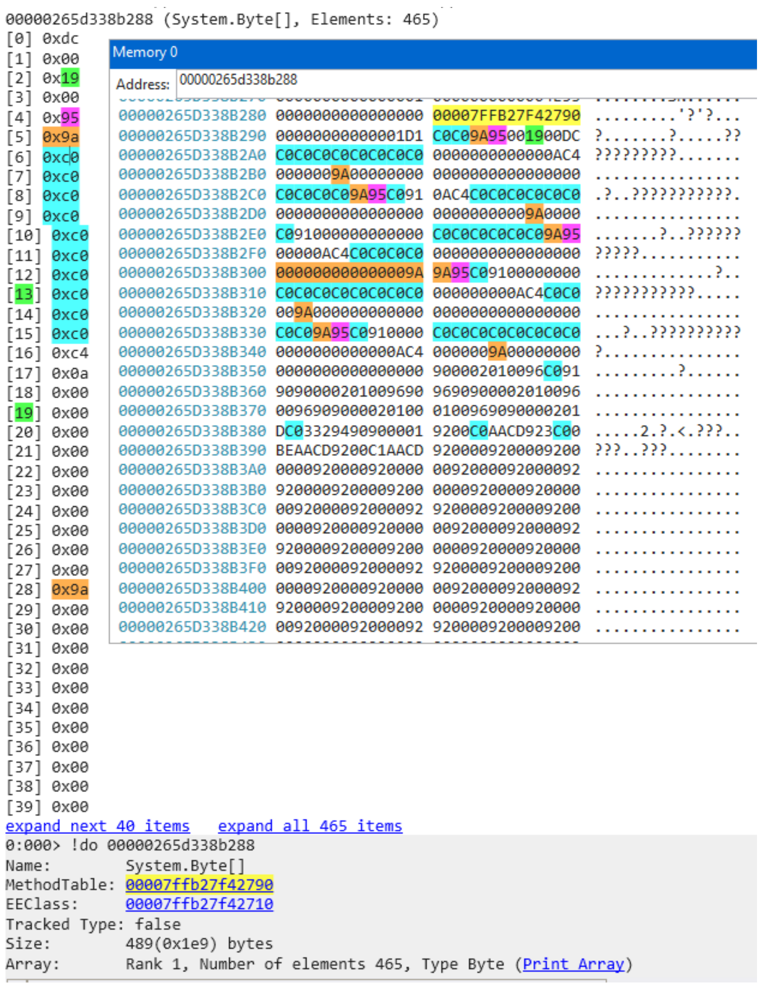
$$M_{time}=[b_1,b_2,\ldots,b_c], \quad (1)$$

where $time$ - is the moment in time when the memory snapshot is taken; are the bytes that make up the snapshot; $b_1,b_2,\ldots,b_c$ - is the number of bytes.

The memory snapshot captures a comprehensive view of the runtime context: it records the operations currently in progress, the scheduling and lifecycle states of execution threads, the set of objects owned or manipulated by those threads, and the loaded program code with its relevant metadata. In practical terms, this includes thread identifiers, call stacks, and scheduler-visible states (e.g., running, runnable, waiting on synchronization), along with the synchronization primitives and queues that govern inter-thread coordination. By correlating these elements, the snapshot enables us to link high-level activities to specific code paths and data structures, revealing where time and memory are actually being consumed. Information about objects is represented in a way that is directly useful for downstream analysis. For each object, the snapshot retains its concrete type (class/interface), salient state (e.g., size, mutability, lifetime/generation, allocation site), and its relationships to other objects expressed as references, ownership, and containment. These relationships form a graph over the heap that allows us to trace producer−consumer chains, identify parent - child aggregates, and detect shared or cyclic structures. Such structured metadata makes it possible to attribute memory pressure to particular subsystems, distinguish long-lived caches from transient buffers, and pinpoint candidates for compaction or pooling.

