№ 10(38) 2024
НАУКА і ТЕХНІКА СЬОГОДНІ
серії: право, економіка, педагогіка, техніка, фізико-математичні науки

UDC 004.6:004.4:004.942

https://doi.org/10.52058/2786-6025-2024-10(38)-39-49

Guk Natalia Anatoliivna, doctor of technical sciences, professor, vice-rector for scientific work, department of computer technologies, Oles Honchar Dnipro National University, https://orcid.org/0000-0001-7937-1039

Mitikov Nikolay Yuriyovych, PhD student, department of computer technologies, Oles Honchar Dnipro National University, https://orcid.org/0009-0002-1297-5676

Selivyorstova Tatjana Vitaliivna, candidate of technical sciences, associate professor, department of information technology and systems, Ukrainian State University of Science and Technologies, https://orcid.org/0000-0002-2470-6986

DETECTING EXTRAORDINARY APPLICATION MEMORY USE BY ANALYZING MEMORY SCREENSHOTS

Abstract. This study investigates excessive memory consumption in .NET applications, with a focus on identifying inefficiencies in memory allocations that lead to unnecessary resource usage. Real-world processing system memory snapshots were gathered using ProcDump, and managed heaps were thoroughly inspected with WinDBG to uncover memory usage patterns and the distribution of space among different types. ClrMD was employed to further analyze runtime data and offer optimization recommendations targeted at reducing the overall memory footprint. To validate these optimizations, Benchmark.NET performance tests were conducted using 'application' data to measure memory usage before and after the suggested changes.

The analysis uncovered that user-defined types were responsible for consuming significantly more memory than required. This overconsumption was due to overallocation, largely driven by an overestimation of the necessary data range for the objects, despite domain-specific data consistently fitting within smaller numeric ranges. The mismatch between object design and actual data requirements led to memory inefficiencies. After conducting targeted optimizations based on the real characteristics of the stored data and adhering to memory alignment principles, the study succeeded in significantly reducing the application's memory consumption.

These optimizations resulted in memory savings potentially measured in gigabytes, demonstrating the effectiveness of aligning object design with data

representation. The research underscores the value of memory snapshot analysis as a tool for identifying and mitigating excessive memory usage in .NET applications. It advocates for a more deliberate object design strategy, one that takes into account the actual range and size of the data being handled. Such an approach can result in significant performance improvements and more efficient memory management. This study offers practical insights for developers aiming to enhance the memory efficiency of .NET applications, contributing to more sustainable and scalable software systems.

Keywords: Memory Snapshot Analysis, .NET Framework, ProcDump Utility, Managed Heap Inspection, Object structure optimization, Memory Footprint Reduction

Гук Наталія Анатоліївна, доктор технічних наук, професор, проректор з наукової роботи, кафедра комп'ютерних технологій, Дніпровський національний університет імені Олеся Гончара, https://orcid.org/0000-0001-7937-1039

Мітіков Микола Юрійович, аспірант, кафедра комп'ютерних технологій, Дніпровський національний університет імені Олеся Гончара, https://orcid.org/0009-0002-1297-5676

Селівьорстова Тетяна Віталіївна, кандидат технічних наук, доцент, кафедра інформаційних технологій та систем, Український державний університет науки і технологій, https://orcid.org/0000-0002-2470-6986

ВИЯВЛЕННЯ НАДМІРНОГО ВИКОРИСТАННЯ ПАМ'ЯТІ ДОДАТКАМИ ШЛЯХОМ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ

Анотація. Це дослідження вивчає надмірне споживання пам'яті у .NET додатках, зокрема, спрямоване на виявлення неефективних виділень пам'яті, що призводять до непотрібного використання ресурсів. Були зібрані знімки пам'яті реальних обчислювальних систем за допомогою ProcDump, а керовані купи були ретельно проаналізовані за допомогою WinDBG для виявлення шаблонів використання пам'яті та розподілу простору серед різних типів. ClrMD був використаний для подальшого аналізу даних під час виконання і надання рекомендацій щодо оптимізації, спрямованих на зменшення загального обсягу використання пам'яті. Для перевірки цих оптимізацій було проведено тестування продуктивності за допомогою Benchmark.NET, використовуючи дані «додатка», щоб виміряти використання пам'яті до і після запропонованих змін.

