

Дніпровський національний університет імені Олеся Гончара
Міністерства освіти і науки України

Дніпровський національний університет імені Олеся Гончара
Міністерства освіти і науки України

Кваліфікаційна наукова
праця на правах рукопису

Мітіков Микола Юрійович

УДК 004.891:004.82:519.7

ДИСЕРТАЦІЯ

**ІДЕНТИФІКАЦІЯ АНОМАЛІЙ В РОБОТІ ПРОГРАМНОГО
ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ**

11 – Математика та статистика
113 – Прикладна математика

Подається на здобуття ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей,
результатів і текстів інших авторів мають посилання на відповідне джерело

_____ М. Ю. Мітіков

Науковий керівник **Гук Наталія Анатоліївна** доктор фізико-математичних
наук, професорка

Дніпро - 2026

АНОТАЦІЯ

Мітіков М. Ю. Ідентифікація аномалій в роботі програмного забезпечення на основі аналізу знімків пам'яті. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття ступеня доктора філософії за спеціальністю 113 «Прикладна математика» (11 – Математика та статистика). – Дніпровський національний університет імені Олеся Гончара, м. Дніпро, 2026.

Дисертаційну роботу присвячено математичному моделюванню та алгоритмічній ідентифікації аномалій, пов'язаних із надмірним використанням оперативної пам'яті програмними застосунками. Об'єктом дослідження є знімок пам'яті як зафіксований стан оперативної пам'яті програмного процесу у визначений момент часу. Предметом дослідження є формальне подання знімка пам'яті, критерії надлишкового подання інформації, алгоритмічні процедури виявлення таких сценаріїв, продукційні правила їх інтерпретації та кількісні оцінки можливого зменшення обсягу пам'яті.

Мета дослідження полягає у розробленні формальної моделі знімка пам'яті, критеріїв надмірності, алгоритмічних процедур і продукційної моделі представлення знань для ідентифікації сценаріїв надмірного використання оперативної пам'яті та оцінювання можливого зменшення її обсягу після допустимих перетворень подання даних.

Наявні засоби моніторингу фіксують динаміку споживання ресурсів, витоки пам'яті або відхилення від типового профілю виконання. Стабільне надлишкове подання інформації в оперативній пам'яті може не мати ознак витоку: кількість об'єктів не зростає монотонно, але структура подання даних містить повтори, надлишкові службові витрати або невідповідність між

інформаційним вмістом і фактичним обсягом пам'яті. Для таких випадків потрібне не лише спостереження за показниками споживання пам'яті, а й аналіз внутрішньої структури знімка пам'яті.

У роботі знімок пам'яті розглянуто як скінченну типізовану структуру даних, що містить множини об'єктів, типів, адрес, розмірів, значень полів, колекцій, масивів і зв'язків між об'єктами. Таке подання переводить аналіз з описового рівня до формалізованої задачі над скінченною структурою. На цій основі задано сценарії надмірного використання пам'яті, критерії їх ідентифікації, предикати застосування правил і функції кількісного оцінювання.

Задачу ідентифікації надмірного використання оперативної пам'яті сформульовано як задачу виявлення класів надлишкового подання інформації у типізованій структурі об'єктів. Для цього використано поняття еквівалентності об'єктів за інформаційним вмістом, безнадлишкового представлення та коефіцієнта надмірності. Окремо розглянуто сценарії дублювання незмінюваних об'єктів, надлишкового діапазону оголошених типів полів, втрат пам'яті через вирівнювання об'єктів, службових витрат хеш-колекцій, надлишкової місткості колекцій і стискання масивів байтів зі збереженням інформаційного вмісту.

Для виявлення дублювання незмінюваних об'єктів удосконалено алгоритмічний підхід, у якому об'єкти попередньо групуються за типом і швидко обчислюваними ознаками інформаційного вмісту. Перевірка еквівалентності виконується лише всередині отриманих підмножин. Це зменшує кількість порівнянь порівняно з повним попарним перебором і зберігає коректність виділення груп об'єктів з однаковим інформаційним вмістом.

Роль продукційної моделі полягає у застосуванні формальних критеріїв, предикатів і кількісних оцінок до фактів, отриманих із типізованої структури знімка пам'яті. Правила задаються як предикати над об'єктами, типами, значеннями полів, колекціями, масивами та службовими структурами.

Результатом застосування правил є множина виявлених сценаріїв надмірного використання пам'яті, відповідних діагностичних висновків і кількісних оцінок очікуваного зменшення обсягу пам'яті.

Емпіричну перевірку виконано на промислових знімках пам'яті програмних застосунків. У межах перевірки досліджено розподіл пам'яті за типами, кількісно оцінено дублювання незмінюваних рядкових значень, перевірено прискорене виявлення дублікатів, оцінено компактне подання ключових типів хеш-колекцій, вартість зберігання інформації у словниках для малих кількостей елементів і доцільність стискання масивів байтів. Для сформованих висновків зіставлено прогнозовані оцінки з фактичними змінами після перетворення подання даних.

Наукова новизна одержаних результатів полягає у наступному:

- **вперше запропоновано** формальну модель знімка пам'яті програмного процесу як скінченної типізованої структури об'єктів, у якій байтове представлення пам'яті інтерпретується через множини об'єктів, типів, адрес, розмірів, значень полів і зв'язків між об'єктами;
- **вперше сформульовано** задачу ідентифікації надмірного використання оперативної пам'яті як задачу виявлення класів надлишкового подання інформації у типізованій структурі об'єктів з урахуванням класів еквівалентності за інформаційним вмістом, безнадлишкового представлення та кількісного коефіцієнта надмірності;
- **дістала подальшого розвитку** продукційна модель представлення знань для аналізу знімків пам'яті, у якій правила задаються предикатами над формалізованою структурою знімка пам'яті, а результат логічного виведення визначається як множина сценаріїв надмірного використання пам'яті, відповідних рекомендацій та кількісних оцінок
- **дістав подальшого розвитку** алгоритмічний підхід до виявлення дублікатів незмінюваних об'єктів для аналізу масштабних знімків пам'яті. За рахунок предметної адаптації методів гешування простору пошуку та попереднього розбиття на підмножини (bucketing) за обчислюваними

структурними ознаками, мінімізовано необхідність попарної перевірки, що дозволяє уникнути квадратичної обчислювальної складності ($O(N^2)$) порівняно з повним перебором і забезпечує застосовність підходу для промислових даних;

- **дістав подальшого розвитку** метод кількісного оцінювання потенційного зменшення використання пам'яті шляхом його поширення на специфічні класи структурної надмірності: дублювання незмінюваних об'єктів, надлишковий діапазон оголошених типів полів, втрати через фізичне вирівнювання об'єктів (alignment), неефективне використання місткості хеш-колекцій та безвтратне стискання масивів байтів;

- **запропоновано** систему формалізованих діагностичних предикатів (rule-based підхід) для інтерпретації результатів аналізу знімків пам'яті. На відміну від неформалізованих інженерних практик, розроблений комплекс продукційних правил формалізує алгоритмічний перехід від обчислених фактів типізованої структури до діагностичних висновків і рекомендацій із застосуванням емпірично обґрунтованих порогових значень τ ;

- **дістала подальшого розвитку** система метрик валідації рекомендацій щодо зменшення використання пам'яті, яка пов'язує виявлені сценарії надмірності, прогнозоване зменшення обсягу пам'яті та фактичну зміну споживання пам'яті після реалізації рекомендацій за допомогою кількісних показників точності, повноти та похибки прогнозування.

Ключові слова: знімок пам'яті, аналіз знімків пам'яті, програмне забезпечення, продуктивність, ідентифікація аномалій, надмірне споживання пам'яті, дублювання об'єктів, неефективні алокації, оптимізація, оптимізація пам'яті, експертна система, база знань, продукційні правила, формальна модель, корпоративні системи.

ABSTRACT

Mitkov M. Y. Identification of Anomalies in Software Operation Based on Memory Snapshot Analysis. – Qualification scientific work in manuscript form.

Dissertation for the degree of Doctor of Philosophy in specialty 113 "Applied Mathematics" (11 – Mathematics and Statistics). – Oles Honchar Dnipro National University, Dnipro, 2026.

The dissertation is devoted to mathematical modeling and algorithmic identification of anomalies related to excessive memory usage in software applications. The subject of the research is a formal memory snapshot representation, redundancy criteria, algorithmic procedures for detecting excessive information representation, production rules for interpreting such scenarios, and quantitative estimates of possible memory reduction.

The objective of the research is to develop a formal memory snapshot model, redundancy criteria, algorithmic procedures, and a production knowledge representation model for identifying excessive memory usage scenarios and estimating possible memory reduction after admissible transformations of data representation.

Existing monitoring tools record resource consumption dynamics, memory leaks, or deviations from typical execution profiles. Stable excessive information representation in memory may have no leak-like behavior: the number of objects does not necessarily grow monotonically, while the data representation may contain duplicates, redundant service costs, or a mismatch between information content and actual memory usage. Such cases require analysis of the internal structure of a memory snapshot rather than observation of external memory counters only.

In the dissertation, a memory snapshot is considered as a finite typed object structure containing sets of objects, types, addresses, sizes, field values, collections, arrays, and relations between objects. This representation transforms memory analysis from a descriptive procedure into a formal problem over a finite structure.

Based on this representation, excessive memory usage scenarios, identification criteria, rule predicates, and quantitative estimation functions are defined.

The problem of identifying excessive memory usage is formulated as the problem of detecting classes of redundant information representation in a typed object structure. The formulation uses equivalence of objects by information content, non-redundant representation, and a quantitative redundancy coefficient. The considered scenarios include duplication of immutable objects, excessive declared ranges of field types, memory losses caused by object alignment, service costs of hash collections, redundant collection capacity, and compression of byte arrays while preserving information content.

For detecting duplicates of immutable objects, the algorithmic approach is improved by preliminary grouping of objects by type and rapidly computable features of information content. Equivalence verification is then performed only within the obtained subsets. This reduces the number of comparisons compared with complete pairwise enumeration while preserving correctness of duplicate group identification.

The production model role is to apply formal criteria, predicates, and quantitative estimates to facts extracted from the typed snapshot structure. Rules are defined as predicates over objects, types, field values, collections, arrays, and runtime service structures. The result of rule application is a set of detected excessive memory usage scenarios, corresponding diagnostic conclusions, and quantitative estimates of expected memory reduction.

The empirical verification was performed on industrial memory snapshots of software applications. The verification included analysis of memory distribution by types, quantitative assessment of duplicate immutable string values, testing of accelerated duplicate detection, evaluation of compact representation for hash collection key types, estimation of information storage cost in dictionaries for small numbers of elements, and assessment of byte array compression. For the obtained diagnostic conclusions, predicted estimates were compared with actual changes after transformations of data representation.

The scientific novelty of the results obtained is as follows:

- for the first time, a formal model of a software process memory snapshot is proposed as a finite typed structure of objects, in which the byte representation of memory is interpreted through sets of objects, types, addresses, sizes, field values, and relations between objects;
- for the first time, the problem of identifying excessive RAM usage is formulated as the problem of detecting classes of redundant information representation in a typed object structure, taking into account equivalence classes by information content, non-redundant representation, and a quantitative redundancy coefficient;
- the production model of knowledge representation for memory snapshot analysis is further developed, in which rules are defined as predicates over the formalized memory snapshot structure, and the result of logical inference is represented as a set of excessive memory usage scenarios, corresponding recommendations, and quantitative estimates;
- the algorithmic approach to detecting duplicates of immutable objects for the analysis of large-scale memory snapshots is further developed. Due to the domain-specific adaptation of hashing methods for the search space and preliminary partitioning into subsets (bucketing) by computable structural features, the need for pairwise verification is minimized, which makes it possible to avoid quadratic computational complexity compared with exhaustive enumeration and ensures the applicability of the approach to industrial data;
- the method for quantitatively estimating the potential reduction of memory usage is further developed by extending it to specific classes of structural redundancy: duplication of immutable objects, excessive ranges of declared field types, losses caused by physical object alignment, inefficient use of hash collection capacity, and lossless compression of byte arrays;
- a system of formalized diagnostic predicates (rule-based approach) is proposed for interpreting the results of memory snapshot analysis. In contrast to non-formalized engineering practices, the developed set of production rules formalizes

the algorithmic transition from computed facts of the typed structure to diagnostic conclusions and recommendations using empirically justified threshold values.

- the system of metrics for validating memory reduction recommendations is further developed, linking detected redundancy scenarios, predicted memory reduction, and actual changes in memory consumption after recommendation implementation through quantitative measures of precision, recall, and prediction error.

Keywords: memory snapshot, memory snapshot analysis, software, performance, anomaly detection, excessive memory consumption, object duplication, inefficient allocations, optimization, memory optimization, expert system, knowledge base, production rules, formal model, enterprise systems.

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Мітіков М.Ю., Гук Н.А. Виявлення проблем у роботі програмного забезпечення на основі аналізу знімків пам'яті. *Питання прикладної математики і математичного моделювання*. 2023. Вип. 23. С. 171–178 (особистий внесок Мітіков М.Ю.: проведення дослідження існуючих підходів, закладання вектору подальшого дослідження; Гук Н.А.: постановка задачі дослідження).

DOI: <https://doi.org/10.15421/322318>

URL: <https://pm-mm.dp.ua/index.php/pmmm/article/view/387>

2. Гук Н.А., Мітіков М.Ю. Сучасні проблеми ідентифікації аномалій в роботі Enterprise Systems. *Системні технології*. Вип. 5. 2024. С. 146–153 (особистий внесок Мітіков М.Ю.: проведення дослідження існуючих викликів та опис підходів по ідентифікації аномальної поведінки. Гук Н.А.: постановка задачі ідентифікації аномалій, узагальнення результатів дослідження)

DOI: <https://doi.org/10.34185/1562-9945-5-154-2024-15>

URL: <https://journals.nmetau.edu.ua/index.php/st/en/article/view/1877>

3. Мітіков М. Ю., Гук Н. А. Дослідження проблем швидкодії програмних додатків. *Математичне та комп'ютерне моделювання. Серія: Технічні науки*. 2024. Вип. 25. С. 22–36. (особистий внесок Мітіков М.Ю.: дослідження чинників, які впливають на швидкодію. Проведення серії розрахункових експериментів. Гук Н.А.: постановка задачі дослідження виявлення основних чинників, які впливають на швидкодію, розробка формальної моделі для опису впливу чинників)

DOI: <https://doi.org/10.32626/2308-5916.2024-25.22-36>

URL: <https://mcm-tech.kpnu.edu.ua/article/view/312531>

4. Guk N.A, Mitikov M.Y, Selivyorstova T. Detecting extraordinary application memory use by analyzing memory screenshots. *Наука і техніка сьогодні (Science and Technology Today)*. 2024. Вип. 10 (38). С. 39–49 (особистий внесок Мітіков М.Ю.: побудова продукційних правил для виявлення

аномалій, проведення розрахункових експериментів. Гук – постановка задачі дослідження, узагальнення результатів дослідження. Селівьорстова – побудова формальної моделі пам'яті та співставлення формальної моделі з результатами розрахункових експериментів).

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

URL: <https://perspectives.pp.ua/index.php/nts/article/view/15692>

5. Мітіков М. Ю., Гук Н. А. Modeling and automation of the process for detecting duplicate objects in memory snapshots. *Herald of Advanced Information Technology*. 2024. Vol. 7, No. 2. P. 147–157 (особистий внесок Мітіков М.Ю.: побудова продукційних правил для виявлення аномалій. Проведення розрахункових експериментів. Гук Н.А.: постановка задачі, розробка формальної моделі для представлення результатів.).

DOI: <https://doi.org/10.15276/hait.07.2024.10>

URL: <https://hait.od.ua/index.php/journal/article/view/29>

6. Гук Н.А., Мітіков М.Ю. Математична модель оптимізації пошуку дублікатів об'єктів типу String у знімках пам'яті. *Системні технології*. 2024. № 6 (155). С. 236-249 (особистий внесок Мітіков М.Ю.: побудова математичної моделі, проведення розрахункових експериментів. Гук Н.А.: математична постановка задачі, верифікація результатів дослідження).

DOI: <https://doi.org/10.34185/1562-9945-6-155-2024-23>

URL: <https://journals.nmetau.edu.ua/index.php/st/en/article/view/1931>

7. Мітіков М. Ю. Математична модель представлення інформації в пам'яті для аналізу продуктивності програмного забезпечення. *Сучасні проблеми моделювання*. 2025. № 28. С. 96–107. DOI: <https://doi.org/10.33842/2313-125X-2025-30-96-107>

DOI: <https://doi.org/10.33842/2313-125X-2025-30-96-107>

URL: <https://magazine.mdpu.org.ua/index.php/spm/article/view/3375>

8. Мітіков М.Ю, Гук Н.А. Підвищення продуктивності колекцій у програмному забезпеченні за допомогою аналізу знімків пам'яті та математичного моделювання. *Сучасні проблеми моделювання*. 2025. № 27. С. 109–122 (особистий внесок Мітіков М.Ю.: побудова формальної моделі використання пам'яті колекціями, її підтвердження в серії розрахункових

експериментів. Гук Н.А.: постановка задачі, побудова формальної моделі колекцій з підвищеною швидкістю, узагальнення результатів дослідження).

DOI: <https://doi.org/10.33842/2313-125X-2025-19-109-122>

URL: <https://magazine.mdpu.org.ua/index.php/spm/article/view/3358>

9. Мітіков М.Ю., Гук Н.А. Архітектура експертної системи для аналізу знімків пам'яті Системні технології. Вип. 6. 2025. С. 199–211 (особистий внесок Мітіков М.Ю.: побудова експертної системи. Гук Н.А.: постановка задачі, побудова експертної системи).

DOI: <https://doi.org/10.34185/1562-9945-5-161-2025-19>

URL: <https://journals.nmetau.edu.ua/index.php/st/en/article/view/2261>

10. Гук Н.А., Єгошкін Д.І., Мітіков М.Ю., Долотов І.О. Універсальний підхід до побудови адаптивних експертних систем на основі зважених правил та патернового аналізу станів систем // *Питання прикладної математики і математичного моделювання*. – Дніпро, 2025. – № 3 (25). – С. 505–61. – <https://doi.org/10.15421/322505> (особистий внесок Мітіков М.Ю.: впровадження матеріалів дослідження за побудованою експертною системою. Єгошкін Д.І. : впровадження матеріалів дослідження за побудованою експертною системою. Долотов І.О.: дослідження існуючих підходів до побудови експертних систем. Гук Н.А.: постановка задачі, узагальнення результатів дослідження).

DOI: <https://doi.org/10.15421/322505>

URL: <https://pm-mm.dp.ua/index.php/pmmm/article/view/492>

Наукові праці, які засвідчують апробацію матеріалів дисертації:

11. Mitikov M.Y., Guk N.A., Honcharova Y. Real-world example of application performance anomaly detection through memory analysis. *Modern scientific and technical research – 2023: матеріали II Всеукраїнської науково-практичної конференції молодих учених та студентів (Дніпро, 11 травня 2023 р.)*. Дніпро, 2023. С. 280–282. URL: https://confcontact.com/2023-suchasni-ntd/conference_materials_suchasni_ntd_in_english_2023.pdf

12. Mitikov M.Y., Guk N.A. Review of methods and tools for system analysis of software performance. *Mathematical and Software of Intelligent Systems (MPIS-*

2023): *тези доповідей XXI Міжнародної науково-практичної конференції (Дніпро, 22–24 листопада 2023 р.)*. Дніпро : ДНУ, 2023. С. 213–214. URL: [https://www.dnu.dp.ua/docs/ndc/2023/materiali%20konf/24_%D0%9C%D0%9F%D0%97%D0%86%D0%A1-2023%20\(1\).pdf](https://www.dnu.dp.ua/docs/ndc/2023/materiali%20konf/24_%D0%9C%D0%9F%D0%97%D0%86%D0%A1-2023%20(1).pdf)

13. Мітіков М. Ю., Гук Н. А. Інформаційна технологія діагностики надмірного використання пам'яті на основі аналізу знімків пам'яті. *Modern Information and Communication Technologies in Transport, Industry and Education: тези доповідей XVII Міжнародної науково-практичної конференції (Дніпро, 13–14 грудня 2023 р.)*. Дніпро, 2023. С. 32. URL: <https://www.tsatu.edu.ua/kn/wp-content/uploads/sites/16/sbornik-xvii-modern-it-conf-2023.pdf>

14. Мітіков М. Ю., Гук Н. А. Виявлення надлишкового використання пам'яті програмними додатками. *Інформаційні технології: теорія і практика: тези доповідей I (VII) Міжнародної науково-практичної конференції здобувачів вищої освіти і молодих учених (Дніпро, 20–22 березня 2024 р.)*. Дніпро: Свідлер А. Л., 2024. С. 107–109. URL: <https://sau.nmu.org.ua/ua/science/conference/ITTP/international/ProgramITTP2024.pdf>

15. Mitikov M.Y., Guk N.A. Enhancing collection performance in software through memory snapshot analysis and mathematical modeling. *Автоматика-2024: тези доповідей XXVII Міжнародної конференції (Дніпро, 20–22 листопада 2024 р.)*. Дніпро, 2024. С. 114–115. URL: <http://mpzis.dnu.dp.ua/wp-content/uploads/2025/11/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0-2024-%D1%82%D0%B5%D0%B7%D0%B8-%D0%B4%D0%BE%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D0%B5%D0%B9.pdf>

16. Мітіков М. Ю., Гук Н. А. Математична модель оцінки операційної вартості використання хеш-колекцій програмного забезпечення за допомогою знімків пам'яті. *Наука і сталий розвиток транспорту – 2024: збірник тез доповідей Всеукраїнської науково-технічної конференції студентів і молодих*

учених (Дніпро, 27 листопада 2024 р.). Дніпро: УДУНТ, 2024. С. 81–82. URL: <https://nmetau.edu.ua/file/tom2.2024.pdf>

17. Мітіков М. Ю. Оптимізація процесу пошуку сегментів пам'яті: математичне моделювання та експериментальна оцінка повноти інформаційного вмісту при фіксованих об'ємах пам'яті. *Інформаційні технології в металургії та машинобудуванні (ІТММ'2025): тези доповідей Міжнародної науково-практичної конференції (Дніпро, 23–24 квітня 2025 р.)*. Дніпро: УДУНТ, 2025. С. 676–680. URL: <https://journals.nmetau.edu.ua/index.php/itmm/uk/issue/view/153>

18. Mitikov M.Y, Guk N.A., Voliansky R., Mozhaiev M., Pranolo A. Mathematical Modeling And Experimental Evaluation Of The Information Content Completeness At Fixed Memory Volumes: Application Of Compression Algorithms. *5th International Conference on Information Technologies: Theoretical and Applied Problems: тези доповідей V Міжнародної науково-практичної конференції (Тернопіль, 22-24 жовтня 2025р.) Scopus: <https://ceur-ws.org/Vol-4146/>*

ЗМІСТ

ВСТУП		17
Розділ 1. ОГЛЯД МЕТОДІВ ВИЯВЛЕННЯ ТА АНАЛІЗУ ПРОБЛЕМ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ		25
1.1	Методи виявлення проблем продуктивності програмного забезпечення	25
1.2	Аналіз знімків пам'яті та його застосування в діагностиці програмних систем	29
1.3	Витоки пам'яті та стабільне надмірне використання оперативної пам'яті	33
1.4	Алгоритмічні підходи до виявлення дублювання, надлишкових структур і неефективного подання даних	36
1.5	Онтологічні, логічні, rule-based та експертні підходи до інтерпретації діагностичних ознак	40
1.6	Невирішені задачі та висновки до розділу	44
1.7	Висновки до розділу	46
Розділ 2. МАТЕМАТИЧНІ МОДЕЛІ ПРЕДСТАВЛЕННЯ ІНФОРМАЦІЇ В ПАМ'ЯТІ ДЛЯ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ		48
2.1	Визначення знімку пам'яті та базові поняття	48
2.2	Формальна модель знімка пам'яті та постановка задачі ідентифікації надмірності пам'яті	49
2.3	Оцінювання ефективності використання пам'яті.	53
2.4	Операції над об'єктами відображеними в знімку пам'яті	56
2.5	Класифікація ситуацій надмірного використання пам'яті	56
2.5.1	Детектування витоку пам'яті	57
2.5.2	Представлення незмінюваної інформації	59
2.5.3	Оптимізація процесу виявлення дублікатів	62
2.5.4	Оцінювання вартості зберігання інформації у хеш-колекціях типу Dictionary<K,V>	65
2.5.5	Оцінювання доцільності звуження типів полів за фактичним діапазоном значень	67
2.5.6	Оцінювання ефекту стискання об'єктів типу byte[] зі збереженням доступності всієї інформації	69
2.6	Висновки до розділу	73
Розділ 3. ЕМПІРИЧНІ ТА ТЕОРЕТИЧНІ ДОСЛІДЖЕННЯ НАДМІРНОГО ВИКОРИСТАННЯ ПАМ'ЯТІ ШЛЯХОМ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ		76
3.1	Методика збору й аналізу промислових знімків пам'яті та первинний аналіз розподілу пам'яті за типами	76
3.2	Ідентифікація та кількісне оцінювання дублювання об'єктів типу System.String	81
3.3	Експериментальна перевірка алгоритму прискореного виявлення дублікатів об'єктів типу System.String	88
3.4	Оцінювання компактного подання ключового типу для Dictionary<K,V>	92
3.4.1	Експериментальне оцінювання компактного подання ключового типу	96

3.5	Експериментальне оцінювання вартості зберігання інформації у Dictionary<K,V> для малих кількостей елементів	99
3.6	Експериментальне оцінювання стискання об'єктів типу System.Byte[]	108
3.7	Валідація рекомендацій та оцінювання фактичного впливу від впровадження	113
3.8	Висновки до розділу	116
Розділ 4. ЕКСПЕРТНА СИСТЕМА ДЛЯ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ ТА ВИЯВЛЕННЯ НАДМІРНОГО ВИКОРИСТАННЯ ПАМ'ЯТІ		120
4.1	Продукційне представлення результатів ідентифікації надмірності пам'яті	120
4.2	Продукційна архітектура застосування правил	124
4.3	База знань і продукційні правила	126
4.3.1	Правило виявлення дублювання незмінюваних об'єктів	127
4.3.2	Правило виявлення надлишкового діапазону оголошених типів полів	129
4.3.3	Правило виявлення втрат пам'яті через вирівнювання об'єктів	130
4.3.4	Правило виявлення надлишкової ємності колекцій	131
4.3.5	Правило оцінювання вартості зберігання інформації у хеш-колекціях	133
4.3.6	Правило оцінювання доцільності стискання масивів байтів	134
4.3.7	Правило виявлення надлишкової гранулярності синхронізації у потокобезпечних словниках	136
4.4	Джерела знань і формування фактів зі знімка пам'яті	137
4.5	Апробація продукційного механізму на промислових знімках пам'яті	139
4.6	Висновки до розділу	142
ВИСНОВКИ		145
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ		148
ДОДАТОК А. АКТ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ		161
ДОДАТОК Б. ВИБІР МОВИ ПРОГРАМУВАННЯ		163
ДОДАТОК В. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА		167

ВСТУП

Актуальність теми.

У програмних застосунках керованого середовища виконання фактичне споживання оперативної пам'яті визначається обсягом даних, способом їх об'єктного подання, службовими структурами середовища виконання, вибором типів, колекцій і схем зберігання значень. За великої кількості об'єктів навіть локальні надлишкові представлення можуть утворювати значні витрати пам'яті.

Засоби моніторингу фіксують узагальнені метрики: обсяг виділеної пам'яті, частоту збирання сміття, пікові навантаження, аварійні стани або відхилення від типового режиму роботи. Такі метрики не завжди виявляють постійне надмірне використання пам'яті, якщо воно закладене у структуру даних і не проявляється як витік або короткочасний стрибок споживання ресурсів.

Знімок пам'яті програмного процесу містить інформацію про фактичний стан оперативної пам'яті у вибраний момент часу: об'єкти, типи, розміри, значення полів, зв'язки між об'єктами, колекції та службові елементи середовища виконання. Тому знімок пам'яті може бути використаний як джерело даних для аналізу структурного подання інформації в оперативній пам'яті.

Для математичного аналізу знімок пам'яті потрібно інтерпретувати як формалізовану структуру. У такій структурі визначаються множини об'єктів, типів, значень, зв'язків, розмірів і службових елементів. На цій основі задаються сценарії надмірного використання пам'яті, критерії їх ідентифікації, предикати спрацювання правил і кількісні оцінки можливого зменшення обсягу пам'яті після допустимих перетворень подання даних.

Наукова задача полягає у побудові формалізованого подання знімка пам'яті, визначенні сценаріїв надмірного використання оперативної пам'яті та розробленні критеріїв і алгоритмічних процедур для їх ідентифікації. Така постановка відокремлює аналіз пам'яті від ручного перегляду знімків і

пов'язує його з математичним моделюванням, оцінюванням надмірності та перевіркою допустимих перетворень подання даних.

Продукційний механізм у роботі розглядається як спосіб застосування формалізованих критеріїв до структури знімка пам'яті. Факти, отримані зі знімка, зіставляються з умовами правил, побудованими на предикатах і кількісних оцінках, введених у формальній моделі. Результатом є сценарії надмірного використання пам'яті, відповідні рекомендації та оцінки можливого зменшення її обсягу. У цій постановці зберігається послідовність: модель знімка пам'яті → критерії надмірності → алгоритмічні процедури → продукційні правила → кількісна перевірка.

Зв'язок роботи з науковими програмами, планами, темами.

Дисертаційні дослідження здійснювалися у відповідності до планів наукових досліджень кафедри комп'ютерних технологій Дніпровського національного університету імені Олеся Гончара в рамках наукових тем “Детерміновані та стохастичні алгоритми комп'ютерного моделювання об'єктів та процесів різної природи”, номер державної реєстрації 0122U001467, “Розробка високоефективних комп'ютерних алгоритмів для аналізу та ідентифікації параметрів математичних моделей”, № держреєстрації 0125U002277, кафедри комп'ютерних технологій у відповідності до тематичних планів науково-дослідних робіт Дніпровського національного університету імені Олеся Гончара.

Мета і задачі дослідження.

Метою дисертаційної роботи є розроблення формальної моделі знімку пам'яті, критеріїв та алгоритмічних процедур ідентифікації сценаріїв надмірного використання оперативної пам'яті у програмних застосунках керованого середовища виконання, а також побудова експертної системи для виявлення фактів надмірного використання пам'яті, надання рекомендацій щодо усунення та оцінки ефекту від впровадження.

Для досягнення поставленої мети необхідно розв'язати такі задачі:

- проаналізувати існуючі підходи до подання, інтерпретації, аналізу та зменшення використання оперативної пам'яті у програмних застосунках керованого середовища виконання;
- побудувати формальну модель знімка пам'яті як скінченної типізованої структури, що відображає об'єкти пам'яті, їхні типи, зв'язки, значення, розміри та службові елементи середовища виконання;
- сформулювати задачу ідентифікації сценаріїв надмірного використання пам'яті на основі формалізованої структури знімка пам'яті;
- визначити критерії, предикати та кількісні характеристики для виявлення окремих класів надмірного подання інформації в оперативній пам'яті;
- розробити алгоритмічні процедури обчислення характеристик надмірності та оцінювання можливого зменшення обсягу пам'яті після допустимих перетворень подання даних;
- побудувати продукційну модель застосування правил до формалізованого подання знімка пам'яті для визначення сценаріїв надмірності, рекомендацій та кількісних оцінок;
- перевірити запропоновані моделі, критерії та алгоритмічні процедури на знімках пам'яті програмних застосунків.

Об'єктом дослідження є знімок пам'яті як зафіксований стан оперативної пам'яті програмного процесу у визначений момент часу.

Предметом дослідження є математичні моделі, критерії, алгоритмічні процедури та продукційні правила ідентифікації сценаріїв надмірного використання оперативної пам'яті за формалізованим поданням її знімка, а також кількісні оцінки можливого зменшення обсягу пам'яті після допустимих перетворень подання даних.

Методи дослідження: для вирішення поставлених задач застосовано методи математичного моделювання, теорії множин, дискретної математики,

структурного аналізу даних, алгоритмічного аналізу, представлення знань, продукційного виведення, математичної статистики та експериментальної перевірки. Інструментальні засоби отримання й обробки знімків пам'яті використано для формування вхідних даних, реалізації алгоритмічних процедур і перевірки результатів.

Достовірність отриманих результатів визначається коректністю математичних постановок задач, узгодженістю моделей, критеріїв та алгоритмічних процедур, відтворюваністю обчислювальних експериментів, зіставленням прогнозованих і фактичних результатів від впровадження змін, а також перевіркою запропонованих підходів на знімках пам'яті програмних застосунків.

Наукова новизна одержаних результатів полягає у наступному:

- вперше запропоновано формальну модель знімка пам'яті програмного процесу як скінченної типізованої структури об'єктів, у якій байтове представлення пам'яті інтерпретується через множини об'єктів, типів, адрес, розмірів, значень полів і зв'язків між об'єктами, що задає математично визначене подання стану оперативної пам'яті;
- вперше сформульовано задачу ідентифікації надмірного використання оперативної пам'яті як задачу виявлення класів надлишкового подання інформації у типізованій структурі об'єктів з урахуванням класів еквівалентності за інформаційним вмістом, безнадлишкового представлення та кількісного коефіцієнта надмірності;
- дістав подальшого розвитку алгоритмічний підхід до виявлення дублікатів незмінюваних об'єктів у знімках пам'яті шляхом попереднього групування об'єктів за типом і швидко обчислюваними ознаками інформаційного вмісту з подальшою перевіркою еквівалентності всередині отриманих підмножин, що зменшує кількість порівнянь порівняно з повним попарним перебором;
- дістав подальшого розвитку метод кількісного оцінювання можливого зменшення використання пам'яті для основних класів надмірного

подання даних, зокрема дублювання незмінюваних об'єктів, надлишкового діапазону типів полів, втрат через вирівнювання об'єктів, неефективного використання хеш-колекцій і стискання масивів байтів зі збереженням інформаційного вмісту;

- дістала подальшого розвитку продукційна модель представлення знань для аналізу знімків пам'яті, у якій правила задаються предикатами над формалізованою структурою знімка пам'яті, а результат логічного виведення визначається як множина сценаріїв надмірного використання пам'яті, відповідних рекомендацій та кількісних оцінок;

- дістала подальшого розвитку система метрик валідації рекомендацій щодо зменшення використання пам'яті, яка пов'язує виявлені сценарії надмірності, прогнозоване зменшення обсягу пам'яті та фактичну зміну споживання пам'яті після реалізації рекомендацій за допомогою кількісних показників точності, повноти та похибки прогнозування.

Практичне значення одержаних результатів. Практичне значення одержаних результатів полягає у застосуванні запропонованих моделей, критеріїв, продукційних правил та алгоритмічних процедур для аналізу знімків пам'яті програмних застосунків і виявлення сценаріїв надмірного використання оперативної пам'яті.

Програмна реалізація використовується як засіб апробації запропонованих моделей та алгоритмічних процедур, а також як інструментальна основа для перевірки сформованих рекомендацій на знімках пам'яті.

Експериментальна перевірка підтвердила можливість кількісного оцінювання допустимих оптимізаційних перетворень і зіставлення прогнозованого зменшення обсягу пам'яті з фактичними результатами після внесення змін у програмний код. Практична апробація показала зменшення пікового використання оперативної пам'яті для досліджених сценаріїв; величина зменшення залежить від структури даних, конфігурації застосунку та характеру навантаження.

Отримані результати можуть бути використані для аналізу знімків пам'яті, виявлення надлишкового подання інформації, формування рекомендацій щодо зміни подання даних і перевірки очікуваного результату таких змін у програмних застосунках, де обсяг оперативної пам'яті є ресурсним обмеженням.

Особистий внесок здобувача. Результати дисертаційної роботи відображено в 18 наукових працях. Усі результати дисертаційної роботи, що виносяться на захист, отримані автором особисто. У працях, що опубліковані у співавторстві, здобувачеві належить:

[16, 28, 68] - узагальнено підходи та інструменти системного аналізу продуктивності програмного забезпечення, окреслено класи аномалій у програмних застосунках та обґрунтовано аналіз надмірного використання пам'яті, яке не відповідає критеріям витоку й потребує аналізу знімків пам'яті;

[20, 52, 101] - розроблено та апробовано методичні засади діагностики проблем у роботі програмного забезпечення на основі аналізу знімків пам'яті, а також показано можливість виявлення нетипового використання пам'яті за структурою об'єктів у знімку пам'яті;

[84, 98] - запропоновано математичні моделі представлення інформації в оперативній пам'яті, що використовуються для формалізованого оцінювання накладних витрат і вартості використання структур даних у керованих середовищах; зокрема, побудовано модель оцінки операційної вартості використання хеш-колекцій за даними знімків пам'яті;

[54, 85] - розроблено модельно-алгоритмічний апарат прискореного виявлення дублікатів об'єктів у знімках пам'яті та математичну модель оптимізації пошуку дублікатів об'єктів типу String; показано можливість зниження обчислювальних витрат порівняно з повним перебором;

[21, 27, 77, 97, 82] - виконано експериментальну апробацію підходів на сценаріях аналізу пам'яті, підтверджено придатність поєднання трасування подій та знімків пам'яті для локалізації вузьких місць і верифікації запропонованих змін, а також обґрунтовано підвищення продуктивності

колекцій на основі аналізу знімків пам'яті та математичного моделювання; додатково досліджено застосування компресійних підходів і оцінку повноти інформаційного вмісту при фіксованих обсягах пам'яті;

[102, 103] - запропоновано підхід до побудови продукційного механізму застосування правил для аналізу знімків пам'яті та формування рекомендацій щодо покращення використання пам'яті на основі зважених правил і патернового аналізу станів.

Апробація результатів дисертації. Результати дисертаційної роботи доповідались і обговорювались на семінарі «Сучасні питання оптимізації та дискретної математики» при Науковій раді НАН України з проблеми «Кібернетика», який функціонує при Дніпровському національному університеті імені Олеся Гончара; на міжнародних конференціях «Modern scientific and technical research» (м. Дніпро, 2023 р.); «Математичне та програмне забезпечення інтелектуальних систем (МПЗІС)» (м. Дніпро, 2023 р.); «Modern Information and Communication Technologies in Transport, Industry and Education» (м. Дніпро, 2023 р.); «Інформаційні технології: теорія і практика» (м. Дніпро, 2024 р.); «Автоматика» (м. Дніпро, 2024 р.); «Наука і сталий розвиток транспорту» (м. Дніпро, 2024 р.); «Сучасні питання оптимізації та дискретної математики» (м. Дніпро, 2024 р.); «Інформаційні технології в металургії та машинобудуванні (ІТММ)» (м. Дніпро, 2025 р.); «Information Technologies: Theoretical and Applied Problems» (м. Тернопіль, 2025 р.).

Публікації. Основні результати дисертаційної роботи опубліковано в 18 наукових працях: одна публікація [82] у виданні, що індексується науково-метричною базою Scopus; 9 статей ([20, 27, 28, 54, 77, 83, 85, 102, 103]) у наукових фахових виданнях України категорії Б, спеціальність 113, 1 стаття ([101]) у наукових фахових виданнях України категорії Б, спеціальність 122, 8 тез доповідей у збірниках матеріалів міжнародних наукових конференцій ([6, 16, 21, 52, 68, 82, 84, 97, 98]).

Структура та обсяг дисертації. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, переліку використаних джерел, що містить 104 найменувань на 13 сторінках та додатків на 8 сторінках. Загальний обсяг дисертації – 170 сторінок, обсяг основного тексту – 147 сторінок. Робота містить 23 рисунка та 7 таблиць.

Подяки. Автор висловлює щирі подяки науковому керівнику, доктору фізико-математичних наук, професору Гук Наталії Анатоліївні за постійну увагу, доброзичливе ставлення та допомогу в роботі.

Розділ 1. ОГЛЯД МЕТОДІВ ВИЯВЛЕННЯ ТА АНАЛІЗУ ПРОБЛЕМ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Методи виявлення проблем продуктивності програмного забезпечення

Дослідження продуктивності програмного забезпечення сформувалося як сукупність підходів до спостереження, вимірювання та пояснення поведінки програмних систем у часі. В. Gregg розглядає продуктивність як багаторівневу властивість, що залежить від взаємодії процесора, оперативної пам'яті, кешів, введення-виведення, мережевих підсистем, планувальника та профілю навантаження [17; 97]. Перевага такого системного підходу полягає у можливості зіставляти симптоми різної природи, наприклад зростання часу відповіді, збільшення черг, зміну частоти системних викликів або підвищене споживання пам'яті. Його обмеження полягає в тому, що агреговані показники описують зовнішній прояв проблеми, але не визначають, які саме об'єкти, значення або структури даних утворюють надлишкове представлення інформації в оперативній пам'яті.