Our modeling stance follows prior art on homomorphic structures: we construct mappings from the concrete runtime heap to an abstract model that preserve essential properties - such as reachability, equivalence classes of objects, and aggregate counts - under transformation, thereby enabling compact yet faithful states [3]. In this view, references become typed edges, object kinds act as labels, and groups of shape-identical objects can be folded without losing the invariants required for analysis. The benefit is that we can answer the same queries (e.g., “which threads retain these buffers?” or “what fraction of memory is accounted for by arrays of type X?”) on a smaller representation, reducing processing time and storage while maintaining semantic fidelity.

Figure 1 shows the appearance of the O_{b_1} byte array in memory.



```
00000265d338b288 (System.Byte[], Elements: 465)
[0] 0xdc
[1] 0x00
[2] 0x19
[3] 0x00
[4] 0x95
[5] 0x9a
[6] 0xc0
[7] 0xc0
[8] 0xc0
[9] 0xc0
[10] 0xc0
[11] 0xc0
[12] 0xc0
[13] 0xc0
[14] 0xc0
[15] 0xc0
[16] 0xc4
[17] 0x0a
[18] 0x00
[19] 0x00
[20] 0x00
[21] 0x00
[22] 0x00
[23] 0x00
[24] 0x00
[25] 0x00
[26] 0x00
[27] 0x00
[28] 0x9a
[29] 0x00
[30] 0x00
[31] 0x00
[32] 0x00
[33] 0x00
[34] 0x00
[35] 0x00
[36] 0x00
[37] 0x00
[38] 0x00
[39] 0x00
expand next 40 items   expand all 465 items
```

```
Memory 0
Address: 00000265d338b288
00000265D338B280 0000000000000000 00007FFB27F42790 .........'?'?...
00000265D338B290 00000000000001D1 C0C09A95001900DC ?.......?.....??
00000265D338B2A0 C0C0C0C0C0C0C0C0 0000000000000AC4 ?????????.......
00000265D338B2B0 0000009A00000000 0000000000000000 ................
00000265D338B2C0 C0C0C0C09A95C091 0AC4C0C0C0C0C0C0 .?..???????????.
00000265D338B2D0 0000000000000000 0000000009A0000 ................
00000265D338B2E0 C091000000000000 C0C0C0C0C0C09A95 .......?..??????
00000265D338B2F0 00000AC4C0C0C0C0 0000000000000000 ?????...........
00000265D338B300 000000000000009A 9A95C09100000000 .............?..
00000265D338B310 C0C0C0C0C0C0C0C0 000000000AC4C0C0 ???????????.....
00000265D338B320 009A000000000000 0000000000000000 ................
00000265D338B330 C0C09A95C0910000 C0C0C0C0C0C0C0C0 ...?..??????????
00000265D338B340 0000000000000AC4 0000009A00000000 ?...............
00000265D338B350 0000000000000000 900002010096C091 .........?......
00000265D338B360 9090000201009690 9690900002010096 ................
00000265D338B370 0096909000020100 0100969090000201 ................
00000265D338B380 DC03329490900001 9200C0AACD923C00 .....2.?.<.???..
00000265D338B390 BEAACD9200C1AACD 9200009200009200 ???..???........
00000265D338B3A0 0000920000920000 0092000092000092 ................
00000265D338B3B0 9200009200009200 0000920000920000 ................
00000265D338B3C0 0092000092000092 9200009200009200 ................
00000265D338B3D0 0000920000920000 0092000092000092 ................
00000265D338B3E0 9200009200009200 0000920000920000 ................
00000265D338B3F0 0092000092000092 9200009200009200 ................
00000265D338B400 0000920000920000 0092000092000092 ................
00000265D338B410 9200009200009200 0000920000920000 ................
00000265D338B420 0092000092000092 9200009200009200 ................
```

```
0:000> !do 00000265d338b288
Name:         System.Byte[]
MethodTable: 00007ffb27f42790
EEClass:      00007ffb27f42710
Tracked Type: false
Size:         489(0x1e9) bytes
Array:        Rank 1, Number of elements 465, Type Byte (Print Array)
```

Figure 1: Example of a byte array.