Аналіз виявив, що користувацькі типи даних споживали значно більше пам'яті, ніж було необхідно. Це надмірне споживання було обумовлене перевиділенням пам’яті, спричиненим переоцінкою необхідного діапазону даних для цих об'єктів, хоча дані в домені зазвичай не перевищували менших числових діапазонів. Невідповідність між проєктуванням об'єктів і фактичними вимогами до даних призвела до неефективного використання пам'яті. Після застосування цілеспрямованих оптимізацій, заснованих на реальних характеристиках збережених даних та принципах вирівнювання пам'яті, дослідження успішно знизило споживання пам’яті додатком.

Ці оптимізації призвели до економії пам’яті, що може вимірюватися гігабайтами, що демонструє ефективність узгодження структури об’єктів з представленими даними. Дослідження підкреслює важливість аналізу знімків пам’яті для виявлення та усунення надмірного використання пам’яті у .NET додатках. Воно пропонує розробникам прагматичний підхід до проєктування об’єктів, що враховує фактичний діапазон і розмір даних, які обробляються, що може призвести до покращення продуктивності та більш ефективного управління пам’яттю.

Ключові слова: Аналіз знімків пам’яті, .NET Framework, утиліта ProcDump, інспекція керованої купи, оптимізація структури об'єктів, зменшення використання пам'яті.

JEL classification: C60, C61, C80, C88, C89.

Statement of the problem. Assessing the ‘correctness’ or efficiency of memory usage in software applications is a challenging task, especially in enterprise environments. Traditional metrics used to evaluate software performance often do not provide a complete picture of memory usage.

The main question this study tries to answer is: how can we reliably assess the memory usage efficiency of an application, especially in the context of large-scale enterprise computing, by analysing the actual application memory containing real application data? Addressing this issue requires: identifying parameters to measure, developing evaluation methodologies, and validating these methods through empirical analysis.

A key aspect that is often overlooked is the need to delve into the actual memory of the computer program that contains the actual application data. Inadequate estimation often fails to take into account the nuances of memory usage, which can potentially lead to erroneous optimisation strategies.

Analysis of recent studies and publications. The issue of detecting excessive memory usage in applications has been extensively researched, with several key strategies and methodologies developed to address this challenge. Memory overuse, particularly in enterprise-level applications, can lead to significant

performance degradation, increased latency, and even system crashes, making it a critical area of focus for developers and performance engineers.

One common approach to detecting excessive memory consumption is memory leak detection, which involves identifying objects or data that are not properly released after use. This can be done using Application Performance Management (APM) tools, which monitor application memory in real-time. These tools, such as those discussed in studies on Android and cloud-based infrastructures, are effective in both manual and automatic integration and are widely used across mobile and enterprise applications [1, 2]. APMs like New Relic and App Dynamics help developers track memory consumption patterns, detect anomalies, and identify memory leaks that lead to excessive usage [3, 4].

Another innovative method involves using algorithms like Precog, which combines offline training with online detection [5]. This machine learning-based algorithm analyzes memory usage patterns in cloud-based environments, using change-point detection and trend-line fitting to spot anomalous behavior over time [6]. By continuously monitoring memory utilization and comparing it with historical data, Precog can flag deviations indicative of excessive memory allocation.

Furthermore, tools like ClrMD and Benchmark.NET have been specifically utilized to analyze memory distribution in managed code environments like .NET. These tools enable detailed analysis of heap memory and allow developers to profile memory allocation in custom data structures. Benchmarking tools measure the performance impact of memory optimizations, helping to validate the effectiveness of proposed changes [1].

In addition, research has highlighted the importance of memory profiling during the development phase [7, 8]. Profiling tools, such as those built into IDEs like Visual Studio or standalone applications like Valgrind, help developers trace memory allocation, detect inefficient memory use, and prevent overuse before applications reach production environments [1, 6].