У прикладних дослідженнях останніх років ця логіка доповнюється побудовою метрик ефективності програмного забезпечення та прогнозних моделей навантаження. Д. А. Логвин і Л. М. Божуха розглядають метрики оцінювання ефективності програмного забезпечення для задач обробки природної мови [23], а С. К. Сімакін і Л. М. Божуха аналізують прогнозування навантаження на сервер із використанням методів штучного інтелекту для оптимізації веб-сервісів [71]. Такі роботи підтверджують важливість кількісного опису продуктивності, проте вони також демонструють типову межу метрик: вимірювання ефективності або прогноз навантаження не розкривають, які саме структури в пам'яті створюють надлишкові витрати.

Профілювання коду є одним із найпоширеніших способів локалізації витрат виконання. А. Осіров систематизує інструменти профілювання C#-застосунків за типом зібраних даних, зокрема за часом виконання методів,

частотою викликів, розподілом пам'яті та поведінкою потоків [14]. С. Huffman у контексті аналізу продуктивності Windows також показує, що профілі виконання та лічильники продуктивності дають змогу локалізувати вузькі місця на рівні процесів, потоків і системних ресурсів [2]. Проте профілювання фіксує переважно динамічні витрати виконання. Воно не відповідає безпосередньо на питання, чи є фактичне подання даних у пам'яті надлишковим, чи існують фізично різні об'єкти з однаковим інформаційним вмістом, або чи відповідає оголошений тип поля фактичному діапазону значень.

Моніторинг і аналіз часових рядів продуктивності застосовуються для неперервного спостереження за промисловими системами. Т. Veasey і S. Dodson розглядають дані моніторингу як джерело для побудови моделей і виявлення нетипової поведінки за часовими рядами [76]. Системи на зразок Application Insights доповнюють цей підхід автоматизованим пошуком аномальних змін у телеметрії та формуванням попереджень за історичними даними [13; 74]. Перевага моніторингових методів полягає у придатності до довготривалого спостереження без безпосереднього втручання в код. Однак вони працюють із симптомами, а не з внутрішньою структурою даних: збільшення обсягу пам'яті, частоти збирання сміття або часу відповіді ще не задає критерію, за яким можна відрізнити необхідне використання пам'яті від надлишкового.

На рівні ІТ-сервісів діагностичний підхід пов'язаний не лише з вимірюванням локальних характеристик програми, а й із керуванням сервісними станами, навантаженням і якістю обслуговування. К. Є. Золотько і Д. В. Красношайка розглядають управління та діагностику надання ІТ-сервісів [24], тоді як С. К. Сімакін і Л. М. Божуха застосовують прогнозування серверного навантаження для оптимізації веб-сервісів [71]. Для задачі аналізу пам'яті ці підходи важливі тим, що показують роль раннього виявлення відхилень, але їхня одиниця аналізу залишається сервісом або часовим рядом, а не об'єктом у керованій купі.

Статичний аналіз коду утворює інший напрям діагностики. P. Louridas показує, що статичні аналізатори можуть виявляти потенційні дефекти, небезпечні конструкції, порушення стилю та фрагменти коду, які потребують перевірки, без запуску програми [71; 73]. Такий підхід корисний на ранніх етапах розроблення, оскільки дозволяє працювати з усією кодовою базою до появи виробничого інциденту. Водночас статичний аналіз не має доступу до фактичного розподілу значень у пам'яті, реальної кількості об'єктів, заповненості колекцій, дублювання рядкових значень або службових витрат конкретного виконання. Тому він може вказати на потенційно неефективну конструкцію, але не дає кількісної оцінки надмірності у фіксованому стані процесу.

Методи машинного навчання та статистичного виявлення аномалій розширюють можливості аналізу складних систем. A. B. Nassif, M. A. Talib, Q. Nasir, F. M. Dakalbab та співавтори систематизують моделі виявлення аномалій за типом ознак, способом навчання і характером даних [72]. K. Choi, J. Yi, S. Park і S. Yoon аналізують глибинні моделі для часових рядів і показують їхню придатність до пошуку нетипової поведінки в послідовностях спостережень [77]. Сильна сторона цих методів полягає у здатності виявляти складні відхилення у просторі ознак. Їхня межа для задач аналізу пам'яті полягає в залежності від попередньо визначених ознак: перед застосуванням класифікаційної або статистичної моделі потрібно задати, які властивості об'єктів, типів, полів, масивів і колекцій мають діагностичне значення.

Близьку роль відіграють кластеризаційні методи, коли потрібно згрупувати стани або об'єкти за подібністю ознак. O. M. Кісельова, O. M. Придуманова і O. A. Філат застосовують DBSCAN та OPTICS для виявлення мережеских загроз [26], а M. Євланов, H. Васільцова, I. Панфорова, B. Мороз, A. Мартиненко і Д. Мороз порівнюють ієрархічні методи кластеризації для ранньої ідентифікації конфігураційних одиниць ІТ-продукту [51]. Для аналізу знімків пам'яті ця група робіт є методично корисною, оскільки показує, як ознаки можуть бути зведені до груп і класів; проте вона не визначає, які саме

ознаки пам'яті мають бути покладені в основу групування об'єктів і як перевіряти еквівалентність інформаційного вмісту.

Звідси випливає важлива межа традиційного аналізу продуктивності: він виявляє події, що вже проявилися в часі виконання або в метриках системи, але не завжди формує пояснення на рівні структури даних. Для задачі надмірного використання пам'яті потрібна не лише інформація про те, що пам'яті використано багато, а й відповідь, які об'єкти утворюють цю величину, які з них містять однаковий інформаційний вміст, які структури мають непропорційні службові витрати і які перетворення можуть зберегти зміст даних. Таке питання виходить за межі стандартного профілювання і вимагає переходу до аналізу знімків пам'яті.

Регресійне тестування та інженерні процедури контролю якості дають інший тип доказів. А. Labuschagne, L. Inozemtseva і R. Holmes досліджують вартість регресійного тестування на практиці [88], а R. Osherove і V. Khorikov систематизують підходи до модульного тестування [90]. Ці методи підвищують надійність змін у коді, проте їхній об'єкт – коректність функціонування, а не ефективність подання даних у конкретному стані пам'яті. Навіть якщо всі тести проходять успішно, програма може зберігати однакові значення у багатьох екземплярах, використовувати колекції з надлишковою ємністю або мати великі службові витрати, які не порушують функціональної поведінки.

Порівняльні експериментальні дослідження продуктивності також мають обмежену пояснювальну здатність щодо пам'яті. R. Kassim, N. A. Bakar і K. H. Awang аналізують продуктивність C#-програм через експериментальні показники виконання [70]. Такі результати дають змогу зіставляти реалізації або конфігурації, але вони залежать від вибраного навантаження, апаратного середовища і набору тестових сценаріїв. Якщо тест не відтворює фактичний розподіл даних, оцінка швидкодії не виявляє приховану надлишковість у пам'яті. Тому експериментальна оцінка продуктивності не замінює аналізу об'єктів, які реально присутні у виконуваному процесі.

Метрики ефективності, зокрема у прикладних задачах обробки природної мови, також вимагають обережного перенесення на аналіз пам'яті. У роботі Д. А. Логвина і Л. М. Божухи метрики використовуються для оцінювання результативності програмного забезпечення в певному класі задач [23]. У контексті знімків пам'яті метрика має бути пов'язана не лише з якістю функціонального результату, а й із відношенням між корисним інформаційним вмістом та фактичним обсягом пам'яті. Саме тому потрібні спеціальні показники надмірності, які не збігаються зі звичайними метриками швидкодії або якості сервісу.

Отже, наявні методи виявлення проблем продуктивності забезпечують вимірювання, локалізацію симптомів і пошук нетипової поведінки, однак переважно працюють із зовнішніми метриками або з кодом, а не з інформаційним вмістом об'єктів у пам'яті. Нерозв'язаною після аналізу цього напрямку залишається задача переходу від симптомів продуктивності до формалізованих ознак надлишкового подання даних у конкретному стані програмного процесу. Саме ця межа зумовлює потребу розглядати знімки пам'яті як окреме джерело структурованої діагностичної інформації.

1.2 Аналіз знімків пам'яті та його застосування в діагностиці програмних систем

Знімок пам'яті процесу фіксує стан виконуваної програми у визначений момент часу. У теорії операційних систем процес описується через адресний простір, потоки виконання, контекст процесора, дескриптори ресурсів і службові структури керування; така модель подана, зокрема, у праці А. Silberschatz, Р. В. Galvin і G. Gagne [1]. Для діагностичного аналізу це означає, що дамп не є лише копією файлу або журналом подій. Він містить фрагмент фактичного стану обчислювального процесу, у якому збережено об'єкти,

області пам'яті, стеки, модулі, службові структури і частину значень, що були доступні під час виконання.

У працях М. Hewardt і D. Pravat, а також Т. Soulamі знімки пам'яті Windows розглядаються як основний матеріал післяаварійного відлагодження: за ними відновлюються винятки, стеки викликів, завантажені модулі, значення реєстрів, фрагменти купи та службові структури процесу [3; 7]. С. Huffman доповнює цей напрям практиками аналізу продуктивності Windows і показує, що дампи та лічильники продуктивності можуть застосовуватися разом для локалізації станів зависання, високого навантаження або аномальної поведінки процесів [2]. Аналогічно О. Ткаченко та О. Голубенко підкреслюють важливість аналізу часових витрат, використання оперативної та буферної пам'яті під час виконання складних запитів у базах даних, що підтверджує доцільність використання інформації про фактичний стан виконуваної системи для дослідження продуктивності програмного забезпечення [15]. Перевага такого підходу полягає у можливості аналізувати стан програми без повторного відтворення інциденту. Його межа полягає в тому, що дампи використовуються переважно як інженерний артефакт відлагодження, а не як математично визначена структура для порівняння об'єктів і кількісного оцінювання надлишковості.

У цифровій криміналістиці оперативна пам'ять трактується як джерело даних, які можуть бути відсутні у файловій системі. R. Ptashkin, O. Hozhyi, Y. Obruch, V. Pavlov і R. Kalinichenko аналізують оперативну пам'ять як окремий метод комп'ютерно-технічного дослідження [9]. М. Н. Ligh, А. Case, J. Levy і А. Walters систематизують підходи до відновлення процесів, мережеских з'єднань, завантажених модулів і артефактів виконання [10]. Ці праці показують, що знімок пам'яті має високу інформаційну цінність для реконструкції стану системи. Однак їхня мета полягає у встановленні фактів виконання, пошуку шкідливої активності або відновленні артефактів, а не в аналізі ефективності об'єктного подання інформації.

Окремі дослідження демонструють, що повнота інтерпретації дампу залежить від наявності службової інформації та структури адресного простору. В. Dolan-Gavitt розглядає дерево VAD як спосіб отримати «погляд процесу» на фізичну пам'ять через діапазони віртуальних адрес [4]. Y. Otsuki, Y. Kawakoya, M. Iwamura, J. Miyoshi і К. Ohkubo досліджують відновлення стеків викликів із дамтів Windows x64, де результат залежить від правил розмотування стека і доступності потрібних фрагментів пам'яті [5]. Для керованих середовищ М. Manna, А. Case, А. Ali-Gombe і G. G. Richard III показують, що аналіз .NET та .NET Core потребує відновлення відомостей про керовану купу, типи, об'єкти та метадані середовища виконання [22; 65]. Отже, байтове збереження пам'яті саме по собі не гарантує повної об'єктної інтерпретації.

Перехід від байтового подання до об'єктної моделі досліджується у працях, присвячених абстрагуванню купи та візуальному аналізу пам'яті. М. Marron, С. Sanchez, Z. Su і М. Fahndrich запропонували підхід до абстрагування runtime heaps, у якому низькорівневі об'єкти об'єднуються в узагальнені структури для розуміння поведінки програми [47]. М. Weninger, L. Makor і Н. Mössenböck використовують метафору «міста пам'яті» для візуалізації еволюції heap memory, що полегшує виявлення зон концентрації пам'яті та структурного зростання [15]. Інструменти на зразок ClrMD забезпечують доступ до типів, адрес, розмірів, полів, масивів, колекцій і зв'язків між об'єктами у керованій купі [100]. Проте ці засоби переважно витягують або візуалізують факти; вони не задають критеріїв, за якими факт дублювання, надмірної ємності або службових витрат стає математично визначеним сценарієм надлишкового використання пам'яті.

Дослідження provenance-based endpoint detection and response також показують межу між збиранням фактів і їхньою інтерпретацією. F. Dong, S. Li, P. Jiang, D. Li, H. Wang та співавтори аналізують промисловий погляд на provenance-based EDR-інструменти і підкреслюють складність практичного використання великих обсягів фактів про події [26; 64]. Для аналізу пам'яті

ситуація подібна: навіть якщо вдається витягнути багато фактів про об'єкти, типи і посилання, потрібні критерії відбору релевантних ознак. Без таких критеріїв знімок пам'яті залишається великим діагностичним масивом, а не математичною моделлю надлишковості.

У суміжних напрямках пам'ять використовується також для перевірки цілісності та реконструкції поведінки застосунків. Т. Göbel, S. Maltan, J. Türr, Н. Baier і F. Mann пропонують ForTrace як фреймворк синтезу криміналістичних наборів даних [23; 66]. J. Wagner, M. Nissan і A. Rasin досліджують пам'ять баз даних для виявлення патернів кешування і перевірки журналів [24; 67]. Ці праці демонструють, що знімок пам'яті може відображати не лише стан операційної системи, а й стан прикладної логіки та сховищ даних. Водночас реконструкція подій, кешів або журналів не дорівнює формальному оцінюванню того, чи є подання даних у пам'яті економним.

Інструментальні системи отримання дамтів і діагностичних даних розв'язують задачу доступу до стану процесу, але не задають змісту подальшої інтерпретації. Microsoft Debug Diagnostic Tool використовується для збирання дамтів процесів IIS у станах зависання або аварій [18]. Такий засіб корисний для накопичення матеріалу аналізу, але не визначає, які об'єкти є надлишковими. Подібна межа характерна і для інструментів прозорі інспекції. К. Leach, С. Spensky, W. Weimer і F. Zhang розглядають transparent introspection як спосіб спостереження за системою без традиційної інструментації [25], а L. Zhou, J. Xiao, К. Leach, W. Weimer, F. Zhang і G. Wang розвивають цю ідею у Nighthawk [68]. Такі підходи підвищують спостережуваність, однак не формують критеріїв надлишкового представлення інформації у пам'яті процесу.

Формат знімка пам'яті має власні обмеження, які безпосередньо впливають на можливість подальшого аналізу. MiniDump може містити різні інформаційні потоки, і склад цих потоків визначається параметрами створення файла. Практики, описані М. Hewardt і D. Pravat, Т. Soulamі та С. Huffman, показують, що неповний дамт може бути достатнім для аналізу винятку або

стека, але недостатнім для відновлення всієї керованої купи [2; 3; 7]. Тому наявність файла .dmp ще не означає наявності повної множини об'єктів і зв'язків. Для задачі оцінювання надмірності це принципово: якщо частина купи або метаданих відсутня, алгоритмічний висновок може втратити коректність.

Отже, аналіз знімків пам'яті дає змогу перейти від зовнішніх метрик до фактичної структури процесу, але наявні підходи не завершують цей перехід побудовою формальної множини об'єктів, типів, значень, розмірів і зв'язків. Нерозв'язаною залишається задача подання знімка пам'яті як типізованої структури, на якій можна задавати предикати, критерії еквівалентності, функції оцінювання та алгоритмічні процедури пошуку надлишкового представлення інформації..

1.3 Витоки пам'яті та стабільне надмірне використання оперативної пам'яті

Найбільш розвинений напрям аналізу пам'яті пов'язаний із виявленням витоків. N. Mitchell і G. Sevitsky у LeakBot запропонували автоматизований підхід до діагностики витоків у великих Java-застосунках, де аналізуються утримувані об'єкти та шляхи посилань, що перешкоджають звільненню пам'яті [43]. M. Jump і K. S. McKinley у Cork розглядають динамічне виявлення витоків у мовах зі збиранням сміття, поєднуючи інформацію про типи, кількість об'єктів і динаміку їх накопичення [41; 42]. Перевага цих підходів полягає у спрямованості на конкретний клас дефектів: об'єкти залишаються досяжними, хоча більше не потрібні. Обмеження полягає в тому, що критерієм аналізу виступає утримання або зростання, а не надлишковість інформаційного подання.

Методи аналізу зростання структур даних поглиблюють leak-oriented підхід. M. Weninger, E. Gander і H. Mössenböck досліджують зміну структур даних у часі для полегшення виявлення витоків [75; 92]. С. Lou, С. Chen, Р. Huang та співавтори у системі RESIN аналізують виробничу хмарну інфраструктуру і використовують дампи пам'яті для локалізації витоків у сервісах, де ручне відтворення інцидентів є складним [37]. R. Kuznetsov, A. Marcolla і D. Khomenko застосовують машинне навчання для виявлення витоків у хмарній інфраструктурі за ознаками поведінки пам'яті [40]. Ці роботи розширюють можливості автоматичного аналізу, але зберігають орієнтацію на аномальне накопичення, тобто на зміну використання пам'яті в часі.

Поняття 'memory bloat' (збільшене використання пам'яті) відокремлює інший клас проблем від класичного витоку пам'яті. G. Novark, E. D. Berger і B. G. Zorn досліджують локалізацію як memory leaks, так і memory bloat, тобто ситуацій, у яких програма використовує більше пам'яті, ніж потрібно для виконання задачі [44; 51; 91]. K. Nguyen, K. Wang, Y. Bu, L. Fang і G. Xu розглядають memory bloat у керованих середовищах і показують, що надмірне використання може виникати без помилки звільнення пам'яті: причинами можуть бути дублювання, неефективне подання об'єктів, надлишкові структури або невдалий вибір контейнерів [79]. S. Lee аналізує вплив memory bloat на продуктивність і розглядає способи його зменшення через зміни у керуванні пам'яттю та повторному використанні об'єктів [80]. Цей напрям є ближчим до задачі аналізу надмірності, оскільки працює не лише з фактом зростання, а й із співвідношенням між корисним вмістом і витратами пам'яті.

Водночас memory bloat не є однорідним явищем. Частина випадків пов'язана з об'єктами, що могли б бути звільнені; інша частина – з об'єктами, які залишаються потрібними, але представлені неефективно. Наприклад, колекція може мати надлишкову ємність, рядкові значення можуть дублюватися у великій кількості фізичних екземплярів, а поле може бути оголошене типом із ширшим діапазоном, ніж потрібно для фактичних значень.

Для таких ситуацій відсутній монотонний приріст пам'яті, тому класичний критерій витoku не спрацьовує. Крім того, стабільне використання великого обсягу пам'яті може вважатися «нормальним» за даними моніторингу, хоча внутрішня структура знімка містить надлишкове подання інформації.

Підходи до трасування роботи збирача сміття також збагачують аналіз об'єктних систем. X. Qi у Elephant Tracks II розглядає фреймворк трасування garbage collection [49]. Такі інструменти дають інформацію про створення, переміщення, утримання й життєвий цикл об'єктів. Проте трасування має іншу цільову функцію: воно пояснює поведінку пам'яті у часі, а не оцінює співвідношення між інформаційним вмістом і фактичним обсягом пам'яті у фіксованому знімку. Для стабільної надлишковості потрібно аналізувати не лише історію об'єкта, а й еквівалентність його значення відносно інших об'єктів того самого або сумісного типу.

Суміжним є напрям аналізу «мертвих» або неефективно використовуваних об'єктів. R. Salkeld і G. Kiczales досліджують взаємодію з dead objects як спосіб виявлення ситуацій, коли об'єкти залишаються у виконанні, але їхній стан або зв'язки втрачають очікувану функціональну роль [48]. U. Degenbaev, J. Eisinger, K. Hara, M. Hlopko, M. Lippautz і H. Payer аналізують cross-component garbage collection, де складність утримання об'єктів пов'язана з межами між компонентами [35]. Ці підходи важливі для розуміння досяжності та життєвого циклу об'єктів, але вони знову працюють із проблемою звільнення або утримання. Вони не визначають, коли два досяжні об'єкти з однаковим значенням становлять надлишкове представлення інформації.

У мовах і платформах із довготривалими клієнтськими сесіями окремо досліджуються витoki JavaScript-застосунків. M. Rudafshani і P. A. S. Ward запропонували LeakSpot для виявлення й діагностики витоків пам'яті в JavaScript-застосунках [46]. A. Shahoor, S. Abdyldayev, H. Hong, J. Yi і D. Kim аналізують проактивне відлагодження витоків пам'яті в single page web applications [45]. Ці роботи показують, що витoki можуть мати різну природу

залежно від платформи і моделі виконання. Проте навіть у цьому розширеному контексті основною ознакою залишається неправильне утримання об'єктів або накопичення, а не стабільне існування надлишкових, але формально потрібних структур.

Методи, орієнтовані на витоки, дають сильні результати для випадків некоректного утримання об'єктів або аномального росту. Методи, орієнтовані на memory bloat, розширюють поле аналізу, однак часто потребують додаткового знання про програму, профіль навантаження або можливі перетворення структури даних. Нерозв'язаною залишається задача відокремлення стабільного, але надлишкового використання пам'яті від необхідного використання пам'яті за даними одного або кількох знімків. Для цього потрібні формальні критерії надмірності, які не зводяться ні до витоку, ні до зовнішньої аномалії часових рядів.

1.4 Алгоритмічні підходи до виявлення дублювання, надлишкових структур і неефективного подання даних

Алгоритмічний аналіз надлишковості пам'яті пов'язаний із питанням, які об'єкти або структури даних можна вважати еквівалентними за інформаційним вмістом і які перетворення не змінюють змісту збережених даних. J. L. Bentley у класичній праці про ефективні програми розглядає економію ресурсів як наслідок добору структур даних, алгоритмів і подання інформації [89; 93]. Цей підхід важливий тим, що відокремлює оптимізацію від локального прискорення окремої операції: неефективність може бути закладена у способі зберігання даних. Проте такі рекомендації мають загальний характер і не визначають алгоритму пошуку конкретних кандидатів на оптимізацію у знімку пам'яті великого програмного процесу.

Окремий клас досліджень стосується дублювання незмінюваних або майже незмінюваних значень. К. Kawachiya, К. Ogata і Т. Onodera аналізують неефективності у Java strings і показують, що рядкові об'єкти можуть утворювати значні накладні витрати через дублювання, службові поля та спосіб зберігання символів [54]. С. Naack, Е. Poll, J. Schäfer і А. Schubert розглядають незмінювані об'єкти для Java-подібної мови, що є важливим для коректного повторного використання значень [95]. Перевага роботи з immutable-об'єктами полягає в тому, що однаковість інформаційного вмісту може бути використана без ризику зміни стану одного екземпляра через інший. Обмеження полягає у необхідності довести або встановити незмінюваність типу, а також у складності перевірки еквівалентності для мільйонів об'єктів у великих знімках пам'яті.

Патерни повторного використання об'єктів та пулів об'єктів пропонують інженерний спосіб зменшення кількості створень і звільнень. А. Freeman описує object pool як шаблон, що зберігає готові до повторного використання об'єкти [58]. І. Т. Christou і S. Efremidis аналізують доцільність застосування пулів і підкреслюють, що вигаш залежить від контексту, вартості створення об'єкта, конкуренції між потоками та режиму використання [94]. Такі підходи можуть зменшувати витрати пам'яті або часу, але вони не відповідають на питання, як автоматично знайти у знімку пам'яті ті класи об'єктів, для яких повторне використання або інтернування є допустимим. Тому між ідеєю повторного використання і процедурою виявлення кандидатів залишається алгоритмічний розрив.

Надлишковість може виникати не лише через дублювання значень, а й через службові витрати структур даних. У керованих середовищах хеш-колекції, списки, словники та масиви мають внутрішню місткість, службові поля, покажчики або масиви допоміжних записів. К. Kokosa у праці про .NET memory management описує особливості керованої купи, службових витрат об'єктів, поколінь збирання сміття та аналізу розподілу пам'яті [50]. Ці відомості потрібні для інтерпретації фактичного розміру об'єктів, але вони не

задають критерію, коли службова вартість конкретної структури є надмірною відносно корисного вмісту. Для цього потрібно поєднати знання про внутрішню структуру контейнера з даними знімка: кількістю елементів, ємністю, розмірами масивів, типами ключів і значень.

Інший напрям зменшення пам'яті пов'язаний зі стисканням і компактним поданням даних. P.-A. Tsai і D. Sánchez запропонували об'єктно-орієнтовану стиснену ієрархію пам'яті, у якій одиницею стискання є об'єкт, а не cache line [82]. Q. Xia, H. Ji, Y. Zhou і N. S. Kim аналізують апаратно прискорене стискання пам'яті у kernel-space з використанням Intel QAT [81]. Ці підходи показують, що стискання може зменшувати фізичне споживання пам'яті, однак воно не вирішує задачу вибору даних для стискання за змістовим критерієм. Якщо застосовувати стискання без аналізу інформаційного вмісту, можна отримати технічне зменшення байтового подання, але не пояснити, чому саме ці об'єкти або масиви є надлишковими і який очікуваний ефект має допустиме перетворення.

Алгоритмічна складність пошуку дублікатів є окремою проблемою. Якщо порівнювати всі об'єкти попарно, кількість порівнянь швидко стає неприйнятною для промислових знімків, що містять мільйони об'єктів. Тому потрібні попередні ознаки групування: тип, розмір, хеш інформаційного вмісту, довжина рядка, структура полів, діапазон значень або службові характеристики контейнера. Таке групування зменшує простір пошуку, але створює вимогу коректності: швидка ознака не повинна призводити до втрати потенційних дублікатів або до хибного об'єднання різних значень. У наявних інженерних підходах ця вимога часто залишається неформальною.

Системи основної пам'яті та механізми checkpointing також використовують знімки або копії стану, але з іншою метою. J. Park, Y. Lee, H. Yeom і Y. Son досліджують memory efficient fork-based checkpointing для in-memory database systems [60]. A. Kemper і T. Neumann у HyPer використовують virtual memory snapshots для поєднання OLTP та OLAP у main memory database [63]. Ці підходи підтверджують, що знімок стану може бути продуктивним

технічним механізмом для високонавантажених систем. Проте їхній об'єкт – забезпечення швидкого доступу, ізоляції або відновлення, а не пошук надлишкового інформаційного подання в об'єктах виконуваного застосунку.

Підходи до оптимізації метаданих у інформаційних системах демонструють споріднену постановку задачі зменшення службового опису. О. Tkachenko і А. Lemeshko аналізують оптимізацію обсягу метаданих у сучасних інформаційних системах [22]. О. Tkachenko, О. Holubenko, V. Vlasenko і А. Antonenko розглядають методи оптимізації використання оперативної пам'яті в процесах підвищення швидкодії комп'ютерних систем [13]. Ці джерела релевантні як приклади оцінювання надлишкових службових витрат, однак вони не розв'язують задачу об'єктно-рівневого аналізу керованої купи. Для знімків пам'яті потрібно визначати не лише наявність метаданих, а й відношення між службовими структурами, об'єктами, контейнерами і фактичними значеннями. Саме тому ідеї оптимізації метаданих [22] та загального зменшення використання оперативної пам'яті [13] мають бути переведені з рівня інформаційної системи на рівень окремих об'єктів, типів, полів, масивів і колекцій.

Проблема службових витрат особливо помітна для контейнерів і колекцій. Хеш-таблиці, словники, списки та інші структури зазвичай мають запас ємності, допоміжні масиви, покажчики, службові поля і правила розширення. У багатьох випадках така надмірність є обґрунтованою платою за швидкість операцій додавання або пошуку. Проте за малих розмірів колекцій або за нерівномірного розподілу ключів службова вартість може перевищувати корисний інформаційний вміст. У літературі про продуктивність ці витрати часто розглядаються як властивість реалізації структури даних, але для аналізу знімка пам'яті потрібно перетворити їх на кількісний критерій, який обчислюється для конкретних об'єктів.

Кешування є окремим випадком, де надмірність може бути як корисною, так і шкідливою. С. К. Сімакін і Л. М. Божуха пропонують метод інтелектуального керування часом життя кешу вебсервісів на основі

алгоритмів навчання з підкріпленням [19]. Такий підхід показує, що рішення про зберігання даних у пам'яті має залежати від контексту використання і прогнозованої користі. Однак для аналізу знімка пам'яті цього недостатньо. Необхідно встановити, чи є конкретні кешовані об'єкти інформаційно корисними, чи вони дублюють уже наявні значення, мають надлишкову місткість або зберігаються у структурах, де частка службових метаданих є непропорційно великою порівняно з обсягом корисної інформації. Подібна проблема надлишкових метаданих та їхнього впливу на ефективність інформаційних систем розглядається у роботі О. Ткаченка та А. Лемешка [22], тоді як С. К. Сімакін і Л. М. Божуха акцентують увагу на необхідності адаптивного керування кешованими даними залежно від контексту використання [19].

Отже, алгоритмічні підходи до зменшення використання пам'яті вже містять окремі рішення для дублювання, immutable-об'єктів, пулів, хеш-структур і стискування. Нерозв'язаною залишається задача об'єднання цих рішень у формальну процедуру аналізу знімка пам'яті: потрібно визначити класи кандидатів, задати еквівалентність за інформаційним вмістом, встановити допустимість перетворення та обчислити кількісну оцінку можливого зменшення пам'яті. Без такої процедури окремі оптимізації залишаються набором інженерних прийомів, а не результатом аналізу формалізованої структури даних.

1.5 Онтологічні, логічні, rule-based та експертні підходи до інтерпретації діагностичних ознак

Після виділення діагностичних ознак виникає задача їх інтерпретації. У задачах аналізу складних систем для цього застосовуються логічні, rule-based, expert systems та ontology-based підходи. Їхня спільна ідея полягає у переході від числових або структурних ознак до пояснюваного висновку через правила,

відношення, класи понять або логічні процедури. Перевага такого підходу полягає в інтерпретованості: висновок можна пов'язати з конкретними умовами, ознаками або зв'язками між сутностями. Обмеження полягає у залежності від якості бази фактів. Якщо факти не мають формального змісту, правило стає евристичним описом, а не процедурою обґрунтованого діагностичного виведення.

V. Lakhno, S. Kazmirchuk, Y. Kovalenko, L. Myrutenko і T. Zhmurko запропонували модель розпізнавання аномалій і кібератак на основі логічних процедур, матриць покриття ознак і поняття елементарного класифікатора [30]. У цьому підході ознаки аномалій подаються так, щоб їх можна було використати для розпізнавання класів загроз. Перевагою є формалізація зв'язку між ознаками та класом події. Водночас така модель працює з множиною попередньо визначених ознак і не розглядає задачу вилучення цих ознак зі структури пам'яті програмного процесу. Для аналізу знімків пам'яті це означає, що логічний висновок може застосовуватися лише після побудови формального шару об'єктів, типів, значень і кількісних характеристик.

V. Protsiuk розглядає виявлення аномалій у сенсорних даних процесу буріння нафтогазових свердловин за умов невизначеності та використовує правила з коефіцієнтами впевненості [31]. Такий підхід демонструє, що rule-based модель може враховувати неповноту, неточність і неоднозначність даних, а діагностичний висновок може мати не лише бінарний, а й зважений характер. Його обмеження полягає в залежності від природи сенсорних ознак: правила працюють із часовими або вимірювальними параметрами процесу, тоді як знімок пам'яті задає дискретну об'єктну структуру з типами, розмірами, полями, посиланнями і службовими витратами. Отже, для пам'яті потрібна інша база фактів, хоча загальний принцип правил із вагами або впевненістю є релевантним.

У працях, пов'язаних із класифікацією та діагностикою ІТ-систем, правила і моделі ознак застосовуються для підтримки рішень на рівні сервісів або конфігурацій. K. Zolotko і D. Krasnoshapka розглядають управління та

діагностику надання IT-сервісів, де діагностичний висновок пов'язується з ознаками стану сервісу [24]. О. М. Kiseleva, О. М. Prytomanova і О. А. Filat застосовують кластеризаційні методи DBSCAN та OPTICS для виявлення мережових загроз [26]. М. Ievlanov, N. Vasilcova, I. Panforova, В. Moroz, А. Martynenko і D. Moroz порівнюють ієрархічні методи кластеризації для ранньої ідентифікації конфігураційних одиниць IT-продукту [51]. Ці джерела релевантні не як прямі методи аналізу пам'яті, а як приклади формалізації ознак і переходу від даних до класифікаційного або діагностичного висновку. Їхнє обмеження для задач пам'яті полягає в тому, що вони не задають критеріїв надлишковості об'єктного подання. Разом з тим вони підтверджують доцільність групування й ієрархізації ознак [26; 51], що може бути використано лише після того, як зі знімка пам'яті формально виділено типи, значення, розміри, зв'язки та службові структури.

Онтологічні підходи застосовуються тоді, коли потрібно уніфікувати терміни, задати класи сутностей, описати відношення між ними та забезпечити логічне виведення. D. K. Pattipati, R. Nasre і S. K. Puligundla запропонували OPAL – фреймворк ontology-based program analysis, у якому синтаксична інформація програми подається RDF-трійками, а додаткові знання про бібліотеки та предметну область використовуються для отримання семантичних трійок і подальшого аналізу [91]. Перевага онтологічного підходу полягає у формалізації понять і відношень між програмними сутностями. Однак онтологія сама по собі не задає кількісної оцінки надмірності, не визначає алгоритму пошуку дублікатів, не встановлює критерію неефективного подання даних і не обчислює очікуване зменшення пам'яті після допустимого перетворення.

Для аналізу знімків пам'яті цей висновок має принципове значення. OPAL демонструє, що програмні сутності можна перевести у формалізований простір відношень [91], але у задачі надмірного використання пам'яті потрібно розширити такий простір кількісними атрибутами: розміром об'єкта, ємністю колекції, значенням поля, належністю до групи еквівалентності та можливим

ефектом перетворення. Інакше онтологічний опис залишиться термінологічним шаром, який пояснює поняття, але не виконує діагностичного обчислення.

Тому найбільш перспективною для аналізу пам'яті є не ізольована експертна система і не ізольована онтологія, а зв'язка трьох рівнів. Перший рівень – формальна структура знімка пам'яті, де визначено об'єкти, типи, значення, розміри та зв'язки. Другий рівень – алгоритмічні критерії, що обчислюють еквівалентність, надлишковість, службову вартість і можливий ефект. Третій рівень – правила або онтологічні відношення, які інтерпретують отримані факти і формують діагностичний висновок. Саме відсутність узгодженості між цими рівнями є основною межею наявних підходів.

Онтологічна модель також не є самодостатнім розв'язанням задачі пам'яті. Вона може описати класи сутностей, наприклад об'єкт, тип, поле, масив, колекцію, посилання, значення, розмір або сценарій надлишковості. Вона може задати відношення між ними, наприклад «є екземпляром», «містить поле», «посилається на», «має значення», «належить до групи еквівалентності». Однак для кількісного висновку потрібно додати функції і процедури: обчислення розміру, перевірку еквівалентності, оцінювання службових витрат, порівняння з безнадлишковим поданням. Без цього онтологія описує словник предметної області, але не обчислює надлишковість.

У цьому контексті показовим є використання ваг, коефіцієнтів довіри або мір впевненості. У сенсорних даних, які аналізує V. Protsiuk, невизначеність виникає через пропуски, шум, дублікати та неоднозначність вимірювань [31]. У пам'яті програмного процесу невизначеність має іншу природу: неповнота знімка, відсутність частини метаданих, зміна доменного діапазону значень після зняття дампу або неоднозначність допустимого перетворення. Це означає, що ваги правил можуть бути корисними, але їх не можна переносити без адаптації. Вага має відображати не лише частоту ознаки, а й достовірність об'єктної інтерпретації та допустимість запропонованого перетворення.

Класичні експертні системи мають сильну сторону саме там, де необхідно пояснювати висновок. У задачах продуктивності це важливо, оскільки рекомендація щодо зміни структури даних не може бути лише числовою. Вона має містити умову застосування, підставу, очікуваний ефект і межі допустимості. Наприклад, правило, яке виявляє надлишкову ємність колекції, повинно відрізнити тимчасовий запас для майбутнього росту від стабільно непотрібної ємності. Правило, що вказує на дублювання рядків, повинно враховувати незмінюваність значень і допустимість спільного використання. Отже, rule-based інтерпретація корисна лише тоді, коли перед нею існує формальна модель фактів.

Це також стосується метрик і прогнозів, що використовуються в ІТ-сервісах: метрики ефективності [23], діагностика сервісних станів [24] та прогнозування навантаження [71] можуть бути джерелом контексту для правила, але не замінюють фактів, отриманих із пам'яті. Для надлишкового подання даних правило має спиратися на обчислені ознаки самого знімка, а не лише на зовнішній опис сервісу або часовий ряд.

Отже, логічні, rule-based, експертні та онтологічні підходи дають засоби інтерпретації ознак, пояснення висновків і формалізації знань. Нерозв'язаною для аналізу знімків пам'яті залишається задача поєднання цього рівня інтерпретації з попередньою математичною моделлю даних. Правило або онтологічне відношення має спиратися не на неформальний опис проблеми, а на факти, одержані зі знімка: типи, значення, розміри, зв'язки, еквівалентність об'єктів, корисний інформаційний вміст і кількісну оцінку надлишковості. Без такого шару продукційний або онтологічний висновок не може бути достатньо перевірним.

1.6 Невирішені задачі та висновки до розділу

Проведений огляд літературних джерел свідчить, що існуючі методи аналізу продуктивності програмного забезпечення переважно орієнтовані на фіксацію зовнішніх симптомів або відхилень у часових рядах. Водночас багато аспектів, пов'язаних із внутрішнім структурним поданням даних в оперативній пам'яті, залишаються недостатньо дослідженими. З огляду на це, невирішеними є такі наукові задачі:

1. Побудова єдиної формальної моделі знімка пам'яті як скінченної типізованої структури об'єктів, значень, розмірів і зв'язків. Відомі підходи, що базуються на метриках ефективності, діагностиці IT-сервісів та прогнозуванні серверного навантаження [23; 24; 71], оперують агрегованими показниками і не визначають формального критерію надлишкового подання інформації. Засоби інструментального аналізу переважно використовують знімок пам'яті як джерело фактів без задання математичних відношень та функцій оцінювання.

2. Дослідження memory leak переважно орієнтовані на динаміку споживання ресурсу в часі, тоді як memory bloat фіксує ширший клас неефективного використання пам'яті. Для стабільного надлишкового подання інформації критерії, що базуються безпосередньо на структурі та інформаційному вмісті об'єктів, у межах розглянутих підходів не сформовано.

3. Розроблення алгоритмічних процедур переходу від фактів знімка до класів надмірності (дублювання незмінюваних об'єктів, надлишкова місткість колекцій, службові витрати хеш-структур, надлишковий діапазон типів, стискання даних зі збереженням змісту). Відомі ізольовані підходи до оптимізації оперативної пам'яті [13; 19; 22] потребують формального об'єднання у процедури з можливістю обчислення числових оцінок очікуваного зменшення обсягу пам'яті.

4. Поєднання кількісних оцінок із продукційною інтерпретацією діагностичних висновків. Застосування кластеризаційних методів для конфігураційних одиниць [26; 51] та ontology-based program analysis у OPAL [91] підтверджує доцільність формалізованого подання ознак. Проте за

відсутності формальної моделі даних та алгоритмічного пошуку продукційні правила залишаються евристичними і не дозволяють кількісно оцінити ефект від перетворення подання даних.

1.7 Висновки до розділу

На основі проведеного огляду та аналізу існуючих методів виявлення проблем продуктивності програмного забезпечення зроблено такі висновки:

1. Встановлено, що задача аналізу надмірного використання пам'яті не може бути повною мірою розв'язана методами моніторингу, прогнозування або загальними метриками продуктивності. Зазначені методи фіксують відхилення поведінки системи від очікуваної [24; 71] та забезпечують порівняння рішень [23], однак виявлення причин надлишкового подання інформації вимагає порівняння об'єктів на рівні їх значень, типів, розмірів і зв'язків.

2. Обґрунтовано необхідність функціонального відокремлення знімка пам'яті від інструментальних засобів його отримання. Показано, що подання знімка як скінченної типізованої структури об'єктів створює математичну основу для задання відношень еквівалентності, предикатів надмірності, оцінювальних функцій та алгоритмічних процедур.

3. Виявлено, що межі класичного поняття *memory leak* не охоплюють ситуацій надлишкового подання об'єктів, які залишаються логічно потрібними. Для високонавантажених керованих систем критичним є клас дефектів *memory bloat*, за якого обсяг пам'яті не демонструє монотонної динаміки витоку, збирач сміття функціонує коректно, але значний ресурс витрачається на повтори, службові структури та невідповідність фактичного діапазону значень оголошеному типу.