At the beginning of the array is information about the object type $t(O_{b_1}) = 00007ffb27f42790$. The next element is information about the number of elements $C(O_{b_1}) = 0x1D1 = 465\,elements$. This information is sufficient for the Common Language Runtime (CLR) to operate on an array-type object.

According to Fig. 1, in a randomly selected byte array, there are many repeated byte values. This trend was observed in a selective analysis of other arrays with a length greater than 200 elements.

5. The method for estimating the amount of RAM reduction

We have a set of objects O_{time} (containing all objects o) and a set of types T_{time} (all types t), which are reflected in a memory snapshot taken at time *time*.

Considering that each object o is an instance of type t, let us introduce a grouping function G, which groups the elements of the input set of objects O by type:

$$G(O): O \rightarrow T. \qquad (2)$$

As a result of the grouping operation, we obtain subsets O_t, which contain all objects of type t. Let's separate the set for type

$$t = System.Byte[\,] \rightarrow O_{System.Byte[\,]}. \tag{3}$$

Given the specifics of data compression algorithms, let's separate arrays with a small number of elements, in this case less than 200:

$$O'_{t(System.Byte[\,])} \subset O_{t(System.Byte[\,])}, \tag{4}$$

and

$$C\left(O'_{t(System.Byte[\,])}\right) \subseteq (200, \infty). \tag{5}$$

Let's introduce the *Zip* compression function, which allows us to preserve the information content INFO of the array using a smaller array:

$$Zip\left(O_{t(System.Byte[\,])}\right) \rightarrow O^{Zip}_{t(System.Byte[\,])}. \tag{6}$$

For informational content

$$INFO\left(O_{t(System.Byte[\,])}\right) = INFO\left(O^{Zip}_{t(System.Byte[\,])}\right) \tag{7}$$

and the number of elements

$$C\left(O_{t(System.Byte[\,])}\right) \geq C\left(O^{Zip}_{t(System.Byte[\,])}\right). \tag{8}$$

This gave us an estimate of the amount of RAM reduction when storing information in objects of type t = System.Byte[]

$$Ratio = \frac{C\left(Zip\left(O'_{t(System.Byte[\,])}\right)\right)}{C\left(O'_{t(System.Byte[\,])}\right)}. \tag{9}$$

The developed method for estimating the amount of RAM reduction is implemented as a software application using .NET and the ClrMD library.

6. Calculation experiment: Application of compression algorithms

To apply and analyse the effectiveness of the proposed approach, a snapshot of an industrial-scale system memory of approximately 100 GB was considered. This memory snapshot contains 4.9 million byte arrays, which collectively contain 2.3 billion bytes. Comparable data footprints are common in industrial energy systems, where compact representations lower runtime and hosting costs without sacrificing observability [9].

Figures 2 and 3 show the 20 most popular array lengths. The most frequent arrays are those with a length of 10 elements.

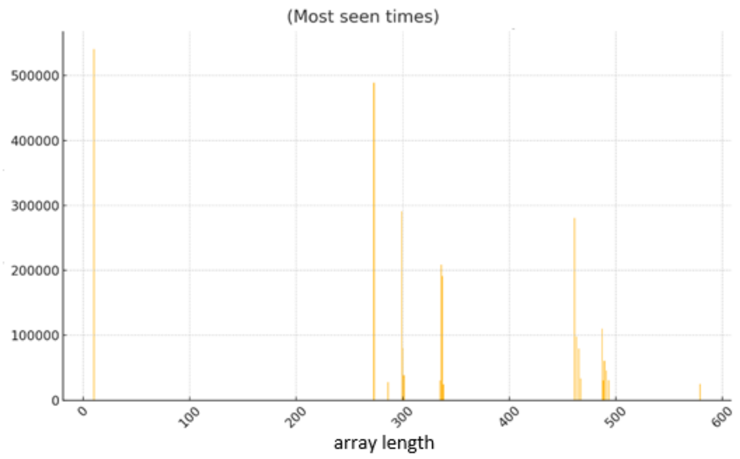


Figure 2: Distribution of occurrences by array length.

