DOI: https://doi.org/10.15276/hait.07.2024.10
UDC 004.942

Modeling and automation of the process for detecting duplicate objects in memory snapshots

Nikolay Y. Mitikov[1)]
ORCID: https://orcid.org/0009-0002-1297-5676; mitikov.m22@fpm.dnu.edu.ua. Scopus Author ID: 59005016500
Natalia A. Guk[1)]
ORCID: https://orcid.org/0000-0001-7937-1039; huk_n@fpm.dnu.edu.ua. Scopus Author ID: 54791066900
[1)] Oles Honchar Dnipro National University, 72, Science Avenue. Dnipro, 49010, Ukraine

ABSTRACT

The paper is devoted to the problem of detecting increased memory usage by software applications. The modern software development cycle is focused on functionality and often overlooks aspects of optimal resource utilization. Limited physical scalability sets an upper limit on the system's capacity to handle requests. The presence of immutable objects with identical information indicates increased memory consumption. Avoiding duplicates of objects in memory allows for more rational use of existing resources and increases the volumes of processed information. Existing scientific publications focus on investigating memory leaks, limiting attention to excessive memory use due to the lack of a unified model for finding excessive memory use. It should be noted that existing programming patterns include the “object pool” pattern, but leave the decision on its implementation to engineers without providing mathematical grounding. This paper presents the development of a mathematical model for the process of detecting duplicates of immutable String type objects in a memory snapshot. Industrial systems that require hundreds of gigabytes of random-access memory to operate and contain millions of objects in memory have been analyzed. At such data scales, there is a need to optimize specifically the process of finding duplicates. The research method is the analysis of memory snapshots of high-load systems using software code developed on.NET technology and the ClrMD library. A memory snapshot reflects the state of the process under investigation at a particular moment in time, containing all objects, threads, and operations being performed. The ClrMD library allows programmatic exploration of objects, their types, obtaining field values, and constructing graphs of relationships between objects. The series of experiments was conducted on Windows-backed machines, although similar results can be obtained on Linux thanks to cross-platform object memory layout pattern. The results of the study proposed an optimization that allows speeding up the process of finding duplicates several times. The scientific contribution of the research lies in the creation of a mathematically substantiated approach that significantly reduces memory resource use and optimizes computational processes. The practical utility of the model is confirmed by the optimization results achieved thanks to the obtained recommendations, reducing hosting costs (which provides greater economic efficiency in the deployment and use of software systems in industrial conditions), and increasing the volumes of processed data.

Keywords: Optimization; algorithm; performance; memory snapshot; duplication; string

***For citation*: Mitikov N. Y., Guk N. A. “Modeling and automation of the process for detecting duplicate objects in memory snapshots”. *Herald of Advanced Information Technology.* 2024; Vol. 7 No. 2: 147–157. DOI: https://doi.org/10.15276/hait.07.2024.10**

INTRODUCTION

The rapid development of information technologies facilitates the comprehensive application of software applications in many areas. The need to process significant volumes of information can be compensated by increasing the computational power of computing systems [1] or optimizing existing software implementations [2]. Considering the linear growth in the cost per unit of RAM in cloud hosting [3], there is a false impression of the possibility of linear scaling of total operational expenses. Available virtual machine options come with RAM sizes in multiples of powers of two – 4GB, 8GB, 16GB, 32GB, etc.

If there is a need for just one extra gigabyte beyond the available RAM, it becomes necessary to scale the computing system to the next tier, which is at least twice as expensive as the previous one (considering the amount of processing power). Optimizing the memory usage of a software application requires a deep understanding of the system's operating algorithms [20] and access to the development process, which also makes this method complex and economically unjustifiable given the option to scale the computing system. Scaling computational capacities is cost-effective at small sizes but becomes progressively less economically viable with each subsequent increase in the memory size of the virtual machine.

It is worth noting that modern development methodologies are based on having multiple environments for testing before releasing a software application into public access [4] and having a

© Mitikov N., Guk N., 2024

This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/deed.uk)

secondary replica/region [5] to enhance availability. Thus, moving to the next level of virtual machine power significantly increases the overall cost of operating expenses.

Existing publications [6], [7] mostly focus on investigating the problem of memory leaks, paying less attention to excessive memory usage. A memory leak is a condition where a program mistakenly fails to release no longer needed memory for reuse [22], thus continuing to allocate new memory blocks.

Fig. 1 illustrates the rapid increase in memory usage by a software application, leading to decreased performance and necessitating a system monitoring restart of the application every few hours. High-load systems working with large volumes of data are particularly susceptible to this problem. Memory snapshots [8], which record the contents of RAM at a specific moment in time, are used to identify the root causes of memory leaks.

Memory snapshots are also used to detect malicious code [9], especially in cases of code injection into a trusted process. While there is a focus on identifying leaks, existing research tends to neglect the issue of excessive memory usage, particularly the situation where an application uses more memory than required for the task at hand. This problem can arise for various reasons, but one of the most common is the duplication of immutable objects. If a program creates multiple objects with identical values instead of using a single immutable object, this leads to the storage of redundant information and increases memory requirements.

A common issue in the operation of modern systems is the duplication of data [24], specifically String type objects (strings), which poses a challenge as it increases the overall cost of operational expenses and remains undetected by existing diagnostic systems since the criterion for a memory leak does not apply.

Finding duplicates requires comparing all strings with each other, which would involve a vast number of operations and negatively impact the performance of the application.

LITERATURE REVIEW

Visualization of cloud computing machine costs [3], depending on the size of RAM, shows a minimum doubling in costs when the memory capacity of the virtual machine is exhausted, necessitating the use of the next size of virtual machine (Fig. 2).