In summary, detecting excessive memory usage involves a combination of real-time monitoring, automated leak detection, machine learning algorithms, and comprehensive memory profiling tools [9, 10]. These strategies are essential to maintain application performance and avoid the pitfalls of memory overconsumption [11, 12].

The purpose of the article. This article aims to consider critical, but often ignored, aspects of memory usage. The study considers ways to solve the following tasks:

– identify the factors that consume the most memory;

– analyse the nature of the data being stored and propose more efficient alternatives;

– to test these theoretical assumptions through empirical benchmarking.

The memory snapshots were collected using the ProcDump API [13] and analysed using WinDBG [14]. ClrMD [15] was used to analyse the data at runtime, while Benchmark.NET [16] was used for performance tests to verify the results of the research.

The focus of this article is on using memory snapshot analysis to identify and reduce excessive memory consumption in .NET applications. It does not cover other performance metrics such as CPU usage or I/O operations.

Methodology. The study was conducted using memory snapshots of a transaction processing system running on the .NET platform in a corporate environment. Snapshots were collected during off-peak hours to minimize the influence of external factors. ProcDump was used to capture memory snapshots without triggering performance counter thresholds, avoiding additional memory load.

Analysis was carried out using WinDBG and SOS to examine the managed heap and memory distribution. The ClrMD library was used to analyze custom data types and their impact on memory usage. Identified memory overuse became the foundation for optimization.

Performance evaluation was done using Benchmark.NET. Several tests were designed to measure the impact of changes in data structure on overall memory consumption. A theoretical model for estimating memory savings was validated through empirical tests, confirming its accuracy.

Empirical and theoretical studies of excessive memory usage by analysing memory snapshots. The memory snapshots were collected using the ProcDump API [13] and analysed using WinDBG [14]. ClrMD [15] was used to analyse the data at runtime, while Benchmark.NET [16] was used for performance tests to verify the results of the research.

The focus of this article is on using memory snapshot analysis to identify and reduce excessive memory consumption in .NET applications. It does not cover other performance metrics such as CPU usage or I/O operations.

A quantitative assessment of the memory distribution between different types of objects in the studied .NET transaction processing system was made for the data (Table 1):

– number of instances: lists the instances of each respective object type, providing a measure of prevalence in the heap;

– memory consumption (bytes): The cumulative memory footprint in bytes for each object type, serving as the primary metric for identifying inefficient memory allocation;

– type description: the full name of each object type, including its namespace, with user-defined and system types separated.

Table 1

Memory Consumption by Object Type in a .NET Transactional Processing System

Instance Count	Memory Consumption (Bytes)	Type Description
6,109,902	244,396,080	Model.Shipping. GroupedShippingMethodZoneKey
62,550	441,122,544	System.Collections.Generic.Dictionary (Entry for Country)
5,489,559	721,114,212	System.Int32[]
786	1,370,734,672	Model.Payments.Gateways. PaymentGatewayPaymentMethod[]
34,456,461	2,620,679,684	System.String

Source: compiled by the authors

The frequent appearance of the internal structure of the 'System Collections.Generic.Dictionary' type, especially the 'Dictionary2+Entry' arrays, among the largest memory consumers indicates that these dictionaries store significant amounts of data. This prevalence not only emphasises the role of dictionaries as a key data structure in the application, but also calls into question their efficiency in terms of memory usage. The significant amount of memory occupied by these internal structures indicates the high cardinality of the stored objects, and thus requires further investigation of potential optimisation strategies to reduce memory consumption.

In the context of a 64-bit environment, where each pointer takes up 8 bytes, the memory overhead associated with class instances can be significant. Combined with the already significant cost of System.Collections.Generic.Dictionary types, this becomes a significant problem.

Given that the keys for these dictionaries are immutable, a possible approach to optimisation could be to replace class instances with structures. In this way, you can take advantage of the semantics of type-values, thus eliminating the need for additional heap allocation to store pointers to these objects.