4. Обґрунтовано потребу формального об'єднання окремих оптимізаційних прийомів (інтернування рядків, *object pool*, звуження типів,

аналіз хеш-колекцій, стискання масивів, керування часом життя кешу та зменшення службових метаданих) [13; 19; 22]. Зазначені перетворення доцільно розглядати як множину допустимих дій із чітко визначеними умовами застосування, критеріями збереження інформаційного вмісту та обмеженнями коректності.

5. Показано, що якість засобів експертної інтерпретації (rule-based, clustering-based та ontology-based підходів) [26; 51; 91] безпосередньо залежить від наявності формальної бази фактів. Застосування обчислених ознак на основі чітко визначеної типізованої структури об'єктів дозволяє перетворити евристичні правила на обґрунтовані діагностичні сценарії з відповідними кількісними оцінками.

Основні результати розділу опубліковані в [6, 16, 19, 20, 21, 26, 28, 53, 55, 69].

Розділ 2. МАТЕМАТИЧНІ МОДЕЛІ ПРЕДСТАВЛЕННЯ ІНФОРМАЦІЇ В ПАМ'ЯТІ ДЛЯ АНАЛІЗУ ПРОДУКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Визначення знімку пам'яті та базові поняття

Знімок пам'яті (memory dump) – це зафіксований стан оперативної пам'яті програмного процесу у визначений момент часу, що містить увесь масив даних, з якими працює програма [84].

Фізично знімок пам'яті являє собою структурований набір байтів, який відображає вміст керованої пам'яті процесу [20]. Кожен об'єкт (o) у адресному просторі програми характеризується унікальною адресою (a_o) в пам'яті. В середовищі виконання .NET (Common Language Runtime, CLR) кожен об'єкт за правилами середовища має стандартний заголовок метаданих за своєю адресою, який містить службову інформацію (зокрема, покажчик на тип об'єкта). Тип визначає склад і структуру полів, які містить екземпляр об'єкта, а також дозволяє обчислити повний розмір цього екземпляра (включно з розміром заголовку метаданих). Таким чином, кожен об'єкт у пам'яті можна розглядати як впорядковану сукупність елементів: службової інформації (заголовку), даних про тип і набору значень полів.

Тип (t) – категорія даних, що визначає допустиму структуру і представлення даних: склад і розміри полів, діапазони значень та операції, які можуть бути виконані над цими значеннями. У мові C# кожна змінна, константа, вираз, параметр і повернуте значення мають певний тип. Типи можуть бути вбудованими (примітивними) – наприклад, *Int32*, *String* – або визначеними користувачем (складні типи, класи й структури).

Об'єкт (o) – це екземпляр деякого типу t , для зберігання якого виділено область пам'яті в керованій купі. Об'єкт складається з сукупності значень своїх полів, організованих за схемою, що визначається типом. Кожен об'єкт у CLR-купі ідентифікується посиланням на свою адресу (a_o) – саме значення

адреси початку об'єкта служить унікальним ідентифікатором екземпляра під час аналізу знімку. В середовищі CLR кожен об'єкт починається з невеликого заголовку фіксованого розміру, що містить службову інформацію (наприклад, покажчик на таблицю методів типу, індикатори стану синхронізації тощо). Після заголовку у пам'яті розміщуються поля об'єкта згідно з описом типу. Таким чином, фізично екземпляр об'єкта може бути представленим як кортеж з адреси, типу та набору значень полів.

Кожен об'єкт у CLR-купі вирівнюється за адресою, кратною розміру машинного слова (8 байт на 64-розрядній архітектурі). Мінімальний розмір будь-якого об'єкта (навіть такого, що не має власних полів) становить 24 байти на 64-розрядній архітектурі. Ці 24 байти містять службовий заголовок об'єкта та можливе вирівнювання (резервування місця) під розмір слова.

2.2 Формальна модель знімка пам'яті та постановка задачі ідентифікації надмірності пам'яті

Знімок пам'яті програми в момент часу u подамо як скінченну послідовність байтів, або як часткове відображення адресного простору в множину байтових значень: [84, 85]:

$$B_u = \{b_i\}, \quad (2.1)$$

де b_i – байти, з яких складається знімок; $i = \overline{1, n}$; n – кількість байтів в знімку пам'яті.

Оскільки кожний знімок пам'яті фіксує стан досліджуваного процесу в певний момент часу u , усі множини, функції та характеристики, що визначаються на основі цього знімка, розглядаються як залежні від u . Надалі, якщо момент часу є фіксованим і не потребує окремого акцентування, аргумент u може опускатися для спрощення запису.

У межах дисертаційної роботи задача ідентифікації надмірного використання пам'яті розглядається як задача виявлення надлишкового подання інформації у формалізованій структурі знімка пам'яті.

На основі аналізу байтового представлення формується множина об'єктів:

$$O = \{o_j\}, \quad (2.2)$$

де $j = \overline{1, m}$, m – кількість об'єктів в знімку пам'яті.

Кожен об'єкт $o_j \in O$ є екземпляром деякого типу t_k з множини типів:

$$T = \{t_k\}, \quad (2.3)$$

де $k = \overline{1, r}$, r – кількість типів в знімку пам'яті.

Тип об'єкта $t_k \in T$ визначає структуру об'єкта (склад і типи його полів), розмір об'єкта в пам'яті s , спосіб інтерпретації байтового представлення та семантику значень об'єкта, а також множину допустимих операцій над ним.

Кожен об'єкт $o_j \in O$ є екземпляром типу $t_k \in T$ та зображується в такий спосіб:

$$\begin{cases} o_j = (a, t_k, s, FLD), \\ FLD(o) = \{f_l(o) = (name_l, type_l, value_l, size_l)\} \end{cases} \quad (2.4)$$

де a - адреса об'єкта в пам'яті; t_k - тип об'єкта; s - розмір об'єкта у байтах; $FLD(o)$ - множина структурованих описів полів об'єкта o ; $f_l(o) = (name_l, type_l, value_l, size_l)$ – кортеж, що задає характеристики l -го поля об'єкта.

Кожному об'єкту відповідає підмножина байтів:

$$bytes(o) \subseteq B, \quad (2.5)$$

причому

$$|bytes(o)| = s(o). \quad (2.6)$$

Оскільки кожен об'єкт визначається послідовністю байтів, введемо відношення еквівалентності:

$$\begin{cases} o_i \sim o_j \Leftrightarrow bytes(o_i) = bytes(o_j) \\ O_t(u) = \{o \in O(u) : \tau(o) = t\} - \text{клас за типом} \\ C_{t,v}(u) = \{o \in O_t(u) : Val_t(o) = v\} - \text{клас за значенням} \end{cases}, \quad (2.7)$$

де $\tau: O(u) \rightarrow T(u)$ функція, яка кожному об'єкту ставить у відповідність його тип. Це відношення задає розбиття множини O на класи еквівалентності $\{C_1, C_2, \dots, C_k\}$.

Кожен клас C_k містить усі об'єкти знімка пам'яті, що належать до відповідного типу t_k :

$$C_k = \{o \in O \mid \tau(o) = t_k\}. \quad (2.8)$$

Таким чином, об'єкт інтерпретується як структурований фрагмент пам'яті з визначеною семантикою, яка задає правила відповідності між байтовим представленням об'єкта та його змістовною інтерпретацією у предметній області. Семантика визначає, яким чином значення окремих байтів та їх сукупностей відображаються у значення полів об'єкта, які інваріанти та обмеження накладаються на ці значення, а також які операції над об'єктом є допустимими. Таким чином забезпечено перехід від синтаксичного рівня представлення даних (послідовності байтів) до їх змістовного тлумачення, що дозволяє використовувати об'єкти у процесі аналізу та виявлення аномалій.

Інтерпретація знімку пам'яті $M(u)$ у момент часу u полягає у відображенні неструктурованого масиву байтів у формалізовану систему об'єктів, типів та потоків виконання. Така інтерпретація базується на аналізі стеків викликів, стану регістрів процесора, скануванні адресного простору пам'яті та ідентифікації типів об'єктів.

Введемо оператор інтерпретації:

$$I: B \rightarrow (O, T, TH), \quad (2.9)$$

де $TH = \{\theta_l\}$ – множина потоків виконання, $l = \overline{1, d}$, d – кількість потоків виконання.

Результат інтерпретації визначає модель знімку пам'яті в вигляді:

$$M(u) = (B, O, T, TH), \quad (2.10)$$

де $B = \{b_i\}$ - множина байтів пам'яті; $O = \{o_j\}$ - множина об'єктів, виділених у пам'яті; $T = \{t_k\}$ - множина типів об'єктів; $TH = \{\theta_l\}$ - множина потоків виконання.

Нехай знімок пам'яті у момент часу u задається моделлю $M(u)$, що містить множину байтів пам'яті, множину об'єктів $O(u)$, множину типів $T(u)$, множину потоків виконання та відображення, які задають типізацію, розміри, значення полів і зв'язки між об'єктами.

Для кожного класу сценаріїв надмірності вводяться предикати виявлення та функції кількісного оцінювання, які визначають наявність надлишкового представлення, оцінку потенційного зменшення обсягу пам'яті та належність виявленого випадку до відповідного класу надмірного використання пам'яті.

Результатом розв'язання задачі є множина сценаріїв надмірного використання пам'яті, визначених на $M(u)$, разом із кількісними оцінками можливого зменшення використання оперативної пам'яті.

Розглянемо приклад об'єкту o' типу $t = System.String$ (рядок), який містить значення "true" [84]. На рис.2.1 цифрами позначено такі елементи: 1 – адреса $a(o')$, 2 – тип $t(o')$, 3 – розмір $s(o')$, 4 – $FLD(o')$.

```

0:000> !mdt 225daffe2c0
0000225daffe2c0 (System.String) length=4, String="true"
0:000> !do 225daffe2c0 1
Name: System.String
MethodTable: 00007ff8d05a59c0 2
EEClass: 00007ff8d0582ec0
Size: 3 34(0x22) bytes
File: C:\Windows\Microsoft.Net\assembly\GAC_64\mscorlib\v4.0.4.0.0_b77a5c561934e089\mscorlib.dll
String: true
Fields:
      MT      Field      Offset      Type VT      Attr      Value Name      4
00007ff8d05a85a0 4000283      8      System.Int32 1 instance      4 m_stringLength
00007ff8d05a6838 4000284      c      System.Char 1 instance      74 m_firstChar
00007ff8d05a59c0 4000288     e0      System.String 0 shared      static Empty
>> Domain:Value 0000221f9b0b680:NotInit 0000226fd0157a0:NotInit <<
0:000> du 225daffe2cc
0000225`daffe2cc "true"
0:000> db 225daffe2cc
0000225`daffe2cc 74 00 72 00 75 00 65 00-00 00 00 00 00 00 00 00 t.r.u.e.....
0000225`daffe2dc 00 00 00 00 00 00 00 00-00 00 00 00 c0 59 5a d0 .....YZ.

```

Рис. 2.1. Приклад об'єкту у знімку пам'яті

Згідно введеного зображення об'єкта (2.4):

$$\left\{ \begin{array}{l} a(o') = 25daffe2c0 \\ t(o') = System.String \\ s(o') = 34 \text{ bytes} \\ FLD(o') = \{m_{stringLength}, m_{fisrtChar}\} \\ o' \in O \end{array} \right. \quad (2.11)$$

Варто відзначити, що розмір 34 байти є власним розміром об'єкту без урахування вирівнювання за розміром вказівника.

Побудована формальна модель $M(u)$ є основою для подальшого введення числових характеристик ефективності використання пам'яті, операцій над множиною об'єктів та класифікації сценаріїв надмірного використання пам'яті.

2.3 Оцінювання ефективності використання пам'яті.

На основі побудованої моделі знімку пам'яті $M(u) = (B, O, T, TH)$ на множині об'єктів знімку пам'яті вводяться узагальнюючі числові характеристики, що дозволяють кількісно оцінити ефективність використання пам'яті.

Сумарний обсяг пам'яті, зайнятий об'єктами, включеними до моделі знімка, дорівнює сумі розмірів усіх об'єктів у знімку:

$$V = \sum_{o \in O^u} s(o) \quad (2.12)$$

Для множини незмінюваних об'єктів введемо відношення еквівалентності за інформаційним вмістом:

$$o_i \sim_V o_j \Leftrightarrow \tau(o_i) = \tau(o_j) \wedge Val(o_i) = Val(o_j),$$

де $Val(o)$ задає інформаційний вміст об'єкта. Саме це відношення використовується для визначення безнадлишкового представлення.

Теорема 1. Нехай O_{imm} - множина об'єктів з властивістю незмінності, а $O^* \subseteq O_{imm}$ - множина представників класів еквівалентності, породжених відношенням еквівалентності за інформаційним вмістом \sim_v . Тоді обсяг пам'яті V_{unique} , визначений на O^* , є мінімальним серед усіх підмножин $\tilde{O} \subseteq O$, що містять принаймні по одному представнику з кожного класу еквівалентності.

Доведення: Кожен клас еквівалентності містить об'єкти з однаковим байтовим представленням і, відповідно, однаковим розміром. Будь-яка підмножина \tilde{O} , що покриває всі класи, повинна містити не менше одного елемента з кожного класу. Множина O^* містить рівно по одному представнику з кожного класу, отже має мінімальну потужність серед таких підмножин. Оскільки всі представники одного класу еквівалентності мають однаковий розмір, заміна одного представника іншим не змінює сумарний обсяг пам'яті. Водночас додавання нових елементів до класу еквівалентності призводить до збільшення сумарного обсягу пам'яті без збільшення інформаційного вмісту. Тому величина

$$V_{unique} = \sum_{o \in O^*} s(o). \quad (2.13)$$

задає мінімальну нижню межу обсягу пам'яті, достатнього для представлення інформаційного вмісту множини об'єктів без дублювання.

Для оцінки надмірного використання пам'яті введемо коефіцієнт надмірності η , який зображується таким відношенням:

$$\eta = \frac{V^u}{V_{unique}^u} \quad (2.14)$$

Теорема 2. Для коефіцієнта надмірного використання пам'яті виконується нерівність $\eta \geq 1$, причому $\eta = 1$ тоді і тільки тоді, коли в множині O відсутні дублікати.

Доведення. Оскільки $O^* \subseteq O_{imm}$, маємо $V_{unique} \leq V$.

Звідси безпосередньо випливає, що $\eta = V/V_{unique} \geq 1$.

Рівність $\eta = 1$ можлива лише тоді, коли $V = V_{unique}$, тобто кожен клас еквівалентності містить рівно один елемент, що означає відсутність дублювання.

Середня кратність дублювання μ_t є кількісною характеристикою, що відображає, наскільки часто в пам'яті повторюються об'єкти з однаковими значеннями для заданого типу даних t . Вона дозволяє оцінити ступінь надмірності представлення інформації у знімку пам'яті та є ключовим показником для виявлення прихованого дублювання.

Обчислення цієї характеристики здійснюється для кожного типу об'єктів окремо і базується на аналізі всіх об'єктів даного типу, що містяться у знімку пам'яті.

Для фіксованого типу $t \in T$ визначається множина всіх об'єктів цього типу, при цьому об'єкти можуть відрізнятися своїм значенням у пам'яті:

$$O_t = \{o \in O_{imm} \mid type(o) = t\}.$$

Далі серед цих об'єктів відокремлюються унікальні значення, на основі відношення еквівалентності множина O_t розбивається на класи еквівалентності, де кожен клас містить об'єкти з ідентичним вмістом у пам'яті. Кількість усіх об'єктів типу t позначається як:

$$n_t = |O_t|,$$

А кількість унікальних значень обчислюється як:

$$v_t = |O_t / \bar{O}|$$

Тоді середня кратність дублювання μ_t для типу $t \in T$ визначається як відношення загальної кількості об'єктів до кількості унікальних значень:

$$\mu_t = n_t / v_t. \quad (2.15)$$

Середня кратність дублювання визначає, скільки разів у середньому кожне унікальне значення об'єкта повторюється в пам'яті. Якщо $\mu_t = 1$, це означає відсутність дублювання для даного типу. Якщо $\mu_t > 1$, це свідчить про наявність повторних екземплярів одних і тих самих значень, причому зі

зростанням μ_t рівень надмірності використання пам'яті для цього типу збільшується.

Введені характеристики є функціями, що визначені на множині об'єктів знімку пам'яті, вони здійснюють агрегування локальних параметрів об'єктів, зокрема їх розмірів, у глобальні показники стану пам'яті.

Введені числові характеристики моделі мають низку важливих властивостей, які підтверджують їхню придатність для оцінювання ефективності використання пам'яті. Зокрема, мають місце такі твердження.

2.4 Операції над об'єктами відображеними в знімку пам'яті

Маємо множину об'єктів O^u та множину типів T , які відображені у знімку пам'яті знятому у момент часу u .

Враховуючи, що кожен об'єкт $o \in O^u$ (2.2) є екземпляром типу $t_k \in T$ (2.3), введемо функцію групування G_1 [84], яка здійснює факторизацію множини об'єктів за типами та визначає потужності відповідних класів еквівалентності:

$$G_1: O^u \rightarrow (t_k^u, c_k^u), k = \overline{1, r}, \quad (2.16)$$

де r – кількість типів в знімку пам'яті.

Результатом операції групування є система пар (t_k, c_k) , яка однозначно визначає розбиття множини об'єктів на класи еквівалентності за типами.

2.5 Класифікація ситуацій надмірного використання пам'яті

На основі введеного розбиття множини об'єктів знімка пам'яті на класи еквівалентності C_k , що відповідають типам t_k , а також аналізу динаміки їхніх потужностей c_k у послідовності знімків, доцільно виділити типові ситуації надмірного використання пам'яті. Така класифікація ґрунтується на характері зміни кількості об'єктів у відповідних класах еквівалентності та дозволяє

інтерпретувати результати застосування функції групування як індикатори потенційних проблем керування пам'яттю. Це дозволить здійснювати аналіз поведінки програмної системи.

2.5.1 Детектування витоку пам'яті

Витік пам'яті розглядається як ситуація, за якої у послідовності знімків пам'яті спостерігається стале накопичення об'єктів певного типу, що залишаються досяжними з коренів середовища виконання та не можуть бути звільнені механізмом автоматичного керування пам'яттю. На відміну від одноразового збільшення обсягу пам'яті, витік характеризується динамікою накопичення об'єктів у часі.

Нехай задано послідовність знімків пам'яті одного програмного процесу:

$$M(u_1), M(u_2), \dots, M(u_q), u_1 < u_2 < \dots < u_q \quad (2.17)$$

де u_j - момент часу створення j -го знімку пам'яті, q - кількість знімків пам'яті.

Згідно з оператором інтерпретації знімку пам'яті (2.9), кожному знімку $M(u_j)$ відповідають множина об'єктів $O(u_j)$ та множина типів $T(u_j)$. Для аналізу динаміки об'єктів у часі введемо об'єднану множину типів:

$$T^\Sigma = \bigcup_{j=1}^q T(u_j) \quad (2.18)$$

Для кожного типу $t \in T^\Sigma$ визначимо множину об'єктів цього типу у знімку пам'яті $M(u_j)$:

$$O_t(u_j) = \{ o \in O(u_j) \mid \text{type}(o) = t \}, \quad (2.19)$$

де $\text{type}(o)$ - тип об'єкта o .

Кількість об'єктів типу t у момент часу u_j визначається як:

$$n_t(u_j) = |O_t(u_j)|. \quad (2.20)$$

Сумарний обсяг пам'яті, зайнятий об'єктами типу t у знімку $M(u_j)$, визначимо як:

$$S_t(u_j) = \sum_{o \in O_t(u_j)} s(o), \quad (2.21)$$

де $s(o)$ – розмір об'єкта o у байтах (2.6).

У середовищі виконання CLR звільнення пам'яті здійснюється збирачем сміття на основі аналізу досяжності об'єктів. Об'єкт підлягає видаленню лише у випадку, якщо він є недосяжним, тобто не існує жодного шляху посилок від коренів виконання до цього об'єкта. Усі досяжні об'єкти вважаються активними незалежно від їх фактичного використання в логіці програми.

Оскільки в середовищі CLR об'єкт може бути звільнений лише за умови відсутності досяжності з коренів виконання, для опису витоку доцільно враховувати лише ті об'єкти, що залишаються досяжними. Нехай $R(u_j)$ - множина коренів у момент часу u_j , а предикат $\text{Reach}(o, R(u_j))$ набуває значення істина, якщо об'єкт o є досяжним з деякого кореня $r \in R(u_j)$. Тоді множина досяжних об'єктів типу t визначається як:

$$O_t^R(u_j) = \{ o \in O_t(u_j) \mid \text{Reach}(o, R(u_j)) \}, \quad (2.22)$$

Відповідно, кількість досяжних об'єктів типу t та сумарний обсяг пам'яті, який вони займають, обчислюються в такий спосіб:

$$\begin{cases} n_t^R(u_j) = |O_t^R(u_j)|, \\ S_t^R(u_j) = \sum_{o \in O_t^R(u_j)} s(o). \end{cases} \quad (2.23)$$

Тип t вважатимемо кандидатом на витік пам'яті, якщо у послідовності знімків пам'яті кількість досяжних об'єктів цього типу та сумарний обсяг зайнятої ними пам'яті не зменшуються, а принаймні одна з цих величин зростає:

$$\forall j \in \{1, \dots, q-1\}: \begin{cases} n_t^R(u_{j+1}) \geq n_t^R(u_j) \\ S_t^R(u_{j+1}) \geq S_t^R(u_j) \end{cases} \quad (2.24)$$

та виконується умова зростання у часі:

$$n_t^R(u_q) > n_t^R(u_1) \vee S_t^R(u_q) > S_t^R(u_1). \quad (2.25)$$

Тоді множина типів, для яких виконується критерій витоку пам'яті, визначається як:

$$T_{\text{leak}} = \{ t \in T^\Sigma \mid C_{\text{leak}}(t) = 1 \}, \quad (2.26)$$

де $C_{\text{leak}}(t)$ - булева функція, що набуває значення 1 у разі виконання умов (2.24)–(2.25).

Множина об'єктів, що належать до потенційного витоку пам'яті в останньому знімку послідовності, задається як:

$$O_{\text{leak}}(u_q) = \bigcup_{t \in T_{\text{leak}}} O_t^R(u_q). \quad (2.27)$$

Для кількісної оцінки інтенсивності витоку пам'яті для кожного типу $t \in T_{\text{leak}}$ введемо приріст кількості об'єктів Δn_t та приріст обсягу пам'яті ΔS_t :

$$\begin{cases} \Delta n_t = n_t^R(u_q) - n_t^R(u_1) \\ \Delta S_t = S_t^R(u_q) - S_t^R(u_1). \end{cases} \quad (2.28)$$

Наведені характеристики дозволяють не лише встановити факт наявності потенційного витоку, але й ранжувати типи об'єктів за внеском у зростання споживання пам'яті. У практичному аналізі найбільший інтерес становлять типи з максимальними значеннями ΔS_t , оскільки саме вони формують основний внесок у збільшення обсягу використаної оперативної пам'яті.

Таким чином, задача детектування витоку пам'яті зводиться до аналізу послідовності знімків пам'яті, групування об'єктів за типами, виділення досяжних об'єктів та перевірки монотонного зростання їхньої кількості або сумарного обсягу. Запропонована формалізація дозволяє одержати множину типів-кандидатів T_{leak} , множину відповідних об'єктів $O_{\text{leak}}(u_q)$, а також кількісні оцінки Δn_t і ΔS_t , необхідні для подальшого автоматизованого аналізу та формування рекомендацій.

2.5.2 Представлення незмінюваної інформації

Одним із типових сценаріїв надмірного використання пам'яті, який не зводиться до витоку пам'яті, є дублювання незмінюваної інформації. На відміну від витоку пам'яті, такий сценарій не супроводжується монотонним

зростанням кількості об'єктів у часі. Його сутність полягає у наявності в пам'яті декількох фізично різних об'єктів, що мають однаковий інформаційний вміст і не можуть бути змінені після створення [20, 84].

Нехай $T(u)$ – множина типів, наявних у знімку пам'яті $M(u)$, а $O(u)$ – множина об'єктів цього знімку. Введемо предикат $Imm(t)$, який набуває значення істина, якщо тип t має властивість незмінності. Під незмінністю типу розуміється властивість, за якої значення екземпляра після його створення не може бути змінене жодною допустимою операцією над цим екземпляром. Множина незмінюваних типів T_{imm} , та множина об'єктів екземплярів цих типів у знімку пам'яті $M(u)$ визначається як:

$$\begin{cases} T_{imm}(u) = \{ t \in T(u) \mid Imm(t) \} \\ O_{imm}(u) = \{ o \in O(u) \mid t(o) \in T_{imm}(u) \} \end{cases} \quad (2.29)$$

Для кожного типу $t \in T_{imm}(u)$ підмножина об'єктів цього типу:

$$O_t^{imm}(u) = \{ o \in O(u) \mid t(o) = t, Imm(t) \} \quad (2.30)$$

На основі попередньо-введеного відношення еквівалентності (2.7) та властивості незмінності, введемо розбиття (хеш-функцію) Val_t , яке встановлює еквівалентність не за байтовою структурою типів, а за збігом інформаційного вмісту об'єктів. Для кожного значення $v \in V_t(u)$ визначається відповідний клас $C_{t,v}(u)$

$$\begin{cases} Val_t: O_t^{imm}(u) \rightarrow V_t(u) \\ o_i \sim Val_t, o_j \Leftrightarrow Val_t(o_i) = Val_t(o_j), o_i, o_j \in O_t^{imm}(u) \\ C_{t,v}(u) = \{ o \in O_t^{imm}(u) \mid Val_t(o) = v \} \\ \mu_{t,v} = | C_{t,v}(u) | \end{cases} \quad (2.31)$$

Якщо для об'єктів o_i та o_j виконується $o_i \sim Val_t(o_j)$, то вони належать до одного типу, однак обернене твердження загалом не є правильним: два об'єкти одного типу можуть мати різні значення. Саме це

додаткове розбиття дозволяє виявляти дублювання незмінюваної інформації у пам'яті.

Клас $C_{t,v}(u)$ містить усі фізично різні об'єкти типу t , що мають однакове значення v . Якщо $\mu_{t,v} > 1$, то в пам'яті наявне дублювання незмінюваної інформації для значення v .

Мінімальна кількість об'єктів $O_t^*(u)$, яка необхідна для представлення незмінної інформації типу t , дорівнює одному об'єкту з кожної з груп (представлена у вигляді операції $rep(C_{t,v}(u))$), а загальна множина унікальних значень $O_{imm}^*(u)$ у знімку пам'яті $M(u)$ є сумою для всіх типів з властивістю незмінності:

$$\begin{cases} O_t^*(u) = \{rep(C_{t,v}(u)) \mid v \in V_t(u)\} \\ O_{imm}^*(u) = \bigcup_{t \in T_{imm}(u)} O_t^*(u) \end{cases} \quad (2.32)$$

Для кожного типу $t \in T_{imm}(u)$ визначається внесок дублювання, який дозволяє впорядковувати незмінювані типи за внеском у надмірне використання пам'яті:

$$s_{dup,t}^{imm}(u) = \sum_{v \in V_t(u)} (|C_{t,v}(u)| - 1) \cdot s(rep(C_{t,v}(u))) \quad (2.33)$$

Твердження. Вибір одного представника з кожного класу еквівалентності $C_{t,v}(u)$ зберігає інформаційний вміст множини об'єктів $O_t^{imm}(u)$ у значеннєвому сенсі:

$$\{Val_t(o) : o \in O_t^{imm}(u)\} = \{Val_t(o) : o \in O_t^*(u)\}.$$

Доведення. За означенням (2.31), для кожного $v \in V_t(u)$ клас $C_{t,v}(u)$ містить усі об'єкти типу t , що мають значення v . Множина $O_t^*(u)$, визначена в (2.32), містить по одному елементу $rep(C_{t,v}(u))$ для кожного такого класу. Тому для кожного значення v , наявного в $O_t^{imm}(u)$, у множині $O_t^*(u)$ існує представник із тим самим значенням. Обернене включення випливає з того, що $O_t^*(u) \subseteq O_t^{imm}(u)$. Отже, перехід до представників класів еквівалентності вилучає повторні фізичні екземпляри, але не змінює множину інформаційних значень.

Таким чином, представлення незмінюваної інформації у пам'яті зводиться до виділення множини незмінюваних типів, побудови класів еквівалентності за значенням та формування множини надлишкових копій. Запропонована формалізація забезпечує математичну основу для подальшого виявлення дублікатів і кількісного оцінювання ефекту від їх усунення.

2.5.3 Оптимізація процесу виявлення дублікатів

Пряме порівняння $O_{t_{imm}}^{imm}(u)$ всіх об'єктів одного незмінюваного типу $t_{imm} \in T_{imm}(u)$ в процесі виявлення дублювання незмінюваної інформації є обчислювально витратним, оскільки кількість попарних перевірок $C_{t_{imm}}^{full}$ зростає квадратично зі збільшенням кількості об'єктів:

$$\begin{cases} N_{t_{imm}}(u) = |O_{t_{imm}}^{imm}(u)| \\ C_{t_{imm}}^{full} = \frac{N_{t_{imm}}(u)(N_{t_{imm}}(u)-1)}{2} \end{cases} \quad (2.34)$$

Тому доцільно ввести проміжний етап попереднього групування за додатковою ознакою.

Для кожного об'єкта $o \in O_{t_{imm}}^{imm}(u)$ введемо байтове представлення його значення $m_{t_{imm}}$, яке на відміну від $s(o)$, що визначає повний розмір об'єкта у пам'яті, характеризує лише довжину байтового представлення інформаційного вмісту. Це представлення використовується для введення обчислювальної ознаки $X_{t_{imm}}$, яка дозволить розбити множину об'єктів $O_{t_{imm}}^{imm}(u)$ на підмножини за допомогою швидко обчислюваного відображення $\varphi_{t_{imm}}$ байтового представлення значення об'єкта у числову характеристику:

$$\begin{cases} B_{t_{imm}}^{val} = (b_1^{val}(o), b_2^{val}(o), \dots, b_{m_{t_{imm}}(o)}^{val}(o)) \\ m_{t_{imm}} = |B_{t_{imm}}^{val}| \\ X_{t_{imm}} : O_{t_{imm}}^{imm}(u) \rightarrow \mathbb{Z}, X_{t_{imm}}(o) = \varphi_{t_{imm}}(B_{t_{imm}}^{val}(o)), \end{cases} \quad (2.35)$$

Для коректності групування обчислювальна ознака $X_{t_{imm}}$ повинна задовольняти умову рівності за умови рівності об'єктів:

$$Val_{t_{imm}}(o_i) = Val_{t_{imm}}(o_j) \Rightarrow X_{t_{imm}}(o_i) = X_{t_{imm}}(o_j) \quad (2.36)$$

При цьому обернене твердження (рівність обчислювальної ознаки) не гарантує рівності об'єктів.

Для незмінних типів, які не містять посилань, обчислювальною ознакою може виступати арифметична сума байтів значення. Така ознака не є унікальним ідентифікатором, оскільки різні значення можуть мати однакову суму байтів. Проте вона дозволяє швидко відокремити об'єкти, які гарантовано не є дублікатами, якщо їхні значення $X_{t_{imm}}$ різні.

Для кожного об'єкту $o \in O_{t_{imm}}^{imm}(u)$, розраховується обчислювальна ознака $X_{t_{imm}}(o)$, відокремлюється мінімальне та максимальне значення, отриманий відрізок значень розбивається на h напівінтервалів $I_{t_{imm},p}^{(h)}$ однакової довжини $\Delta_{t_{imm},h}(u)$. Останній інтервал вважається замкненим справа, щоб включити $X_{t_{imm}}^{max}$ значення:

$$\left\{ \begin{array}{l} X_{t_{imm}}^{min} = \min_{o \in O_{t_{imm}}^{imm}(u)} X_{t_{imm}}(o), \quad X_{t_{imm}}^{max} = \max_{o \in O_{t_{imm}}^{imm}(u)} X_{t_{imm}}(o) \\ \Delta_{t_{imm},h}(u) = \frac{X_{t_{imm}}^{max} - X_{t_{imm}}^{min}}{h}, \quad h \in N \\ p = \overline{1, h} \\ I_{t_{imm},p}^{(h)}(u) = [X_{t_{imm}}^{min}(u) + (p-1)\Delta_{t_{imm},h}(u), X_{t_{imm}}^{min}(u) + p\Delta_{t_{imm},h}(u)) \\ P_{t_{imm},p}^{(h)}(u) = \{o \in O_{t_{imm}}^{imm}(u) \mid X_{t_{imm}}(o) \in I_{t_{imm},p}^{(h)}(u)\} \end{array} \right. \quad (2.37)$$

Кожна підмножина $P_{t_{imm},p}^{(h)}(u)$ містить об'єкти, для яких значення обчислювальної ознаки потрапляє в один і той самий інтервал. З умови (2.36) випливає, що об'єкти з однаковим значенням не можуть потрапити до різних інтервалів. Водночас один інтервал може містити об'єкти з різними значеннями, тому всередині таких підмножин необхідна додаткова уточнююча перевірка.

Твердження. Попереднє групування за ознакою χ_t , що задовольняє умову (2.36), не втрачає пар об'єктів з однаковим інформаційним вмістом.

Доведення. Нехай $o_i, o_j \in O_t^{imm}(u)$ і $Val_t(o_i) = Val_t(o_j)$. З умови (2.36) маємо $\chi_t(o_i) = \chi_t(o_j)$. Розбиття (2.37) задає підмножини $P_{t,p}^h(u)$ за належністю значення $\chi_t(o)$ до відповідного інтервалу. Оскільки значення ознаки для o_i та o_j збігаються, обидва об'єкти належать до одного й того самого інтервалу, а отже, до однієї підмножини $P_{t,p}^h(u)$. Тому точне порівняння всередині підмножин $P_{t,p}^h(u)$ виявляє всі пари з однаковим значенням. Попереднє групування може створювати лише хибні кандидати всередині однієї підмножини, але не відкидає справжні дублікати.

З цього твердження випливає, що попереднє групування скорочує кількість уточнюючих порівнянь, але не змінює множину можливих дублікатів. Обчислювальна ознака не є критерієм рівності значень: об'єкти з однаковою ознакою потребують подальшої перевірки інформаційного вмісту.

Оцінка кількості операцій після попереднього групування $C_{t_{imm},h}$ враховує $N_{t_{imm}}$ - вартість лінійних витрат на обчислення ознаки $X_\tau(o)$ для всіх об'єктів t_{imm} , та використовується для оцінки коефіцієнту скорочення кількості операцій $K_{t_{imm},h}(u)$:

$$\begin{cases} C_{t_{imm},h}(u) = N_{t_{imm}} + \sum_{p \in P_{t_{imm},h}(u)} \frac{(|P|)(|P|-1)}{2} \\ K_{t_{imm},h}(u) = \frac{C_{t_{imm}}^{full}}{C_{t_{imm},h}(u)} \end{cases} \quad (2.38)$$

За припущенням рівномірного розподілу об'єктів між h підмножинами, збільшення кількості підмножин $h \in H_{t_{imm}}$ зменшує очікувану вартість точних порівнянь, однак збільшує накладні витрати на підтримку структури групування:

$$\begin{cases} |P_{t_{imm},p}^{(h)}(u)| \approx \frac{N_{t_{imm}}(u)}{h} \\ C_{t_{imm},h}^{unif}(u) \approx N_{t_{imm}} + h * \frac{\frac{N_{t_{imm}}(u)}{h} (\frac{N_{t_{imm}}(u)}{h} - 1)}{2} \end{cases} \quad (2.39)$$

Вибір параметра h має здійснюватися як компроміс між зменшенням кількості порівнянь і додатковими витратами на обробку підмножин, і представлений у вигляді задачі мінімізації:

$$h_{t_{imm}}^* = \arg \min_{h \in H_{t_{imm}}} (C_{t_{imm},h}(u) + \lambda h). \quad (2.40)$$

Оптимізація виявлення дублікатів незмінюваних об'єктів ґрунтується на введенні обчислювальної ознаки $X_{t_{imm}}$, яка визначається для будь-якого незмінюваного типу t_{imm} . Ця ознака використовується для попереднього групування об'єктів і зменшення кількості точних порівнянь. Остаточне встановлення дублювання виконується за рівністю значень $Val_{t_{imm}}$, що забезпечує коректність результату навіть за наявності колізій ознаки, коли декілька різних значень відображаються в однакове значення функції $X_{t_{imm}}$.

2.5.4 Оцінювання вартості зберігання інформації у хеш-колекціях типу **Dictionary<K,V>**

Хеш-колекції типу $t=Dictionary<K,V>$ є поширеним засобом подання асоціативних структур даних у середовищі .NET. Їх використання забезпечує ефективний доступ до значень за ключами, однак супроводжується додатковими витратами пам'яті на службові структури, необхідні для реалізації хешування, обробки колізій, підтримки операцій додавання, видалення та пошуку елементів [78, 99].

У загальному випадку фактичний обсяг пам'яті, зайнятий колекцією $Dictionary<K,V>$, складається з двох частин: обсягу корисної інформації, тобто даних пар ключ–значення, та обсягу службових структур. Для малих значень кількості елементів службові витрати можуть суттєво перевищувати обсяг корисного вмісту, що призводить до надмірного використання оперативної пам'яті без виконання критерію витоку пам'яті.

Нехай $T_D(u) \subseteq T(u)$ - множина типів, що відповідають реалізаціям $Dictionary < K, V >$ у знімку пам'яті $M(u)$; $N(d)$ визначає кількість пар ключ-

значення представлених у колекції d . Для фіксованого типу $t_D \in T_D(u)$ множина екземплярів об'єктів визначається:

$$\begin{cases} O_{D,t_D}(u) = \{ d \in O(u) \mid t(d) = t_D \} \\ N(d) = | \{ (k_1, v_1), (k_2, v_2) \dots (k_i, v_i) \} \end{cases} \quad (2.41)$$

Введемо функції $w_K(k_i)$ та $w_V(v_i)$, що визначають обсяг пам'яті, необхідний для збереженні ключа k_i та значення v_i у внутрішній структурі словника. Для значимих типів ці величини відповідають розміру значення відповідного типу, а для посилальних типів - розміру посилання у середовищі виконання. Ці функції дозволяють описати обсяг пам'яті для збереження інформації $I(d)$, та фактичний $V_{fact}(d)$:

$$\begin{cases} I(d) = \sum_{i=1}^{N(d)} (w_K(k_i) + w_V(v_i)) \\ V_{fact}(d) = s(d) + \sum_{a \in A_D(d)} s(a) \\ W_D(d) = V_{fact}(d) - I(d) \\ \mu_D(d) = \frac{I(d)}{V_{fact}(d)} \\ 0 < \mu_D(d) \leq 1 \\ k_D(d) = \frac{V_{fact}(d)}{I(d)} = \frac{1}{\mu_D(d)}, \end{cases} \quad (2.42)$$

де $s(d)$ - розмір самого об'єкта словника, $A_D(d)$ - множина допоміжних об'єктів і внутрішніх структур, пов'язаних з його реалізацією, $W_D(d)$ - службові витрати пам'яті для словника, $\mu_D(d)$ - оцінка ефективності зберігання корисної інформації, $k_D(d)$ - коефіцієнт, який показує у скільки разів фактичний обсяг пам'яті, зайнятий словником, перевищує обсяг корисної інформації, що в ньому зберігається.

Для виявлення класів словників із низьким рівнем ефективності зберігання інформації в оперативній пам'яті доцільно групувати їх за кількістю збережених елементів:

$$\begin{cases} O_{D,t_D}^{(n)}(u) = \{ d \in O_{D,t_D}(u) \mid N(d) = n \}, \\ c_{t_D}(n, u) = |O_{D,t_D}^{(n)}(u)| \end{cases} \quad (2.43)$$

де $c_{t_D}(n, u)$ – кількість словників типу t_D , що містять n елементів.

Для аналізу сукупності словників одного типу t_D з n елементами введено агреговані характеристики:

$$\begin{cases} I_{t_D}^{(n)}(u) = \sum_{d \in O_{D,t_D}^{(n)}(u)} I(d) \\ V_{fact}^{(n)}(u) = \sum_{d \in O_{D,t_D}^{(n)}(u)} V_{fact}(d) \\ W_{t_D}^{(n)}(u) = V_{fact}^{(n)}(u) - I_{t_D}^{(n)}(u) \\ \mu_{t_D}^{(n)}(u) = \frac{I_{t_D}^{(n)}(u)}{V_{fact}^{(n)}(u)} \\ k_{t_D}^{(n)}(u) = \frac{V_{fact}^{(n)}(u)}{I_{t_D}^{(n)}(u)}. \end{cases} \quad (2.44)$$

Якщо для деякого n виконується умова

$$\mu_{t_D}^{(n)}(u) < \tau_D \quad (2.45)$$

де $\tau_D \in (0,1)$ - задане порогове значення ефективності зберігання, то група словників $O_{D,t_D}^{(n)}(u)$ вважається кандидатом на заміну.