Researchers Adriaan Labuschagne and Laura Inozemtseva in their work [4] on regression testing have clearly demonstrated methods for enhancing the quality of software products. Testing the product in environments similar to the real world significantly reduces the risk of errors in application functionality [10], but requires specialized testing environments, thereby increasing the number of virtual machines.

Significant attention is paid to the mechanisms of memory leak detection in the works of Gene Novark, Emery D. Berger, Benjamin G. [6], and Markus Weninger [7]. Jon Louis Bentley [2] describes approaches to reducing memory usage, but the mechanisms for finding candidates for optimization are outside the scope of his work.

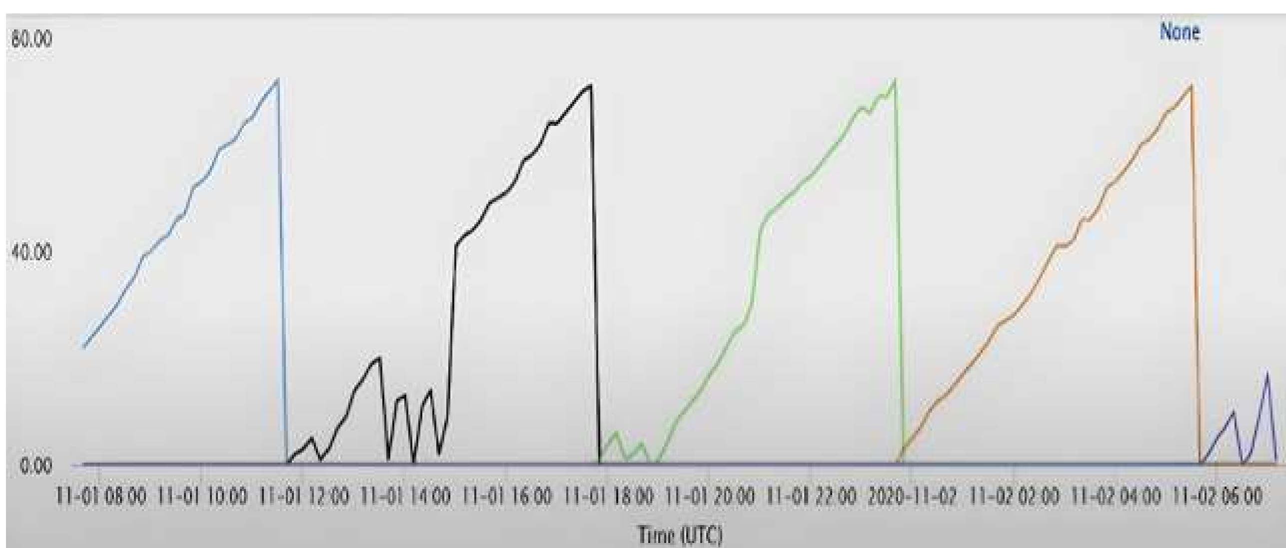


Fig. 1. **Memory usage dynamics**
Source: **monitoring system supervised by the authors**

Ioannis T. Christou and Sofoklis Efremidis [11] describe the use of the “object pool” design pattern. According to their research, using this design pattern can significantly reduce the response time of high-performance multi-threaded applications, especially in environments with limited memory. However, the mechanism for finding appropriate uses in their work is not described.

The authors of these studies agree that the applicability of this pattern depends on the specific software product and the data it operates. Excessive memory use can be an issue for high load systems. However, with an understanding of the causes of suboptimal memory distribution [19] and the use of appropriate strategies to avoid them, the stability and efficiency of system operations can be enhanced.

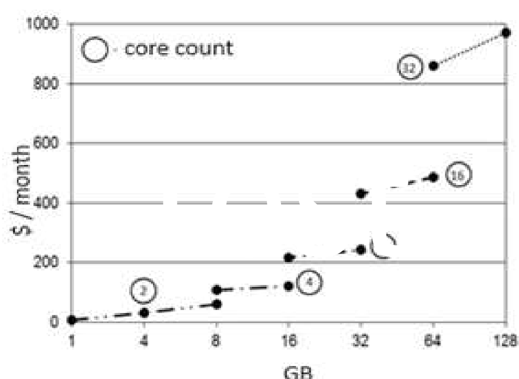


Fig. 2. **Hosting cost based on the RAM amount**

Source: **compiled by the authors**

One possible scenario for hosting development [12], [13] is the implementation of dynamic cost for using a virtual machine based on the percentage of resource usage, due to the nonlinear increase in energy consumption when the load exceeds 80% of the system's capacity.

The scenario of excessive memory usage by a software application [21] is difficult to detect since it does not manifest overtly – the memory leak criterion is not met, specifically, there is no monotonic increase in memory usage by the application over time.

Data duplication appears when diverse data streams enter the system [25], which may contain identical values, such as reading data from a database where the query results contain identical values. A vivid example of potential information duplication is relational databases with queries that join several tables.

To detect data duplication, it is necessary to analyze the program's memory and identify objects that have the property of immutability [14]. The property of immutability can be set at the programming language level [15], or manifest in existing code due to the absence of method calls that can change the state of the object.

Finding the property of immutability [16] is key to reducing memory costs through the reuse of identical objects [17], and requires access to the complete codebase of the running program.

Memory snapshots are extensively utilized for analyzing and detecting malicious code that may be introduced into the memory of a trusted process. Traditional static and dynamic methods prove ineffective in identifying malware residing in memory [18]. Furthermore, existing solutions employing forensic memory analysis exhibit unsatisfactory efficiency in terms of detection speed and rely heavily on extensive expertise in memory analysis. A solution to this problem involves capturing memory snapshots from healthy processes to serve as a training base for machine learning algorithms. Upon detecting anomalies, the security system is capable of responding swiftly despite the absence of a physical threat file.