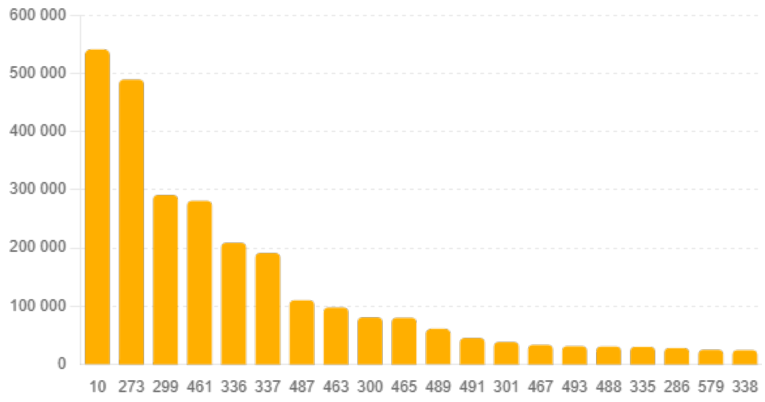


Figure 3: Distribution of occurrences by array length.

The system under study contains data about user segments encoded by an array of ten values. All other sizes do not have clear links to the data and are mostly compact representations of user data in binary form.

Figures 4 and 5 show the 20 most popular array lengths according to the amount of memory occupied. Despite the largest number of arrays with a length of 10 elements, they are not the largest consumers of RAM. Arrays with a length of 273 elements occupy approximately 27 times more memory, and arrays with a length of 461 elements occupy approximately the same amount.

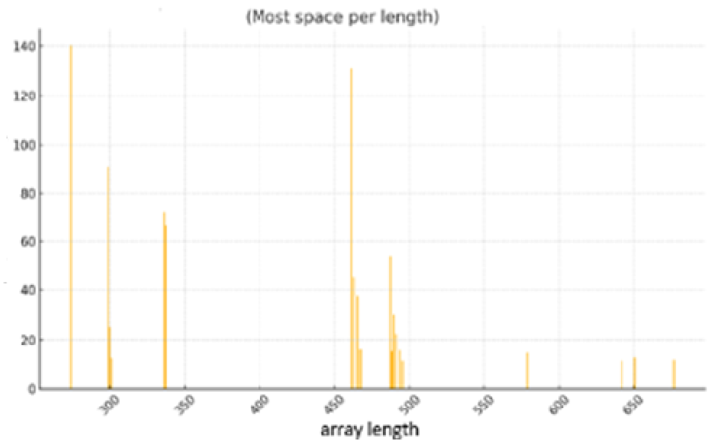


Figure 4: Distribution of memory usage by array length.

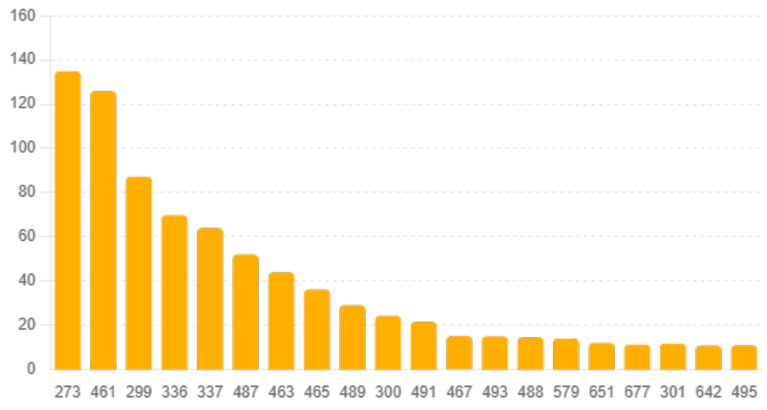


Figure 5: Distribution of memory usage by array length.