Запропонована модель оцінювання вартості зберігання інформації у *Dictionary* $\langle K, V \rangle$ дозволяє кількісно відокремити корисний інформаційний обсяг від службових накладних витрат. Вона забезпечує визначення коефіцієнтів μ_D, κ_D , а також агрегованих показників для груп словників з однаковою кількістю елементів. Це дає змогу виявляти випадки нерационального використання хеш-колекцій з погляду витрат оперативної пам'яті та формувати кількісно обґрунтовану оцінку потенційного зменшення обсягу пам'яті при переході до компактнішої структури зберігання. [78, 99].

2.5.5 Оцінювання доцільності звуження типів полів за фактичним діапазоном значень

Одним із джерел надмірного використання пам'яті є застосування типів даних із надлишковим діапазоном значень. Така ситуація виникає, якщо поле об'єкта оголошене типом, що допускає значно ширший діапазон значень, ніж фактично використовується. У цьому випадку частина байтів, відведених для зберігання поля, не використовується для подання інформаційного вмісту, але входить до фактичного розміру кожного екземпляра об'єкта.

Відповідно до (2.4), кожен об'єкт $o \in O(u)$ подано як

$$o = (a, t, s, FLD),$$

де FLD є множиною значень полів об'єкта. Для подальшого аналізу елемент множини полів об'єкта подамо у вигляді

$$FLD(o) = \{ (p, \delta_p, x_p(o)) \}, \quad (2.46)$$

де p - ідентифікатор поля, δ_p - оголошений тип поля, $x_p(o)$ - значення поля p в об'єкті o .

Наявна у знімку пам'яті множина значень поля p , що входить до $FLD(o)$ для об'єктів типу $t_{sh} \in T(u)$ визначається:

$$\begin{cases} O_{t_{sh}}(u) = \{ o \in O(u) \mid t(o) = t_{sh} \} \\ x_{t_{sh},p}(u) = \{ x_p(o) \mid o \in O_{t_{sh}}(u), (p, \delta_p, x_p(o)) \in FLD(o) \}. \end{cases} \quad (2.47)$$

Для кожного з числових полів p , відношення фактичного діапазону значень $R_{t_{sh},p}^{obs}(u)$ до множини допустимих значень D_{δ_p} оголошеним типом поля δ_p дозволяє надати оцінку використання діапазону $\rho_{t_{sh},p}$:

$$\begin{cases} x_{t_{sh},p}^{min}(u) = \min_{o \in O_{t_{sh}}(u)} x_p(o), \quad x_{t_{sh},p}^{max}(u) = \max_{o \in O_{t_{sh}}(u)} x_p(o), \\ R_{t_{sh},p}^{obs}(u) = [x_{t_{sh},p}^{min}(u), x_{t_{sh},p}^{max}(u)] \\ D_{\delta_p} = [L_{\delta_p}, U_{\delta_p}] \\ \rho_{t_{sh},p} = \frac{x_{t_{sh},p}^{max}(u) - x_{t_{sh},p}^{min}(u) + 1}{U_{\delta_p} - L_{\delta_p} + 1}. \end{cases} \quad (2.48)$$

Мале значення $\rho_{t_{sh},p}$ означає, що оголошений тип поля має суттєво ширший діапазон, ніж фактично використовується у досліджуваному знімку пам'яті.

Тип d вважається допустимою заміною для поля p , якщо фактичний діапазон значень цього поля $R_{t_{sh},p}^{obs}(u)$ з урахуванням коефіцієнту запасу γ , повністю міститься в області допустимих значень D_d типу d :

$$\begin{cases} \varepsilon_{t_{sh},p}(u) = \max \{1, [\gamma * (x_{t_{sh},p}^{max}(u) - x_{t_{sh},p}^{min}(u))]\} \\ R_{t_{sh},p}^{obs,\gamma}(u) = [x_{t_{sh},p}^{min}(u) - \varepsilon_{t_{sh},p}(u), x_{t_{sh},p}^{max}(u) + \varepsilon_{t_{sh},p}(u)] \\ R_{t_{sh},p}^{obs,\gamma}(u) \subseteq D_d \end{cases} \quad (2.49)$$

Якщо для поля відомі семантичні обмеження предметної області, наприклад невід'ємність ідентифікатора, ці обмеження варто враховувати при розрахунку діапазону значень.

Потенційне зменшення розміру одного значення поля p визначається як різниця між розміром початкового типу поля δ_p та запропонованого $\delta_{t_{sh},p}^*$:

$$\Delta w_{t_{sh},p}(u) = w(\delta_p) - w(\delta_{t_{sh},p}^*(u))$$

Запропонований підхід дозволяє на основі аналізу фактичних значень, що входять до $FLD(o)$, виявляти поля з надлишковим діапазоном оголошеного типу, визначати мінімальний допустимий тип для подання спостережуваних значень та оцінювати потенційне зменшення використання оперативної пам'яті.

2.5.6 Оцінювання ефекту стискування об'єктів типу `byte[]` зі збереженням доступності всієї інформації

Окремим сценарієм надмірного використання оперативної пам'яті є зберігання великих обсягів бінарної інформації у вигляді об'єктів типу `System.Byte[]`. Такі об'єкти широко використовуються для подання серіалізованих структур, вмісту файлів, мережевих повідомлень, буферів введення-виведення та інших послідовностей байтів. Якщо інформаційний

вміст таких масивів містить внутрішню надлишковість, то його можна подати у стисненому вигляді без втрати доступності всієї інформації [82].

Для кожного об'єкта з типом $t = \text{System.byte}[]$, інформаційний вміст подамо у вигляді послідовності байтів:

$$\begin{cases} t_b = \text{System.Byte}[] \\ O_b(u) = \{o \in O(u) \mid t(o) = t_b\} \\ \text{INFO}(o) = (b_1(o), b_2(o) \dots, b_{n_b(o)}(o)), \end{cases} \quad (2.50)$$

де $n_b(o)$ – кількість елементів масиву байтів. Введемо множину алгоритмів стискання Z , де кожен алгоритм відображає початковий інформаційний вміст у стиснене подання:

$$\begin{cases} Z = \{(Z_i)\} \quad i = \overline{1, m} \\ Z_\alpha: \{0, \dots, 255\}^{n_b(o)} \rightarrow \{0, \dots, 255\}^{m_b(o)}, \quad z_\alpha \in Z, \end{cases} \quad (2.51)$$

де $m_\alpha(o)$ - довжина стисненого подання.

Для забезпечення збереження доступності всієї інформації алгоритм стискання має бути безвтратним. Це означає, що для кожного алгоритму Z_α існує відповідне відображення відновлення Z_α^{-1} , для якого виконується умова:

$$Z_\alpha^{-1}(Z_\alpha(\text{INFO}(o))) = \text{INFO}(o), \quad o \in O_b(u). \quad (2.52)$$

Виконання цієї умови є формальним критерієм збереження інформаційної повноти: після стискання та подальшого відновлення послідовність байтів має збігатися з початковою.

Для оцінювання ефективності стискання одного об'єкта введемо коефіцієнт стискання r_α та зміну використаного об'єму пам'яті Δs_α :

$$\begin{cases} \Delta s_\alpha(o) = s(o) - s_\alpha^{\text{cmp}}(o) \\ r_\alpha(o) = \frac{s_\alpha^{\text{cmp}}(o)}{s(o)} \end{cases} \quad (2.53)$$

З урахуванням того, що для малих масивів службові витрати можуть перевищувати виграш від стискання, введемо мінімальну допустиму довжину масиву n_{\min} . Крім того, не кожен масив байтів є коректним кандидатом на стискання з погляду логіки програмної системи: наприклад, буфери активного введення-виведення або об'єкти, що інтенсивно модифікуються, можуть не бути придатними для заміни на стиснене подання. Для цього введемо предикат

$$\text{Adm}_{\text{cmp}}(o),$$

який набуває значення істина, якщо об'єкт o може бути поданий у стисненому вигляді без порушення семантики доступу до даних.

Тоді множина об'єктів-кандидатів на стискання за алгоритмом Z_α визначається як

$$O_\alpha^{\text{cmp}}(u) = \left\{ o \in O_b(u) \mid \begin{array}{l} n_b(o) \geq n_{\min}, \Delta s_\alpha(o) > 0 \\ r_\alpha(o) \leq \theta_{\text{cmp}}, \text{Adm}_{\text{cmp}}(o) \end{array} \right\}, \quad (2.54)$$

де $\theta_{\text{cmp}} \in (0,1)$ - задане порогове значення коефіцієнта стискання.

Твердження. Для кожного об'єкта $o \in O_\alpha^{\text{cmp}}(u)$ заміна інформаційного вмісту $INFO(o)$ на стиснене подання $Z_\alpha(INFO(o))$ є коректною щодо збереження інформаційного вмісту, якщо виконується умова (2.52).

Доведення. Для алгоритму Z_α умова (2.52) задає існування відображення відновлення Z_α^{-1} , для якого

$$Z_\alpha^{-1}(Z_\alpha(INFO(o))) = INFO(o).$$

Отже, після стискання та відновлення отримується та сама послідовність байтів, що була початковим інформаційним вмістом об'єкта. Належність o до множини $O_\alpha^{\text{cmp}}(u)$, визначеної в (2.54), додатково фіксує додатне зменшення розміру, виконання порогової умови для коефіцієнта стискання та істинність предиката $\text{Adm}_{\text{cmp}}(o)$. Тому для таких об'єктів стиснене подання не змінює інформаційний вміст і не порушує задану умову допустимості доступу до даних.

Для множини кандидатів $O_\alpha^{cmp}(u)$ ефект від провадження стискування ρ_α^{cmp} визначається:

$$\left\{ \begin{array}{l} S_\alpha^{orig}(u) = \sum_{o \in O_\alpha^{cmp}(u)} s(o), \quad S_\alpha^{cmp}(u) = \sum_{o \in O_\alpha^{cmp}(u)} s_\alpha^{cmp}(o) \\ \Delta S_\alpha^{cmp}(u) = S_\alpha^{orig}(u) - S_\alpha^{cmp}(u) \\ \rho_\alpha^{cmp}(u) = \frac{\Delta S_\alpha^{cmp}(u)}{S_b(u)} \end{array} \right. \quad (2.55)$$

Величина $\rho_\alpha^{cmp}(u)$ показує, яку частку пам'яті, зайнятої масивами байтів, потенційно можна зменшити шляхом застосування алгоритму Z_α до об'єктів-кандидатів.

В результаті, задача оцінювання ефекту стискування об'єктів типу *System.Byte[]* зводиться до виділення множини масивів байтів, визначення їхнього інформаційного вмісту, застосування безвтратного відображення стискування, перевірки умови повного відновлення інформації та кількісного порівняння початкового й стисненого подання. Запропонована формалізація дозволяє обчислити потенційну економію пам'яті $\Delta S_*^{cmp}(u)$, відносний ефект $\rho_*^{cmp}(u)$, а також сформуванати множини кандидатів $O_*^{cmp}(u)$ для подальшої експертної оцінки та генерації рекомендацій.

2.6 Висновки до розділу

1. У другому розділі дисертаційної роботи сформовано математичний апарат подання та аналізу інформації в оперативній пам'яті програмного процесу. Знімок пам'яті подано як скінченну впорядковану множину байтів, а результат його інтерпретації - як сукупність множин об'єктів, типів і потоків виконання. Об'єкт пам'яті формалізовано через адресу, тип, розмір та множину значень полів, що забезпечує перехід від байтового представлення даних до їх змістовної інтерпретації.
2. Сформульовано задачу ідентифікації сценаріїв надмірного використання оперативної пам'яті як задачу виявлення класів надлишкового подання інформації у типізованій структурі об'єктів, визначених за формалізованим описом знімка пам'яті. Визначено критерії виявлення таких класів на основі структурних характеристик об'єктів, відношення еквівалентності за інформаційним вмістом та кількісних оцінок надмірності.
3. Розроблено математичну модель детектування витoku пам'яті на основі аналізу послідовності знімків пам'яті. Витік пам'яті формалізовано як ситуацію, за якої для певного типу об'єктів спостерігається монотонне незменшення кількості досяжних об'єктів або сумарного обсягу пам'яті, що ними займається. У межах моделі визначено множину типів-кандидатів на витік, множину відповідних об'єктів та кількісні характеристики приросту кількості об'єктів і зайнятої пам'яті.
4. Запропоновано модель представлення незмінюваної інформації у пам'яті, яка дозволяє описувати сценарії надмірного використання пам'яті без застосування критерію витoku. Для незмінюваних типів введено значеннево-орієнтоване розбиття об'єктів на класи еквівалентності за збігом інформаційного вмісту. На цій основі

визначено множину безнадлишкових представників, множину надлишкових копій та оцінку внеску дублювання для відповідного типу об'єктів.

5. Розроблено підхід до прискореного виявлення дублікатів незмінюваних об'єктів. Для цього введено обчислювальну ознаку, яка визначається для об'єктів фіксованого незмінюваного типу та використовується для попереднього групування перед точним порівнянням значень. Показано, що таке групування зменшує кількість попарних перевірок порівняно з повним перебором, при цьому зберігається коректність результату за рахунок остаточної перевірки рівності інформаційного вмісту.
6. Побудовано математичну модель оцінювання доцільності звуження типів полів за фактичним діапазоном значень, наявних у знімку пам'яті. Для числових полів визначено фактичний діапазон значень, коефіцієнт використання діапазону оголошеного типу та умову допустимої заміни типу з урахуванням заданого запасу. Запропонований підхід дозволяє формалізовано виявляти поля, для яких оголошений тип має надлишковий діапазон, та оцінювати потенційне зменшення використання пам'яті при збереженні інформаційної повноти.
7. Запропоновано математичну модель оцінювання вартості зберігання інформації у хеш-колекціях типу $\text{Dictionary}\langle K, V \rangle$. У моделі відокремлено корисний інформаційний обсяг пар ключ–значення від службових витрат, пов'язаних із внутрішніми структурами словника. Введені коефіцієнти ефективності та вартості зберігання дозволяють виявляти випадки, коли використання хеш-колекції є неефективним з погляду споживання пам'яті, зокрема за малої кількості елементів.
8. Розроблено модель оцінювання ефекту стискання об'єктів типу `byte[]` зі збереженням доступності всієї інформації. Інформаційний вміст масиву байтів подано як послідовність значень, до якої застосовується безвтратне відображення стискання з подальшим відновленням.

Введено умови відбору об'єктів-кандидатів на стискання, а також кількісні показники потенційного зменшення обсягу пам'яті.

9. Отримані у розділі моделі створюють формальну основу для подальшого емпіричного дослідження надмірного використання пам'яті та побудови бази знань експертної системи. Вони дозволяють описувати як класичні сценарії витоків пам'яті, так і інші сценарії надмірного споживання, зокрема дублювання незмінюваних об'єктів, надлишковий діапазон типів полів, неефективне використання хеш-колекцій та потенціал стискання масивів байтів. Запропоновані формалізації є підґрунтям для експериментальної перевірки моделей і кількісного оцінювання ефекту від впровадження змін у наступному розділі.

Основні результати розділу опубліковані в [20, 52, 54, 68, 77, 82, 83, 84, 85, 97, 98].

Розділ 3. ЕМПІРИЧНІ ТА ТЕОРЕТИЧНІ ДОСЛІДЖЕННЯ НАДМІРНОГО ВИКОРИСТАННЯ ПАМ'ЯТІ ШЛЯХОМ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ

У межах експериментальної частини було проаналізовано понад 80 знімків пам'яті .NET-застосунків. У підпунктах розділу наведено окремі випадки, для яких відповідні кількісні критерії мають виражене виконання. Ці випадки не вичерпують усієї вибірки, але узгоджуються з повторюваними ситуаціями, що фіксувалися під час аналізу. Якщо умови певного правила не виконувались, це інтерпретувалося як відсутність ознак відповідного сценарію, а не як відсутність надмірного використання пам'яті в інших формах.

3.1 Методика збору й аналізу промислових знімків пам'яті та первинний аналіз розподілу пам'яті за типами

У цьому розділі виконується емпірична апробація математичних підходів до аналізу інформації в оперативній пам'яті, побудованих у розділі 2. Об'єктом емпіричного дослідження є промислові знімки пам'яті .NET-застосунків, отримані під час функціонування реальних програмних систем. Метою первинного етапу аналізу розподілу є перехід від байтового подання знімку пам'яті до впорядкованої множини об'єктів, типів і кількісних характеристик, придатних для подальшого аналізу надмірного використання оперативної пам'яті.

Теоретичною основою такого переходу є формальне подання знімку пам'яті як множини байтів відповідно до (2.1), множини об'єктів відповідно до (2.2) та множини типів відповідно до (2.3). Кожен об'єкт керованої купи розглядається згідно з поданням (2.4), тобто як сутність, що має адресу, тип, розмір і множину значень полів. Розмір об'єкта в байтах визначається відповідно до (2.6). Після застосування оператора інтерпретації (2.9) знімок

пам'яті у фіксований момент часу u^* розглядається у вигляді моделі, заданої формулою (2.10).

Застосування саме цієї формалізації дозволяє уніфікувати подальший аналіз незалежно від конкретної прикладної системи: усі об'єкти знімку пам'яті подаються через спільну множинну модель, а їхні кількісні характеристики можуть бути обчислені за єдиними правилами. Загальний обсяг керованої пам'яті визначається відповідно до (2.12), що забезпечує основу для порівняння внеску окремих типів у сумарне споживання оперативної пам'яті.

Методичні положення щодо використання знімків пам'яті для аналізу проблем продуктивності програмного забезпечення розглянуто в роботах [16], [20], [84]. Питання збереження інформаційної повноти при фіксованих обсягах пам'яті та пришвидшення пошуку релевантних сегментів пам'яті розглянуто в [6], [83]. Практична апробація підходу до виявлення надмірного використання пам'яті за допомогою аналізу промислових знімків пам'яті наведена в [102].

Збирання знімків пам'яті здійснювалося із застосуванням ProcDump, що дозволяє фіксувати стан процесу у визначений момент часу або за настання заданих умов. Первинну перевірку структури керованої купи та ідентифікацію об'єктів виконано засобами WinDBG, які є типовим інструментом низькорівневого аналізу .NET-застосунків. Для програмного вилучення фактів про об'єкти, типи, розміри та значення полів використано бібліотеку ClrMD. Для подальшої експериментальної перевірки ефекту окремих змін у швидкодії у наступних підпунктах застосовано Benchmark.NET.

У межах дослідження аналіз обмежено характеристиками керованої пам'яті. Інші показники продуктивності, зокрема використання процесора, мережеві операції, операції введення-виведення та затримки зовнішніх сервісів, не розглядаються як самостійні об'єкти дослідження.

Первинний аналіз розподілу пам'яті виконується як етап зменшення розмірності задачі. Промисловий знімок пам'яті може містити мільйони об'єктів різних типів, тому повний аналіз усіх об'єктів без попереднього

групування є обчислювально недоцільним. Для цього використовується функція групування за типами, введена у (2.16). Після такого групування для кожного типу t визначається множина його екземплярів відповідно до (2.19), кількість екземплярів відповідно до (2.20) та сумарний обсяг пам'яті, зайнятий об'єктами цього типу, відповідно до (2.21).

Процедура первинного аналіз розподілу пам'яті складається з таких етапів:

1. Створення знімку пам'яті промислового .NET-застосунку у момент часу u^* ;
2. Інтерпретація байтового вмісту знімку пам'яті відповідно до оператора (2.9);
3. Побудова множин об'єктів, типів і потоків виконання згідно з моделлю (2.10);
4. Групування об'єктів за типами за допомогою функції (2.16);
5. Визначення для кожного типу кількості екземплярів за (2.20) і сумарного обсягу пам'яті за (2.21);
6. Ранжування типів за спаданням сумарного обсягу пам'яті, а також аналіз типів із великою кількістю екземплярів;
7. Вибір типів-кандидатів, які мають потенційні ознаки спричинення надмірного використання пам'яті, для подальшого поглибленого аналізу

Важливо зазначити, що первинний аналіз розподілу пам'яті одного знімку не є критерієм витоку пам'яті. Критерій витоку потребує аналізу послідовності знімків (2.17)-(2.28). Знімок пам'яті розглядається як статичний зріз стану програмної системи, що дозволяє виявити типи, які формують найбільший внесок у використання оперативної пам'яті, але не доводить наявності монотонного накопичення об'єктів у часі.

У таблиці 3.1. наведено розподіл об'єктів за типами, що отримано за результатами застосування процедури попереднього аналізу.

Табл. 3.1 Розподіл споживання пам'яті за типами об'єктів

Тип об'єкта, <i>t</i>	Кількість екземплярів відповідно до (2.20)	Сумарний обсяг пам'яті, байт, відповідно до (2.21)	Інтерпретація
System.String (рядок)	34 456 461	2 620 679 684	найбільший сумарний обсяг пам'яті та значна кількість екземплярів
PaymentGatewayPaymentMethod[]	786	1 370 734 672	мала кількість екземплярів при значному сумарному обсязі
System.Int32[]	5 489 559	721 114 212	значна кількість масивів примітивного типу
Generic.Dictionary / Entry	62 550	441 122 544	значний внесок службових структур хеш-колекцій
GroupedShipping MethodZoneKey	6 109 902	244 396 080	велика кількість екземплярів користувачького типу

З аналізу таблиці 3.1 випливає, що найбільшим споживачем керованої пам'яті є тип *System.String*. Його екземпляри займають 2 620 679 684 байти, що

свідчить про домінування рядкових даних у структурі досліджуваної керованої купи. Оскільки *System.String* належить до незмінюваних типів, такий результат є підставою для застосування моделі представлення незмінюваної інформації (2.29)-(2.33).

Другою групою типів-кандидатів є масиви, зокрема *PaymentGatewayPaymentMethod[]* та *System.Int32[]*. Для першого з цих типів характерною є невелика кількість екземплярів при значному сумарному обсязі пам'яті, що вказує на можливу наявність великих масивів або надмірно зарезервованих внутрішніх структур. Для *System.Int32[]*, навпаки, суттєвим є поєднання великої кількості екземплярів із відносно значним сумарним обсягом пам'яті. Такі типи потребують окремого аналізу структури зберігання даних, фактичного наповнення та можливого резервування.

Третьою групою є хеш-колекції та пов'язані з ними користувацькі типи. Наявність серед найбільших споживачів пам'яті внутрішніх структур *Dictionary* свідчить про доцільність аналізу службових витрат хеш-колекцій та є підставою для застосування моделі оцінювання вартості зберігання інформації у *Dictionary<K,V>*.

Тип *GroupedShippingMethodZoneKey* має значну кількість екземплярів, що вказує на потенційну доцільність аналізу його полів, діапазонів значень і способу подання у 64-бітному середовищі виконання. У таких випадках надмірне використання пам'яті може бути спричинене не дублюванням об'єктів, а невідповідністю фактичних діапазонів значень обраним типам даних, витратами на посилавальні типи або вирівнюванням структури об'єкта в пам'яті.

Таким чином, первинний аналіз розподілу пам'яті виконує роль формального етапу відбору типових ситуацій надмірного використання пам'яті. За його результатами визначено такі типові ситуації: незмінювані рядкові об'єкти *System.String*, масиви примітивних і користувацьких типів, хеш-колекції та користувацькі ключові структури.

Застосування методики збору, інтерпретації та первинного аналізу розподілу пам'яті дозволило отримати впорядковану множину типів-кандидатів для подальшого аналізу надмірного використання оперативної пам'яті.

У вибірці також були знімки, для яких групування за типами не відокремлювало один або декілька типів із переважним внеском у сумарний обсяг пам'яті. Частка таких випадків була порядку п'яти відсотків. Для них подальший аналіз виконувався не за найбільшим типом, а за структурними ознаками: наявністю колекцій, масивів, незмінюваних типів або груп із великою кількістю екземплярів. Первинне групування в таких випадках задає лише порядок подальшого розгляду і не є критерієм наявності надмірного використання пам'яті.

3.2 Ідентифікація та кількісне оцінювання дублювання об'єктів типу *System.String*

Об'єкти типу *System.String* займають одну з найбільших часток керованої пам'яті досліджуваного застосунку. Тип *System.String* належить до незмінюваних типів даних, тому для нього використовується модель представлення незмінюваної інформації, наведена у (2.29)–(2.33).

У середовищі виконання CLR тип *System.String* використовується для представлення скінченної послідовності символів. Відповідно до подання об'єкта у (2.4), кожен рядковий об'єкт характеризується адресою, типом, розміром та інформаційним вмістом. Адреса визначає фізичне розміщення об'єкта у пам'яті, тоді як інформаційний вміст визначається послідовністю символів. Рядкові об'єкти з різними адресами можуть містити однакове значення. Такі об'єкти належать до одного класу еквівалентності за значенням відповідно до (2.31).

Особливість типу *System.String* полягає в тому, що після створення рядкового об'єкта його значення не змінюється. Отже, для множини фізично різних рядкових об'єктів з однаковим значенням достатньо зберігати один

представник відповідного класу еквівалентності. У математичному сенсі це відповідає переходу від множини всіх екземплярів незмінюваного типу, визначеної у (2.29)-(2.30), до класів еквівалентності за значенням згідно з (2.31) та до множини представників, визначеної у (2.32). Кількісний внесок таких повторів у надмірне використання пам'яті оцінюється відповідно до (2.33).

Для об'єктів типу *System.String* виконується групування за значенням відповідно до (2.31). Дублювання визначається належністю рядкових об'єктів до одного класу еквівалентності за інформаційним вмістом. Слід підкреслити, що адреса рядкового об'єкта не використовується як критерій рівності значень. Адреса дозволяє встановити фізичну відмінність екземплярів, але дублювання визначається не збігом адрес, а належністю об'єктів до одного класу еквівалентності за інформаційним вмістом. Тому два рядкові об'єкти з різними адресами, але однаковою послідовністю символів, вважаються повторними представленнями одного й того самого значення.

На рисунку 3.1 наведено приклад фрагмента знімку пам'яті, у якому фізично різні об'єкти містять однакові рядкові значення. У цьому прикладі адреси об'єктів демонструють, що вони є окремими екземплярами у керованій купі, а збіг їхнього інформаційного вмісту свідчить про належність до одного класу еквівалентності за значенням.

```

00000181696ce700 (Model.CountryToCountrySetting) ← 2
  <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" 3
  <SettingValue>k__BackingField:00000181696ce7a0 (System.String) Length=6, String="5;;USD"
  <SettingDescription>k__BackingField:00000181696ce7c8 (System.String) Length=111, String="Value for Commercial and Parcel Invoice,
00000181696ce8c0 (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" 3
  <SettingValue>k__BackingField:00000181696ce960 (System.String) Length=6, String="5;;USD"
  <SettingDescription>k__BackingField:00000181696ce988 (System.String) Length=111, String="Value for Commercial and Parcel Invoice,
00000181696cea80 (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" 3
  <SettingValue>k__BackingField:00000181696ceb20 (System.String) Length=6, String="5;;USD"
  <SettingDescription>k__BackingField:00000181696ceb48 (System.String) Length=111, String="Value for Commercial and Parcel Invoice,
  
```

Рис. 3.1 Приклад дублювання рядкової інформації у знімку пам'яті

Практична інтерпретація цього явища наведена на рисунку 3.2. У випадку дублювання декілька різних об'єктів зберігають однакове значення. У випадку повторного використання один фізичний об'єкт виступає спільним представником відповідного значення. Другий варіант не змінює інформаційного змісту системи, але може зменшити обсяг пам'яті, необхідний для його представлення. Відомим підходом до такого повторного використання є пул об'єктів, а для рядкових значень також можуть застосовуватись інтернування або прикладне кешування.

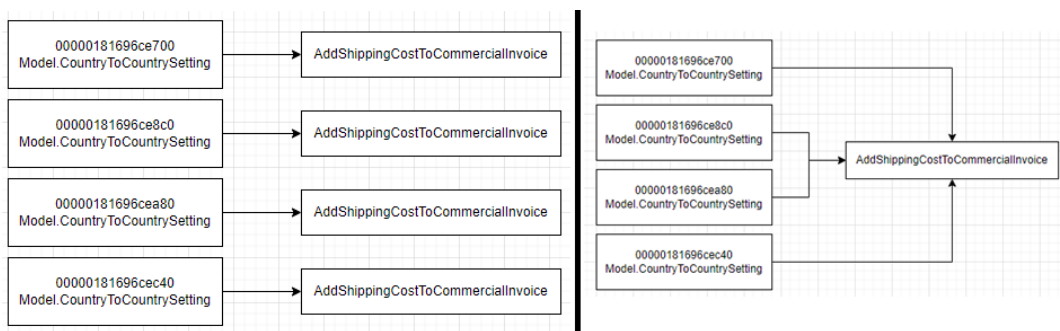


Рис. 3.2 Порівняння дублювання та повторного використання незмінюваних об'єктів

Оброблення знімку пам'яті було виконано програмно із застосуванням бібліотеки CLRMD, що дозволяє отримувати відомості про об'єкти керованої купи, їхні типи, адреси, розміри та значення полів. Оскільки промисловий знімок пам'яті містить десятки мільйонів об'єктів, пряме попарне порівняння всіх рядкових значень є обчислювально недоцільним. Тому під час формування класів еквівалентності використовується підхід попереднього групування, обґрунтований у (2.34)-(2.40).

Для візуальної ідентифікації дублювання було побудовано емпіричний розподіл об'єктів типу System.String за значенням обчислювальної ознаки $X_{t_{imm}}$. Отриманий початковий результат наведено на рис. 3.3. У розподілі чітко спостерігаються три домінуючі глобальні максимуми, що відповідають групам рядкових значень з аномально великою кількістю фізичних представлень (понад 3 млн екземплярів кожне). До таких значень належать «5;;USD»,

«AddShippingCostToCommercialInvoice» та «Value for Commercial and Parcel Invoice», які структурно пов'язані з об'єктами типу Model.CountryToCountrySetting.

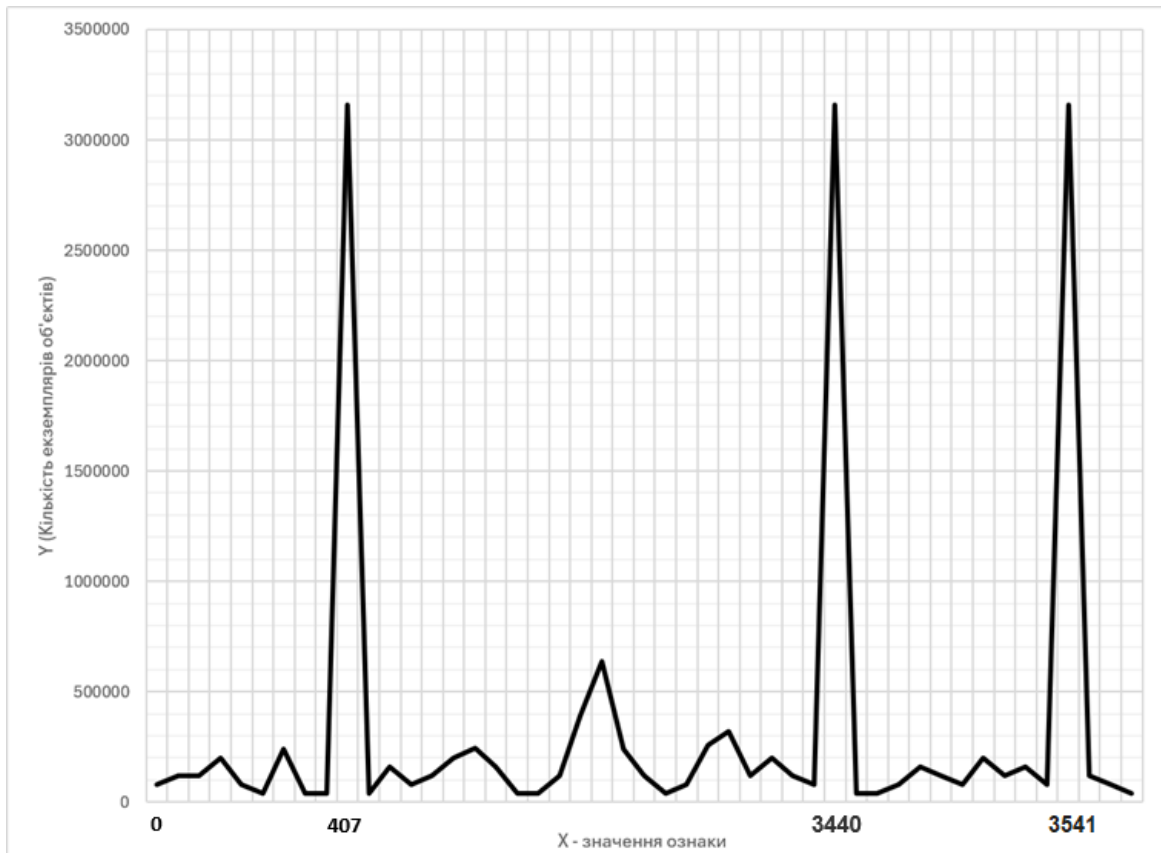


Рис 3.3 Розподіл об'єктів типу рядок за значенням $X_{t_{imm}}$

Оскільки ці три аномальні піки сумарно утворюють понад 9 мільйонів дублікатів і візуально приховують розподіл решти даних, їх було тимчасово вилучено з вибірки. Після цього розподіл за ознакою $X_{t_{imm}}$ було побудовано повторно (рис. 3.4). Зміна масштабу та форми розподілу дозволила ідентифікувати множину інших класів еквівалентності, що формують загальносистемне «фонове» дублювання пам'яті.

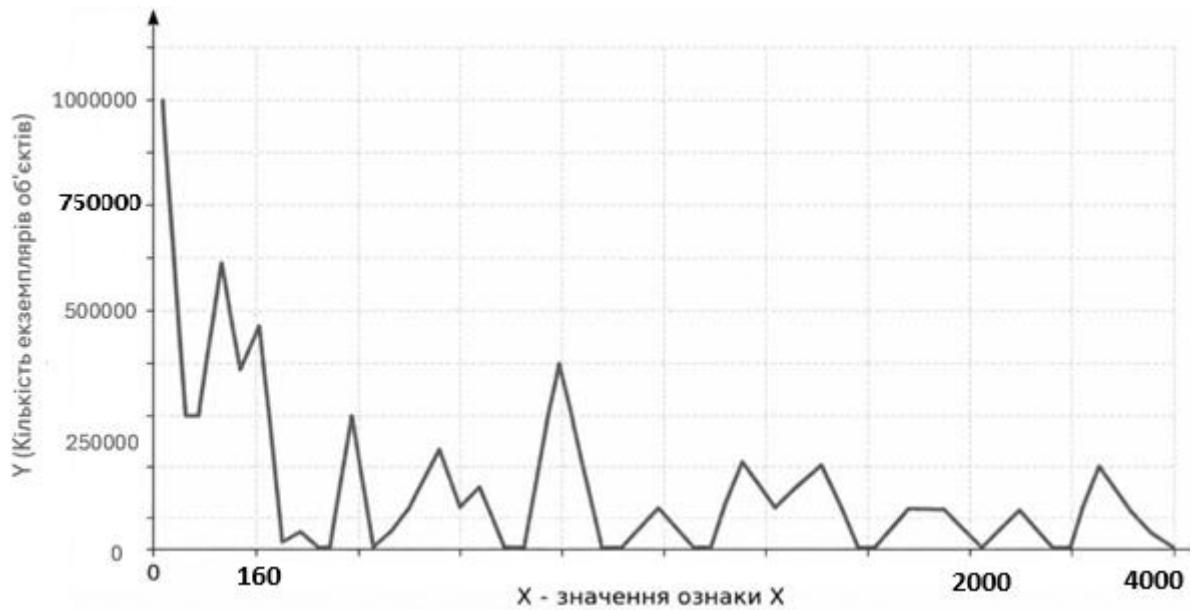


Рис 3.4 Розподіл об'єктів типу рядок після видалення найбільш повторюваних значень

Аналіз цього очищеного розподілу дозволив виділити наступні за масовістю групи дубльованих рядків. Результати аналізу цих найбільш повторюваних фонових значень наведено у таблиці 3.2. У таблиці кількість екземплярів означає кількість фізично різних рядкових об'єктів, що належать до одного класу еквівалентності за значенням відповідно до (2.31).

Кількість надлишкових екземплярів визначається з урахуванням того, що один представник класу має залишатися для збереження інформаційного змісту, як це передбачено (2.32). Сукупний обсяг пам'яті наведено для всіх екземплярів відповідного значення.

Табл. 3.2 Найбільш повторювані фонові значення типу System.String у промисловому знімку пам'яті

Рядкове значення	Кількість екземплярів у класі еквівалентності	Сукупний обсяг пам'яті	Інтерпретація
true	635181	2.4 МВ	повторюване логічне

			значення, подане у рядковій формі
HideStandardShippingMethod	390073	9.7 MB	повторюване службове або конфігураційне значення
false	318273	1.5 MB	повторюване логічне значення, подане у рядковій формі
Restricted due to ticket	257600	8.4 MB	повторюване текстове значення прикладної області
visa	241913	945 KB	повторюване значення прикладного класифікатора

Дані таблиці 3.2 показують, що дублювання характерне як для коротких рядкових значень, так і для довших службових або прикладних текстових значень. Короткі значення “*true*” та “*false*” мають найбільшу кількість екземплярів, але через малий розмір кожного окремого рядка їхній сумарний обсяг є меншим, ніж у довших значень з меншою кількістю повторів. Це свідчить про необхідність оцінювати дублювання не лише за кількістю екземплярів, а й за внеском у сумарний обсяг пам’яті відповідно до (2.33).

У результаті групування об'єктів типу *System.String* за значенням відповідно до моделі (2.29)–(2.33) встановлено, що у досліджуваному знімку пам'яті наявно приблизно 220 тис. різних рядкових значень, для яких існують повторні фізичні представлення. Сумарна кількість таких повторних екземплярів становить близько 14,5 млн. Цей обсяг формується з двох складових: понад 9 млн надлишкових екземплярів припадає на три головні аномалії, пов'язані з *Model.CountryToCountrySetting* (рис. 3.3), а решта утворює широкий спектр фонового дублювання прикладних і конфігураційних значень (лідери якого наведені у табл. 3.2). За загальної кількості близько 33 млн об'єктів типу *System.String* це відповідає приблизно 43 % рядкових об'єктів.

Сукупний обсяг пам'яті, зайнятий об'єктами типу *System.String*, становить 2164 МБ. Обсяг пам'яті, що припадає на повторні представлення вже наявних значень, становить приблизно 930 МБ. Отже, обсяг безнадлишкового представлення рядкових значень дорівнює приблизно 1234 МБ. Для підмножини об'єктів типу *System.String* значення коефіцієнта надмірності, введеного у (2.14), становить

$$\eta_{string} = \frac{2164}{2164 - 930} \approx 1.75$$

Значення η_{string} показує, що фактичний обсяг пам'яті, зайнятий рядковими об'єктами, приблизно у 1,75 рази перевищує обсяг, необхідний для подання відповідної множини значень без повторних фізичних представлень. Відносна частка надлишкового обсягу, що впливає з цього співвідношення, становить

$$1 - \frac{1}{\eta_{string}} \approx 0,43.$$

Таким чином, приблизно 43 % пам'яті, зайнятої об'єктами типу *System.String* у досліджуваному знімку, припадає на дубльовані представлення значень. За умови використання механізмів повторного застосування

незмінюваних значень цей обсяг може розглядатися як оцінка потенційного зменшення використання пам'яті для даного типу об'єктів.

Отримані результати не інтерпретуються як ознака витоку пам'яті, оскільки аналіз виконується для одного знімку пам'яті та не встановлює монотонного зростання кількості об'єктів у часі.

Для частини знімків, у яких *System.String* входив до групи найбільших за обсягом пам'яті типів, умови правила дублювання рядкових значень не виконувались. Частка таких випадків була порядку тридцяти відсотків. У цих станах кількість повторних фізичних представлень не досягала порогового рівня, однак у пам'яті зберігалася розширена множина інтернованих значень. До неї входили комбінації рядків, що проходили через обчислення раніше, але не обов'язково використовувалися у поточному стані програми. Тому відсутність спрацювання правила дублювання не інтерпретується як відсутність надмірного використання пам'яті для рядкових об'єктів. У таких випадках предметом аналізу є не кратність фізичних копій одного значення, а обсяг множини утримуваних у пам'яті значень та умови їх подальшого використання.

3.3 Експериментальна перевірка алгоритму прискореного виявлення дублікатів об'єктів типу *System.String*

Для типу $t_{imm} = \text{System.String}$ розглянуто підмножину об'єктів $O_{t_{imm}}^{imm}(u)$, виділену зі знімку пам'яті відповідно до формалізації незмінюваних типів, наведеної у (2.29)-(2.33). Об'єкти цього типу є незмінюваними, тому наявність декількох фізично різних екземплярів з однаковим значенням розглядається як дублювання інформаційного вмісту.