Analysis indicates that contemporary research focuses heavily on identifying memory leaks, yet there are no effective methods for addressing excessive memory usage in high-load systems. Concurrently, existing studies converge on the importance of optimal memory utilization to enhance application performance. Therefore, the development of mathematical models and methods for detecting excessive memory use remains a pressing issue.

PROBLEM STATEMENT

In the domain of software development, particularly in environments where high-load systems are prevalent, the efficient utilization of memory resources [23] is crucial for maintaining optimal performance and reducing operational costs. The challenge of detecting excessive memory usage in such systems is compounded by the complexities involved in managing and understanding memory dynamics, which include the presence of immutable object duplicates that consume unnecessary memory space.

The problem of excessive memory detection in software products can be articulated as the development of a robust quantitative model that effectively identifies and quantifies duplicate immutable objects within the memory of these systems. This model should enable developers to assess the amount of memory wastefully consumed by these duplicates, thereby informing strategies to

minimize memory usage without compromising system performance.

The proposed model will utilize advanced memory snapshot analysis techniques to accurately detect these duplicates, incorporating a set of metrics designed to expedite the identification process. Although, the article focuses on single scenario, further research shall be built on top of the foundation laid. The effectiveness of this model will be evaluated through computational experiments within industrial systems, aiming to provide a scalable and reliable solution to the excessive memory usage problem. This approach is analogous to the challenge in machine learning of evaluating model quality, where a quantitative metric based on current standards and trends is crucial for selecting the optimal model for specific tasks. The goal is to develop a methodological framework that not only addresses the immediate concerns of memory waste but also contributes to the broader discourse on resource efficiency in software development.

PURPOSE AND OBJECTIVES OF THE STUDY

The aim of this work is to optimize memory usage by developing a new mathematical model for identifying duplicate String objects in the memory of software products. This will be achieved by analyzing memory snapshots and evaluating the model's effectiveness. Detecting duplicates with immutability properties allows for an accurate assessment of the amount of excessively used memory and reduces its consumption in subsequent development cycles.

To achieve this goal, the following objectives were set:

1. Construct a mathematical model of objects in memory.
2. Isolate String type objects with immutability properties into a set.
3. Select a supplementary metric to accelerate the search for duplicates.
4. Develop a method for searching for duplicates.
5. Apply the proposed approach to an existing industrial system and perform an analysis of the computational experiment results.

MATERIALS AND METHODS OF RESEARCH

1. Overview of modern development practices focus in software engineering

To ensure the stable operation of high-load systems, it is critically important to have sufficient computational resources. Due to the high complexity of industrial systems and significant variability of data, the sizes of computational capacities are calculated based on load testing that exceeds expected levels. As a result of these tests, requirements are formed for the amount of RAM and computational power needed for the software product to function.

Errors identified during program execution and code deficiencies are subject to refinement. After successfully passing key test scenarios, the software product is ready to be launched for end-users. However, the development cycle often does not account for the possibility of increased resource usage, which leads to an increase in computational demands, raises operating costs, and limits the number of requests the system can handle according to requirements. It is necessary to optimize the memory usage of software applications to improve performance and increase the potential number of operations performed simultaneously.

2. Memory snapshots applicability

Within a memory snapshot, information is displayed about the operations being performed, the state of execution threads, the objects managed by these threads, and the program code. Details about the objects include their types, states, and relationships.

A memory snapshot of a program can be represented as a structured array of bytes in the following way:

$$M_{time} = [b_1, b_2, \ldots b_N],$$

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

Considering the presence of a large number of objects in the memory of industrial software applications, manually searching for duplicates among thousands of different object types in a memory snapshot is not feasible. It is necessary to limit the search to types that occupy the most memory space, in most cases, these are objects of type String.

The creation, modification, and deletion of String type objects require certain resources. Fig. 3 shows an example of object duplication from the studied industrial system, which consumes over 20GB of memory. This software application does not exhibit signs of memory leakage, namely the monotonic increase in used memory, so existing monitoring systems do not detect any memory usage problems in it.

```
00181696ce700 (Model.CountryToCountrySetting)
 <ObjectState>k__BackingField:0x0 (Undefined) (Data.CacheObjectStates)
 <SourceCountryId>k__BackingField:0xe2 (System.Int64)
 <TargetCountryId>k__BackingField:0x20 (System.Int64)
 <SettingName>k__BackingField:00000181696ce740 (System.String) Length=34, String="AddShippingCostToCommercialInvoice"
 <SettingValue>k__BackingField:00000181696ce7a0 (System.String) Length=6, String="5;;USD"
 <SettingDescription>k__BackingField:00000181696ce7c8 (System.String) Length=111, String="Value for Commercial and Par
00181696ce8c0 (Model.CountryToCountrySetting)
 <ObjectState>k__BackingField:0x0 (Undefined) (Data.CacheObjectStates)
 <SourceCountryId>k__BackingField:0xe2 (System.Int64)
 <TargetCountryId>k__BackingField:0x21 (System.Int64)
 <SettingName>k__BackingField:00000181696ce900 (System.String) Length=34, String="AddShippingCostToCommercialInvoice"
 <SettingValue>k__BackingField:00000181696ce960 (System.String) Length=6, String="5;;USD"
 <SettingDescription>k__BackingField:00000181696ce988 (System.String) Length=111, String="Value for Commercial and Par
00181696cea80 (Model.CountryToCountrySetting)
 <ObjectState>k__BackingField:0x0 (Undefined) (Data.CacheObjectStates)
 <SourceCountryId>k__BackingField:0xe2 (System.Int64)
 <TargetCountryId>k__BackingField:0x22 (System.Int64)
 <SettingName>k__BackingField:00000181696ceac0 (System.String) Length=34, String="AddShippingCostToCommercialInvoice"
 <SettingValue>k__BackingField:00000181696ceb20 (System.String) Length=6, String="5;;USD"
 <SettingDescription>k__BackingField:00000181696ceb48 (System.String) Length=111, String="Value for Commercial and Par
```