Fig. 6 shows the 20 largest arrays. It is worth noting that the lengths of the arrays coincide with powers of two. Arrays of this length are used in array reuse algorithms (Object Pooling) [20] when receiving input requests from the network and decoding information. Also, when reading files, buffers with a standard growth corresponding to the power of two are used.

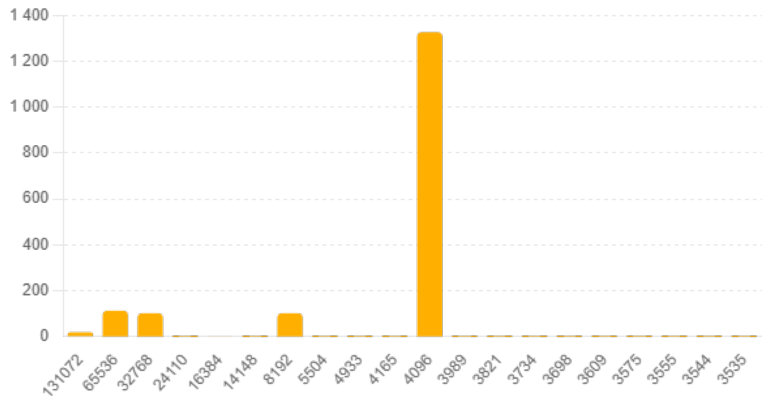


Figure 6: Distribution of the largest arrays.

7. Analysis of results of Calculation experiment

The data obtained indicates a large number of objects representing information about approximately one million users. When the system is running, user information is loaded into memory upon the first request. After a period of user inactivity, this information is transformed into a byte array to reduce memory usage and the number of objects.

The size of the binary representation varies depending on the amount of user information available.

Using the compression function for this information made it possible to reduce the amount of memory by 0.95 GB, which is 45% of the total volume occupied by the byte array type. This dovetails with description-minimization results reported for static technological objects, where concise encodings retain decision-useful content while shrinking memory budgets [7].

A series of computational experiments was conducted on systems from other domains, and similar data distribution patterns were found. Arrays with sizes not divisible by powers of two reflect real data. Data can appear in the process memory from several sources:

• Request from another system. In this case, objects are transmitted in binary format. Information transfer protocols usually include compression at the transfer protocol level.

• Data collection by the system. In this case, the system chooses the data storage format and data compression algorithms may be applied.

• Reading from the file system. In this case, compression may be built into the file format.

8. Conclusions

Most publications on the subject focus on the problem of memory leaks, bypassing excessive consumption due to the impossibility of diagnosing it. Research on the use of design patterns to reduce memory usage cites implementation mechanisms and effects, but leaves aside the criteria and mechanics of candidate search due to the high variability of scenarios and software products.

This publication focuses specifically on the algorithm for searching for candidates using memory snapshots. Although memory snapshots are used to search for memory leaks or malicious code, their use to search for excessive memory consumption is a recent development.

For objects of the 'byte array' type, a metric was created that provides a qualitative assessment of the implementation of compression algorithms based on a process memory snapshot. During the study, sets with sizes matching the lengths of arrays in object pools and buffer sizes when reading files were separated.

Broadly, our findings are consistent with cross-domain evidence that structure-aware representations deliver outsized efficiency gains relative to naïve scaling [3], [7]. According to the assumption made, the best effect of implementing compression algorithms is achieved for arrays belonging to data collected by the system. This assumption was confirmed by a series of experiments with memory snapshots of the industrial systems under study. Reducing the minimum array size threshold does not significantly reduce memory usage, but it does require compression operations to be performed in memory. Related optimization case studies in electrical drives and power electronics underscore the transferability of representation-driven efficiency gains to production environments [8], [10], [11], [12], [13].

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] B. Gregg, *Scaling solutions*, in: *Systems Performance* (2nd ed.), Addison-Wesley, Boston, MA, 2021, Sec. 2.7.3, p. 929.

[2] J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[3] T. Serdiuk, et al., Mathematical model of dynamics of homomorphic objects, in: *CEUR Workshop Proceedings*, vol. 2516, 2019, pp. 190–205. URL: https://ceur-ws.org/Vol-2516/paper22.pdf.