Obviously, switching to structures can significantly reduce the memory requirements for these dictionary key-value pairs. However, this optimisation strategy requires careful consideration of trade-offs, such as increased stack load, to ensure that the resulting memory gain does not negatively impact other aspects of system performance.

The ClrMD program was written to analyse the range of data for each field of the 'GroupedShippingMethodZoneKey' type (Table 2).

Table 2

‘GroupedShippingMethodZoneKey’ class field value ranges

Field Name	Minimum value	Maximum value
shippingMethodId	1	40,048,460
shippingMethodZoneId	1	429
HashCode	int.Min	int.Max

Despite data fields are declared with ‘long’ type which spans to 9.2 quintillion, the actual values are much lower. The ‘shippingMethodId’ represents the physical shipping method for delivery, and cannot reach the reserved data range, so reducing its size twice will provide at least 32 bits benefit on each object.

The ‘shippingMethodZoneId’ field holds data range that fits ‘ushort’ range, which consumes16 bits saving 48 bits.

The hashcode can be calculated with simple bitwise routines from fields saving size of integer.

The original object size is:

$$S_{orig} = A\left(O_{over} + 2*long + int - O_{res}\right),$$

where O_{over} – object overhead 24 bytes, includes runtime info (like type pointer) and reserves space for one pointer;

O_{res} – 8 bytes, the reserved space for one pointer inside object;

$long$ – 8 bytes, size of long;

int – 4 bytes, size of integer,

$A(\ )$ – alignment function, which aligns objects by pointer size (aliquot 8 bytes).

The total size of the original object is 40 bytes, which is confirmed by the layout of the object in WinDBG (Figure 1).

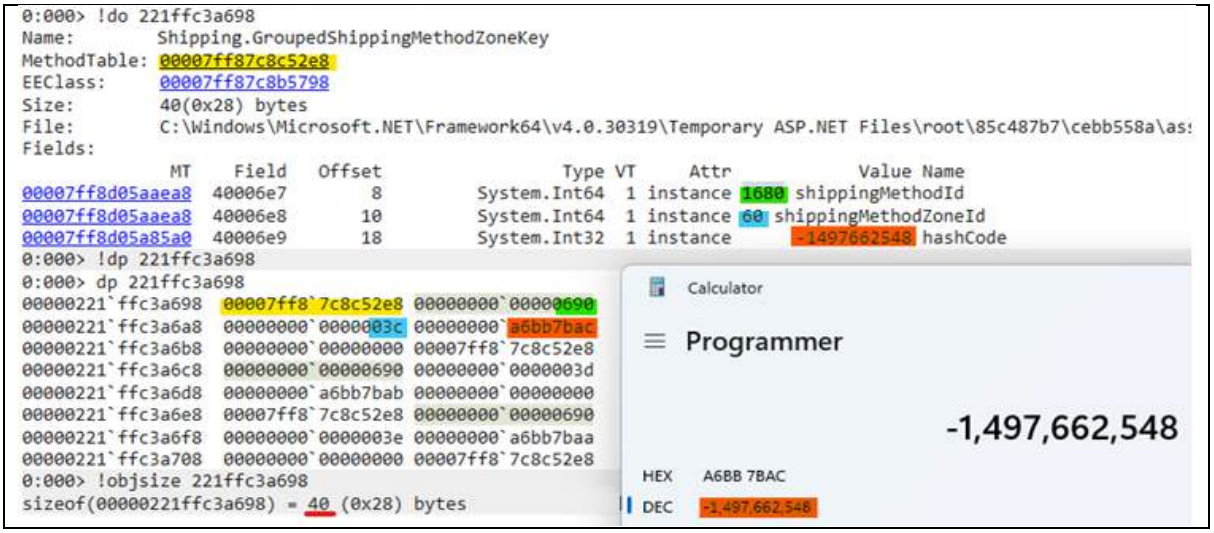

Figure 1. Original object size in memory

Since the object is used as a reference, an additional pointer size is allocated from the dictionary to point to it, which is 48 bytes of memory.

On the other hand, the fields used require 26 and 9 bits retrospectively, which totals 35 bits, or 5 bytes.