Повне попарне порівняння всіх об'єктів множини $O_{t_{imm}}^{imm}(u)$ потребує кількості перевірок, що визначається співвідношенням (2.34). Для зменшення

кількості таких перевірок застосовано попереднє групування за обчислювальною ознакою $X_{t_{imm}}$, введеною у (2.35). Для кожного об'єкта $o \in O_{t_{imm}}^{imm}(u)$ розглядається байтове представлення його значення $B_{t_{imm}}^{val}(o)$, після чого обчислюється значення

$$X_{t_{imm}}(o) = \varphi_{t_{imm}}(B_{t_{imm}}^{val}(o)).$$

У досліджуваному випадку відображення $\varphi_{t_{imm}}$ реалізовано як швидко обчислювана числова характеристика байтового представлення значення рядка. Така ознака не є унікальним ідентифікатором рядкового значення, тому остаточне встановлення дублювання виконується за рівністю значень $Val_{t_{imm}}$, як це передбачено умовою (2.36).

Після обчислення $X_{t_{imm}}(o)$ для всіх об'єктів типу *System.String* визначаються мінімальне та максимальне значення цієї ознаки. Відповідний діапазон розбивається на h напівінтервалів $I_{t_{imm},p}^h(u)$ згідно з (2.37). Для кожного напівінтервалу формується підмножина $P_{t_{imm},p}^h(u)$, що містить об'єкти, значення ознаки яких належать цьому інтервалу. Точне порівняння рядкових значень виконується лише всередині таких підмножин. Як було наочно продемонстровано у підрозділі 3.2 (рис. 3.3 та рис. 3.4), запропонований алгоритм попереднього групування за ознакою $X_{t_{imm}}$ дозволяє чітко локалізувати класи еквівалентності як для глобальних аномалій, так і для фонового дублювання. У цьому ж підпункті алгоритм розглядається виключно з точки зору аналізу його обчислювальних властивостей та оптимізації часу виконання.

Для оцінювання впливу кількості підмножин h на час виконання алгоритму проведено серію обчислювальних експериментів. У кожному експерименті фіксувалися кількість підмножин, максимальна кількість елементів у підмножині, довжина інтервалу Δ та час виконання. Результати наведено у табл. 3.3.

Табл. 3.3 - Експериментальна оцінка впливу кількості підмножин на час пошуку дублікатів об'єктів типу System.String

Кількість підмножин h	Максимальна кількість елементів у підмножині	Розмір Δ	Час виконання, мс.
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	1994.70

Без попереднього групування час обробки становив 7464 мс. Для $h = 512$ час виконання становив 1894,38 мс, що відповідає зменшенню часу приблизно у 3,94 рази. Найменше значення часу у наведеній серії отримано для $h = 10000$, однак подальше збільшення кількості підмножин не приводить до пропорційного скорочення часу виконання.

Залежність часу виконання від кількості підмножин наведено на рис. 3.5. Зі збільшенням h зменшується кількість елементів у найбільших підмножинах, однак одночасно зростає кількість службових груп, які необхідно створювати

та обробляти. Тому вибір параметра h не зводиться лише до мінімізації часу одного запуску, а має враховувати також накладні витрати на формування та підтримку структури групування, що узгоджується з постановкою (2.40).

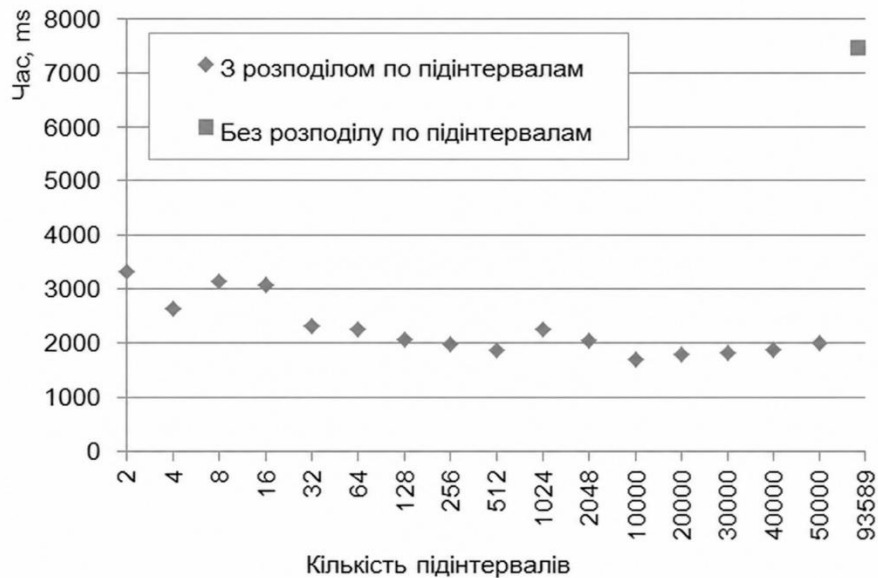


Рис 3.5 Залежність часу виконання від кількості підмножин

На рис. 3.6 наведено залежність максимальної кількості елементів у підмножині від кількості підмножин. Зменшення максимальної потужності підмножини є наслідком збільшення h , однак ця залежність також не є пропорційною. Це пов'язано з нерівномірним розподілом рядкових значень за ознакою $X_{t_{imm}}$ та наявністю класів еквівалентності великої потужності.

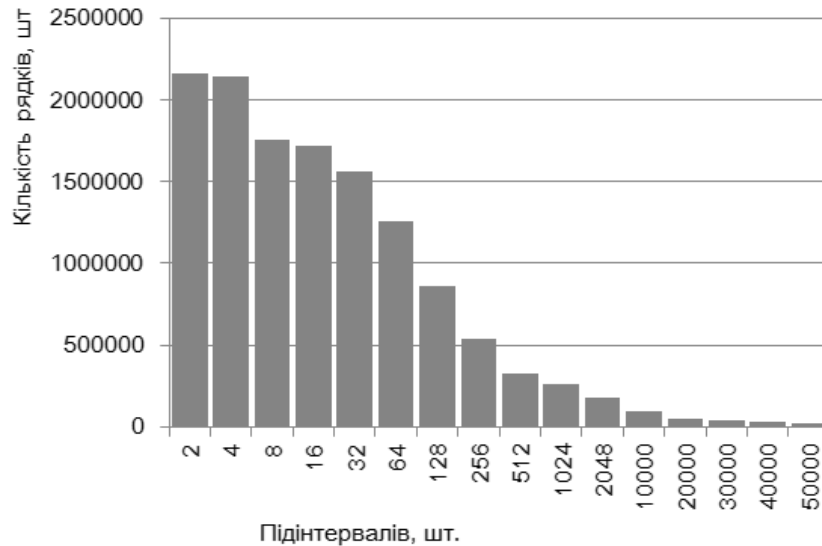


Рис 3.6 Залежність максимальної кількості елементів у підмножині від кількості підмножин

Отримані результати узгоджуються з оцінками (2.38)–(2.40): попереднє групування зменшує кількість точних порівнянь, але не гарантує лінійного зменшення часу виконання через додаткові витрати на обробку підмножин. Значення h , близькі до $5 \cdot 10^2$, забезпечують значне скорочення часу порівняно з повним перебором при помірній кількості підмножин. Подальше збільшення h може зменшувати потужність окремих груп, але потребує додаткового обсягу пам'яті для зберігання та обробки структури розбиття.

Отже, застосування попереднього групування за обчислювальною ознакою $X_{t_{imm}}$ з подальшою перевіркою рівності значень $Val_{t_{imm}}$ дозволяє скоротити час виявлення дублікатів об'єктів типу *System.String* без зміни критерію встановлення дублювання. Отримані результати не змінюють кількісну оцінку надмірного використання пам'яті, наведену у підпункті 3.2, а характеризують обчислювальні властивості алгоритму виявлення повторних фізичних представлень незмінюваних рядкових значень.

3.4 Оцінювання компактного подання ключового типу для Dictionary<K,V>

За результатами первинного аналізу використання пам'яті, наведеної у табл. 3.1, тип `Model.Shipping.GroupedShippingMethodZoneKey` має велику кількість екземплярів і використовується у складі хеш-колекцій. Для такого типу надмірне використання пам'яті може бути пов'язане не тільки з дублюванням значень, а з особливостями його представлення у 64-бітному середовищі CLR: збереженням екземплярів як посилальних об'єктів, службовим заголовком об'єкта, покажчиками в структурах `Dictionary<K,V>` та надлишковим діапазоном оголошених типів полів.

Для аналізу було розглянуто тип

$$t_{sh} = \text{Model.Shipping.GroupedShippingMethodZoneKey}.$$

Відповідно до подання поля об'єкта у вигляді трійки $(p, \delta_p, x_p(o))$, введеного у (2.46), для кожного поля p цього типу було визначено множину фактичних значень $x_{t_{sh},p}(u)$ за (2.47). На основі цих значень побудовано фактичні діапазони $R_{t_{sh},p}^{obs}(u)$, що використовуються для оцінювання коефіцієнта використання діапазону $\rho_{t_{sh},p}$ згідно з (2.48).

Результати аналізу діапазонів значень полів типу `GroupedShippingMethodZoneKey` наведено у табл. 3.4.

Табл. 3.4 - Діапазони значень полів типу `GroupedShippingMethodZoneKey`

Поле p	Оголошений тип δ_p	Фактичний діапазон	Коефіцієнт використання діапазону $\rho_{t,p}$	Можливе компактне подання
<code>shippingMethodId</code>	<code>long</code>	[1-40,048,460]	$2,17 \cdot 10^{-12}$	<code>int</code>
<code>shippingMethodZoneId</code>	<code>long</code>	[1-429]	$2,33 \cdot 10^{-17}$	<code>ushort</code>
<code>HashCode</code>	<code>int</code>	[<code>int.Min</code> - <code>int.Max</code>]	1	-

Для поля `shippingMethodId` фактичний діапазон значень належить підмножині додатних значень і не перевищує верхню межу типу `int`. Тому за умови виконання включення $R_{t_{sh},p}^{obs,\gamma}(u) \subset D_d$, визначеного у (2.49),

допустимою заміною для цього поля може бути тип $d = \text{int}$, $\rho_{t,p} \approx 0,00932$ (0,93%). Потенційне зменшення розміру одного значення поля у цьому випадку визначається як

$$\Delta w_{t_{sh,p}}(u) = w(\delta_p) - w(\delta_{t_{sh,p}}^*(u)) = 8 - 4 = 4 \text{ байти.}$$

Для поля `shippingMethodZoneId` фактичні значення належать діапазону $[1; 429]$. З урахуванням предметного обмеження невід'ємності і виконання умови (2.49) це поле може бути подане типом $d = \text{ushort}$, $\rho_{t,p} \approx 0,00655$ (0,65%). У цьому випадку зменшення розміру одного значення становить

$$\Delta w_{t_{sh,p}}(u) = 8 - 2 = 6 \text{ байтів.}$$

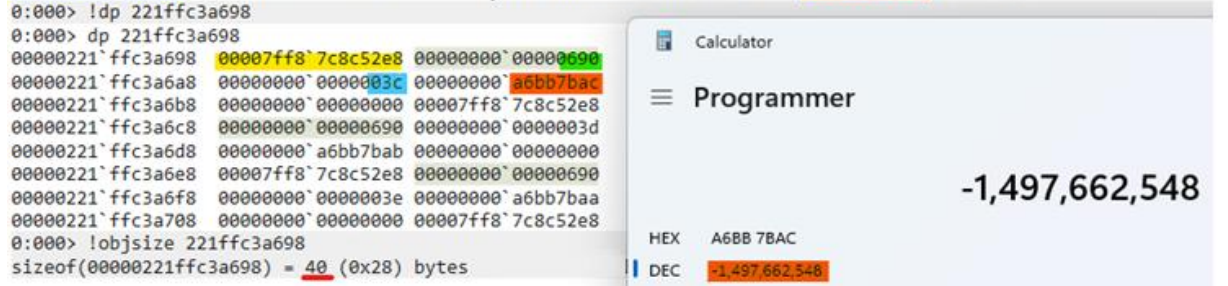
Поле `HashCode` не розглядається як кандидат на звуження діапазону за моделлю (2.48)-(2.49), оскільки його значення відповідають діапазону типу `int`. Воно може бути виключене зі складу збережених полів, бо значення хеш-коду однозначно обчислюється на основі інших полів ключа. У такому випадку зменшення пам'яті пов'язане не зі звуженням типу, а зі зміною структури подання об'єкта.

Початкове подання `GroupedShippingMethodZoneKey` як класу у 64-бітному середовищі CLR передбачає виділення окремого об'єкта у керованій купі. Згідно з даними WinDBG, розмір одного такого об'єкта становить 40 байтів. Оскільки об'єкт використовується як ключ у словнику, у внутрішній структурі `Dictionary<K,V>` додатково зберігається покажчик на нього розміром 8 байтів. Таким чином, для одного ключа, представленого як посилальний тип, фактичний обсяг пам'яті з урахуванням покажчика у словнику становить 48 байтів.

```

0:000> !do 221ffc3a698
Name: Shipping.GroupedShippingMethodZoneKey
MethodTable: 00007ff87c8c52e8
EEClass: 00007ff87c8b5798
Size: 40(0x28) bytes
File: C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET Files\root\85c487b7\cebb558a\as:
Fields:
      MT      Field      Offset      Type VT      Attr      Value Name
00007ff8d05aaea8 40006e7      8      System.Int64 1 instance 1680 shippingMethodId
00007ff8d05aaea8 40006e8     10      System.Int64 1 instance 60 shippingMethodZoneId
00007ff8d05a85a0 40006e9     18      System.Int32 1 instance -1497662548 hashCode
0:000> !dp 221ffc3a698
0:000> dp 221ffc3a698
00000221`ffc3a698 00007ff8`7c8c52e8 00000000`00000690
00000221`ffc3a6a8 00000000`0000003c 00000000`a6bb7bac
00000221`ffc3a6b8 00000000`00000000 00007ff8`7c8c52e8
00000221`ffc3a6c8 00000000`00000690 00000000`0000003d
00000221`ffc3a6d8 00000000`a6bb7bab 00000000`00000000
00000221`ffc3a6e8 00007ff8`7c8c52e8 00000000`00000690
00000221`ffc3a6f8 00000000`0000003e 00000000`a6bb7baa
00000221`ffc3a708 00000000`00000000 00007ff8`7c8c52e8
0:000> !objsize 221ffc3a698
sizeof(00000221ffc3a698) = 40 (0x28) bytes

```



The image shows a debugger window with a memory dump and a calculator window. The memory dump shows the fields of a GroupedShippingMethodZoneKey object. The calculator window shows the decimal value -1,497,662,548, which is the hashCode of the shippingMethodZoneId field.

Рис 3.7. Початковий розмір об'єкта GroupedShippingMethodZoneKey у пам'яті

Інформаційний вміст двох полів, необхідних для ідентифікації ключа, після звуження типів може бути поданий значеннями розміром 4 та 2 байти. Отже, мінімальний обсяг для збереження цих значень стандартними числовими типами становить 6 байтів. Фактичний розмір структури у пам'яті визначається правилами вирівнювання CLR, однак перехід від класу до значимого типу усуває службовий заголовок окремого об'єкта та необхідність зберігати покажчик на нього.

З урахуванням наведеного, компактне подання ключа може бути реалізоване як readonly struct, що містить поля зі звуженими типами та обчислюваний хеш-код. Властивість readonly фіксує незмінність значення після створення, що відповідає використанню об'єкта як ключа у Dictionary<K,V>. При цьому ключ зберігається без додаткового об'єкта в керованій купі, а його значення розміщується безпосередньо у відповідній внутрішній структурі словника.

Отримані результати показують, що для типу GroupedShippingMethodZoneKey надмірне використання пам'яті формується за рахунок поєднання двох чинників: посилального подання ключа та надлишкового діапазону оголошених типів полів. Застосування співвідношень

(2.46)-(2.49) дозволяє формально встановити допустимість звуження окремих полів за фактичними діапазонами значень, а аналіз розміщення об'єкта у пам'яті показує ефект від переходу до значимого типу.

У частині розглянутих користувацьких типів умови звуження типів полів не виконувались. Частка таких випадків становила близько тридцяти відсотків серед типів, для яких аналізувалися фактичні діапазони значень. Основними причинами були близьке до повного використання діапазону оголошеного типу або недостатній запас для заміни згідно з (2.49). Для таких типів початкове подання поля не класифікується як надлишкове за діапазоном.

Оцінювання коефіцієнта використання діапазону $\rho_{t_{sh},p}$ для одного знімка пам'яті не є достатнім для остаточного висновку щодо допустимості звуження типу. Для підтвердження стабільності фактичного діапазону значень аналіз необхідно виконувати для знімків пам'яті, отриманих у різні періоди роботи програмного додатку. У таких випадках перевіряється сталість меж $R_{t_{sh},p}^{obs}(u)$ та відсутність тенденції до наближення фактичного діапазону до меж оголошеного типу.

3.4.1 Експериментальне оцінювання компактного подання ключового типу

Для обчислювального порівняння розглянуто дві реалізації типу `Model.Shipping.GroupedShippingMethodZoneKey`. Перша реалізація відповідає початковому поданню у вигляді класу, екземпляри якого розміщуються у керованій купі як окремі об'єкти. Друга реалізація подає ключ як `readonly struct`, у якій поле `shippingMethodId` зберігається типом `int`, поле `shippingMethodZoneId` - типом `ushort`, а значення хеш-коду обчислюється на основі значень цих полів.

У кожному експерименті використовувався однаковий набір значень полів `shippingMethodId` та `shippingMethodZoneId`, отриманий зі знімку пам'яті. Для обох реалізацій виконувались операції формування ключів, завантаження даних до `Dictionary<K,V>` та пошуку елементів за ключем. Вимірювання часу виконання та обсягу виділеної пам'яті здійснювалося за допомогою `Benchmark.NET` [101].

На рис. 3.8 наведено результати експерименту для сценарію завантаження даних до словника.

```
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

Рис. 3.8 Результати експерименту із завантаження даних

Для початкового подання обсяг виділеної пам'яті становив 533 МБ, а час виконання - близько 1 с. Для компактного подання відповідні значення становили 46,6 МБ та 54 мс; обсяг виділеної пам'яті зменшився приблизно у 11,4 рази; час виконання - приблизно у 18,5 рази.

Отримані значення узгоджуються з оцінкою розміру ключового типу, наведеною у підпункті 3.4. Початкове подання одного ключа як посилального типу складається з окремого об'єкта розміром 40 байтів та покажчика на нього у внутрішній структурі словника розміром 8 байтів. Отже, фактичний обсяг, пов'язаний з одним ключем у такому поданні, становить 48 байтів.

У компактному поданні ключ розміщується як значення. Після звуження типів полів та вирівнювання його розмір становить 8 байтів. Таким чином, різниця між початковим і компактним поданням одного ключа дорівнює 40 байтам. За кількості 6 109 902 екземплярів маємо

$$6\,109\,902 \cdot 40 = 244\,396\,080$$

байтів, тобто приблизно 233 МБ. Ця оцінка стосується подання ключового типу і не враховує інші службові структури Dictionary<K,V>.

Окремо було виконано порівняння операції пошуку елемента за ключем. Результати наведено на рис. 3.9.

```
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

Рис. 3.9 Результати оцінювання швидкодії пошуку елемента в кеші

У початковому варіанті виконання пошуку супроводжується формуванням ключа як посилального об'єкта. У цьому випадку кожна така операція потребує виділення пам'яті для окремого екземпляра ключового типу. У компактному варіанті ключ подається значенням, тому формування ключа не потребує створення додаткового об'єкта у керованій купі. За даними експерименту, час пошуку елемента зменшився приблизно у десять разів.

Зміна способу подання ключа не змінює логіку роботи Dictionary<K,V> за умови збереження правил порівняння ключів та обчислення хеш-коду. У розглянутому випадку змінюється лише форма представлення ключового типу в пам'яті: замість окремого об'єкта з службовим заголовком і посиланням використовується значиме подання з полями меншого розміру.

Таким чином, результати експериментів узгоджуються з аналізом, наведеним у підпункті 3.4. Звуження типів полів відповідно до фактичних діапазонів значень за (2.48)-(2.49) зменшує обсяг даних, необхідний для подання ключа, а перехід від класу до readonly struct усуває службові витрати, пов'язані з окремим об'єктом у керованій купі. Отримані числові значення

характеризують вплив компактного подання ключового типу на обсяг виділеної пам'яті та час виконання операцій завантаження й пошуку у проведених експериментах.

Для частини типів, що використовувалися як ключі у *Dictionary*<*K*,*V*>, перехід від класу до значимого типу не розглядався як допустима заміна. Частка таких випадків була порядку сорока відсотків серед перевірених ключових типів. До цієї групи належали типи з посиляльними компонентами, нетривіальною логікою порівняння. У таких випадках зміна форми подання могла змінити семантику ключа або не дати кількісно помітного зменшення пам'яті.

3.5 Експериментальне оцінювання вартості зберігання інформації у *Dictionary*<*K*,*V*> для малих кількостей елементів

У цьому підпункті розглянуто застосування моделі оцінювання вартості зберігання інформації у хеш-колекціях типу *Dictionary*<*K*,*V*>, наведеної у (2.41)–(2.45). Аналіз виконується для колекцій з малою кількістю елементів, оскільки для таких випадків службові структури словника можуть становити більшу частину фактичного обсягу пам'яті.

На рис. 3.10 наведено приклад об'єкта типу *Dictionary*<*K*,*V*> у знімку пам'яті. Об'єкт подано відповідно до формального опису (2.4): він має адресу, тип, розмір та множину значень полів. За даними знімку пам'яті, розмір самого об'єкта словника без урахування допоміжних об'єктів становить 80 байтів.

```

0:000> !do 0000203f2000928 1
Name: System.Collections.Generic.Dictionary`2[[System.Int32, System.Private.CoreLib],[Model.Search.IndexedProduct,Engine]]
MethodTable: 00007ffae756b270 2
EEClass: 00007ffae5a4d838
Tracked Type: false
Size: 80(0x50) bytes 3
File: C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.12\System.Private.CoreLib.dll
Fields:
    MT      Field      Offset      Type      VT      Attr      Value      Name
00007ffae51ebf38 4002113      8      System.Int32[] 0 instance 0000203f2000978 _buckets
00007ffae615bb30 4002114     10 ...ivate.CoreLib]][] 0 instance 0000203f20009a0 _entries
00007ffae51b3cf0 4002115     30      System.UInt64 1 instance 6148914691236517206 _fastModMultiplier
00007ffae516e8d0 4002116     38      System.Int32 1 instance          1 _count
00007ffae516e8d0 4002117     3c      System.Int32 1 instance         -1 _freeList
00007ffae516e8d0 4002118     40      System.Int32 1 instance          0 _freeCount
00007ffae516e8d0 4002119     44      System.Int32 1 instance          1 _version
00007ffae5f16f40 400211a     18 ...Private.CoreLib]] 0 instance 0000000000000000 _comparer
00007ffae79fc0d8 400211b     20 ...Private.CoreLib]] 0 instance 0000000000000000 _keys
00007ffae5edcc28 400211c     28 ...Private.CoreLib]] 0 instance 0000206fe93aec0 _values

```

Рис. 3.10 Приклад об'єкта типу Dictionary<K,V> у знімку пам'яті

На рис. 3.11 наведено склад полів екземпляра словника. До них належать посилання на внутрішні масиви `_buckets` та `_entries`, службові поля для обліку кількості елементів, вільних позицій, версії колекції, множника для обчислення номера комірки, а також посилання на об'єкти представлення ключів і значень.

```

0:000> !do 0000203f2000928
Name: System.Collections.Generic.Dictionary`2[[System.Int32, System.Private.CoreLib],[Relwise.Engine.DataTypes.Serializable.Model.Search.IndexedProduct, Relwise.Engine]]
MethodTable: 00007ffae756b270
EEClass: 00007ffae5a4d838
Tracked Type: false
Size: 80(0x50) bytes
File: C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.12\System.Private.CoreLib.dll
Fields:
    MT      Field      Offset      Type      VT      Attr      Value      Name
00007ffae51ebf38 4002113      8      System.Int32[] 0 instance 0000203f2000978 _buckets
00007ffae615bb30 4002114     10 ...ivate.CoreLib]][] 0 instance 0000203f20009a0 _entries
00007ffae51b3cf0 4002115     30      System.UInt64 1 instance 6148914691236517206 _fastModMultiplier
00007ffae516e8d0 4002116     38      System.Int32 1 instance          1 _count
00007ffae516e8d0 4002117     3c      System.Int32 1 instance         -1 _freeList
00007ffae516e8d0 4002118     40      System.Int32 1 instance          0 _freeCount
00007ffae516e8d0 4002119     44      System.Int32 1 instance          1 _version
00007ffae5f16f40 400211a     18 ...Private.CoreLib]] 0 instance 0000000000000000 _comparer
00007ffae79fc0d8 400211b     20 ...Private.CoreLib]] 0 instance 0000000000000000 _keys
00007ffae5edcc28 400211c     28 ...Private.CoreLib]] 0 instance 0000206fe93aec0 _values

```

Рис. 3.11 Поля об'єкта типу Dictionary<K,V>

У розглянутому прикладі масив `_buckets` займає 36 байтів, а масив `_entries` - 96 байтів. Крім того, словник містить допоміжний об'єкт представлення значень, розмір якого становить 24 байти. Розмір допоміжних об'єктів, пов'язаних з реалізацією словника, наведено на рис. 3.12.

```

0:000> !DumpObj /d 00000203f2000978
Name:      System.Int32[]
MethodTable: 00007ffae51ebf38
EEClass:   00007ffae51eb8
Tracked Type: false
Size:      36(0x24) bytes
Array:     Rank 1, Number of elements 3, Type Int32 (Print Array)
Fields:
None
0:000> !DumpObj /d 00000203f20009a0
Name:      System.Collections.Generic.Dictionary`2+Entry[[System.Int32, System.Private.CoreLib],]
MethodTable: 00007ffae756da60
EEClass:   00007ffae756d9c8
Tracked Type: false
Size:      96(0x60) bytes
Array:     Rank 1, Number of elements 3, Type VALUETYPE (Print Array)
Fields:
None
0:000> !DumpObj /d 00000206fe93aec0
Name:      System.Collections.Generic.Dictionary`2+ValueCollection[[System.Int32, System.Private.
MethodTable: 00007ffae75d3de0
EEClass:   00007ffae5ebf808
Tracked Type: false
Size:      24(0x18) bytes
File:      C:\Program Files\dotnet\shared\Microsoft.NETCore.App\7.0.12\System.Private.CoreLib.dll

```

Рис. 3.12 Розмір допоміжних об'єктів усередині Dictionary<K,V>

Отже, для словника з однією парою ключ–значення фактичний обсяг пам'яті складається з розміру самого словника, масиву `_buckets`, масиву `_entries` та допоміжного об'єкта представлення значень:

$$80 + 36 + 96 + 24 = 236 \text{ байтів.}$$

Для пари `int-object` корисний інформаційний обсяг становить 12 байтів: 4 байти для ключа типу `int` та 8 байтів для посилання на значення. Тоді частка корисного інформаційного обсягу у фактичному розмірі словника з одним елементом дорівнює $12 / 236 \approx 0,051$. Відповідно, фактичний обсяг пам'яті перевищує обсяг корисної інформації приблизно у $236 / 12 \approx 19,67$ рази. Це співвідношення узгоджується з моделлю (2.42), у якій фактичний розмір словника подається як сума корисного інформаційного обсягу та службових витрат, пов'язаних з реалізацією хеш-колекції.

На рис. 3.13 наведено розподіл словників за фактичною кількістю збережених елементів. У досліджуваному знімку пам'яті наявна велика кількість словників з малою кількістю елементів. Для подальшої кількісної оцінки розглянуто підмножину словників, що містять одну пару ключ–значення.

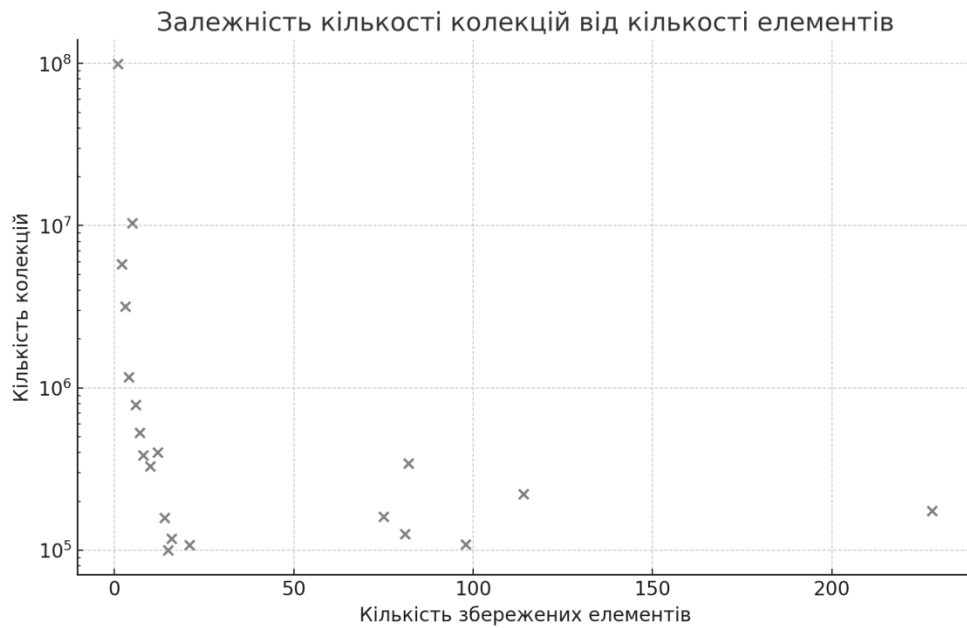


Рис. 3. 13 Розподіл словників за кількістю збережених елементів

Для зменшення службових витрат пам'яті розглянуто два варіанти компактного подання словника з малою кількістю елементів.

Перший варіант орієнтований на випадок, коли ключ має числовий тип. У цьому разі можливе скорочення частини службових структур, пов'язаних із загальною реалізацією словника: відмова від окремого поля порівнювача, динамічне створення представлень ключів і значень лише за потреби, використання сталого значення множника для 64-бітного середовища, об'єднання частини внутрішніх структур і зменшення зарезервованої ємності до фактичної кількості елементів. За наведеними розрахунками розмір одного такого словника становить 88 байтів.

Другий варіант розглядається для випадку, коли після формування колекції вона використовується лише для читання. Для словника з одним елементом у цьому випадку можна зберігати пару ключ–значення без масиву `_entries`, оскільки операції додавання та видалення більше не виконуються. За таких умов частина службових полів, зокрема `_count`, `_freeList` та `_freeCount`, не бере участі у подальшому доступі до даних. Розмір одного компактного об'єкта для зберігання однієї пари ключ–значення становить 32 байти.

Кількісні оцінки для сукупності з 99 172 580 словників з одним елементом наведено у табл. 3.5.

Табл. 3.5 - Кількість пам'яті для збереження однієї пари значень

Кількість об'єктів: 99172580	Байт			ГБ		
	Один об'єкт			Сумарний розмір даних		
	Розмір даних	Додатковий технічний розмір	Зменшення відносно стандартного	Розмір даних	Додатковий технічний розмір	Зменшення відносно стандартного
Корисна інформація	12	0	0	1,11	0,00	0,00
Стандартний словник	236	224	0	21,80	20,69	0,00
Компактне подання для числового ключа	88	76	148	8,13	7,02	13,67
Компактне подання для одного значення	32	20	204	2,96	1,85	18,84

З даних табл. 3.5 випливає, що стандартне подання 99 172 580 словників з одним елементом потребує 21,80 ГБ пам'яті, тоді як корисний інформаційний обсяг відповідних пар ключ–значення становить 1,11 ГБ. Для компактного подання, орієнтованого на числовий ключ, сумарний обсяг становить 8,13 ГБ, а зменшення відносно стандартного подання - 13,67 ГБ. Для подання, орієнтованого на один незмінний елемент, сумарний обсяг становить 2,96 ГБ, а зменшення відносно стандартного подання - 18,84 ГБ.

Наведені оцінки стосуються пам'яті, що використовується для подання словників з одним елементом, і не включають інші об'єкти, які можуть бути пов'язані з логікою прикладної системи. Перехід до компактного

подання також накладає умови на сценарій використання колекції. Зокрема, подання для одного значення є коректним лише тоді, коли етап наповнення даними відокремлений від етапу читання і після формування колекція не змінюється.

Для перевірки впливу компактних подань на час виконання операцій було проведено серію обчислювальних експериментів із застосуванням Benchmark.NET [101]. Порівнювались стандартна реалізація словника та спеціалізовані реалізації для малих кількостей елементів. У досліджах вимірювався час створення екземпляра колекції, обсяг виділеної пам'яті, час пошуку наявного елемента та час пошуку відсутнього елемента.

На рис. 3.14 наведено результати вимірювання часу створення екземпляра колекції. Для словника з одним елементом стандартна реалізація створюється приблизно у 8 разів довше, ніж компактне подання. Для словника із сімома елементами стандартна реалізація потребує приблизно у 6 разів більше часу, ніж подання з фіксованою кількістю елементів, і приблизно удвічі більше часу, ніж компактне подання зі змінною кількістю елементів.

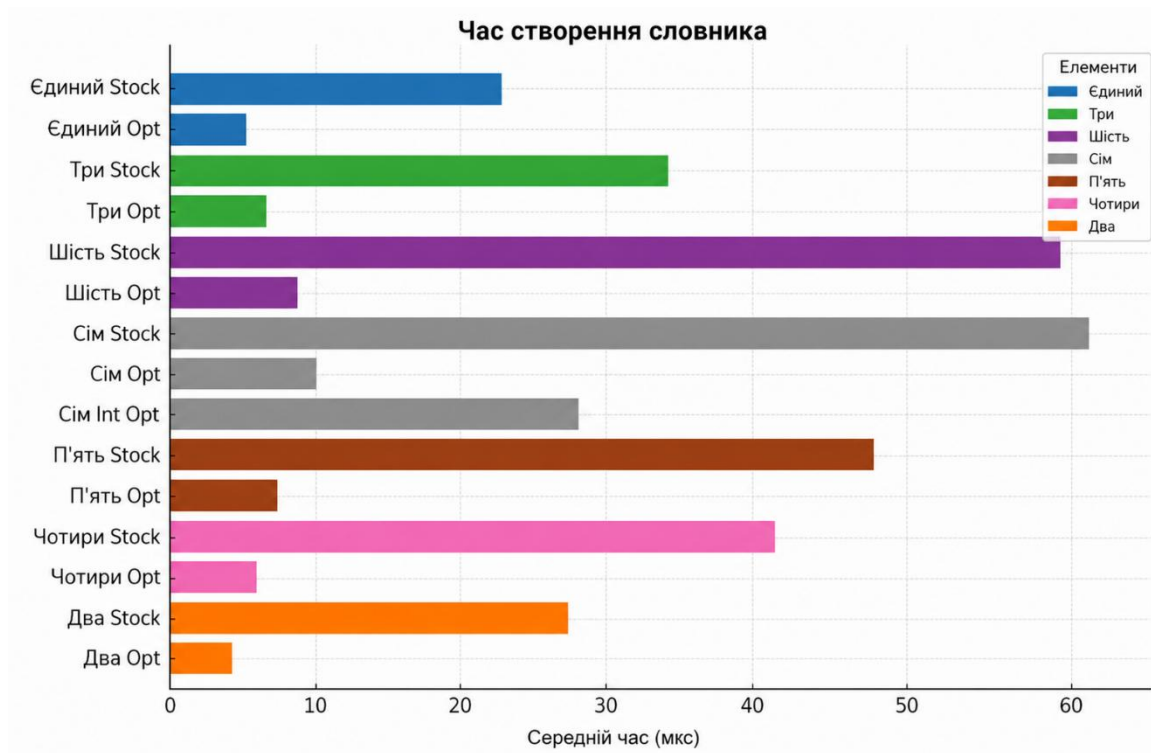


Рис. 3.14 Час створення екземпляра колекції

На рис. 3.15 наведено обсяг пам'яті, що виділяється під час створення екземпляра колекції. Для словника з одним елементом стандартна реалізація потребує приблизно у 7,5 рази більше пам'яті, ніж компактне подання. Для словника із сімома елементами різниця становить приблизно 3 рази.

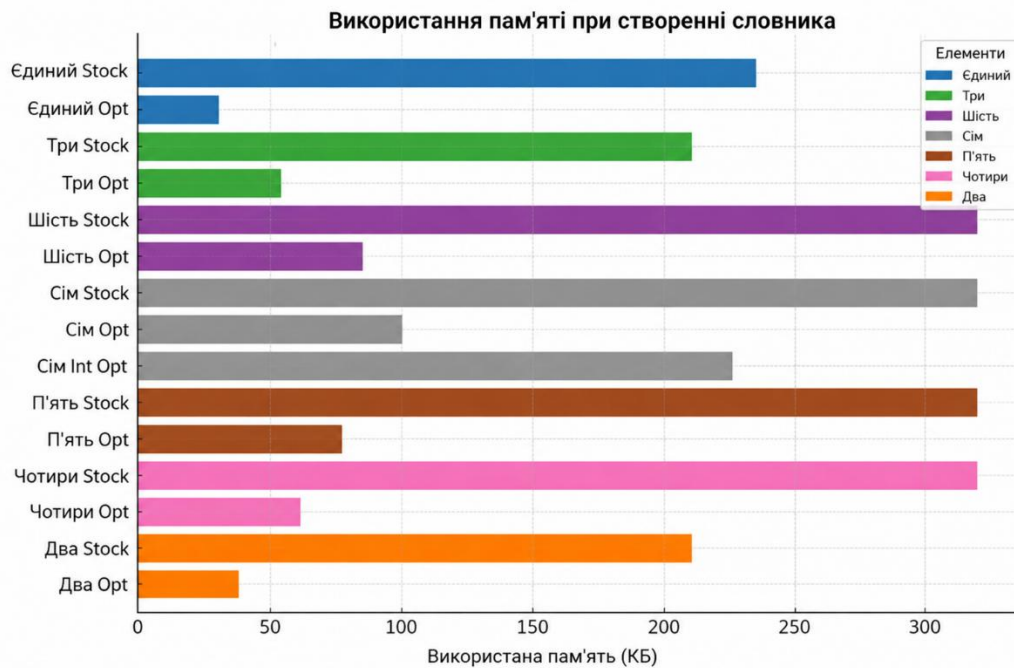


Рис. 3.15 Обсяг пам'яті, що виділяється під час створення екземпляра колекції

На рис. 3.16 наведено результати вимірювання часу пошуку наявного елемента. Для колекції з одним елементом пошук у компактному поданні виконується приблизно у 17 разів швидше, ніж у стандартному словнику. Для колекції з шістьма елементами різниця становить приблизно 15,5 рази, а для колекції із сімома елементами - близько 2 разів.

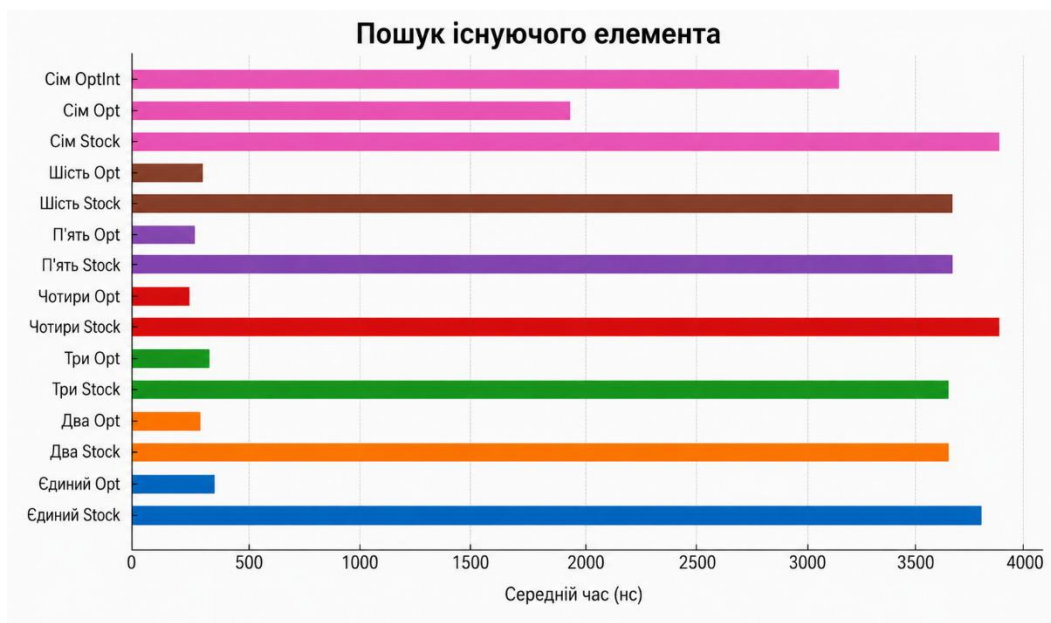


Рис. 3.16 Час пошуку наявного елемента в колекції

На рис. 3.17 наведено результати вимірювання часу пошуку відсутнього елемента. Отримані значення мають той самий характер залежності: для малих кількостей елементів компактно подання зменшує час виконання операції, тоді як зі збільшенням кількості елементів перевага лінійного або фіксованого подання зменшується.