Fig. 3. **String duplicates example**
Source: **compiled by the authors**

Thanks to the memory snapshot, duplicates of objects with the property of immutability were found, leading to the identification of increased memory usage. To understand the mechanism of the software application and diagnose problems, memory snapshots are scanned [8].

In the case of using high-load systems, the process of scanning memory snapshots requires automation, which allows for rapid analysis. Therefore, the automation of scanning provides significant advantages, including reducing the time for analysis, increasing the accuracy of problem detection, and improving the overall performance of the system [17].

3. Algorithm for segregating objects

In this work, it is proposed to identify objects in a memory snapshot based on their properties and to use this as a metric. The metric is introduced to define the "distance" between objects and their similarity.

An automated scanning system during a memory snapshot will create a set of objects O. The next step is to separate the types that possess the property of immutability, specifically those that do not change their state after creation. We will apply a function to group by types and find the set O_t of objects with immutable types. We will consider this property only for String type objects due to their statistically highest occurrence in the memory snapshots of the studied systems. In this set, we will identify the subset STRN of all String type objects and the subset STR of objects that possess the property of immutability$STR \subset STRN \subset O_{t_i} \subset O$.

Under the directive of the software application's CLR (Common Language Runtime) execution environment, objects of type String have a zero byte *b(0),* the length of the object in bytes *b*, and the content information located in these bytes. A String type object can be represented as *String(b(0), b).* Introduce an additional property s of the object, which is the arithmetic sum of the bytes allocated for storing the object's content. Thus, the expression for a String type object can be written as $String(b(0), b, s)$.

Each instance of the object $String(b(0), b, s)$ belongs to the set:

$$STRN = \{String_1(b_1(0), b_1, s_1), String_2(b_2(0), b_2, s_2), \ldots . String_N(b_N(0), b_N, s_N)\},$$

where N is the number of instances of String type objects. This set includes all String type objects, including duplicates.

Each instance of a String object is identified by the parameter s.

The values of the parameters s_i for each of the objects form the set

$$S=\{s_1, s_2, \ldots, s_n\},$$

where n is the number of unique sizes of String type objects, $n<=N$.

To optimize the grouping of strings, we will develop an approach that allows for quick segregation of strings based on the introduced metric, thus significantly reducing the number of comparisons between strings, leaving only comparisons between elements within one group.

In the set S, we will find the minimum and maximum values, denoted as and b, respectively. The range [a, b] will be divided into m intervals Δ of length $\Delta= \frac{(b-a)}{m}, m \le n$. Each interval may include several values of the parameter s of String type objects, or none at all.

The distribution of the parameter s across intervals is done by aligning subsets:

$$k = \prod_{i=1}^{N}\left(\frac{s_i}{\Delta}\right), k \le n, \quad (1)$$

where k is the interval number.

Thus, a partial ordering of the set S by intervals occurs. It should be noted that the obtained intervals may contain strings with the same value of s, but differing in content. A clear example would be strings in which the order of words is changed, but the algebraic sum of bytes allocated for their storage remains the same. The sum function was chosen as a quick hash function. Among the set of intervals, there will be those that contain several unordered values. Additional ordering should be conducted within such intervals. Therefore, the task arises to perform the minimum number of such orderings. Let us denote by the function G (m, L) the total number of orderings and find the minimum G (m, L).

Let us assume that each interval contains the same number of numbers, L.

In this case, the total number of orderings equals:

$$G(m,L) = mL\frac{L-1}{2}.$$

It is necessary to determine the values of m and L for which the function G (m, L) assumes the smallest value.

Since $m = \frac{n}{L}$, we have:

$$G(n,L) = n\frac{L-1}{2}. \quad (2)$$

The minimum of the function G(m,L) will depend on the value of parameter L. The trivial case L=1 is not considered because sorting operations require at least two values; therefore, the minimum value of $(m,)$ is achieved for L=2:

$$min\, G(n,L) = \frac{n}{2}. \quad (3)$$

With this value of L, the number of intervals will be $m = \frac{n}{2}$.

Since it is not known in advance how to divide the set S into a number of unequal intervals such that these intervals contain an equal number of numbers, we introduce the estimate $inf\ G(n,L) = \frac{n}{2}$

The upper bound of values for the function G is found when L=n and m=1 $sup\ G(n,L) = n\frac{n-1}{2}$

Let's define the effectiveness of the proposed method in the study by comparing the number of operations required with that of a full exhaustive search. A full exhaustive search involves pairwise comparisons of all strings with each other. In this case, the total number of comparisons is $n\frac{n-1}{2}$

Let's introduce an efficiency coefficient K_{ef} for the extreme cases $inf\ G(n,L)$ and $sup\, G(n,L)$, considering the number of operations for distribution into intervals and the number of operations required to determine the values a and b:

$$K_{\text{eф}}^{inf} = \frac{n\frac{n-1}{2}}{\frac{n}{2}+2n} = \frac{n-1}{5},$$

$$K_{\text{eф}}^{sup} = \frac{n\frac{n-1}{2}}{n\frac{n-1}{2}+2n} = \frac{n-1}{n+3}.$$

Thus, the efficiency coefficient of the proposed method lies within the range of

$$\frac{n-1}{n+3} \le K_{\text{eф}} \le \frac{n-1}{5}.$$

The maximum value of the efficiency coefficient indicates a linear dependency of efficiency on the number of processed elements, demonstrating that with each additional element, processing costs increase linearly. This suggests that the overhead for handling each additional string grows in a linear fashion, which is typical when the number of operations scales directly with the size of the input.