Thus, the observed overhead is more than 8 times higher than the actual amount of data stored.

When using the native .NET number types, the memory used for data transfer is 6 bytes, which is 8 times less memory.

It is worth noting that the original implementation of the class had the 'hashcode' function, which took up an excessive amount of memory.

The proposed version of the code will be 'readonly struct' (to avoid storing a pointer), taking into account the data ranges used by the application.

The correctness of the model can be verified by extracting data from the memory snapshot in the performance benchmark to emulate loading a dictionary with accurate data:

```
[Benchmark]
public void Cctr_KnowTheRangesStruct()
{
    var testGround = new List<KnowTheRangesStruct>(_dataset.Count);
    foreach (Original o in _dataset)
    {
        testGround.Add(new KnowTheRangesStruct(o.shippingMethodId, o.shippingMethodZoneId));
    }
}
```

```
BenchmarkDotNet v0.13.8, Windows 11 (10.0.22621.2283/22H2/2022Update/SunValley2)
AMD Ryzen 7 6800H with Radeon Graphics, 1 CPU, 16 logical and 8 physical cores
.NET SDK 7.0.400
  [Host]     : .NET 7.0.11 (7.0.1123.42427), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.11 (7.0.1123.42427), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev	Ratio	Gen0	Gen1	Gen2	Allocated	Alloc Ratio
Cctr_WeightOfList	8.515 ms	0.0659 ms	0.0515 ms	0.008	187.5000	187.5000	187.5000	46.61 MB	0.08
Cctr_Original	1,044.898 ms	10.3225 ms	9.1507 ms	1.000	64000.0000	32000.0000	1000.0000	553.13 MB	1.00
Cctr_OnFlyHashcode	283.965 ms	3.8156 ms	3.1862 ms	0.272	23500.0000	23000.0000	500.0000	233.07 MB	0.42
Cctr_KnowTheRanges	331.078 ms	6.4850 ms	10.2860 ms	0.309	18000.0000	17500.0000	500.0000	186.46 MB	0.34
Cctr_KnowTheRangesStruct	54.105 ms	0.9489 ms	0.8876 ms	0.052	111.1111	111.1111	111.1111	46.62 MB	0.08

Figure 2. Loading data benchmark result

When evaluating the revised object creation process, a significant gain was achieved not only in memory usage, but also in computing time. In particular, the memory bandwidth showed an impressive decrease from 533 MB to a much smaller 46.6 MB. At the same time, time performance has improved significantly, dropping from 1 second to just 54 milliseconds. These improvements are particularly important given that object creation is a procedure that occurs early in the application lifecycle, thus setting the stage for more efficient performance throughout.

As shown in Figure 3, the optimized approach focused on searching for items from the cache resulted in a tenfold speedup in search speed. This performance improvement can be attributed primarily to the restructuring of the search operation. In the original implementation, each search required the construction of a unique key, which led to additional memory allocation for each individual operation.

```
BenchmarkDotNet v0.13.8, Windows 11 (10.0.22621.2283/22H2/2022Update/SunValley2)
AMD Ryzen 7 6800H with Radeon Graphics, 1 CPU, 16 logical and 8 physical cores
.NET SDK 7.0.400
  [Host]     : .NET 7.0.11 (7.0.1123.42427), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.11 (7.0.1123.42427), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev	Ratio	Gen0	Allocated	Alloc Ratio
Locate_Original	1,243.5 ms	6.12 ms	5.73 ms	1.00	63000.0000	531117672 B	1.000
Locate_OnFlyHashcode	219.8 ms	3.31 ms	3.10 ms	0.18	23333.3333	195517104 B	0.368
Locate_KnowTheRanges	140.3 ms	0.88 ms	0.82 ms	0.11	17500.0000	146637838 B	0.276
Locate_KnowTheRangesStruct	124.2 ms	1.17 ms	1.10 ms	0.10	-	166 B	0.000

Figure 3. Finding element in the cache

Transitioning to a structure-based solution has successfully ameliorated this inefficiency, significantly curtailing the number of required memory allocations.