[4] G. Novark, et al., Efficiently and precisely locating memory leaks and bloat, in: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, Dublin, 2009, pp. 1–14. doi:10.1145/1542476.1542507.

[5] M. Weninger, Analyzing data structure growth over time to facilitate memory leak detection, in: *Proceedings of the 27th ACM Conference on Systems and Programming Languages*, Association for Computing Machinery, Mumbai, 2019. doi:10.1145/3368089.9781450362399.

[6] V. V. Hnatushenko, et al., Homomorphic filtering in digital multichannel image processing, *Naukovyi Visnyk Natsionalnoho Hirnychoho Universytetu* (3) (2023) 118–124. doi:10.33271/nvngu/2023-3/118.

[7] M. Tryputen, et al., Minimization of the description of images in the problem of adaptive control of static technological objects, in: *20th IEEE International Conference on Modern Electrical and Energy Systems (MEES 2021)*, 2021. doi:10.1109/MEES52427.2021.9598651.

[8] V. K. Tytiuk, et al., Online-identification of electromagnetic parameters of an induction motor, *Energetika* 63 (5) (2020) 423–440. doi:10.21122/1029-7448-2020-63-5-423-440.

[9] V. Kuznetsov, et al., Modeling of thermal process in the energy system “Electrical network–asynchronous motor”, *E3S Web of Conferences* 280 (2021) 05003. doi:10.1051/e3sconf/202128005003.

[10] A. V. Dymerets, et al., Dynamic characteristics of zero-current-switching quasi-resonant buck converter under variation of resonant circuit and load parameters, in: *IEEE International Conference on Electronics and Nanotechnology (ELNANO 2020)*, 2020, pp. 848–853. doi:10.1109/ELNANO50318.2020.9088879.

[11] V. Busher, et al., Optimal control method of high-voltage frequency converters with damaged cells, *IOP Conference Series: Materials Science and Engineering* 985 (1) (2020) 012021. doi:10.1088/1757-899X/985/1/012021.

[12] V. Tytiuk, et al., Control of the start of high-powered electric drives with the optimization in terms of energy efficiency, *Naukovyi Visnyk Natsionalnoho Hirnychoho Universytetu* (5) (2020) 101–108. doi:10.33271/NVNGU/2020-5/101.

[13] V. Busher, et al., Advanced space vector modulation with “fractional” power cells, *Results in Engineering* 26 (2025) 105415. doi:10.1016/j.rineng.2025.105415.

[14] K. Nguyen, et al., Understanding and combating memory bloat in managed environments, *ACM Transactions on Software Engineering and Methodology* 26 (4) (2018). doi:10.1145/3162626.

[15] S. Lee, *Mitigating the performance impact of memory bloat*, Ph.D. Dissertation, Georgia Institute of Technology, 2015. URL: http://hdl.handle.net/1853/56174.

[16] Q. Xia, et al., Hardware-accelerated kernel-space memory compression using Intel QAT, *IEEE Computer Architecture Letters* 24 (1) (2025) 57–60. doi:10.1109/LCA.2025.3534831.

[17] P.-A. Tsai, et al., Compress objects, not cache lines: An object-based compressed memory hierarchy, in: *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*, ACM, 2019, pp. 229–242. doi:10.1145/3297858.330400.

[18] N. Y. Mitikov, et al., Detection of software issues based on memory dump analysis, *Problems of Applied Mathematics and Mathematical Modeling* (18) (2023) 171–178. doi:10.15421/322318.

[19] C. Lou, et al., Chintalapati, RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure, in: M. K. Aguilera, H. Weatherspoon (Eds.), *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’22)*, USENIX Association, 2022, pp. 109–125. ISBN 978-1-939133-28-1.

[20] I. T. Christou, S. Efremidis, To pool or not to pool? Revisiting an old pattern, Athens Information Technology, Marousi, 2018. arXiv:1801.03763.