The minimum value approximates to constant complexity at large n, showing that the efficiency of the algorithm increases and approaches a constant value as n increases. This implies that as the dataset grows; the relative cost of processing per element diminishes, potentially leading to a more efficient use of resources at scale.

However, the efficiency coefficient of the method does not account for the increase in the number of additional resources required to create sets as the number of elements and the number of sets increase. This can include memory for storing

the intervals and the computational overhead associated with managing larger sets. This additional cost could offset some of the gains from the reduced number of comparisons, especially in cases where the distribution of elements across intervals is uneven or requires frequent reevaluation.

In practical applications, it would be necessary to also consider these factors to obtain a holistic view of the method's performance and to optimize the parameters m and L not just for theoretical efficiency but also taking into account practical resource utilization and operational constraints.

EXPERIMENT SETUP

1. Enterprise system characteristics

The validation of the proposed approach was carried out on a memory snapshot of an industrial system, which occupies more than 20 GB of operational memory under normal load conditions.

The system is responsible for transactional data processing, handling hundreds of tax-related, product stock, and delivery enquiries in real time. A distribution of String type objects was constructed based on the metric *s*.

On the graph (Fig. 4), there are three peaks that demonstrate a significant increase in the number of elements with the same checksum value *s*, corresponding to three String type objects: “*5;USD*”,*AddShippingCostToCommercialInvoice*”, “*Value for Commercial and Parcel Invoice*”. According to the developed approach, each peak on the graph corresponds to one *s* value and uniquely identifies an object. This allows for rapid analysis and quantitative identification of duplicates.

The grouping by the *s* parameter confirmed the presence of 220 thousand unique values and 14.5 million duplicates. The most duplicated strings (Fig. 4) were identified, confirming the initial assumption about the presence of abnormal duplication of objects in the memory snapshot. A set s^1 of *s* parameters of objects identified in the previous step will be formed as $s^1=\{s_k^1, s_k^2, s_k^3\}$. Set s^1 belong to *S*, meaning $s^1 \subset S$.

The resulting set will contain String type objects that have not yet been processed by the algorithm. A significant number of duplicates were found in this set.

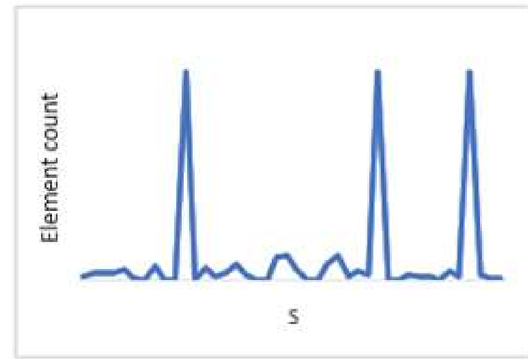


Fig. 4. **Distribution Across Set S**
Source: **compiled by the authors**

By removing the set s^1 from set *S*, we obtain a new set $S^1=S\backslash s^1$ (Fig. 5) and will construct a distribution for the set S^1.

Fig. 5. **Distribution across the set S`**
Source: **compiled by the authors**

In (Table 1), examples of strings with the highest number of repetitions in memory are provided, highlighting both their duplication count and the total amount of memory they collectively consume. For example, the string True appears 635,181 times, occupying a total of 2.4 MB of memory.

Table 1. **Duplicates count and space taken**

String value	Duplicates	Total size
True	635181	2.4 MB
HideStandardShipping Method	390073	9.7 MB
false	318273	1.5 MB
Restricted due to ticket	257600	8.4 MB
Visa	241913	945 KB

Source: **compiled by the authors**

Table 2 illustrates how the time it takes to find duplicates varies depending on the number of sets used in the process. Each row in the table represents a different configuration of sets, detailing the maximum number of elements per set, the size of the

intervals (△), and the execution time in milliseconds (ms) for the duplicate search operation.

Fig. 6 illustrates how the maximum number of elements within set changes as the number of sets increases. It provides a visual representation showing that as the number of sets increases, the maximum number of elements per set tends to decrease. This trend suggests that dividing the data into more sets leads to smaller groups or subsets, thus distributing the data more finely across the sets.

Table 2. **Dependency of duplicate search time on the number of sets**

Set count	Max elements in set	Size △	Execution time, ms.
2	2160701	46795	3321.95
4	2142365	23398	2653.41
8	1753658	11699	3152.39
16	1715889	5850	3100.36
32	1558702	2925	2323.03
64	1260964	1463	2274.14
128	862321	732	2089.94
256	535165	366	1968.16
512	325051	183	1894.38
1024	262314	92	2253.57
2048	174401	46	2074.31
10000	91012	10	1708.93
20000	50420	5	1789.49
30000	40239	4	1831.46
40000	30639	3	1881.13
50000	20531	2	1994.7