Conclusions. The optimization efforts yielded remarkable improvements across multiple metrics. Specifically, object creation became twenty times faster while simultaneously consuming ten times less memory. In terms of actual storage allocation for the key, an eightfold reduction was achieved, saving approximately 200 MB out of an initial 233 MB. This constitutes a significant portion of the overall memory footprint and, by extension, contributes meaningfully to the optimization of the application's performance.

Importantly, these gains are not isolated to a singular user-defined type. The architecture allows for scalability in performance improvements, as multiple user types within the system are conducive to similar restructuring. Consequently, the aggregated gain in both time and memory efficiency is substantially higher when considered in the broader context of the application.

The findings from this study lay the groundwork for future research aimed at automating the detection of sparse allocations within top-consuming objects. Subsequent investigations will focus on the development of a system capable of autonomously identifying these inefficient memory allocations and generating actionable recommendations for footprint reduction. The goal is to create a self-optimizing system that not only identifies inefficiencies but also proposes real-time solutions to enhance memory usage and overall application performance.

References:

1. Tang, Y., Wang, H., Zhan, X., Luo, X., Zhou, Y., Zhou, H., Yan, Q., Sui, Y., & Keung, J. (2022). A systematical study on Application Performance Management Libraries for apps. *IEEE Transactions on Software Engineering*, *48*(8), 3044–3065. DOI: https://doi.org/10.1109/tse.2021.3077654

2. Tang, Y., Zhou, H., Luo, X., Chen, T., Wang, H., Xu, Z., & Cai, Y. (2022). XDebloat: towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering*, *48*(11), 4501–4520. https://doi.org/10.1109/tse.2021.3120213

3. Thung, F., Liu, J., Rattanukul, P., Maoz, S., Toch, E., Gao, D., & Lo, D. (2024). Towards speedy Permission-Based debloating for Android apps. *Proceedings - 2024 IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2024*. https://doi.org/10.1145/3647632.3651390

4. Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., & Liu, Y. (2022). Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature review. *IEEE Transactions on Software Engineering*, *48*(10), 4181–4213. https://doi.org/10.1109/tse.2021.3114381

5. Li, L. (2024). Smart Software Analysis for Software Quality Assurance. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3674399.3674475

6. Jindal, A., Staab, P., Cardoso, J., Gerndt, M., & Podolskiy, V. (2021). Online memory leak detection in the Cloud-Based infrastructures. In *Lecture notes in computer science* (pp. 188–200). https://doi.org/10.1007/978-3-030-76352-7_21

7. Yeruva, A. R., & Ramu, V. B. (2023). AIOps research innovations, performance impact and challenges faced. *International Journal of System of Systems Engineering*, *13*(3), 229–247. https://doi.org/10.1504/ijsse.2023.133013

8. Clause, J., & Orso, A. (2010). LEAKPOINT. *32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*. https://doi.org/10.1145/1806799.1806874

9. Blanco, A. F., Córdova, A. Q., Bergel, A., & Alcocer, J. P. S. (2024). Asking and answering questions during memory profiling. *IEEE Transactions on Software Engineering*, *50*(5), 1096–1117. https://doi.org/10.1109/tse.2024.3377127

10. Zhou, J., Li, D., & Liu, T. (2023). Profile Dynamic Memory Allocation in Autonomous Driving Software. *10th International Conference on Dependable Systems and Their Applications, DSA 2023*. https://doi.org/10.1109/dsa59317.2023.00127

11. Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J., & Zhang, C. (2019). SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. *41st IEEE/ACM International Conference on Software Engineering, ICSE 2019*. https://doi.org/10.1109/icse.2019.00025

12. Utture, A., & Palsberg, J. (2023). From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. *31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*. https://doi.org/10.1145/3611643.3616267

13. Russinovich, M. E., & Margosis, A. (2016). *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press.

14. Hewardt, M. (2009). *Advanced .NET debugging*. Pearson Education.

15. Microsoft, & Maximov, I. [sungam3r]. (2020). *clrmd/doc/GettingStarted.md at main · microsoft/clrmd*. GitHub. https://github.com/microsoft/clrmd/blob/main/doc/GettingStarted.md#clr-debugging-a-brief-introduction