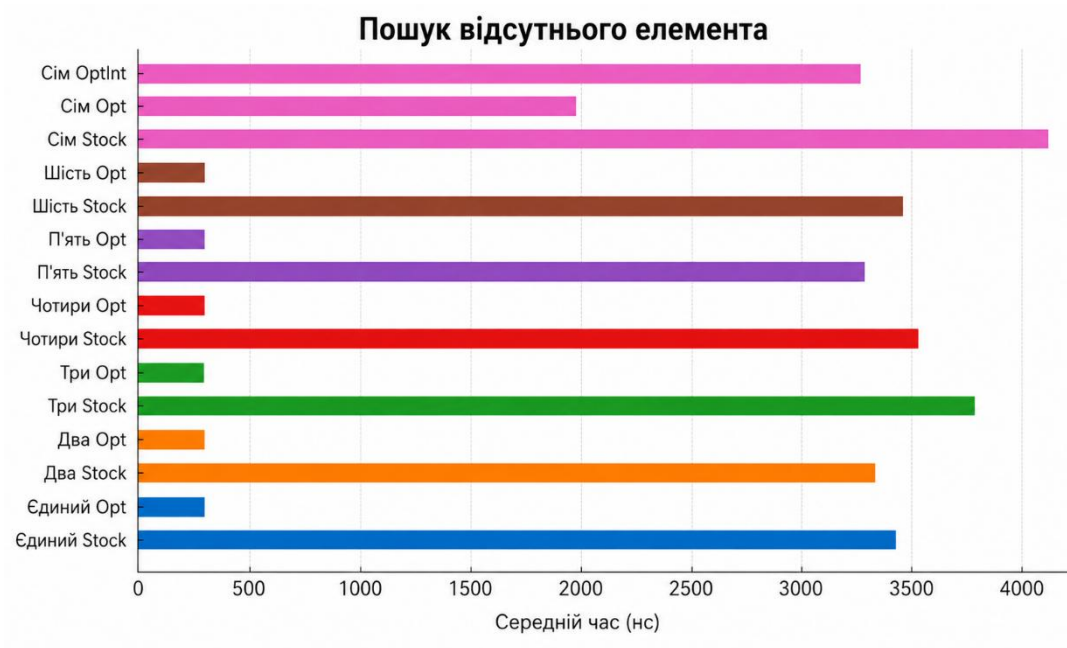


Рис 3.17 Час пошуку відсутнього елемента в колекції

Результати вимірювань показують, що для малих кількостей елементів службові витрати стандартної реалізації Dictionary<K,V> можуть перевищувати обсяг корисної інформації на порядок. Компактні подання зменшують кількість допоміжних об'єктів і розміщують дані у більш щільній формі. Це знижує обсяг виділеної пам'яті та зменшує кількість переходів за посиланнями під час доступу до елементів.

Зі збільшенням кількості елементів характер залежності змінюється. Для більших колекцій стандартна хеш-структура поступово набуває переваги завдяки сталому за порядком доступу до елемента. Тому спеціалізовані подання доцільно розглядати лише для колекцій з малою кількістю елементів, що узгоджується з групуванням словників за кількістю елементів у (2.43) та агрегованим оцінюванням груп у (2.44).

Отже, результати цього підпункту узгоджуються з моделлю (2.41)-(2.45): для словників з малою кількістю елементів фактичний обсяг пам'яті значною мірою визначається службовими структурами, а не корисним інформаційним вмістом. Кількісне оцінювання за даними знімку пам'яті дозволяє виділити групи словників, для яких компактне подання зменшує обсяг виділеної пам'яті. Обчислювальні експерименти показують, що для таких груп зменшується також час виконання операцій створення та пошуку при малих значеннях кількості елементів.

У вибірці були групи словників, для яких умови компактного подання не виконувались. Частка таких груп була близько тридцяти відсотків від кількості груп, оцінених за числом елементів. До них здебільшого належали словники з активними операціями додавання або видалення. Для таких груп стандартна хеш-структура не розглядається як надлишкова за критеріями (2.41)–(2.45).

3.6 Експериментальне оцінювання стискання об'єктів типу System.Byte[]

Для оцінювання підходу, заданого співвідношеннями (2.50)-(2.55), було використано множину об'єктів типу System.Byte[], виділену зі знімку пам'яті. Для кожного об'єкта розглядалися довжина масиву, інформаційний вміст та результат застосування безвтратного відображення стискання.

Знімок пам'яті мав обсяг близько 100 ГБ. У ньому було виявлено приблизно 4,9 млн об'єктів типу System.Byte[]. Сукупний інформаційний вміст цих масивів становив близько 2,3 млрд байтів. Основні числові характеристики наведено у табл. 3.6.

Табл. 3.6 Характеристики множини об'єктів типу System.Byte[]

Показник	Значення
Обсяг знімку пам'яті	~100 ГБ
Кількість об'єктів типу System.Byte[]	~4.9 млн
Сукупний інформаційний вміст масивів	~ 2,3 млрд байтів
Зменшення обсягу після стискання об'єктів-кандидатів	0,95 ГБ
Відносне зменшення від сумарного обсягу об'єктів типу System.Byte[]	~ 45 %

На рис. 3.18 та рис. 3.19 наведено розподіл кількості масивів за їх довжиною. Найбільшу кількість екземплярів мають масиви довжиною 10 елементів. У досліджуваних даних такі масиви використовуються для подання сегментів користувачів у бінарній формі.

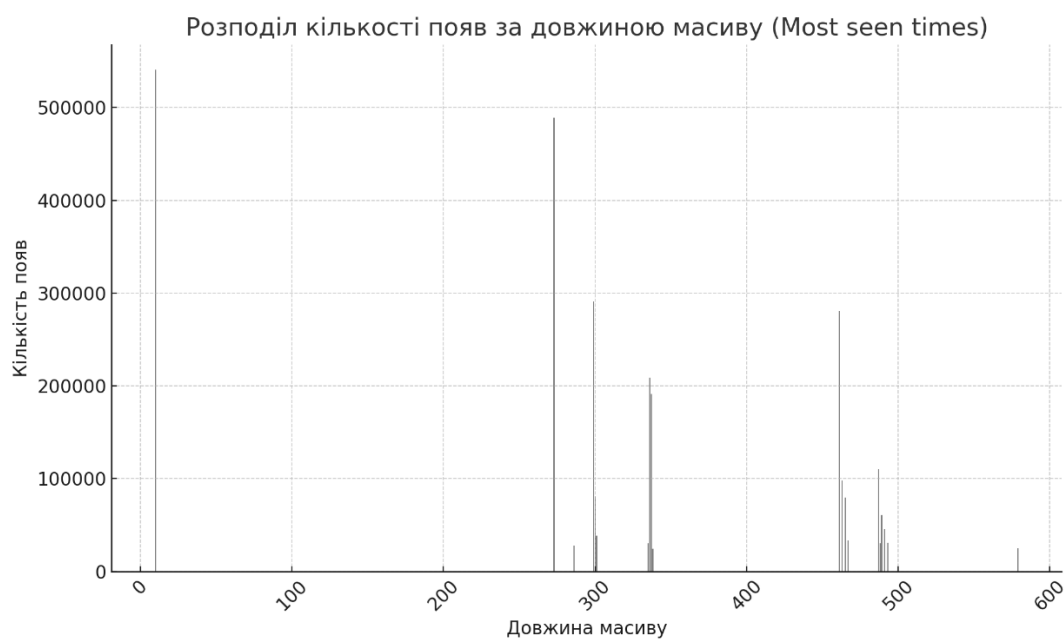


Рис. 3.18 Розподіл кількості масивів за довжиною

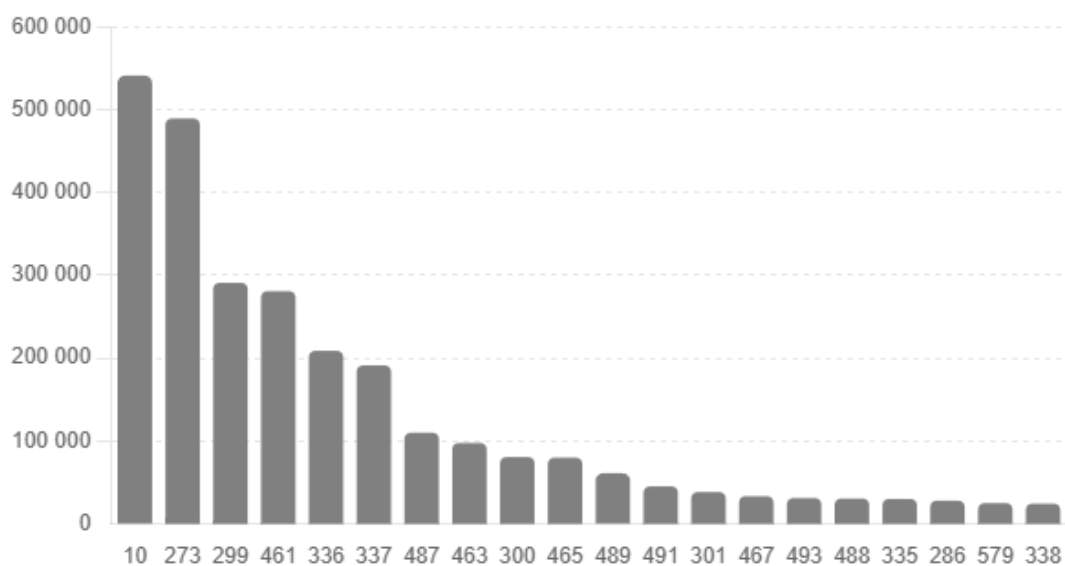


Рис. 3.19 Розподіл кількості масивів за довжиною

Розподіл за кількістю екземплярів не визначає внесок відповідної групи масивів у сумарний обсяг пам'яті. Тому додатково було побудовано розподіл за обсягом пам'яті, зайнятим масивами однакової довжини. Відповідні результати наведено на рис. 3.20 та рис. 3.21.

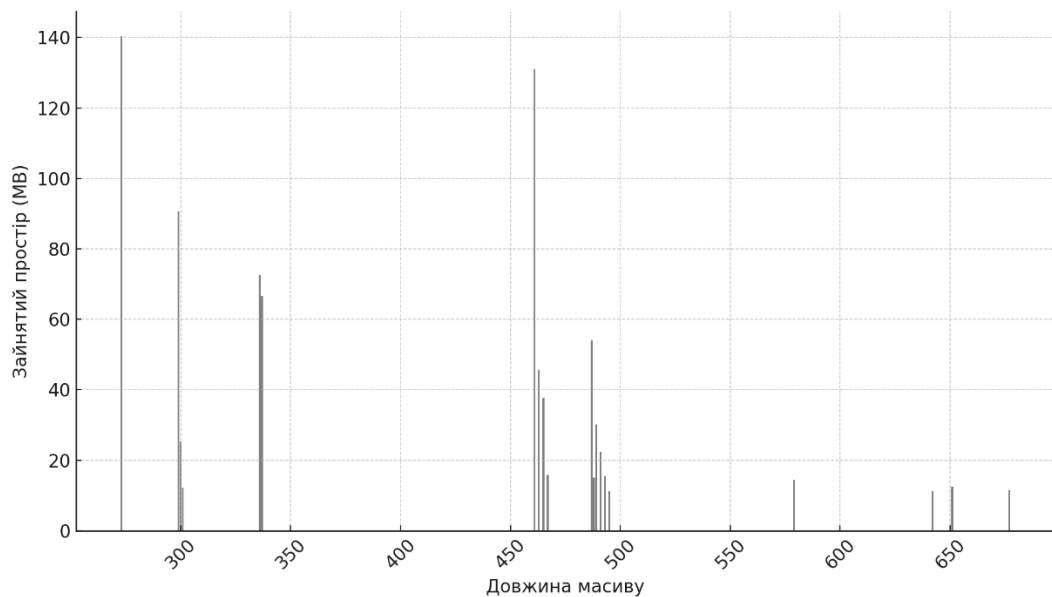


Рис. 3.20 Розподіл обсягу пам'яті за довжиною масиву

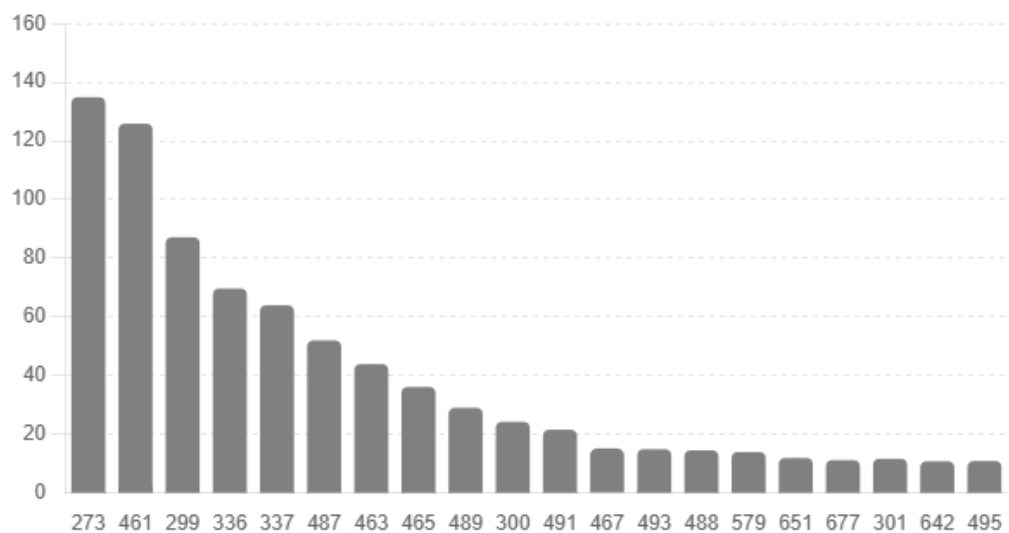


Рис. 3.21 Розподіл обсягу пам'яті за довжиною масиву

З наведених розподілів випливає, що масиви довжиною 10 елементів мають найбільшу кількість екземплярів, але не формують найбільший внесок у сумарний обсяг пам'яті. Для масивів довжиною 273 та 461 елемент сумарний обсяг пам'яті є більшим, оскільки розмір кожного окремого об'єкта у цих групах істотно більший. Отже, відбір об'єктів-кандидатів на стискання доцільно виконувати з урахуванням не лише кількості екземплярів, а й сумарного обсягу пам'яті відповідної групи.

На рис. 3.22 наведено розподіл найбільших масивів. Частина довжин таких масивів є кратною степеням двійки. Такі об'єкти можуть відповідати буферам введення-виведення або структурам повторного використання масивів. Для них застосування стискування не визначається лише значенням коефіцієнта стискування, оскільки ці об'єкти можуть брати участь в активних операціях читання, запису або мережевого обміну.

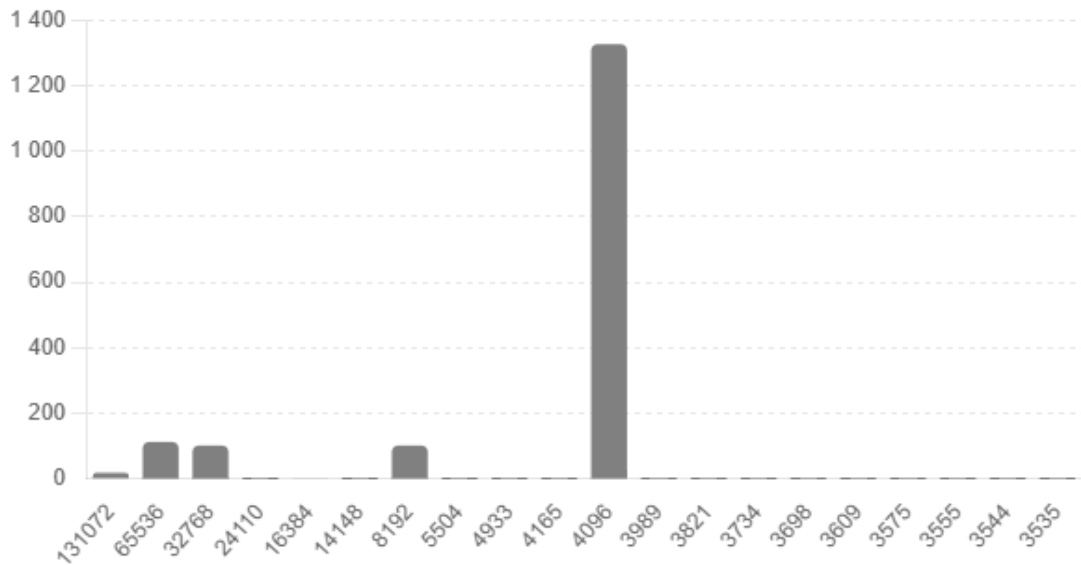


Рис. 3.22 Розподіл найбільших масивів

Відповідно до (2.54), множина об'єктів-кандидатів на стискування формується не для всіх масивів байтів, а лише для тих, що задовольняють умови мінімальної довжини, зменшення розміру після стискування, порогового значення коефіцієнта стискування та допустимості заміни з погляду способу доступу до даних. У цьому експерименті до таких кандидатів було віднесено масиви, що подають збережений бінарний стан користувацьких даних і не належать до активних буферів введення-виведення.

Для кожного об'єкта-кандидата застосовувалося безвтратне відображення стискування. Умова збереження інформаційного вмісту перевірялася відповідно до (2.52). Після стискування для кожного масиву

визначалися індивідуальні характеристики, введені у (2.53), а для всієї множини кандидатів - сумарні характеристики за (2.55).

Частина масивів байтів відповідає серіалізованому стану користувачьких даних. Такі масиви утворюються після перетворення об'єктного подання даних у бінарну форму. Їх довжина залежить від обсягу інформації, пов'язаної з відповідним користувачем, тому у цій групі довжини масивів не мають фіксованої кратності степеням двійки.

Після застосування стискання до відібраної множини кандидатів отримано зменшення обсягу пам'яті на 0,95 ГБ. Відносно сумарного обсягу пам'яті, зайнятого об'єктами типу System.Byte[], це становить приблизно 45 %. позначеннях (2.55) для розглянутого алгоритму стискання це відповідає значенню

$$\rho_{\alpha}^{cmp}(u) \approx 0.45.$$

Це значення характеризує частку пам'яті, яка може бути зменшена для відповідної множини об'єктів-кандидатів без втрати інформаційного вмісту. Воно не поширюється на всі масиви байтів у знімку пам'яті, оскільки частина таких об'єктів є службовими буферами або структурами, для яких стискання не є допустимою заміною за умовами (2.54).

Додатковий аналіз знімків пам'яті інших програмних систем показав наявність двох груп масивів байтів. До першої групи належать масиви з довжинами, кратними степеням двійки. Вони переважно відповідають буферам або внутрішнім службовим структурам. До другої групи належать масиви, довжина яких визначається фактичним обсягом збереженої інформації. Саме друга група є основною для застосування умов (2.54).

Джерело появи масивів байтів у пам'яті також враховується під час відбору кандидатів. Якщо масив утворений унаслідок отримання даних з іншої системи, стискання може вже виконуватися на рівні протоколу передавання. Якщо масив є результатом збирання або збереження даних у межах самої системи, спосіб подання може бути змінений без порушення інформаційного вмісту. Якщо масив отриманий із файлової системи, стискання може бути

властивістю формату файлу, тому повторне стискання може не зменшувати розмір подання.

Отримані результати показують, що застосування умов (2.54) дозволяє відокремити масиви, для яких стискання є допустимим з погляду збереження інформаційного вмісту та способу доступу до даних. Значення $\rho_{\alpha}^{cmp}(u)$, визначене у (2.55), використовується як кількісна характеристика відносного зменшення обсягу пам'яті для вибраного алгоритму стискання.

Для частини знімків, у яких об'єкти типу System.Byte[] входили до груп із помітним обсягом пам'яті, умови стискання не виконувались. Частка таких випадків була порядку сімдесяти відсотків. Типовими причинами були мала довжина масивів, відсутність зменшення розміру після стискання, належність до активних буферів введення-виведення або попереднє стискання на рівні протоколу чи формату даних. Для таких масивів значення $\Delta s_{\alpha}(o)$ або предикат допустимості з (2.54) не задавали підстави для заміни подання.

3.7 Валідація рекомендацій та оцінювання фактичного впливу від впровадження

Однією з ключових задач запропонованої продукційної моделі є не лише виявлення сценаріїв надмірного використання пам'яті, але й кількісне оцінювання практичного ефекту від впровадження сформованих рекомендацій. Для цього було проведено валідацію рекомендацій на основі аналізу промислових знімків пам'яті та повторного дослідження систем після внесення відповідних змін у програмний код.

Для кожної рекомендації фіксувався спосіб встановлення її істинності. У роботі використовувались такі варіанти підтвердження або відхилення:

- підтверджено розробником;
- підтверджено зміною коду;
- підтверджено повторним знімком;

- відхилено як семантично недопустиме.

Рекомендація вважалась істинно позитивною, якщо вона була підтверджена розробником, реалізована у програмному коді та підтверджена повторним знімком пам'яті. Якщо формальний критерій спрацьовував, але запропоноване перетворення суперечило семантиці програмного коду або вимогам предметної області, така рекомендація класифікувалась як хибнопозитивна.

Для кожної рекомендації обчислювалася відносна похибка прогнозування:

$$\varepsilon_i = \frac{|\Delta_{\text{pred},i} - \Delta_{\text{real},i}|}{\Delta_{\text{real},i}} * 100\%$$

Табл. 3.7 – Валідація прогнозованого та фактичного ефекту рекомендацій

Сценарій	Початковий вимір	Рекомендація	Δ_{pred}	Δ_{real}	$\varepsilon, \%$
Дублювання строк	2.64 ГБ	Інтернування	0,54 ГБ	0,50 ГБ	8,0
Надлишкове подання ключового типу	533 МБ	Звуження типів полів	488 МБ	486,4 МБ	0.33
Надмірні службові витрати для словника з одним елементом	21.8 ГБ	Компактне подання для числового ключа	13,67 ГБ	12,94 ГБ	5,64
Масиви байтів	2.3 ГБ	Безвтратне стискання об'єктів-кандидатів	0,95 ГБ	0,88 ГБ	7,95

Сукупний ефект не враховується як окрема істинно позитивна рекомендація, оскільки він є агрегованою характеристикою застосування кількох незалежних правил у часі, а не результатом одного продукційного правила.

Нехай $M_i^{(0)}$ – знімок пам'яті, за яким було сформовано рекомендацію r_i , а $M_i^{(1)}$ – знімок пам'яті після внесення відповідних змін у програмний код.

Оскільки досліджувані промислові системи оперують великими обсягами даних, які змінюються в часі, загалом виконується

$$M_i^{(1)} \equiv M_i^{(0)}.$$

Тому фактична перевірка ефекту не є повторним вимірюванням на тотожно однаковій множині об'єктів. Вона розглядається як перевірка на новому стані системи, співмірному з початковим за порядком обсягу даних:

$$S(M_i^{(1)}) = \Theta(S(M_i^{(0)})), |O(M_i^{(1)})| = \Theta(|O(M_i^{(0)})|),$$

де $S(M)$ – сумарний обсяг пам'яті, а $O(M)$ – множина об'єктів, відображених у знімку пам'яті.

Для забезпечення коректності порівняння також передбачається близькість емпіричних розподілів структурних характеристик:

$$d(\rho(M_i^{(0)}), \rho(M_i^{(1)})) \leq \delta,$$

де $\rho(M)$ – емпіричний розподіл типів, розмірів або довжин об'єктів у знімку пам'яті, $d(\cdot, \cdot)$ – метрика відстані між розподілами, δ – допустиме відхилення.

Отже, значення Δ_{real} , наведені у таблиці 3.7, слід інтерпретувати як оцінку фактичного ефекту на статистично співмірному, але не еквівалентному стані системи:

$$\Delta_{\text{real}} = \widehat{\Delta}(M_i^{(1)}), M_i^{(1)} \sim_{\Theta, \delta} M_i^{(0)}.$$

Таке подання відповідає практичному сценарію використання високонавантажених систем, у яких між моментом отримання початкового знімка пам'яті та повторною валідацією система продовжує функціонувати, змінюючи множину об'єктів, їх кількість та розподіл значень. Проте при збереженні одного порядку обсягів і структурних характеристик можливо виконувати коректне порівняння прогнозованого та фактичного ефекту.

3.8 Висновки до розділу

У третьому розділі наведено результати експериментального дослідження типових ситуацій надмірного використання оперативної пам'яті на основі аналізу знімків пам'яті .NET-застосунків. Результати розділу пов'язані з математичними моделями та алгоритмами, введеними у розділі 2, і використовуються для кількісного оцінювання окремих сценаріїв неефективного подання інформації в пам'яті.

1. Проведено первинний аналіз розподілу пам'яті за типами об'єктів. За результатами групування об'єктів за типами та обчислення сумарного обсягу пам'яті визначено групи об'єктів, що потребують подальшого аналізу: об'єкти типу `System.String`, масиви примітивних і користувацьких типів, хеш-колекції та користувацькі типи, що використовуються як ключі у `Dictionary<K,V>`. Показано, що такий етап не є критерієм витоку пам'яті, а використовується для визначення типових ситуацій надмірного використання пам'яті у фіксований момент часу.
2. Для об'єктів типу `System.String` виконано групування за інформаційним вмістом відповідно до моделі представлення незмінюваної інформації. У досліджуваному знімку пам'яті виявлено приблизно 220 тис. різних рядкових значень, що мають повторні фізичні представлення, та близько 14,5 млн повторних екземплярів. Частка повторних рядкових об'єктів становить близько 43 % від загальної кількості об'єктів типу `System.String`. Обсяг пам'яті, зайнятий повторними представленнями рядкових значень, оцінено приблизно у 930 МБ, а значення коефіцієнта надмірності для цієї підмножини становить приблизно 1,75.
3. Виконано експериментальну перевірку алгоритму прискореного виявлення дублікатів об'єктів типу `System.String`. Застосування

попереднього групування за обчислювальною ознакою дозволило зменшити кількість точних порівнянь рядкових значень. Без попереднього групування час обробки становив 7464 мс; для $h = 512$ отримано час 1894,38 мс, що відповідає зменшенню часу приблизно у 3,94 рази. Встановлено, що подальше збільшення кількості підмножин не забезпечує пропорційного зменшення часу через додаткові витрати на формування та оброблення структури групування.

4. Для типу `Model.Shipping.GroupedShippingMethodZoneKey` проведено аналіз фактичних діапазонів значень полів. Встановлено, що поля `shippingMethodId` та `shippingMethodZoneId`, оголошені як `long`, використовують лише малу частину допустимого діапазону цього типу. За умовами допустимості заміни типів ці поля можуть бути подані типами `int` та `ushort` відповідно. Також показано, що подання ключа як класу у 64-бітному середовищі CLR супроводжується службовими витратами на окремий об'єкт та покажчик у структурі словника. Перехід до подання у вигляді `readonly struct` зменшує обсяг, необхідний для зберігання одного ключа, з 48 до 8 байтів з урахуванням вирівнювання та способу розміщення у словнику.
5. Для компактного подання ключового типу виконано обчислювальне порівняння двох реалізацій. Для початкового подання обсяг виділеної пам'яті під час завантаження даних становив 533 МБ, а час виконання - близько 1 с. Для подання у вигляді `readonly struct` відповідні значення становили 46,6 МБ та 54 мс. Отримані результати узгоджуються з оцінкою розміру ключового типу, наведеною у підпункті 3.4, і характеризують вплив способу подання ключа на обсяг виділеної пам'яті та час виконання операцій завантаження й пошуку.
6. Для хеш-колекцій типу `Dictionary<K,V>` виконано оцінювання вартості зберігання інформації для малих кількостей елементів. Для словника з однією парою ключ-значення фактичний обсяг пам'яті становить 236 байтів, тоді як корисний інформаційний обсяг для пари `int-object`

- становить 12 байтів. Для сукупності з 99 172 580 словників з одним елементом стандартне подання потребує 21,80 ГБ пам'яті при корисному інформаційному обсязі 1,11 ГБ. Компактне подання для числового ключа дає оцінку зменшення обсягу пам'яті на 13,67 ГБ, а подання для одного незмінного елемента - на 18,84 ГБ.
7. Для спеціалізованих подань колекцій з малою кількістю елементів проведено обчислювальне порівняння часу створення, обсягу виділеної пам'яті та часу пошуку елементів. Отримані результати показують, що для малих кількостей елементів службові витрати стандартної реалізації Dictionary<K,V> можуть перевищувати обсяг корисної інформації на порядок. Зі збільшенням кількості елементів перевага компактних подань зменшується, що відповідає властивостям хеш-структур при роботі з більшими множинами елементів.
 8. Для об'єктів типу System.Byte[] виконано оцінювання застосування безвтратного стискання. У знімку пам'яті виявлено близько 4,9 млн масивів байтів із сукупним інформаційним вмістом близько 2,3 млрд байтів. Розподіл масивів за довжиною та обсягом пам'яті показав, що кількість екземплярів окремої довжини не завжди відповідає її внеску в сумарний обсяг пам'яті. Після відбору об'єктів-кандидатів відповідно до умов моделі стискання отримано зменшення обсягу пам'яті на 0,95 ГБ, що відповідає значенню $\rho_{\alpha}^{cmp}(u) \approx 0,45$ для розглянутої множини кандидатів.
 9. Отримані у розділі результати характеризують окремі сценарії надмірного використання оперативної пам'яті, які не обов'язково відповідають критерію витоку пам'яті. До таких сценаріїв належать дублювання незмінюваних рядкових значень, надлишковий діапазон типів полів, службові витрати хеш-колекцій для малих кількостей елементів та зберігання бінарної інформації у нестисненому вигляді. Кількісні характеристики, отримані у цьому розділі, можуть бути

використані як факти для подальшого формування правил експертної системи.

10. Під час аналізу вибірки з понад 60 знімків пам'яті було встановлено, що жодне з розглянутих правил не є універсальним для всіх досліджуваних станів пам'яті. Для кожного правила існують випадки неспрацювання, пов'язані з відсутністю відповідного сценарію надмірного використання пам'яті або з невиконанням умов безпечної заміни подання даних. Тому результати розділу 3 слід розглядати як набір кількісних критеріїв для формування фактів експертної системи, а не як єдиний узагальнений критерій неефективного використання пам'яті. Такий підхід дозволяє надалі будувати правила, які спрацювують лише для тих структур пам'яті, де відповідний сценарій має достатнє кількісне підтвердження.
11. Виконано зіставлення розрахункових оцінок надмірності та прогнозованого зменшення обсягу пам'яті з фактичними результатами після застосування рекомендованих перетворень структури даних. Отримані результати підтвердили працездатність запропонованих алгоритмічних процедур для досліджених сценаріїв надмірного використання оперативної пам'яті.

Основні результати розділу опубліковані в [6, 20, 55, 78, 83, 86, 98, 99, 102].

Розділ 4. ЕКСПЕРТНА СИСТЕМА ДЛЯ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ ТА ВИЯВЛЕННЯ НАДМІРНОГО ВИКОРИСТАННЯ ПАМ'ЯТІ

4.1 Продукційне представлення результатів ідентифікації надмірності пам'яті

Призначення продукційного механізму полягає у застосуванні правил до фактів, сформованих із типізованої структури знімка пам'яті. Правила не вводять нову модель пам'яті; вони задають умови використання предикатів, критеріїв і оцінювальних функцій, побудованих у розділі 2 та перевірених у розділі 3. Результатом є сценарій надмірності, група об'єктів, рекомендація та кількісна оцінка можливого зменшення обсягу пам'яті.

У межах розробленої моделі знімок пам'яті містить дані про об'єкти, їх типи, адреси, розміри, значення полів, колекції, масиви та зв'язки між елементами керованої пам'яті. Такі характеристики можуть бути використані не лише для опису поточного стану процесу, а й для встановлення структурних причин надмірного використання оперативної пам'яті. До таких причин належать дублювання незмінної інформації, надлишкове резервування пам'яті у структурах даних, низький рівень заповнення колекцій, необґрунтований вибір типів даних, втрати пам'яті через вирівнювання об'єктів, а також зберігання інформації у формі, що не відповідає її фактичному обсягу.

У такій постановці задача продукційного механізму полягає у реалізації відображення від множини фактів, побудованих за формальною моделлю знімка пам'яті, до множини діагностичних висновків, сценаріїв надмірності та кількісних оцінок можливого зменшення обсягу пам'яті. Продукційні правила лише фіксують умови застосування вже введених у розділі 2 предикатів, критеріїв і оцінювальних функцій.

Для фіксованого знімка пам'яті M_u продукційне подання задається кортежем

$$\mathcal{E}_u = (F_u, R, I, Q_u),$$

де F_u - база фактів, отриманих із M_u ; R - множина продукційних правил; I - оператор логічного виведення; Q_u - множина результатів застосування правил до фактів цього знімка.

База фактів F_u містить формалізовані характеристики, обчислені за моделлю знімка пам'яті: множини об'єктів і типів, адресні та розмірні характеристики, значення полів, зв'язки між об'єктами, результати групування, характеристики колекцій, масивів і службових структур. Ці факти не містять висновку про доцільність впровадження зміни; вони задають вхідні ознаки для подальшої інтерпретації.

Множина правил $R = \{r_i\}_{i=1}^m$ задає умови переходу від фактів до висновків. Кожне правило подається у вигляді

$$r_i: \varphi_i(F_u) \Rightarrow q_i,$$

де φ_i - предикат над фактами знімка пам'яті, а $q_i \in Q_u$ - результат спрацювання правила. Результат має вигляд

$$q_i = (s_i, a_i, \hat{\delta}_i),$$

де s_i - виявлений сценарій надмірного використання пам'яті, a_i - відповідна рекомендація щодо допустимого перетворення подання даних, $\hat{\delta}_i$ - кількісна оцінка можливого зменшення обсягу пам'яті.

Оператор виведення I виконує зіставлення фактів із предикатами правил:

$$Q_u = I(F_u, R).$$

Він не вводить нових критеріїв надмірності, а застосовує предикати, порогові умови та оцінювальні функції, визначені на основі моделі знімка пам'яті. У такому поданні продукційний механізм є алгоритмічним шаром інтерпретації формалізованих фактів: від структури пам'яті він переходить до множини сценаріїв, рекомендацій та кількісних оцінок, які підлягають подальшій перевірці.

Результати аналізу, отримані під час дослідження знімків пам'яті, мають різну природу. Частина з них є кількісними характеристиками, зокрема обсяг пам'яті, зайнятий певним типом, кількість екземплярів, середній розмір об'єкта, розмір масиву або резервна ємність колекції. Інша частина пов'язана зі структурними властивостями об'єктів: незмінністю значень, повторюваністю рядкових даних, співвідношенням між фактичним і зарезервованим обсягом, наявністю однакових значень у різних екземплярах. Для отримання рекомендацій ці характеристики мають бути зіставлені з правилами, що описують типові випадки надмірного використання пам'яті.

Призначення продукційного механізму полягає в автоматизованому перетворенні фактів, отриманих зі знімка пам'яті, у висновки щодо наявності окремих сценаріїв нераціонального використання оперативної пам'яті. Така система повинна не лише встановлювати факт дублювання або надлишкового резервування, а й визначати спосіб усунення відповідної причини з урахуванням очікуваної економії пам'яті [103]. Рекомендації можуть стосуватися інтернування або кешування рядкових значень, застосування пулів об'єктів, зменшення резервної ємності колекцій, заміни типів даних, зміни структури зберігання об'єктів або стискування масивів байтів.

Побудова продукційного подання передбачає узгодження трьох складових. Перша складова пов'язана з формальним поданням знімка пам'яті та виділенням фактів, придатних для логічного аналізу. Друга складова визначає базу знань, у якій фіксуються правила інтерпретації властивостей об'єктів і структур даних. Третя складова задає механізм виведення, за допомогою якого факти зіставляються з умовами правил, а результатом такого зіставлення стає рекомендація щодо зміни програмного коду або способу подання даних.

Задача продукційного представлення результатів ідентифікації формулюється як побудова відображення, що за даними знімка пам'яті формує базу фактів, зіставляє її з базою правил і визначає наявність типових сценаріїв надмірного використання оперативної пам'яті. Для кожного встановленого

сценарію система має сформувавши рекомендацію та оцінити очікуваний обсяг пам'яті, який може бути вивільнений або зменшений унаслідок застосування запропонованої зміни [102; 103].

У межах цієї постановки продукційний механізм виконує такі операції:

- формує базу фактів зі знімка пам'яті;
- зіставляє агреговані факти з умовами продукційних правил;
- визначає сценарії надмірного використання оперативної пам'яті;
- використовує кількісні оцінки, отримані з моделей розділу 2;
- формує діагностичну рекомендацію, яка підлягає перевірці з

урахуванням семантики програмного коду.

Для реалізації зазначеної постановки необхідно розв'язати такі задачі:

1. побудувати формальне подання знімка пам'яті та множини об'єктів, які в ньому містяться;
2. сформувавши базу знань, що охоплює властивості типів об'єктів, статистичні характеристики алокацій і типові сценарії надмірного використання пам'яті;
3. побудувати систему продукційних правил типу «умова → дія» для виявлення проблем і генерації рекомендацій;
4. визначити склад бази фактів і механізм логічного виведення;
5. провести апробацію системи на знімках пам'яті програмних додатків, отриманих під час експлуатації.

Продукційне представлення результатів аналізу знімків пам'яті є способом застосування формалізованих правил до фактів, отриманих із моделі знімка пам'яті. Воно не підміняє математичну постановку задачі і не замінює критерії надмірності. Його функція полягає у переході від обчислених характеристик об'єктів і структур даних до діагностичних висновків, рекомендацій та кількісних оцінок, які надалі перевіряються на експериментальних даних.

4.2 Продукційна архітектура застосування правил

Задача, сформульована у підрозділі 4.1, реалізується через продукційну архітектуру, у якій дані знімка пам'яті відокремлюються від правил їх інтерпретації. Такий поділ дає змогу застосовувати одну базу знань до різних знімків пам'яті та формувати висновки на основі фактичного стану керованої пам'яті. Архітектура системи включає базу фактів, базу знань і механізм логічного виведення, взаємодія яких забезпечує перехід від інструментально отриманих даних до рекомендацій щодо зміни програмного коду [103].

База фактів формується після опрацювання знімка пам'яті засобами доступу до структури керованої пам'яті. До її складу належать множини об'єктів і типів, адресні та розмірні характеристики об'єктів, значення полів, зв'язки між об'єктами, результати групування за типами, значеннями та структурними ознаками, а також характеристики колекцій, масивів і службових структур. База фактів не містить оцінок щодо доцільності змін. Вона задає формалізований опис конкретного знімка пам'яті у вигляді ознак, які можуть бути використані механізмом логічного виведення.

База знань містить продукційні правила, які зіставляють факти з типовими сценаріями надмірного використання пам'яті. Ліва частина правила задає умови спрацювання на основі кількісних і структурних характеристик об'єктів. Права частина пов'язує встановлений сценарій із рекомендацією щодо впровадження зміни або з проміжним висновком, який може бути використаний іншими правилами. До таких сценаріїв належать дублювання незмінних об'єктів, надлишкова ємність колекцій, використання типів даних із надмірним діапазоном значень, втрати пам'яті через вирівнювання об'єктів, накладні витрати службових структур словників, доцільність стискання масивів байтів і надлишкова гранулярність синхронізації.

Механізм логічного виведення виконує послідовне зіставлення фактів з умовами продукційних правил. Якщо умови правила виконуються, система

формує новий факт про наявність відповідного сценарію або створює рекомендацію щодо усунення встановленої причини. Для обмеження несуттєвих спрацювань у правилах використовуються порогові параметри, зокрема мінімальна кількість повторів, граничне значення коефіцієнта заповнення колекції, допустимий рівень надлишку діапазону значень або мінімальний очікуваний обсяг економії пам'яті. Такі параметри дають змогу відокремити випадкові локальні відхилення від ситуацій, що мають практичне значення для аналізу.

За функціональним призначенням правила системи виконують дві пов'язані операції. Перша операція полягає у встановленні наявності проблемного сценарію у знімку пам'яті. Друга операція пов'язує цей сценарій із дією, придатною для подальшого розгляду розробником. Наприклад, групування однакових екземплярів незмінних рядків може привести до висновку про дублювання рядкових значень і рекомендації щодо інтернування або кешування. Низький коефіцієнт заповнення внутрішнього буфера колекції може бути підставою для рекомендації зменшити її ємність або змінити структуру зберігання елементів.