Source: **compiled by the authors**

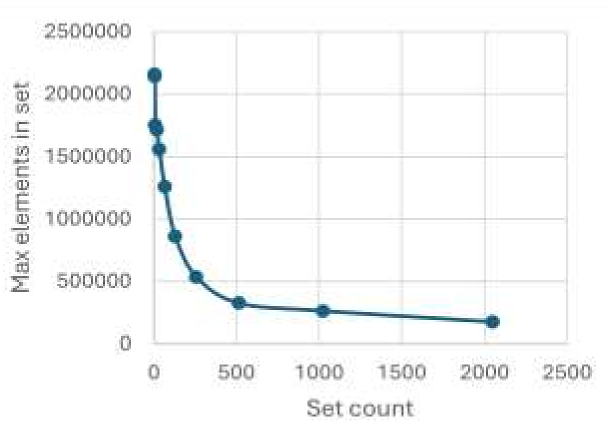


Fig. 6. **Dependency of the maximum number of elements on the number of sets**
Source: **compiled by the authors**

The Fig. 7 displays the relationship between the number of sets and the time taken (in milliseconds) to complete the partitioning operation. This chart would typically show trends where the operation time might decrease as the number of sets increases, indicating more efficient processing due to parallelization or more manageable data sizes per set. Conversely, it could also reveal points where operation time increases due to overhead associated with managing a larger number of sets, or where the distribution of data among sets is less than optimal, requiring adjustments in the partitioning logic.

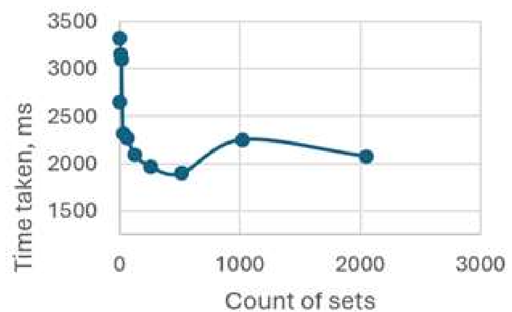


Fig. 7. **Dependency of the time (ms) of execution of partitioning on the number of sets**
Source: **compiled by the authors**

DISCUSSION OF THE RESULTS

Based on the assumption made, the best performance of partitioning is achieved when there are a similar number of elements in the subsets, which helps reduce the proportion of strings with the same checksum but different contents. This assumption was confirmed through a series of experiments with memory snapshots of the studied industrial systems, finding that the optimal number of subsets to accelerate partitioning falls within the range of 460 to 520. Increasing the number of subsets beyond this range does not significantly speed up the operation but does lead to the need to maintain a significantly larger number of groups in memory, thus increasing the memory requirements for conducting the analysis.

The series of experiments showed a dominant number of duplicates in strings up to 150 characters in length. This is explained by “constant” values that enter the system at the startup phase (for example, read from a database). Longer strings are usually formed dynamically and have a greater variability of data. Strings containing a large number of culture-specific characters, such as hieroglyphs, must also be considered separately. Each hieroglyph in UTF-8

has a large index/value, leading to a large sum of values and thus increasing the interval Δ.

Introducing these constraints allows for reducing the interval Δ and achieving a more even distribution of values across intervals.

CONCLUSIONS

Most publications on the topic of research focus on the problem of memory leaks, bypassing excessive consumption due to the inability to diagnose it. Studies on the application of design patterns to reduce memory usage present mechanisms of implementation and effect but leave aside the criteria and mechanics of searching for candidates due to the great variability of scenarios and software products. There is a justified need to pay attention to non-functional requirements, specifically the minimization of object duplicates in memory to increase performance and the volume of information with which the system can operate.

This work focuses specifically on the algorithm for searching for candidates using memory snapshots. While memory snapshots are used to search for memory leaks or introduced malicious code, their use to search for excessive memory consumption is a novel implementation.

A model of objects in the memory snapshot of the process has been developed, which allows obtaining the type and values of each object's fields. String type objects with the property of immutability were separated from the general set of objects. An improved method for searching for duplicates of String type objects in memory snapshots using the introduced metric has been developed.

The proposed approach was tested on several high-load industrial systems for which significant increases in used memory were detected. The conclusions obtained from the analysis of industrial systems were conveyed to the developers of the studied industrial systems. Based on these, the software code was modified, and in each case, the amount of RAM required by the process was significantly reduced.

A foundation has been laid for further search for mechanisms of memory preservation in industrial systems.

REFERENCES

1. Gregg, B. “2.7.3 scaling solutions”. “*Systems Performance, Second Edition.”* Boston: Addison-Wesley. 2021. ISBN-10 0136820158.

2. Bentley, J. L. Writing efficient programs, Englewood Cliffs, N.J.: Prentice-Hall. 1982. ISBN-10 013970244X.

3. “Microsoft. Windows virtual machines pricing”. – Available from: https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows. – [Accessed: 8, February 2023].

4. Labuschagne, A., Inozemtseva, L. & Holmes, R. “Measuring the cost of regression testing in practice”. *ESEC/FSE: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 2017, https://www.scopus.com/record/display.uri?eid=2-s2.0-85030776060&origin=resultslist.
DOI: https://doi.org/10.1145/3106237.3106288.

5. Gao, L., Dahlin, M., Nayate, A., Zheng, J. & Iyengar, A. “Improving availability and performance with application-specific data replication”. *IEEE Transactions on Knowledge and Data Engineering*. 2005; 17 (1): 106–120, https://www.scopus.com/inward/record.uri?eid=2-s2.0-17444420807&doi=10.1109%2fTKDE.2005.10&partnerID=40&md5=372f00d4ec430291228a21a2ede57ca9.
DOI: https://doi.org/10.1109/TKDE.2005.10.