16. Akinshin, D. [AndreyAkinshin]. (2024). *Releases dotnet/BenchmarkDotNet*. GitHub. https://github.com/dotnet/BenchmarkDotNet/releases

Література:

1. Tang, Y., Wang, H., Zhan, X., Luo, X., Zhou, Y., Zhou, H., Yan, Q., Sui, Y., & Keung, J. (2022). A systematical study on Application Performance Management Libraries for apps. *IEEE Transactions on Software Engineering*, *48*(8), 3044–3065. DOI: https://doi.org/10.1109/tse.2021.3077654

2. Tang, Y., Zhou, H., Luo, X., Chen, T., Wang, H., Xu, Z., & Cai, Y. (2022). XDebloat: towards Automated Feature-Oriented App Debloating. *IEEE Transactions on Software Engineering*, *48*(11), 4501–4520. https://doi.org/10.1109/tse.2021.3120213

3. Thung, F., Liu, J., Rattanukul, P., Maoz, S., Toch, E., Gao, D., & Lo, D. (2024). Towards speedy Permission-Based debloating for Android apps. *Proceedings - 2024 IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2024*. https://doi.org/10.1145/3647632.3651390

4. Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., & Liu, Y. (2022). Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature review. *IEEE Transactions on Software Engineering*, *48*(10), 4181–4213. https://doi.org/10.1109/tse.2021.3114381

5. Li, L. (2024). Smart Software Analysis for Software Quality Assurance. *ACM International Conference Proceeding Series*. https://doi.org/10.1145/3674399.3674475

6. Jindal, A., Staab, P., Cardoso, J., Gerndt, M., & Podolskiy, V. (2021). Online memory leak detection in the Cloud-Based infrastructures. In *Lecture notes in computer science* (pp. 188–200). https://doi.org/10.1007/978-3-030-76352-7_21

7. Yeruva, A. R., & Ramu, V. B. (2023). AIOps research innovations, performance impact and challenges faced. *International Journal of System of Systems Engineering*, *13*(3), 229–247. https://doi.org/10.1504/ijsse.2023.133013

8. Clause, J., & Orso, A. (2010). LEAKPOINT. *32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010*. https://doi.org/10.1145/1806799.1806874

9. Blanco, A. F., Córdova, A. Q., Bergel, A., & Alcocer, J. P. S. (2024). Asking and answering questions during memory profiling. *IEEE Transactions on Software Engineering*, *50*(5), 1096–1117. https://doi.org/10.1109/tse.2024.3377127

10. Zhou, J., Li, D., & Liu, T. (2023). Profile Dynamic Memory Allocation in Autonomous Driving Software. *10th International Conference on Dependable Systems and Their Applications, DSA 2023*. https://doi.org/10.1109/dsa59317.2023.00127

11. Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J., & Zhang, C. (2019). SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code. *41st IEEE/ACM International Conference on Software Engineering, ICSE 2019*. https://doi.org/10.1109/icse.2019.00025

12. Utture, A., & Palsberg, J. (2023). From Leaks to Fixes: Automated Repairs for Resource Leak Warnings. *31st ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*. https://doi.org/10.1145/3611643.3616267

13. Russinovich, M. E., & Margosis, A. (2016). *Troubleshooting with the Windows Sysinternals Tools*. Microsoft Press.

14. Hewardt, M. (2009). *Advanced .NET debugging*. Pearson Education.

15. Microsoft, & Maximov, I. [sungam3r]. (2020). *clrmd/doc/GettingStarted.md at main · microsoft/clrmd*. GitHub. https://github.com/microsoft/clrmd/blob/main/doc/GettingStarted.md#clr-debugging-a-brief-introduction

16. Akinshin, D. [AndreyAkinshin]. (2024). *Releases dotnet/BenchmarkDotNet*. GitHub. https://github.com/dotnet/BenchmarkDotNet/releases