Вихідний результат роботи архітектури подається як множина рекомендацій, пов'язаних із відповідними фактами та правилами. Для кожної рекомендації зберігається тип виявленого сценарію, множина або група об'єктів, на яких спрацювало правило, умова спрацювання та кількісна оцінка очікуваної економії пам'яті. Такий формат результату дає змогу відокремити власне факт нераціонального використання пам'яті від способу його усунення і зберегти зв'язок між рекомендацією та даними знімка пам'яті, з яких її отримано.

Послідовність роботи системи має такий вигляд. Спочатку знімок пам'яті перетворюється на множину формалізованих фактів. Далі факти надходять до механізму логічного виведення, де вони зіставляються з правилами бази знань. Після спрацювання правил формується множина виявлених сценаріїв та пов'язаних з ними рекомендацій. За наявності кількох

рекомендацій для одного знімка пам'яті їх можна впорядкувати за очікуваним обсягом економії пам'яті або за типом сценарію, що спрощує подальший аналіз результатів.

Архітектура системи узгоджує формальні моделі, побудовані у розділі 2, з експериментально встановленими сценаріями, дослідженими у розділі 3. Модель знімка пам'яті задає структуру фактів, а результати експериментального аналізу визначають умови спрацювання правил і склад рекомендацій [102; 103]. Завдяки цьому знімок пам'яті використовується не лише як джерело статистичних характеристик, а як вхідний об'єкт для логічної інтерпретації. Запропонована архітектура забезпечує послідовний перехід від фактичного стану пам'яті до формалізованих рекомендацій щодо зменшення надмірного використання оперативної пам'яті.

4.3 База знань і продукційні правила

Архітектура, визначена у підрозділі 4.2, використовує базу знань як множину правил, що інтерпретують факти, сформовані зі знімка пам'яті. Побудована у розділі 2 модель задає множини об'єктів, типів, полів, значень і функцій групування, а результати розділу 3 визначають типові сценарії надмірного використання оперативної пам'яті та кількісні характеристики їх прояву. У межах цього підрозділу ці результати подаються у формі продукційних правил, придатних для застосування механізмом логічного виведення [102; 103].

База знань містить правила двох функціональних типів. Правила першого типу встановлюють наявність певного сценарію неефективного використання пам'яті за фактами, отриманими зі знімка. Правила другого типу пов'язують встановлений сценарій із рекомендацією щодо зміни подання даних, параметрів структур або способу використання об'єктів. Умови правил

задаються через множини об'єктів і типів, значення полів, характеристики колекцій, масивів і службових структур. Результатом спрацювання правила є новий факт про виявлений сценарій або рекомендація з відповідною кількісною оцінкою.

Дії продукційних правил мають статус кандидатних перетворень. Спрацювання правила фіксує наявність сценарію надмірного подання даних і кількісну оцінку можливого зменшення обсягу пам'яті, але не означає безумовної зміни подання об'єктів або структур даних. Допустимість дії перевіряється окремо за семантикою значень, способом доступу до них, стабільністю фактичних діапазонів і характером життєвого циклу відповідних об'єктів. Якщо ці умови не виконані або не можуть бути перевірені за наявними фактами, результат правила залишається діагностичним висновком і не переходить до множини допустимих дій.

У правилах використовуються порогові параметри, що відокремлюють одиничні локальні випадки від ситуацій, які мають практичне значення для поведінки системи. До таких параметрів належать мінімальна кратність повторення об'єктів, допустима частка використання діапазону значень, гранична частка втрат через вирівнювання, мінімальний коефіцієнт заповнення колекції, поріг вартості зберігання у хеш-колекціях, поріг доцільності стискання та максимальна припустима гранулярність синхронізації.

4.3.1 Правило виявлення дублювання незмінюваних об'єктів

Модель представлення незмінюваної інформації, побудована у підрозділі 2.5.2, задає множину незмінюваних типів, відповідні множини об'єктів, класи еквівалентності за значенням і кількісну оцінку дублювання згідно з (2.29)-(2.33). У межах продукційної архітектури ці об'єкти та

результати групування належать до бази фактів, а правило бази знань виконує їх інтерпретацію як сценарію надмірного використання пам'яті.

Для типу $t \in R$ використовується вже визначене розбиття об'єктів на класи еквівалентності за інформаційним вмістом. Наявність класів, що містять більше одного фізично різного об'єкта, означає повторне представлення одного й того самого значення у пам'яті. Такий випадок не інтерпретується як витік пам'яті, оскільки аналіз виконується для фіксованого знімка, але він задає підставу для оцінювання надмірності подання незмінюваної інформації.

Умова спрацювання правила задається через показник середньої кратності повторення значень μ_t , визначений для незмінюваного типу t . Для відокремлення одиничних повторів від систематичного дублювання вводиться пороговий параметр N_{dup} . Тоді продукційне правило має вигляд

$$\mathbf{IF} t \in T_{imm} \wedge \mu_t > N_{dup} \mathbf{THEN} \mathit{Recommend}(\mathit{InternOrCache}(t)), \quad (4.1)$$

де N_{dup} задає мінімальний рівень повторення, за якого дублювання вважається достатнім для формування рекомендації. Дія $\mathit{Recommend}(\mathit{InternOrCache}(t))$ означає вибір одного зі способів повторного використання значень відповідного типу: інтернування, кешування або застосування пулу об'єктів. Конкретний спосіб реалізації залежить від семантики типу, частоти створення об'єктів і режиму доступу до значень.

Для об'єктів типу `System.String` це правило відповідає сценарію, дослідженому у розділі 3: фізично різні рядкові об'єкти можуть мати однаковий інформаційний вміст і належати до одного класу еквівалентності за значенням. У такому випадку рекомендація спрямована не на зміну значення об'єктів, а на зменшення кількості їх повторних фізичних представлень у пам'яті [54].

Межа застосовності цього правила пов'язана з життєвим циклом значень. Інтернування, кешування або застосування пулу спільних об'єктів не розглядаються як допустима дія, якщо повторювані значення мають короткий час існування, залежать від окремого запиту, сесії або користувачького

контексту, утворюють необмежену множину нових значень чи використовуються в кодї з урахуванням тотожності посилань. Для рядкових значень інтернування не застосовується лише за фактом наявності однакового інформаційного вмісту у знімку пам'яті. Потрібна перевірка, що повторювані значення мають стабільне повторне використання протягом життєвого циклу процесу і не призведуть до додаткового утримання пам'яті. За відсутності такої перевірки правило фіксує дублювання незмінюваної інформації та кількісну оцінку надмірності, але не формує допустиму дію інтернування.

4.3.2 Правило виявлення надлишкового діапазону оголошених типів полів

Підхід до оцінювання доцільності звуження типів полів за фактичним діапазоном значень побудовано у підрозділі 2.5.5. У межах бази знань ця модель використовується не для повторного обчислення характеристик поля, а як джерело фактів: множина значень поля, фактичний діапазон, коефіцієнт використання діапазону та допустима компактніша заміна типу визначаються за (2.46)-(2.49).

Для числового поля p типу t_{sh} використовується коефіцієнт $\rho_{t_{sh},p}$, введений у (2.48). Якщо цей коефіцієнт є малим, оголошений тип поля має ширший діапазон, ніж фактично використовується у знімку пам'яті. Для відокремлення таких випадків від несуттєвих відхилень вводиться порогове значення τ_{range} . Рекомендація формується лише тоді, коли існує тип $\delta_{t_{sh},p}^*(u)$, що покриває фактичний діапазон із запасом за умовою (2.49) і має менший розмір, ніж початковий тип поля δ_p .

Продукційне правило для цього сценарію має вигляд

$$\mathbf{IF} \rho_{t_{sh},p} \leq \tau_{range} \wedge R_{t_{sh},p}^{obs,y}(u) \subseteq D_{\delta_{t_{sh},p}^*(u)} \wedge w(\delta_{t_{sh},p}^*(u)) < w(\delta_p)$$

$$\mathbf{THEN} \textit{Recommend} \left(\textit{ShrinkFieldType} \left(t_{sh}, p, \delta_{t_{sh},p}^*(u) \right) \right), \quad (4.2)$$

де t_{range} задає граничне значення коефіцієнта використання діапазону, D означає область допустимих значень типу, а дія *ShrinkFieldType* означає рекомендацію замінити оголошений тип поля на компактніший тип, допустимий для спостережуваних значень. Така рекомендація не є автоматичною зміною структури даних: її застосування потребує перевірки предметних обмежень поля, зокрема можливих значень, які не були представлені у досліджуваному знімку пам'яті. Практичний приклад застосування цього правила до ключового типу *GroupedShippingMethodZoneKey* наведено у підрозділі 3.4.

Для правила звуження оголошених типів полів фактичний діапазон, знайдений у знімку пам'яті, не є достатньою підставою для заміни типу. Звуження не належить до допустимих дій, якщо спостережуваний діапазон не є стабільним обмеженням предметної області або не має підтвердження за кількома режимами роботи. Особливо це стосується полів, що зберігають ідентифікатори, лічильники, зовнішні коди, значення з баз даних, версійні ознаки або параметри, діапазон яких може змінюватися після розширення функціональності. Якщо майбутній діапазон значень не обмежений відомим інваріантом, правило зберігає статус оцінки надлишкового діапазону оголошеного типу, але не задає допустимої заміни типу поля.

4.3.3 Правило виявлення втрат пам'яті через вирівнювання об'єктів

Відомості про службову частину об'єкта, розміщення полів і вирівнювання у 64-розрядному середовищі CLR уже враховано у формальному поданні об'єкта знімка пам'яті. У межах бази знань ці характеристики використовуються як факти, що дають змогу оцінити частку

додаткового обсягу, який виникає не через інформаційний вміст полів, а через правила фізичного розміщення об'єкта в пам'яті.

Для типу t розглядаються два розмірні показники: $S_{raw}(t)$ - розмір подання до вирівнювання, обчислений за службовою частиною та полями типу, і $S_{actual}(t)$ - фактичний розмір екземпляра після застосування правил вирівнювання. Відносна надбавка, зумовлена вирівнюванням, задається коефіцієнтом $\lambda_{pad}(t)$. Для відокремлення несуттєвих випадків від систематичних втрат вводиться порогове значення τ_{pad} . Продукційне правило має вигляд

$$\left\{ \begin{array}{l} \lambda_{pad}(t) = \frac{S_{actual}(t) - S_{raw}(t)}{S_{raw}(t)} \\ \text{IF } \lambda_{pad}(t) \geq \tau_{pad} \text{ THEN } \text{Recommend}(\text{RepackFields}(t)) \end{array} \right., \quad (4.3)$$

де τ_{pad} задає граничне значення відносної надбавки, за якого втрати пам'яті через вирівнювання вважаються достатніми для формування рекомендації. Дія $\text{RepackFields}(t)$ означає рекомендацію переглянути порядок розміщення полів типу t , їхні оголошені типи або спосіб подання об'єкта, якщо така зміна не порушує семантику даних.

Це правило узгоджується з правилом звуження оголошених типів полів, наведеним у 4.3.2, але не зводиться до нього. Звуження типу поля може зменшити інформаційний розмір окремого значення, тоді як правило вирівнювання оцінює вплив службових і структурних особливостей подання об'єкта на його фактичний розмір. У практичному випадку, розглянутому в підрозділі 3.4, перехід від посилального подання ключа до `readonly struct` супроводжується зміною фактичного розміру з урахуванням вирівнювання та способу розміщення значення у структурі словника.

4.3.4 Правило виявлення надлишкової ємності колекцій

У базі фактів, сформованій після опрацювання знімка пам'яті, для об'єктів-колекцій фіксуються структурні характеристики, зокрема фактична кількість елементів і виділена внутрішня ємність. У межах цього правила такі характеристики використовуються для виявлення випадків, коли резерв внутрішнього буфера або хеш-таблиці істотно перевищує фактичну потребу в зберіганні елементів. До таких структур належать, зокрема, поля типів $List<T>$ та $Dictionary<K,V>$, які в цьому підпункті розглядаються лише за співвідношенням між кількістю елементів і виділеною ємністю [78].

Нехай $FL_{col}(t)$ - множина полів типу t , значення яких є колекціями. Для поля $p \in FL_{col}(t)$ через $Val_p(o)$ позначено значення цього поля в об'єкті o , а через $Count(\cdot)$ і $Capacity(\cdot)$ - відповідно фактичну кількість елементів і ємність колекції. Для уникнення невизначеності при нульовій ємності розглядається тільки підмножина об'єктів, у яких відповідна колекція має додатну ємність. Коефіцієнт заповнення та продукційне правило задаються у вигляді

$$\left\{ \begin{array}{l} O_{t,p}^+ = \{o \in O_t \mid Val_p(o) \neq null, Capacity(Val_p(o)) > 0\} \\ \varphi_{t,p} = \frac{1}{|O_{t,p}^+|} \sum_{o \in O_{t,p}^+} \frac{Count(Val_p(o))}{Capacity(Val_p(o))} \\ \text{IF } p \in FL_{col}(t) \wedge O_{t,p}^+ \neq \emptyset \wedge \varphi_{t,p} \leq \tau_{cap} \\ \text{THEN } Recommend(ReduceCapacity(t,p)) \end{array} \right. , (4.4)$$

де $O_{t,p}^+$ - усі об'єкти типу t , у яких поле p має ненульову ємність та визначено, $\tau_{cap} \in (0,1)$ задає граничне значення коефіцієнта заповнення, нижче якого ємність колекції вважається надлишковою. Дія $ReduceCapacity(t,p)$ означає рекомендацію переглянути початкову або поточну ємність колекції, а для малих наборів елементів - розглянути компактніше подання даних. Така рекомендація формується не за самим фактом наявності внутрішнього резерву, а за його стійким проявом у фактах знімка пам'яті.

Правило не підміняє оцінювання вартості зберігання інформації у хеш-колекціях. Коефіцієнт $\varphi_{t,p}$ характеризує саме рівень використання виділеної

ємності, тоді як службові витрати словника, пов'язані з внутрішніми структурами, аналізуються через окрему модель, побудовану в підрозділі 2.5.4 і перевірену в підрозділі 3.5.

4.3.5 Правило оцінювання вартості зберігання інформації у хеш-колекціях

Модель оцінювання вартості зберігання інформації у хеш-колекціях типу Dictionary<K,V> побудовано у підпункті 2.5.4. У базі фактів для таких колекцій використовуються кількість пар ключ–значення, обсяг корисної інформації, фактичний обсяг пам'яті та агреговані коефіцієнти ефективності зберігання, визначені у (2.41)-(2.45). У межах бази знань ці величини інтерпретуються як ознаки надлишкових службових витрат, пов'язаних із використанням хеш-структури для малих наборів елементів.

Для групи словників $O_{D,t_D}^{(n)}(u)$, що відповідає типу t_D і містить n елементів, використовується агрегований коефіцієнт $\mu_{t_D}^{(n)}(u)$, введений у (2.44). Якщо цей коефіцієнт менший за порогове значення τ_D , то частка корисної інформації у фактичному обсязі пам'яті є недостатньою. Для відокремлення саме тих випадків, де компактне подання може бути доцільним, додатково вводиться параметр N_D , що задає максимальну кількість елементів у словнику для застосування цього правила.

Продукційне правило має вигляд

$$\mathbf{IF} t_D \in T_D(u) \wedge O_{D,t_D}^{(n)}(u) \neq \emptyset \wedge n \leq N_D \wedge \mu_{t_D}^{(n)}(u) < \tau_D$$

THEN *Recommend(ReplaceWithCompactCollection(t_D, n))*, (4.5)

де N_D визначає межу, в межах якої хеш-колекція розглядається як мала за кількістю елементів, а τ_D використовується як поріг ефективності зберігання, заданий у (2.45). Дія *ReplaceWithCompactCollection(t_D, n)* означає рекомендацію перевірити можливість заміни стандартного словника

на компактніше подання: колекцію з фіксованою кількістю елементів, спеціалізовану структуру для одного або кількох елементів, або інше нехешоване подання, якщо воно зберігає семантику доступу до даних.

Це правило не дублює правило виявлення надлишкової ємності колекцій, наведене у 4.3.4. Коефіцієнт заповнення колекції оцінює співвідношення між кількістю елементів і виділеною ємністю, тоді як $\mu_{t_D}^{(n)}(u)$ характеризує співвідношення між корисним інформаційним обсягом і фактичними витратами пам'яті на зберігання словника. Експериментальне оцінювання у підпункті 3.5 показує, що для словників з малою кількістю елементів фактичний обсяг пам'яті може значною мірою визначатися службовими структурами, а не самими даними пар ключ–значення.

Заміна *Dictionary* $\langle K, V \rangle$ на компактніше подання не є допустимою лише за фактом малого числа елементів або низького коефіцієнта ефективності зберігання. Для такої дії потрібний аналіз профілю операцій доступу: частоти пошуку за ключем, вставлення, видалення, оновлення, обходу та побудови колекції. Якщо словник використовується в режимі частих звернень за ключем, має змінну або наперед необмежену кількість елементів, використовує спеціальну функцію порівняння ключів, бере участь у потокобезпечному доступі або зберігає семантику, пов'язану з хешованим пошуком, перехід до нехешованого чи спеціалізованого подання не формується як допустима дія. У такому випадку правило фіксує надлишкові службові витрати пам'яті для групи словників, але остаточне перетворення потребує перевірки операційної вартості доступу.

4.3.6 Правило оцінювання доцільності стискання масивів байтів

У моделі, побудованій у підпункті 2.5.6, визначено інформаційний вміст масиву байтів, безвтратне відображення стискання, множину об'єктів-

кандидатів на стискання та відносний ефект зменшення обсягу пам'яті згідно з (2.50)-(2.55). У межах бази знань ці величини використовуються як факти, отримані після аналізу знімка пам'яті.

Правило застосовується не до всіх масивів байтів, а лише до множини кандидатів $O_{cmp}^*(u)$, сформованої за умовами (2.54). До цієї множини належать об'єкти, для яких виконуються обмеження щодо мінімальної довжини, безвтратності відновлення, зменшення розміру після стискання та допустимості заміни з погляду способу доступу до даних. Для таких кандидатів використовується відносний ефект стискання $\rho_{cmp}^*(u)$, визначений у (2.55).

Для відокремлення несуттєвого зменшення обсягу пам'яті від практично значущого вводиться порогове значення $\tau_{cmp} \in (0,1)$. Продукційне правило має вигляд

$$\begin{aligned} & \text{IF } O_{cmp}^*(u) \neq \emptyset \wedge \rho_{cmp}^*(u) \geq \tau_{cmp} \\ & \text{THEN } Recommend(ApplyCompression(O_{cmp}^*(u))). \end{aligned} \quad (4.6)$$

τ_{cmp} задає мінімальну частку пам'яті, яку доцільно зменшити для формування рекомендації. Дія *ApplyCompression* означає рекомендацію розглянути стиснене подання для об'єктів-кандидатів, а не для всіх масивів типу *System.Byte[]*. Вибір конкретного алгоритму стискання виконується з урахуванням значення $\rho_{cmp}^*(u)$, вартості стискання та відновлення, а також характеру доступу до відповідних даних.

Для масивів типу *System.Byte[]* стискання не застосовується до всієї множини масивів, навіть якщо коефіцієнт стискання має прийнятне значення. Недопустимими для такої дії є активні буфери введення-виведення, мережевого обміну, читання або запису, масиви повторного використання, службові буфери середовища виконання, а також об'єкти з уже стисненим, зашифрованим або близьким до випадкового вмістом. Окремо відхиляються масиви, для яких потрібний прямий довільний доступ до окремих байтів або низька затримка доступу, несумісна з витратами на стискання та відновлення.

Отже, дія стискання має розглядатися лише для об'єктів-кандидатів, що подають збережений бінарний стан, не беруть участі в активних операціях обміну і проходять перевірку безвтратного відновлення інформаційного вмісту.

4.3.7 Правило виявлення надлишкової гранулярності синхронізації у потокобезпечних словниках

Окремий сценарій надмірного використання пам'яті пов'язаний із потокобезпечними словниками, у яких для підтримання одночасного доступу використовуються службові об'єкти синхронізації. У цьому підпункті розглядаються об'єкти типу *ConcurrentDictionary* $\langle K, V \rangle$, де K і V позначають відповідно тип ключа та тип значення. Для таких колекцій кількість блоків синхронізації впливає на службові витрати пам'яті, тому ця характеристика може бути використана як факт бази знань.

Нехай $O_{K,V}(u)$ - множина екземплярів *ConcurrentDictionary* $\langle K, V \rangle$ у знімку пам'яті u , а $Locks_{cd}(o)$ - кількість блоків синхронізації, пов'язаних з екземпляром o . Для виділення об'єктів із надлишковою гранулярністю вводиться граничне значення L_{max} . Якщо кількість таких об'єктів для фіксованої пари типів (K, V) не менша за поріг N_{cd} , база знань формує рекомендацію щодо перегляду рівня синхронізації:

$$\left\{ \begin{array}{l} O_{K,V}^{locks}(u) = \{o \in O_{K,V}(u) \mid Locks_{cd}(o) > L_{max}\}, \\ \mathbf{IF} \mid O_{K,V}^{locks}(u) \mid \geq N_{cd} \mathbf{THEN} \mathit{Recommend ReduceConcurrency}(K, V), \end{array} \right. (4.7)$$

де $L_{max} \in \mathbb{N}$ задає гранично допустиму кількість блоків синхронізації для одного екземпляра, а $N_{cd} \in \mathbb{N}$ - мінімальну кількість екземплярів одного типу, за якої сценарій розглядається як систематичний. Дія *ReduceConcurrency* (K, V) означає рекомендацію перевірити необхідність поточного рівня синхронізації для словників із парою типів (K, V) . Якщо

режим доступу до даних не потребує високої конкурентності для кожного окремого екземпляра, може бути розглянуто зменшення рівня синхронізації або інше подання колекції.

Це правило не встановлює помилку синхронізації та не робить висновку про поведінку потоків лише за знімком пам'яті. Його призначення полягає у виявленні групи потокобезпечних словників, для яких кількість службових блоків синхронізації може бути надлишковою відносно фактичного використання пам'яті. Остаточне рішення щодо зміни рівня синхронізації має враховувати вимоги до паралельного доступу в програмному коді.

4.4 Джерела знань і формування фактів зі знімка пам'яті

Побудована у підрозділі 4.2 продукційна архітектура відокремлює фактичний стан пам'яті від правил його інтерпретації. У підрозділі 4.3 сформовано правила, які працюють не безпосередньо з байтовим вмістом дампу, а з формалізованими фактами. Тому формування бази фактів є проміжним етапом між інструментальним аналізом знімка пам'яті та застосуванням правил бази знань.

Джерелом фактів є стан керованої пам'яті, зафіксований у момент створення знімка. До таких фактів належать відомості про об'єкти, типи, розміри, значення полів, посилання між об'єктами, характеристики колекцій, масивів і службових структур. Ці характеристики не містять висновку про доцільність внесення змін; вони лише задають формалізований опис досліджуваного стану пам'яті, який надалі інтерпретується продукційними правилами.

Джерела знань мають іншу роль. Вони визначають, які саме факти потрібно вилучити зі знімка пам'яті та як ці факти зіставляти з типовими сценаріями надмірного використання пам'яті. До таких джерел належать

математичні моделі представлення об'єктів у пам'яті, побудовані у розділі 2, експериментальні результати розділу 3, а також правила, сформовані у підрозділі 4.3. Саме ці джерела задають набір ознак, порогових параметрів і рекомендацій, які використовуються базою знань.

Інструментальною основою формування фактів є програмне опрацювання дампу пам'яті з використанням бібліотеки `Microsoft.Diagnostics.Runtime`, або `ClrMD`. Її застосування дає змогу отримати програмний доступ до керованої купи, типів, об'єктів, полів і зв'язків між об'єктами [100]. У межах цієї роботи `ClrMD` використовується не як самостійний засіб інтерпретації причин надмірного використання пам'яті, а як інструмент вилучення первинних даних, на основі яких будується база фактів.

Формування фактів виконується у кілька послідовних етапів. На першому етапі зі знімка пам'яті виділяються об'єкти керованої купи та встановлюється їх належність до типів, визначених у формальній моделі розділу 2. Для кожного об'єкта фіксуються адресні й розмірні характеристики, склад полів, значення доступних полів і посилання на інші об'єкти. У результаті формується множина первинних фактів, які описують локальні властивості окремих об'єктів.

На другому етапі первинні факти перетворюються на структурні факти. До них належать групи об'єктів одного типу, групи об'єктів з однаковим інформаційним вмістом, множини об'єктів-колекцій, словників, масивів байтів і потокобезпечних контейнерів. На цьому рівні використовуються вже введені у розділі 2 способи групування за типами, значеннями та структурними ознаками. Таке групування зменшує розмірність подальшого аналізу, оскільки правила бази знань працюють не з кожним об'єктом окремо, а з узагальненими множинами та їхніми характеристиками.

На третьому етапі обчислюються агреговані факти, пов'язані з умовами продукційних правил. До них належать середня кратність повторення незмінюваних значень, коефіцієнт використання діапазону значень поля, відносні втрати через вирівнювання, коефіцієнт заповнення колекцій,

коефіцієнт ефективності зберігання у словниках, відносний ефект стискання масивів байтів і кількість блоків синхронізації у потокобезпечних словниках. Саме ці агреговані факти є вхідними даними для правил, сформульованих у 4.3.1–4.3.7.

У результаті опрацювання знімка пам'яті база фактів може бути подана як сукупність трьох рівнів: первинних фактів про окремі об'єкти, структурних фактів про групи об'єктів і агрегованих фактів про кількісні характеристики цих груп. Такий поділ забезпечує узгодження між формальною моделлю пам'яті та продукційними правилами. Первинні факти відтворюють стан пам'яті, структурні факти задають об'єкти аналізу, а агреговані факти визначають умови спрацювання правил.

Формування фактів ґрунтується на даних знімка пам'яті й не потребує повного доступу до вихідного коду програмного додатка. Це дає змогу аналізувати програмні комплекси, для яких доступна фактична структура пам'яті, але обмежена інформація про внутрішню реалізацію. Водночас сформована рекомендація не підміняє аналізу предметної семантики програмного коду: вона вказує на виявлений сценарій і на групу об'єктів, для яких потрібно перевірити можливість зміни подання даних.

Отже, джерела знань визначають правила інтерпретації, а знімок пам'яті надає фактичні дані для їх застосування. Механізм формування фактів забезпечує перехід від байтового подання дампу до формалізованих ознак, придатних для логічного виведення. Це створює зв'язок між моделями розділу 2, експериментальними сценаріями розділу 3 і рекомендаціями, що формуються базою знань у розділі 4.

4.5 Апробація продукційного механізму на промислових знімках пам'яті

Апробацію розробленої експертної системи виконано на промислових знімках пам'яті .NET-застосунків, отриманих під час експлуатації. У цьому підрозділі результати розділу 3 використовуються не як повтор експериментального аналізу, а як перевірка здатності системи формувати факти, застосовувати продукційні правила та генерувати рекомендації на основі фактичного стану керованої пам'яті.

Вхідними даними для апробації були знімки пам'яті обсягом від 20 до 180 ГБ, що відповідає масштабу високонавантажених програмних систем, розглянутих у розділі 3. На першому етапі зі знімків пам'яті формувалася база фактів: множини об'єктів і типів, значення полів, розмірні характеристики, групи об'єктів за значенням, характеристики колекцій, словників, масивів байтів і службових структур. На другому етапі до цих фактів застосовувалися правила, сформовані у підрозділі 4.3.

Апробація охоплювала ті сценарії, для яких у досліджуваних знімках пам'яті були наявні достатні факти для спрацювання правил. Для незмінюваних рядкових значень система формувала класи еквівалентності за інформаційним вмістом і виявляла групи фізично різних об'єктів System.String з однаковими значеннями. На основі правила 4.3.1 такі групи інтерпретувалися як дублювання незмінюваної інформації, а рекомендація полягала у перевірці можливості інтернування, кешування або застосування пулу спільних значень [54].

Для користувацьких типів, що використовувалися у складі хеш-колекцій, система застосовувала правила, пов'язані зі звуженням оголошених типів полів і втратами через фізичне подання об'єкта у 64-бітному середовищі CLR. У випадках, де фактичний діапазон значень поля був істотно меншим за область допустимих значень його оголошеного типу, система формувала рекомендацію щодо перевірки компактнішого типу поля. Таке спрацювання узгоджується з результатами підрозділу 3.4, де було розглянуто компактне подання ключового типу для Dictionary<K,V>.

Окрему групу становили правила для колекцій і хеш-колекцій. Для об'єктів із низьким коефіцієнтом заповнення система формувала рекомендації щодо перегляду початкової або поточної ємності. Для словників з малою кількістю елементів додатково використовувалася оцінка співвідношення між корисним інформаційним обсягом і фактичними витратами пам'яті на службові структури. У таких випадках рекомендація полягала не в безумовній заміні Dictionary<K,V>, а в перевірці можливості компактнішого подання для груп словників з малою кількістю пар ключ–значення [78].

Для масивів типу System.Byte[] система використовувала факти про довжину масивів, їх сумарний обсяг і результати оцінювання безвтратного стиснення. Правило 4.3.6 спрацьовувало лише для множини кандидатів, у яких стиснення забезпечувало збереження інформаційного вмісту та мало практично значущий очікуваний ефект. Це дало змогу відокремити масиви, для яких стиснене подання може бути доцільним, від службових або вже стиснених бінарних даних [83].

Сформовані рекомендації перевірялися шляхом порівняння очікуваного ефекту з результатами експериментальних змін, розглянутих у розділі 3. Для частини сценаріїв, зокрема дублювання рядків, компактного подання ключів, спеціалізованих колекцій для малих кількостей елементів і стиснення масивів байтів, отримані оцінки підтвердили, що правила експертної системи виділяють саме ті групи об'єктів, які дають основний внесок у надмірне використання пам'яті.

Практичний ефект від застосування сформованих рекомендацій оцінювався за зміною пікового використання пам'яті та супутніх характеристик роботи застосунку. Для розглянутих сценаріїв упровадження рекомендацій давало зменшення пікового споживання пам'яті в межах 15–30 % залежно від конфігурації програмного додатка та характеру навантаження. Це зменшення супроводжувалося зниженням навантаження на механізм збирання сміття та вивільненням частини ресурсів, пов'язаних з обробленням надлишкових об'єктів [27; 78; 101].

Результати апробації також підтвердили, що система може формувати висновки без повного аналізу вихідного коду. Висновки будуються на фактах, отриманих зі знімка пам'яті, тому система виявляє не декларативну структуру програмного коду, а фактичний стан об'єктів у пам'яті. Водночас сформована рекомендація не є автоматичною зміною реалізації: вона визначає групу об'єктів, тип сценарію та очікуваний напрям змін, після чого потребує перевірки з урахуванням семантики програмного коду.

Отже, апробація на промислових знімках пам'яті підтвердила працездатність побудованої експертної системи як засобу переходу від фактів знімка пам'яті до інтерпретованих рекомендацій. Система забезпечує формування фактів, застосування правил бази знань, виявлення типових сценаріїв надмірного використання пам'яті та кількісне оцінювання очікуваного ефекту від впровадження змін.

4.6 Висновки до розділу

У четвертому розділі побудовано формальну продукційну модель представлення знань для аналізу знімків пам'яті, у якій факти, отримані зі знімка пам'яті, перетворюються правилами-предикатами на множину сценаріїв надмірного використання пам'яті, рекомендацій та кількісних оцінок очікуваного ефекту. Побудова системи спирається на формальну модель знімка пам'яті, розроблену в розділі 2, та експериментально досліджені сценарії, наведені в розділі 3.

1. Сформульовано призначення продукційного представлення результатів ідентифікації надмірності пам'яті як механізму переходу від фактів знімка пам'яті до діагностичних висновків, рекомендацій та кількісних оцінок. Визначено, що система має виконувати перехід від фактів, отриманих зі знімка пам'яті, до інтерпретованих рекомендацій щодо

зміни подання даних, структури об'єктів або параметрів використання колекцій.

2. Побудовано продукційну архітектуру застосування правил, що містить базу фактів, базу знань і механізм логічного виведення. Такий поділ відокремлює фактичний стан пам'яті від правил його інтерпретації та дає змогу застосовувати одну систему правил до різних знімків пам'яті.
3. Сформовано базу знань у вигляді системи продукційних правил для основних сценаріїв надмірного використання пам'яті. До неї включено правила виявлення дублювання незмінюваних об'єктів, надлишкового діапазону оголошених типів полів, втрат через вирівнювання об'єктів, надлишкової ємності колекцій, нераціональної вартості зберігання у хеш-колекціях, доцільності стискання масивів байтів і надлишкової гранулярності синхронізації.
4. Визначено механізм формування фактів зі знімка пам'яті. Факти подано на трьох рівнях: первинні факти про окремі об'єкти, структурні факти про групи об'єктів і агреговані факти, що відповідають умовам продукційних правил. Це забезпечує узгоджений перехід від інструментально отриманих даних до логічного виведення.
5. Під час апробації система формувала факти зі знімків пам'яті, застосовувала правила бази знань і виділяла групи об'єктів, для яких можливі рекомендації щодо інтернування або кешування значень, звуження типів полів, зміни способу подання ключів, перегляду ємності колекцій, заміни хеш-колекцій для малих кількостей елементів або стискання масивів байтів.
6. Показано, що сформовані рекомендації мають діагностичний характер і не підміняють аналіз семантики програмного коду. Експертна система визначає сценарій, групу об'єктів і очікуваний напрям дії, після чого відповідна зміна має перевірятися з урахуванням режиму доступу до даних, предметних обмежень і вимог до коректності програмної реалізації.

7. Результати апробації підтвердили працездатність продукційного механізму як засобу переходу від фактичного стану пам'яті до формалізованих рекомендацій. Для розглянутих сценаріїв упровадження сформованих рекомендацій забезпечувало зменшення пікового використання пам'яті в межах 15–30 % залежно від конфігурації програмного додатка та характеру навантаження.

Отримані результати розділу встановлюють зв'язок між математичним апаратом представлення пам'яті, емпіричними результатами аналізу промислових знімків і продукційною моделлю формування рекомендацій. Основні результати розділу опубліковано в [20, 26, 78, 84, 103, 104].

ВИСНОВКИ

У дисертаційній роботі розв'язано актуальну наукову задачу, що полягає у розробленні формальної математичної моделі знімка пам'яті, критеріїв надмірності, алгоритмічних процедур та продукційної моделі для застосування правил для ідентифікації сценаріїв надмірного використання оперативної пам'яті у програмних застосунках керованого середовища виконання та кількісного оцінювання можливого зменшення її обсягу після допустимих перетворень подання даних. Отримано такі основні наукові та практичні результати:

1. Проаналізовано існуючі підходи до подання, інтерпретації, аналізу та зменшення використання оперативної пам'яті у програмних застосунках керованого середовища виконання. Встановлено, що наявні методи діагностики орієнтовані переважно на фіксацію динаміки споживання ресурсів та витоків пам'яті, проте є недостатніми для виявлення стабільного надлишкового подання інформації без ознак витоків. Це обґрунтовує необхідність переходу до формалізованого математичного аналізу внутрішньої типізованої структури знімків пам'яті.

2. Побудовано формальну модель знімка пам'яті як скінченної типізованої структури, що відображає об'єкти пам'яті, їхні типи, адреси, розміри, значення полів, зв'язки між об'єктами та службові елементи керованого середовища виконання. Введення множин об'єктів, типів, потоків виконання, функцій групування, відношень еквівалентності та кількісних характеристик забезпечило перехід від байтового подання пам'яті до математично визначеної структури, придатної для аналізу сценаріїв надмірності.

3. Сформульовано задачу ідентифікації сценаріїв надмірного використання пам'яті на основі формалізованої структури знімка пам'яті. Задачу подано як виявлення класів надлишкового подання інформації у типізованій структурі об'єктів із використанням класів еквівалентності за

інформаційним вмістом, поняття безнадлишкового представлення, допустимого перетворення подання даних та кількісного коефіцієнта надмірності. Така формалізація дозволила відокремити сценарії стабільної структурної надмірності від класичних сценаріїв витоку пам'яті.

4. Визначено формальні критерії, предикати та кількісні характеристики для виявлення окремих класів надмірного подання інформації в оперативній пам'яті. Для відмежування сценаріїв стабільної структурної надмірності від витоку пам'яті використано критерій монотонного зростання кількості досяжних об'єктів або сумарного обсягу зайнятої ними пам'яті у послідовності знімків. Визначено критерії дублювання незмінюваних об'єктів, предикати виявлення надлишкового діапазону оголошених типів полів, втрат через фізичне вирівнювання об'єктів, надлишкової ємності колекцій, неефективного використання службових структур хеш-колекцій та доцільності безвтратного стискання масивів байтів.

5. Розроблено алгоритмічні процедури обчислення характеристик надмірності та оцінювання можливого зменшення обсягу пам'яті після допустимих перетворень подання даних. Побудовано алгоритмічну процедуру прискореного виявлення дублікатів незмінюваних об'єктів шляхом попереднього групування за швидко обчислюваними ознаками інформаційного вмісту з подальшою перевіркою еквівалентності всередині отриманих підмножин. Визначено процедури кількісного оцінювання службових витрат хеш-колекцій, доцільності звуження типів полів за фактичним діапазоном значень, втрат через вирівнювання об'єктів та ефекту безвтратного стискання масивів байтів.

6. Побудовано продукційну модель застосування правил до формалізованого подання знімка пам'яті для визначення сценаріїв надмірності, формування діагностичних висновків, рекомендацій та кількісних оцінок. Умови застосування правил задано предикатами над характеристиками об'єктів, типів, значень полів, колекцій, масивів та службових структур середовища виконання, що формалізує перехід від фактів

знімка пам'яті до множини виявлених сценаріїв надмірності, відповідних рекомендацій та оцінок очікуваного зменшення обсягу пам'яті.

7. Проведено перевірку запропонованих моделей, критеріїв та алгоритмічних процедур на промислових знімках пам'яті програмних застосунків у 64-бітному середовищі виконання. Виконано кількісне оцінювання дублювання незмінюваних об'єктів, експериментальну перевірку прискореного виявлення дублікатів, оцінювання компактного подання ключових типів для хеш-колекцій, аналіз службових витрат колекцій з малою кількістю елементів та оцінювання ефекту стискання масивів байтів. Програмну реалізацію використано виключно як інструментальний засіб експериментальної апробації моделей, критеріїв та алгоритмічних процедур; вона не становить самостійного наукового результату. Для досліджених сценаріїв виконано зіставлення розрахункових оцінок надмірності та прогнозованого зменшення обсягу пам'яті з фактичними результатами після реалізації рекомендованих перетворень структури даних. Практичне значення одержаних результатів підтверджено впровадженням запропонованих моделей, алгоритмічних процедур та продукційних правил у промислових програмних системах, що засвідчено актом впровадження, наведеним у додатку А.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Silberschatz A., Galvin P. B., Gagne G. Operating System Concepts. – 10th ed. – Hoboken, NJ: Wiley, 2018. – 1040 p. ISBN: 9781119320913.
2. Huffman C. Windows Performance Analysis Field Guide. – Waltham, MA: Elsevier (Syngress), 2015. – XXIII, 349 p. ISBN: 9780124167018.
3. Hewardt M., Pravat D. Advanced Windows Debugging. – Boston: Addison-Wesley, 2007. – 840 p. ISBN: 9780321374462.
4. Dolan-Gavitt B. The VAD tree: A process-eye view of physical memory // Digital Investigation. 2007. – Vol. 4, Suppl. 1. – P. S62–S64. DOI: 10.1016/j.diin.2007.06.008.
5. Otsuki Y., Kawakoya Y., Iwamura M., Miyoshi J., Ohkubo K. Building stack traces from memory dump of Windows x64 // Digital Investigation. – 2018. – Vol. 24, Suppl. 1. – P. S101–S110. DOI: 10.1016/j.diin.2018.01.013.
6. Мітіков М. Ю. Оптимізація процесу пошуку сегментів пам'яті: математичне моделювання та експериментальна оцінка повноти інформаційного вмісту при фіксованих об'ємах пам'яті. Інформаційні технології в металургії та машинобудуванні (ІТММ'2025): тези доповідей Міжнародної науково-практичної конференції (Дніпро, 23–24 квітня 2025 р.). Дніпро: УДУНТ, 2025. С. 676–680.
7. Soulamı T. Inside Windows Debugging. – Redmond, WA: Microsoft Press, 2012. – 448 p. ISBN: 9780735662780.
8. Russinovich M., Margosis A. Troubleshooting with the Windows Sysinternals Tools. – 2nd ed. Redmond, WA: Microsoft Press, 2016. – 648 p. ISBN: 9780735684447.
9. Ptashkin, R., Hozhyi, O., Obruch, Y., Pavlov, V., & Kalinichenko, R. (2020). Ram analysis as one of the methods for computer forensics. Bulletin of Cherkasy State Technological University, 25(4), 39-47. <https://doi.org/10.24025/2306-4412.4.2020.214993>.