6. Novark, G., Berger, E. D. & Zorn, B. G. “Efficiently and precisely locating memory leaks and bloat”. *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 2009, https://www.scopus.com/record/display.uri?eid=2-s2.0-70450250104&origin=resultslist.
DOI: https://doi.org/10.1145/1542476.1542521.

7. Weninger, M. “Analyzing data structure growth over time to facilitate memory leak detection» Mumbai”. *10th ACM/SPEC International Conference on Performance Engineering.* 2019, https://www.scopus.com/record/display.uri?eid=2-s2.0-85064630765&origin=resultslist.
DOI: https://doi.org/10.1145/3297663.3310297.

8. Shin, W., Kim, W. H., & Min, C. “Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot”. *European Conference on Computer Systems*. 2022; 17: 663–677.

9. Hamad, N., Dong, S., Olorunjube J. & Farhan, U. “Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification”. *Expert Systems with Applications*. 223, https://www.scopus.com/record/display.uri?eid=2-s2.0-85151039062&origin=resultslist. DOI: https://doi.org/10.1016/j.eswa.2023.119952.

10. Osherove, R. & Khorikov, V. “The art of unit testing, manning”. 2024. ISBN-10 1617297488.

11. Ioannis, S. E. & Christou, T. “To pool or not to pool? Revisiting an old pattern”. *Athens Information Technology*. 2018. DOI: https://doi.org/10.48550/arXiv.1801.03763.

12. Aldossary, M., Djemame, K., Alzamil, I. & Kostopoulos, A. ”Energy-aware cost prediction and pricing of virtual machines in cloud computing environments”. *Future Generation Computer Systems*. 2019. p. 442–459, https://www.scopus.com/record/display.uri?eid=2-s2.0-85056448412&origin=resultslist. DOI: https://doi.org/10.1016/j.future.2018.10.027.

13. Zhang, X., Wu, T., Chen, M., Wei, T. & Zhou, J.”Energy-aware virtual machine allocation for cloud with resource reservation”. *Journal of Systems and Software*. 2019. p. 147–161, https://www.scopus.com/record/display.uri?eid=2-s2.0-85055471140&origin=resultslist. DOI: https://doi.org/10.1016/j.jss.2018.09.084.

14. Flageol, W., Guéhéneuc, Y., Badri, M., Monnier, S., “Design pattern for reusing immutable methods in object-oriented languages”. *European Conference on Pattern Languages of Programs*. 2023; 28 (6): 1–9. DOI: https://doi.org/10.1145/3628034.3628040.

15. Haack, C., Poll, E., Schäfer, J. & Schubert, A. “Immutable objects for a java-like language”. 2007, https://www.scopus.com/record/display.uri?eid=2-s2.0-37149051628&origin=resultslist. DOI: https://doi.org/10.1007/978-3-540-71316-6_24.

16. Pengfei, S., Qingsen, W., Milind, C. & Xu, L. “Pinpointing performance inefficiencies in Java”. 2019, https://www.scopus.com/record/display.uri?eid=2-s2.0-85071907205&origin=resultslist. DOI: https://doi.org/10.1145/3338906.3338923.

17. Mitikov, N. & Guk, N. A. “Detection of software problems based on memory dump analysis”. *Applied Mathematics and Mathematical Modeling Issues*. 2023; 23: 171–178. DOI: https://doi.org/10.15421/32232301.

18. Liu, J., Feng, Y. & Liu, X. “MRm-DLDet: a memory-resident malware detection framework based on memory forensics and deep neural network”. *Cybersecurity*. 2023; 6 (21). DOI: https://doi.org/10.1186/s42400-023-00157-w.

19. Helm, C. & Kenjiro, T. “PerfMemPlus: A tool for automatic discovery of memory performance problems”. *34th International Conference on High Performance Computing*. 2019; 11501: 209–226, https://www.scopus.com/record/display.uri?eid=2-s2.0-85067495342&origin=resultslist. DOI: https://doi.org/10.1007/978-3-030-20656-7_11.

20. Bennour, I., Ettouil, M., Zarrouk, R. & Abderrazak J. “Study of runtime performance for Java-multithread PSO on multicore machines”. *International Journal of Computational Science and Engineering*. 2019; 19 (4): 483–493, https://www.scopus.com/record/display.uri?eid=2-s2.0-85072122618& origin=resultslist. DOI: https://doi.org/10.1504/IJCSE.2019.101881.

21. Xulong, T., Karakoy, M., Kandemir, M. T. & Arunachalam, M. “Co-optimizing memory-level parallelism and cache-level parallelism”. *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019; 40: 935–949, https://www.scopus.com/record/display.uri?eid=2-s2.0-85067638402&origin=resultslist. DOI: https://doi.org/10.1145/3314221.3314599.

22. Ryoo, J., Kandemir, M. T. & Karakoy, M. “Memory space recycling”. *Proceedings of the ACM on Measurement and Analysis of Computing Systems.* 2022; 6 (1): 14, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85125844851&doi=10.1145%2f3508034&partnerID=40&md5=081a08166101c225ab14a4915aa72a5f. DOI: https://doi.org/10.1145/3508034.

23. Kandemir, M., Tang, X., Kotra, J. & Karakoy, M. “Fine-granular computation and data layout reorganization for improving locality”. *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers.* 2022; art. no. 5, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85145665934&doi=10.1145%2f3508352.3549386&partnerID=40&md5=c94f97d17d306caa1051b0c5cf13fbc8. DOI: https://doi.org/10.1145/3508352.3549386.

24. Helm, C. & Taura, K. “Automatic identification and precise attribution of DRAM bandwidth contention”. *ACM International Conference Proceeding Series*. 2020. 3404422, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85090562735&doi=10.1145%2f3404397.3404422&partnerID=40&md5=1ee2463914a91b32d52f28b60a7fb9be. DOI: https://doi.org/10.1145/3404397.3404422.