10. Ligh M. H., Case A., Levy J., Walters A. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. – Indianapolis, IN: Wiley, 2014. – 912 p. ISBN: 9781118825099.
11. Bulat, A., Kiseleva, E., Hart, L., Prytomanova, O. (2023). Generalized Models of Logistics Problems and Approaches to Their Solution Based on the Synthesis of the Theory of Optimal Partitioning and Neuro-Fuzzy Technologies. In: Zgurovsky, M., Pankratova, N. (eds) *System Analysis and Artificial Intelligence . Studies in Computational Intelligence*, vol 1107. Springer, Cham. https://doi.org/10.1007/978-3-031-37450-0_21.
12. Ferrandez, Tess. Debugging .NET Core memory issues (on Linux) with dotnet dump. www.tessferrandez.com. [З мережі] Microsoft, 2021 p. <https://www.tessferrandez.com/blog/2021/03/18/debugging-a-netcore-memory-issue-with-dotnet-dump.html>.
13. Ткаченко О., Голубенко О., Власенко В., Антоненко А. Методи оптимізації використання оперативної пам'яті в процесах підвищення швидкодії комп'ютерних систем. Вимірювальна та обчислювальна техніка в технологічних процесах. 2025. № 4(84). С. 465–472. DOI: <https://doi.org/10.31891/2219-9365-2025-84-57>.
14. Андреев Д. А., Золотько К. Є. Моделювання бізнес-процесів ІТ-проектів за допомогою теорії черг. *Математичне та комп'ютерне моделювання. Серія: Фізико-математичні науки*. 2025. Вип. 28. С. 5–16. DOI: 10.32626/2308-5878.2025-28.5-16.
15. Ткаченко О., Голубенко О. Оптимізація однопроцесорного оброблення мультизапитів. *Advanced Information Technology*. 2023. Т. 1, № 2. С. 32–37. DOI: 10.17721/AIT.2023.1.05.
16. Мітіков М.Ю., Гук Н.А. Огляд методів та інструментів системного аналізу продуктивності програмного забезпечення. Математичне та програмне забезпечення інтелектуальних систем (МПЗІС-2023): Тези доповідей XXI Міжнародної науково-практичної конференції, Дніпро, 22-24 листопада

- 2023 р. / Під загальною редакцією О.М. Кісельової. – Дніпро: ДНУ, 2023. С. 213–214.
17. Gregg, Brendan. *Systems Performance, Enterprise and the Cloud*, second edition. 2021. ISBN-13: 978-0-13-682015-4.
18. Microsoft. How to use the Debug Diagnostics tool to troubleshoot a process that has stopped responding in IIS. [З мережі] Microsoft, 04 2018 р. <https://support.microsoft.com/en-gb/topic/how-to-use-the-debug-diagnostics-tool-to-troubleshoot-a-process-that-has-stopped-responding-in-iis-995db9a3-a3be-6d20-cf2f-c48101a64444>.
19. Сімакін С. К., Божуха Л. М. Метод інтелектуального керування часом життя кешу вебсервісів на основі алгоритмів навчання з підкріпленням. Актуальні проблеми автоматизації та інформаційних технологій. 2025. Т. 29. С. 363–369. DOI: <https://doi.org/10.15421/432533>.
20. Мітіков М.Ю., Гук Н.А. Виявлення проблем у роботі програмного забезпечення на основі аналізу знімків пам'яті. Питання прикладної математики і математичного моделювання. 2023. Вип. 23. С. 171–178. DOI: <https://doi.org/10.15421/322318>.
21. Mitikov M.Y, Guk N.A, Honcharova Y. Real-world example of application performance anomaly detection through memory analysis. *Modern scientific and technical research – 2023: матеріали II Всеукраїнської науково-практичної конференції молодих учених та студентів (Дніпро, 11 травня 2023 р.)*. Дніпро, 2023. С. 280–282.
22. Tkachenko O., Lemeshko A. Optimization of metadata volume in modern information systems: methods, tools and algorithmic approaches. *Advanced Information Technology*. 2025. Vol. 1, No. 4. DOI: <https://doi.org/10.17721/AIT.2025.1.01>.
23. Логвин Д. А., Божуха Л. М. Метрики оцінювання ефективності програмного забезпечення для обробки природної мови. Актуальні проблеми автоматизації та інформаційних технологій. 2025. Т. 29. С. 266–276. DOI: <https://doi.org/10.15421/432524>.

- 24.Золотько К. Є., Красношарпа Д. В. Управління та діагностика надання ІТ-сервісів. Питання прикладної математики і математичного моделювання. 2022. Вип. 22. С. 60–66. DOI: <https://doi.org/10.15421/322206>.
25. Leach, K., Spensky, C., Weimer, W., & Zhang, F. (2016). Towards transparent introspection. 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 248–259. <https://doi.org/10.1109/SANER.2016.25>
26. Kiseleva O. M., Prytomanova O. M., Filat O. A. Application of clustering methods for detecting network threats. Питання прикладної математики і математичного моделювання. 2023. Вип. 23. С. 129–137. DOI: <https://doi.org/10.15421/322311>.
27. Мітіков М. Ю., Гук Н. А. Дослідження проблем швидкодії програмних додатків. Математичне та комп'ютерне моделювання. Серія: Технічні науки. 2024. Вип. 25. С. 22–36. DOI: <https://doi.org/10.32626/2308-5916.2024-25.22-36>.
28. Гук Н.А., Мітіков М.Ю. Сучасні проблеми ідентифікації аномалій в роботі Enterprise Systems. Системні технології. Вип. 5. 2024. С. 146–153 DOI: <https://doi.org/10.34185/1562-9945-5-154-2024-15>.
29. Lakhno, V., Tkach, Y., et al. (2016) 'Development of adaptive expert system of information security using a procedure of clustering the attributes of anomalies and cyber attacks', Eastern-European Journal of Enterprise Technologies, 6(9 (84)), pp. 32–44. doi:10.15587/1729-4061.2016.85600.
30. Lakhno, V. et al. (2016) 'Design of adaptive system of detection of cyber-attacks, based on the model of logical procedures and the coverage matrices of features', Eastern-European Journal of Enterprise Technologies, 3(9(81)), p. 30. doi:10.15587/1729-4061.2016.71769.
31. Protsiuk, V. (2024) 'Anomaly detection models of for sensor data of oil and gas well drilling process under uncertainty', Herald of Khmelnytskyi National University. Technical sciences, 333(2), pp. 177–188. doi:10.31891/2307-5732-2024-333-2-29.

32. Meleshko, Y. et al. (2024) 'Development a set of mathematical models for anomaly detection in high-load complex computer systems', *Eastern-European Journal of Enterprise Technologies*, 6(4 (132)), pp. 14–25. doi:10.15587/1729-4061.2024.316779.
33. Semenov, S. et al. (2022) 'Devising a procedure for defining the general criteria of abnormal behavior of a computer system based on the improved criterion of uniformity of input data samples', *Eastern-European Journal of Enterprise Technologies*, 6(4 (120)), pp. 40–49. doi:10.15587/1729-4061.2022.269128.
34. Jensen, Simon & Sridharan, Manu & Sen, Koushik & Chandra, Satish. (2015). MemInsight: platform-independent memory debugging for JavaScript. 345-356. 10.1145/2786805.2786860.
35. Degenbaev, Ulan & Eisinger, Jochen & Hara, Kentaro & Hlopko, Marcel & Lippautz, Michael & Payer, Hannes. (2018). Cross-component garbage collection. *Proceedings of the ACM on Programming Languages*. 2. 1-24. doi:10.1145/3276521.
36. Chelpanova, O. & Turchyna, V.. (2022). Узагальнення аномальних випадків у задачах упорядкування. *Problems of applied mathematics and mathematic modeling*. doi:10.15421/322122.
37. Lou, C., Chen, C., Huang, P., Dang, Y., Qin, S., Yang, X., Li, X., Qingwei Lin, & Chintalapati, M. (2025, May 19). Resin: A holistic service for dealing with memory leaks in production cloud infrastructure. Microsoft Research. <https://www.microsoft.com/en-us/research/publication/resin-a-holistic-service-for-dealing-with-memory-leaks-in-production-cloud-infrastructure/>
38. Nikolov, V., Kapitza, R. & Hauck, F.J. Recoverable Class Loaders for a Fast Restart of Java Applications. *Mobile Netw Appl* 14, 53–64 (2009). <https://doi.org/10.1007/s11036-008-0115-8>
39. Maliienko O., Turchyna V. Analysis of the Impact of Task Prioritization Lists on the Potential for Avoiding Anomalies in Task Scheduling // *Системні технології*. 2024. № 6(155). С. 167–174. DOI: 10.34185/1562-9945-6-155-2024-16

40. Kuznetsov R., Marcolla A., Khomenko D. Anomaly Detection of Memory Leaks Using Machine Learning in Cloud-Based Infrastructure // arXiv preprint arXiv:2109.12408. – 2021.
41. Jump, Maria & McKinley, Kathryn. (2009). Detecting memory leaks in managed languages with Cork. *Softw., Pract. Exper.*. 40. 1-22. 10.1002/spe.945.
42. Jump, M., & McKinley, K. S. (2007). Cork: Dynamic memory leak detection for garbage-collected languages. *ACM SIGPLAN Notices*, 42(1), 31–38. <https://doi.org/10.1145/1190215.1190224>
43. Mitchell, Nick & Sevitsky, Gary. (2003). LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. 351-377. 10.1007/978-3-540-45070-2_16.
44. Novark, G., Berger, E. D., & Zorn, B. G. (2009). Efficiently and precisely locating memory leaks and bloat. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 397–407. <https://doi.org/10.1145/1542476.1542521>
45. Shahoor, Arooba & Abdylidayev, Satbek & Hong, Hyeonggi & Yi, Jooyong & Kim, Dongsun. (2025). Proactive Debugging of Memory Leakage Bugs in Single Page Web Applications. *IEEE Transactions on Software Engineering*. PP. 1-27. 10.1109/TSE.2025.3571192.
46. Rudafshani, M., & Ward, P. A. S. (2017) LeakSpot: detection and diagnosis of memory leaks in JavaScript applications. *Software: Practice and Experience*, 47(1), 97–123. <https://doi.org/10.1002/spe.2406>
47. Marron, M., Sanchez, C., Su, Z., & Fahndrich, M. (2013). Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering*, 39(6), 774–786. <https://doi.org/10.1109/TSE.2012.69>
48. Salkeld, Robin & Kiczales, Gregor. (2013). Interacting with Dead Objects. *ACM SIGPLAN Notices*. 48. 10.1145/2509136.2509543.
49. Qi X. A Practical and Extensible Framework for Garbage Collection Tracing: Extended Abstract (Elephant Tracks II) // *Proc. SPLASH SRC 2018*. – 2018.

50. Kokosa, Konrad. Pro .NET Memory Management. Warsaw, 2018. ISBN 978-1-4842-4026-7.
51. Ievlanov M., Vasilcova N., Panforova I., Moroz B., Martynenko A., Moroz D. Comparison of solutions to the task of IT product configuration items early identification using hierarchical clusterization methods. Eastern-European Journal of Enterprise Technologies. 2024. Vol. 3, No. 2(129). P. 20–33. DOI: <https://doi.org/10.15587/1729-4061.2024.303526>.
52. Moroz B., Kabak L., Varekh N., Moroz D. Text document classification system with Big Data technologies usage. Information Technology: Computer Science, Software Engineering and Cyber Security. 2023. No. 2. P. 34–40. DOI: <https://doi.org/10.32782/IT/2023-2-4>.
53. Мітіков М. Ю., Гук Н. А. Інформаційна технологія діагностики надмірного використання пам'яті на основі аналізу знімків пам'яті. Modern Information and Communication Technologies in Transport, Industry and Education: тези доповідей XVII Міжнародної науково-практичної конференції (Дніпро, 13–14 грудня 2023 р.). Дніпро, 2023. С. 32.
54. Analysis and reduction of memory inefficiencies in Java strings. Kiyokuni Kawachiya, Kazunori Ogata, Tamiya Onodera.: 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, 2008. <https://doi.org/10.1145/1449764.1449795>.
55. 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. P. 147–157. DOI: <https://doi.org/10.15276/hait.07.2024.10>
56. System Deadlocks. E. G. Coffman, M. Elphick, A. Shoshani. 2: ACM Computing Surveys, 1971 p., T. 3. <https://doi.org/10.1145/356586.356588>.
57. Multicore SDK: A Practical and Efficient Deadlock Detector for Real-World Applications. Zhi Da Luo, Raja Das, Yao Qi. Berlin, Germany : IEEE, 2011. ISBN:978-1-61284-174-8.

58. Freeman, Adam. The Object Pool Pattern. Pro Design Patterns in Swift. Berkeley : https://doi.org/10.1007/978-1-4842-0394-1_7, 2014.
59. Implementing typed intermediate languages. Zhong Shao, Christopher League, Stefan Monnier. 1: ACM SIGPLAN Notices, 1999 p., T. 34. <https://doi.org/10.1145/291251.289460>.
60. Park, Jiwoong & Lee, Yunjae & Yeom, Heon & Son, Yongseok. (2020). Memory efficient fork-based checkpointing mechanism for in-memory database systems. 420-427. 10.1145/3341105.3375782.
61. Neto, Antonio & Caulfield, Adam & Alvares, Chistabelle & De Oliveira Nunes, Ivan. (2023). DiCA: A Hardware-Software Co-Design for Differential Checkpointing in Intermittently Powered Devices. 10.48550/arXiv.2308.12819.
62. Guo, Jinwei & Xiao, Bing & Cai, Peng & Qian, Weining & Zhou, Aoying. (2017). Efficient Multi-version Storage Engine for Main Memory Data Store. 205-220. 10.1007/978-3-319-68786-5_17.
63. Kemper, Alfons & Neumann, Thomas. (2011). HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. Proceedings - International Conference on Data Engineering. 195-206. 10.1109/ICDE.2011.5767867.
64. Dong, Feng & Li, Shaofei & Jiang, Peng & Li, Ding & Wang, Haoyu & Huang, Liangyi & Xiao, Xusheng & Chen, Jiedong & Luo, Xiapu & Guo, Yao & Chen, Xiangqun. (2023). Are we there yet? An Industrial Viewpoint on Provenance-based Endpoint Detection and Response Tools. 2396-2410. 10.1145/3576915.3616580.
65. Manna, Modhuparna & Case, Andrew & Ali-Gombe, Aisha & Richard III, Golden G.. (2022). Memory analysis of .NET and .Net Core applications. Forensic Science International: Digital Investigation. 42. 301404. 10.1016/j.fsidi.2022.301404.
66. Göbel, Thomas & Maltan, Stephan & Türr, Jan & Baier, Harald & Mann, Florian. (2022). ForTrace - A holistic forensic data set synthesis framework.

- Forensic Science International: Digital Investigation. 40. 301344.
10.1016/j.fsidi.2022.301344.
67. Wagner, James & Nissan, Mahfuzul & Rasin, Alexander. (2023). Database memory forensics: Identifying cache patterns for log verification. Forensic Science International: Digital Investigation. 45. 301567.
10.1016/j.fsidi.2023.301567.
68. Zhou, L., Xiao, J., Leach, K., Weimer, W., Zhang, F., Wang, G. (2019). Nighthawk: Transparent System Introspection from Ring-3. In: Sako, K., Schneider, S., Ryan, P. (eds) Computer Security – ESORICS 2019. ESORICS 2019. Lecture Notes in Computer Science, vol 11736. Springer, Cham.
https://doi.org/10.1007/978-3-030-29962-0_11
69. Мітіков М. Ю., Гук Н. А. Виявлення надлишкового використання пам'яті програмними додатками. Інформаційні технології: теорія і практика: тези доповідей I (VII) Міжнародної науково-практичної конференції здобувачів вищої освіти і молодих учених (Дніпро, 20–22 березня 2024 р.). Дніпро: Свідлер А. Л., 2024. С. 107–109.
70. Application performance benchmark: An experimental analysis on C# programs. Kassim, Roslaili, Bakar, Nordin Abu та Awang, Khalil Hj. Kuala Lumpur : 2008 International Symposium on Information Technology, 2008.
10.1109/ITSIM.2008.4631616.
71. Сімакін С. К., Божуха Л. М. Прогнозування навантаження на сервер з використанням ШІ для оптимізації веб-сервісів. Актуальні проблеми автоматизації та інформаційних технологій. 2024. Т. 28. С. 234–243. DOI: <https://doi.org/10.15421/432422>.
72. Machine Learning for Anomaly Detection: A Systematic Review. Nassif, Ali Bou, та ін., та ін.: IEEE, 2021. <https://doi.org/10.1109/ACCESS.2021.3083060>.
73. Louridas P. Static code analysis / P. Louridas. // IEEE Software. – 2006. – Vol. 23, no. 4. – С. 58–61, doi: 10.1109/MS.2006.114.
74. Smart detection in Application Insights [Електронний ресурс] / AbbyMSFT, KennedyDenMSFT, AaronMaxwell, v-jbasden // Microsoft. – 2024. – Режим

доступу до ресурсу: <https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/proactive-diagnostics>.

75. Weninger M. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection / M. Weninger, E. Gander, H. Mössenböck. // In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19). – 2019. – С. 273–284. doi: 10.1145/3297663.3310297.
76. Performance Monitoring Data / T. Veasey, S. Dodson. // International Journal of Machine Learning. – 2014. – Vol. 4, no. 2. – С. 120–126. doi: 10.7763/IJMLC.2014.V4.398.
77. Deep Learning for Anomaly Detection in Time-Series Data: Review, Analysis, and Guidelines / K. Choi, J. Yi, C. Park, S. Yoon. // IEEE. – 2021. – Vol. 9. – С. 120043–120065. doi: 10.1109/ACCESS.2021.3107975.
78. Мітіков М.Ю., Гук Н.А. Підвищення продуктивності колекцій у програмному забезпеченні за допомогою аналізу знімків пам'яті та математичного моделювання. Сучасні проблеми моделювання. 2025. № 27. С. 109–122. DOI: <https://doi.org/10.33842/2313-125X-2025-19-109-122>.
79. Understanding and Combating Memory Bloat in Managed. Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 4, University of California, Irvine : Athens Information Technology, 2018 p., Т. 26. <https://doi.org/10.1145/3162626>.
80. Lee, S. (2015). Mitigating the performance impact of memory bloat [Doctoral dissertation, Georgia Institute of Technology]. Georgia Tech Institutional Repository. <http://hdl.handle.net/1853/56174>.
81. Xia, Q., Ji, H., Zhou, Y., & Kim, N. S. (2025). Hardware-accelerated kernel-space memory compression using Intel QAT. IEEE Computer Architecture Letters, 24(1), 57–60. <https://doi.org/10.1109/LCA.2025.3534831>.
82. Tsai, P.-A., & Sánchez, D. (2019, April). Compress objects, not cache lines: An object-based compressed memory hierarchy. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming

- Languages and Operating Systems (ASPLOS 2019) (pp. 229–242). Association for Computing Machinery. <https://doi.org/10.1145/3297858.330400>.
83. Mitikov M.Y, Guk N.A., Voliansky R., Mozhaiev M., Pranolo A. Mathematical Modeling And Experimental Evaluation Of The Information Content Completeness At Fixed Memory Volumes: Application Of Compression Algorithms. 5th International Conference on Information Technologies: Theoretical and Applied Problems: тези доповідей V Міжнародної науково-практичної конференції (Тернопіль, 22-24 жовтня 2025р.) <https://ceur-ws.org/Vol-4146/short05.pdf> Scopus.
84. Мітіков М. Ю. Математична модель представлення інформації в пам'яті для аналізу продуктивності програмного забезпечення. Сучасні проблеми моделювання. 2025. № 28. С. 96–107. DOI: <https://doi.org/10.33842/2313-125X-2025-30-96-107>.
85. Мітіков М. Ю., Гук Н. А. Математична модель представлення інформації в пам'яті для аналізу продуктивності програмного забезпечення. Сучасні питання оптимізації та дискретної математики: тези доповідей наукового семінару при Науковій раді НАН України з проблеми «Кібернетика» (Дніпро, 15 жовтня 2024 р.). Дніпро, 2024.
86. Гук Н.А., Мітіков М.Ю. Математична модель оптимізації пошуку дублікатів об'єктів типа String у знімках пам'яті. Системні технології. 2024. № 6 (155). С. 236-249. DOI: <https://doi.org/10.34185/1562-9945-6-155-2024-23>
87. Microsoft. Windows Virtual Machines Pricing. Cloud Computing Services | Microsoft Azure. [Онлайновий] Microsoft . [Цитовано: 8 February 2024 p.] <https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/>.
88. Measuring the Cost of Regression Testing in Practice. Labuschagne, Adriaan, Laura Inozemtseva, Reid Holmes. New York: Association for Computing Machinery, 2017. 978-1-4503-5105-8.
89. Bentley, Jon Louis. Writing efficient programs. Pittsburgh, Pennsylvania : Prentice-hall, 1982. ISBN:978-0-13-970251-8.

90. Roy Osherove, Vladimir Khorikov. The Art of Unit Testing. місце видання невідоме : Manning, 2024. ISBN 9781617297489.
91. Pattipati D. K., Nasre R., Puligundla S. K. OPAL: An extensible framework for ontology-based program analysis. *Software: Practice and Experience*. 2020. Vol. 50, No. 8. P. 1425–1462. DOI: 10.1002/spe.2821
92. Weninger M., Gander E., Mössenböck H. Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance // *Proceedings 11th Symposium on Software Performance (SSP 2020)*. Leipzig, Germany, 2020. URL: [JKU ePUB Repository](#).
93. Kiseleva, E., Prytomanova, O., Padalko, V. (2021). An Algorithm for Constructing Additive and Multiplicative Voronoi Diagrams Under Uncertainty. In: Babichev, S., Lytvynenko, V., Wójcik, W., Vyshemyrskaya, S. (eds) *Lecture Notes in Computational Intelligence and Decision Making. ISDMCI 2020. Advances in Intelligent Systems and Computing*, vol 1246. Springer, Cham. https://doi.org/10.1007/978-3-030-54215-3_46978-0139702440.
94. Ioannis T. Christou, Sofoklis Efremidis. To Pool or Not To Pool? Revisiting an Old Pattern. Marousi : Athens Information Technology, 2018. arXiv:1801.03763.
95. Immutable Objects for a Java-Like Language. C. Haack, E. Poll, J. Schäfer, A. Schubert. Berlin : Springer, 2007. 978-3-540-71316-6.
96. MRm-DLDet: a memory-resident malware detection framework based on memory forensics and deep neural network. Liu, J., Feng, Y., Liu, X. 21,; *Cybersecurity*, 2023 p., T. 6. 10.1186/s42400-023-00157-w.
97. Moroz B., Kabak L., Varekh N., Moroz D. Text document classification system with Big Data technologies usage. *Information Technology: Computer Science, Software Engineering and Cyber Security*. 2023. No. 2. P. 34–40. DOI: <https://doi.org/10.32782/IT/2023-2-4>.
98. Mitikov M.Y., Guk N.A. Enhancing collection performance in software through memory snapshot analysis and mathematical modeling. *Автоматика-2024: тези*

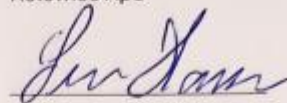
доповідей XXVII Міжнародної конференції (Дніпро, 20–22 листопада 2024 р.). Дніпро, 2024. С. 114–115.

99. Мітіков М. Ю., Гук Н. А. Математична модель оцінки операційної вартості використання хеш-колекцій програмного забезпечення за допомогою знімків пам'яті. Наука і сталий розвиток транспорту – 2024: збірник тез доповідей Всеукраїнської науково-технічної конференції студентів і молодих учених (Дніпро, 27 листопада 2024 р.). Дніпро: УДУНТ, 2024. С. 81–82.
100. 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>
101. Akinshin, D. [AndreyAkinshin]. (2024). Releases dotnet/BenchmarkDotNet. GitHub. <https://github.com/dotnet/BenchmarkDotNet/releases>
102. Guk N.A, Mitikov M.Y, Selivyorstova T.V. Detecting extraordinary application memory use by analyzing memory screenshots. Наука і техніка сьогодні (Science and Technology Today). 2024. Вип. 10 (38). С. 39–49. DOI: [https://doi.org/10.52058/2786-6025-2024-10\(38\)-39-49](https://doi.org/10.52058/2786-6025-2024-10(38)-39-49)
103. Мітіков М.Ю., Гук Н.А. Архітектура експертної системи для аналізу знімків пам'яті Системні технології. Вип. 6. 2025. С. 199–211. DOI: <https://doi.org/10.34185/1562-9945-5-161-2025-19>
104. Гук Н.А., Єгошкін Д.І., Мітіков М.Ю., Долотов І.О. Універсальний підхід до побудови адаптивних експертних систем на основі зважених правил та патернового аналізу станів систем // Питання прикладної математики і математичного моделювання. – Дніпро, 2025. – № 3 (25). – С. 505–61. – <https://doi.org/10.15421/322505>

ДОДАТОК А. АКТ ВПРОВАДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ

APPROVED

Chief Operating Executive
Relewise ApS



Sune Hansen

2026

АКТ

on the implementation of the results of the research submitted to obtain the Doctor of Philosophy degree by Mitikov Mykola

"Identification of anomalies in software operation based on memory snapshot analysis"
into the software development process of the **Relewise** product.

1. Consideration of the research results

The Relewise engineering team reviewed the results of the scientific research conducted by Mykola Mitikov concerning the identification of anomalies in software systems through the analysis of memory snapshots and their potential application in the development and operation of large-scale software platforms.

The results presented were evaluated regarding their applicability for improving the reliability, performance diagnostics, and operational stability of the Relewise software product.

2. Brief description of the implemented results

Modern distributed software systems operate under conditions of high load, dynamic scaling, and complex interaction between numerous components. In such environments, early detection of anomalies in system behavior becomes a critical task for maintaining reliability and performance.

Within the framework of the research, methods and algorithms for analyzing runtime memory snapshots of software systems were developed. These methods allow the identification of abnormal patterns in memory structures, object distributions, and resource utilization.

The proposed approach enables automated detection of inefficient data structures, abnormal object duplication, excessive memory allocations, and other anomalies that negatively affect the performance and stability of software systems.

During the research, a set of analytical techniques for processing memory snapshot data was developed, including analysis of object graphs, identification of duplicated immutable objects, detection of inefficient collection usage patterns, and evaluation of memory utilization characteristics.

The application of the proposed methods in the development and diagnostic processes of the Relewise software platform contributes to improved observability of runtime behavior, facilitates faster identification of performance issues, and supports informed decision-making during software optimization.

3. Conclusion

The results of the research conducted by Mykola Mitikov are considered relevant and practically valuable for improving the methods of diagnosing and optimizing complex software systems.

The proposed methods and analytical approaches are recommended for use in the development, analysis, and optimization processes of the Relewise software product.

4. Information on implemented intellectual property results

1. Mitikov, M.Y., & Huk, N.A (2025). Improving the performance of collections in software using memory snapshot analysis and mathematical modeling. *Modern Problems of Modeling*, (27), 109–122. <https://doi.org/10.33842/2313-125X-2025-19-109-122>
2. Mitikov, M. Y., & Huk, N. A. (2024). Identification of anomalies in software operation based on memory snapshot analysis. *System Technologies*, 6(155). <https://journals.nmetau.edu.ua/index.php/st/article/view/1931>
3. Mitikov, M. Y., & Huk, N. A. (2025). Architecture of an expert system for memory snapshot analysis. *System Technologies*, 6(161). <https://doi.org/10.34185/1562-9945-5-161-2025-19>

ДОДАТОК Б. ВИБІР МОВИ ПРОГРАМУВАННЯ

Розроблене програмне забезпечення призначене для автоматизованого аналізу знімків пам'яті .NET-застосунків, виділення об'єктів керованої купи, групування їх за типами та значеннями, обчислення статистичних характеристик і формування фактів, що надалі використовуються в експертній системі. З огляду на це вибір мови програмування визначався не лише загальними можливостями реалізації алгоритмів, а й необхідністю безпосередньої роботи з внутрішніми структурами CLR та файлами знімків пам'яті.

Для реалізації програмного засобу було використано мову програмування C# та платформу .NET. Такий вибір обумовлений тим, що об'єктом дослідження є пам'ять програмних систем, які виконуються у середовищі CLR. Застосування мови C# дозволяє працювати з типами, об'єктами, полями, масивами, колекціями та іншими елементами керованого середовища у тій самій технологічній області, у якій функціонують досліджувані програмні додатки.

Основними вимогами до мови програмування та середовища реалізації були такі:

1. підтримка роботи з файлами знімків пам'яті процесів у форматі .dmp;
2. можливість доступу до структури керованої купи CLR, зокрема до об'єктів, типів, полів, потоків виконання та статичних значень;
3. можливість ітеративного обходу великої кількості об'єктів пам'яті без попереднього завантаження всіх результатів у проміжні структури;
4. підтримка узагальнених типів, колекцій, словників та засобів групування даних;
5. можливість реалізації окремих модулів аналізу для різних типових ситуацій надмірного використання пам'яті;
6. підтримка 64-бітного режиму виконання, необхідного для аналізу великих знімків пам'яті;

7. можливість формування результатів аналізу у структурованому вигляді для подальшої обробки;
8. наявність бібліотек для роботи з CLR-знімками пам'яті та об'єктами керованої купи.

Мова C# відповідає зазначеним вимогам, оскільки вона є основною мовою розробки для платформи .NET і має безпосередню сумісність із бібліотеками, що забезпечують доступ до CLR. У реалізації програмного засобу використовується бібліотека Microsoft.Diagnostics.Runtime, яка надає засоби для відкриття знімку пам'яті, створення об'єкта середовища виконання, доступу до керованої купи, потоків, модулів, типів та окремих об'єктів. Це дозволяє працювати зі знімком пам'яті не як із неструктурованою послідовністю байтів, а як із множиною об'єктів, що мають типи, адреси, розміри та значення полів.

Програмний засіб реалізовано як консольний застосунок .NET. Його структура передбачає наявність окремого компонента для встановлення з'єднання зі знімком пам'яті, а також набору аналізаторів, кожен з яких виконує окремий вид обробки даних. Компонент роботи зі знімком пам'яті відкриває файл .dmp, створює представлення CLR-середовища, надає доступ до керованої купи, потоків виконання та об'єктів за адресами. Це забезпечує можливість виконувати подальші операції аналізу у термінах об'єктної моделі CLR.

Аналізатори реалізовано у вигляді окремих класів, що мають спільний інтерфейс обробки контексту аналізу. Така структура дозволяє відокремити різні задачі дослідження пам'яті: побудову розподілу об'єктів за типами, пошук повторюваних рядкових значень, аналіз ефективності використання словників, аналіз масивів байтів, дослідження кешів, статичних полів та інших структур. Винесення цих задач в окремі класи забезпечує можливість розширення програмного засобу без зміни базової логіки доступу до знімку пам'яті.

Для побудови первинної статистики використовується обхід об'єктів керованої купи з групуванням за іменем типу та накопиченням сумарного обсягу пам'яті. Такий підхід відповідає етапу первинного аналізу розподілу пам'яті, описаному в розділі 3, і дозволяє визначати типи, що мають найбільший внесок у загальне використання оперативної пам'яті.

Для аналізу дублювання рядкових значень реалізовано обробку об'єктів типу `System.String`. Під час такої обробки об'єкти групуються за значенням рядка, після чого для кожного значення визначається кількість повторних екземплярів, адреса одного з представників і сумарний обсяг пам'яті, пов'язаний із цим значенням. Отримані дані використовуються для кількісного оцінювання дублювання незмінюваної інформації.

Для аналізу хеш-колекцій реалізовано окремі модулі, що працюють із внутрішніми структурами `Dictionary<K,V>`. Зокрема, аналізуються масиви `_entries` та `_buckets`, кількість фактично збережених елементів, резерв внутрішньої ємності, наявність допоміжних об'єктів, а також співвідношення між корисним інформаційним вмістом і фактичним обсягом пам'яті. Це відповідає задачі оцінювання вартості зберігання інформації у хеш-колекціях.

Для об'єктів типу `System.Byte[]` реалізовано побудову розподілів масивів за довжиною, кількістю екземплярів та сумарним обсягом пам'яті. Така обробка дозволяє відокремити масиви з малим внеском у пам'ять від масивів, що формують значний сумарний обсяг, і використовується під час відбору кандидатів для подальшого стискання.

Результати роботи аналізаторів зберігаються у структурованому форматі JSON. Це дозволяє використовувати отримані дані як проміжний шар між програмним аналізом знімку пам'яті та подальшою експертною інтерпретацією. Такий формат є придатним для повторної обробки, агрегації результатів, побудови таблиць, формування правил експертної системи та порівняння результатів між різними знімками пам'яті.

Платформа .NET також забезпечує можливість виконання програми у 64-бітному режимі. Це має значення для аналізу великих знімків пам'яті,

оскільки такі файли можуть містити десятки мільйонів об'єктів і потребувати значного адресного простору під час обробки. У реалізації програмного засобу цільову платформу задано як x64, що відповідає характеру досліджуваних даних.

Альтернативні мови програмування, зокрема Python, R або MATLAB, можуть бути зручними для числових розрахунків, статистичної обробки та візуалізації результатів, однак вони не забезпечують безпосереднього доступу до структури CLR-купи без додаткових проміжних засобів. Використання таких мов потребувало б окремого шару для вилучення даних зі знімку пам'яті, що збільшувало б складність програмної реалізації та відокремлювало б етап аналізу від середовища виконання досліджуваних об'єктів.

Мова C++ може використовуватися для низькорівневого аналізу пам'яті та роботи з нативними структурами, однак у задачі дослідження керованої пам'яті .NET її застосування не дає переваг порівняно з C# та CLRMD. У цьому випадку значна частина аналізу пов'язана не з побайтовою інтерпретацією довільних областей пам'яті, а з роботою з CLR-типами, об'єктами, полями, масивами та колекціями. Тому використання C# дозволяє зберегти відповідність між програмною реалізацією та предметом дослідження.

Отже, вибір мови C# і платформи .NET є узгодженим із предметом дисертаційного дослідження, оскільки забезпечує безпосередній доступ до керованої пам'яті CLR, підтримку бібліотеки Microsoft.Diagnostics.Runtime, можливість модульної реалізації аналізаторів і формування структурованих результатів. Така реалізація дозволяє виконувати аналіз знімків пам'яті, одержувати кількісні характеристики об'єктів та формувати факти, необхідні для подальшої роботи експертної системи.

ДОДАТОК В. СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА

Наукові праці, в яких опубліковані основні наукові результати дисертації:

1. Мітіков М.Ю., Гук Н.А. Огляд методів виявлення та аналізу проблем продуктивності в програмному забезпеченні: підходи, виклики та перспективи. *Питання прикладної математики і математичного моделювання*. 2023. Вип. 23. С. 171–178. DOI: <https://doi.org/10.15421/322318>
2. Гук Н.А., Мітіков М.Ю. Математична модель оптимізації пошуку дублікатів об'єктів типа String у знімках пам'яті. *Системні технології*. 2024. № 6 (155). С. 236-249. DOI: <https://doi.org/10.34185/1562-9945-6-155-2024-23>
3. Гук Н.А., Мітіков М.Ю. Сучасні проблеми ідентифікації аномалій в роботі Enterprise Systems. *Системні технології*. Вип. 5. 2024. С. 146–153 DOI: <https://doi.org/0.34185/1562-9945-5-154-2024-15>
4. Мітіков М.Ю., Гук Н.А. Підвищення продуктивності колекцій у програмному забезпеченні за допомогою аналізу знімків пам'яті та математичного моделювання. *Сучасні проблеми моделювання*. 2025. № 27. С. 109–122. DOI: <https://doi.org/10.33842/2313-125X-2025-19-109-122>
5. Мітіков М. Ю. Математична модель представлення інформації в пам'яті для аналізу продуктивності програмного забезпечення. *Сучасні проблеми моделювання*. 2025. № 28. С. 96–107. DOI: <https://doi.org/10.33842/2313-125X-2025-30-96-107>
6. Guk N.A, Mitikov M.Y, Selivyorstova T. Detecting extraordinary application memory use by analyzing memory screenshots. *Наука і техніка сьогодні (Science and Technology Today)*. 2024. Вип. 10 (38). С. 39–49. DOI: [https://doi.org/10.52058/2786-6025-2024-10\(38\)-39-49](https://doi.org/10.52058/2786-6025-2024-10(38)-39-49)
7. Мітіков М. Ю., Гук Н. А. Дослідження проблем швидкодії програмних додатків. *Математичне та комп'ютерне моделювання. Серія: Технічні науки*. 2024. Вип. 25. С. 22–36. DOI: <https://doi.org/10.32626/2308-5916.2024-25.22-36>
8. Мітіков М. Ю., Гук Н. А. Modeling and automation of the process for detecting duplicate objects in memory snapshots. *Herald of Advanced Information*

Technology. 2024. Vol. 7, No. 2. P. 147–157. DOI: <https://doi.org/10.15276/hait.07.2024.10>

9. Мітіков М.Ю., Гук Н.А. Архітектура експертної системи для аналізу знімків пам'яті *Системні технології*. Вип. 6. 2025. С. 199–211 <https://doi.org/10.34185/1562-9945-5-161-2025-19>

10. Гук Н.А., Єгошкін Д.І., Мітіков М.Ю., Долотов І.О. Універсальний підхід до побудови адаптивних експертних систем на основі зважених правил та патернового аналізу станів систем // *Питання прикладної математики і математичного моделювання*. – Дніпро, 2025. – № 3 (25). – С. 505–61. – <https://doi.org/10.15421/322505>

Наукові праці, які засвідчують апробацію матеріалів дисертації:

11. Mitikov M.Y., Guk N.A., Honcharova Y. Real-world example of application performance anomaly detection through memory analysis. *Modern scientific and technical research – 2023: матеріали II Всеукраїнської науково-практичної конференції молодих учених та студентів (Дніпро, 11 травня 2023 р.)*. Дніпро, 2023. С. 280–282.

12. Mitikov M.Y., Guk N.A. Review of methods and tools for system analysis of software performance. *Mathematical and Software of Intelligent Systems (MPIS-2023): тези доповідей XXI Міжнародної науково-практичної конференції (Дніпро, 22–24 листопада 2023 р.)*. Дніпро : ДНУ, 2023. С. 213–214.

13. Мітіков М. Ю., Гук Н. А. Інформаційна технологія діагностики надмірного використання пам'яті на основі аналізу знімків пам'яті. *Modern Information and Communication Technologies in Transport, Industry and Education: тези доповідей XVII Міжнародної науково-практичної конференції (Дніпро, 13–14 грудня 2023 р.)*. Дніпро, 2023. С. 32.

14. Мітіков М. Ю., Гук Н. А. Виявлення надлишкового використання пам'яті програмними додатками. *Інформаційні технології: теорія і практика: тези доповідей I (VII) Міжнародної науково-практичної конференції*

здобувачів вищої освіти і молодих учених (Дніпро, 20–22 березня 2024 р.). Дніпро: Свідлер А. Л., 2024. С. 107–109.

15. Mitikov M.Y., Guk N.A. Enhancing collection performance in software through memory snapshot analysis and mathematical modeling. *Автоматика-2024: тези доповідей XXVII Міжнародної конференції (Дніпро, 20–22 листопада 2024 р.)*. Дніпро, 2024. С. 114–115.

16. Мітіков М. Ю., Гук Н. А. Математична модель оцінки операційної вартості використання хеш-колекцій програмного забезпечення за допомогою знімків пам'яті. *Наука і сталий розвиток транспорту – 2024: збірник тез доповідей Всеукраїнської науково-технічної конференції студентів і молодих учених (Дніпро, 27 листопада 2024 р.)*. Дніпро: УДУНТ, 2024. С. 81–82.

17. Мітіков М. Ю. Оптимізація процесу пошуку сегментів пам'яті: математичне моделювання та експериментальна оцінка повноти інформаційного вмісту при фіксованих об'ємах пам'яті. *Інформаційні технології в металургії та машинобудуванні (ІТММ'2025): тези доповідей Міжнародної науково-практичної конференції (Дніпро, 23–24 квітня 2025 р.)*. Дніпро: УДУНТ, 2025. С. 676–680.

18. Mitikov M.Y., Guk N.A., Voliansky R., Mozhaiev M., Pranolo A. Mathematical Modeling And Experimental Evaluation Of The Information Content Completeness At Fixed Memory Volumes: Application Of Compression Algorithms. *5th International Conference on Information Technologies: Theoretical and Applied Problems: тези доповідей V Міжнародної науково-практичної конференції (Тернопіль, 22-24 жовтня 2025р.) Scopus*