25. Akbulut, G. G., Kandemir, M. T., Karakoy M. & Choi, W. “Data recompilation for multithreaded applications”. *Data Recomputation for Multithreaded Applications*. 2023, https://www.scopus.com/inward/record.uri?eid=2-s2.085181397164&doi=10.1109%2fICCAD57390.2023.10323776&partnerID=40&md5=fabfedc3945ee184a991e0e537d138d1. DOI: https://doi.org/10.1109/ICCAD57390.2023.10323776.

Conflicts of Interest: the authors declare no conflict of interest

Received 11.03.2024
Received after revision 07.05.2024
Accepted 15.05.2024

DOI: https://doi.org/10.15276/hait.07.2024.10
УДК 004.942

Моделювання та автоматизація процесу пошуку дублікатів об'єктів у знімках пам'яті

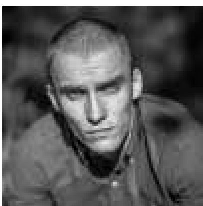
Мітіков Микола Юрійович[1)]
ORCID: https://orcid.org/0009-0002-1297-5676; mitikov.m22@fpm.dnu.edu.ua. Scopus Author ID: 59005016500
Гук Наталія Анатоліївна[1)]
ORCID: https://orcid.org/0000-0001-7937-1039; huk_n@fpm.dnu.edu.ua. Scopus Author ID: 54791066900
[1)] Дніпровський національний університет ім. Олеся Гончара, проспект Науки, 72. Дніпро, Україна

АНОТАЦІЯ

Мета цієї роботи полягає у виявленні збільшеного використання пам'яті програмними застосунками. Сучасний цикл розробки програмного забезпечення зосереджений на функціональності і часто ігнорує аспекти оптимального використання ресурсів. Обмежене фізичне масштабування задає верхній ліміт на пропускну здатність системи оброблювати запити. Наявність незмінних об’єктів з однаковою інформацію є ознакою збільшеної витрати пам’яті. Уникнення дублікатів об’єктів в пам’яті дозволяє більш раціонально використовувати існуючий ресурс і збільшити обсяги оброблюваної інформації. Існуючі наукові публікації фокусуються на дослідженні проблем витоків пам’яті, та обмежують увагою саме надмірне використання пам’яті через відсутність уніфікованої моделі пошуку надмірного використання пам’яті. Варто зазначити, що існуючі шаблони програмування містять шаблон «пул об’єктів», але залишають висновок про доцільність його впровадження інженерам, не надаючи математичного підґрунтя. Представлено розробку математичної моделі для процесу виявлення дублікатів об'єктів з властивістю незмінності типу String в знімку пам’яті. Проаналізовано промислові системи, які вимагають сотні гігабайт оперативної пам’яті для роботи та містять мільйони об’єктів в оперативній пам’яті. За таких масштабів даних, існує необхідність оптимізувати саме процес пошуку дублікатів. Методом дослідження є аналіз знімків пам’яті високонавантажених систем за допомогою програмного коду, розробленого на технології .NET та бібліотеці ClrMD. Знімок пам’яті відображає стан досліджуваного процесу у момент часу, містить усі об’єкти, потоки та виконувані операції. Бібліотека ClrMD дозволяє програмно досліджувати об’єкти, їх типи, отримувати значення полів, будувати графи зв’язків між об’єктами. Серію експериментів було проведено на віртуальних машинах під керуванням операційної системи Windows, але схожі результати можуть бути отримані для операційної системи Linux через крос-платформений стандарт позиціювання даних в пам’яті. За результатами дослідження було запропоновано оптимізацію яка дозволяє пришвидшити процес пошуку дублікатів у декілька разів. Науковий внесок дослідження полягає в створенні математично обґрунтованого підходу, який сприяє значному зменшенню використання ресурсів пам'яті та оптимізації обчислювальних процесів. Практична користь моделі підтверджується результатами оптимізації досягнутих завдяки отриманим рекомендаціям, зниженням витрат на хостинг (що забезпечує більшу економічну ефективність у розгортанні та використанні програмних систем у промислових умовах), а також збільшення обсягів оброблених даних.

Ключові слова: оптимізація, алгоритм; продуктивність; знімок пам’яті; дублювання; строка

ABOUT THE AUTHORS



Nikolay Y. Mitikov - Postgraduate student, Faculty of Applied Mathematics. Oles Honchar Dnipro National University, 72, Science Ave. Dnipro, 49010, Ukraine
ORCID: https://orcid.org/0009-0002-1297-5676; mitikov.m22@fpm.dnu.edu.ua. Scopus Author ID: 59005016500
***Research field*:** Math modeling; application performance; resource consumption

Мітіков Микола Юрійович - аспірант, факультет Прикладної математики. Дніпровський національний університет імені Олеся Гончара , пр. Науки, 72. Дніпро, 49010, Україна

Natalia A. Guk - Doctor of Physical and Mathematical Sciences, Professor, Faculty of Applied Mathematics. Oles Honchar Dnipro National University, 72, Science Ave. Dnipro, 49010, Ukraine
ORCID: https://orcid.org/0000-0001-7937-1039; huk_n@fpm.dnu.edu.ua. Scopus Author ID: 54791066900
Research field: Machine Learning; intelligent information technologies; mechanics

Гук Наталія Анатоліївна - доктор фізико-математичних наук, професор, факультет Прикладної математики. Дніпровський національний університет імені Олеся Гончара , пр. Науки, 72. Дніпро, 49010, Україна