УДК 519.6:004.4:004.942

ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ КОЛЕКЦІЙ У ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ ЗА ДОПОМОГОЮ АНАЛІЗУ ЗНІМКІВ ПАМ'ЯТІ ТА МАТЕМАТИЧНОГО МОДЕЛЮВАННЯ

Мітіков М.Ю., аспірант*,
mitikov.m22@fpm.dnu.edu.ua, ORCID ID: 0009-0002-1297-5676
Гук Н.А., доктор фіз.-мат. наук,
huk_n@fpm.dnu.edu.ua, ORCID ID: 0000-0001-7937-1039
Дніпровський національний університет імені Олеся Гончара
(м. Дніпро, Україна)

У сфері сучасної розробки програмного забезпечення базові бібліотеки мають вирішальне значення для створення розробниками ефективних і надійних систем. Ці бібліотеки, що служать основою архітектури програмного забезпечення, повинні забезпечувати стабільну продуктивність і сумісність у різних додатках. Однак жорсткі вимоги до стабільності інтерфейсу та передбачуваної поведінки можуть обмежувати можливості оптимізації цих основних компонентів, особливо в сценаріях використання, які спочатку не були пріоритетними під час проектування.

Однією з істотних проблем є неточна оцінка використання пам'яті в хеш-колекціях, що є загальною особливістю багатьох базових бібліотек. При використанні кешів або хеш-колекцій фактичний обсяг пам'яті може бути недооцінений, що призводить до неефективного використання ресурсів. Ця помилка не тільки впливає на управління пам'яттю, але й збільшує операційні витрати, пов'язані з роботою програмного забезпечення, особливо в середовищах, де ефективність використання ресурсів має першочергове значення.

Крім того, технічні накладні витрати хеш-колекцій також сприяють збільшенню цих витрат. Наприклад, хеш-таблиці та словники базуються на хеш-алгоритмах, які розподіляють ключі по серії сегментів. Щоб мінімізувати хеш-колізії – ситуації, коли кілька ключів відображаються на один і той самий сегмент – ці колекції часто виділяють більше пам'яті, ніж необхідно, створюючи буфер для підвищення продуктивності під час великого навантаження. Однак така стратегія призводить до недовикористання пам'яті та може спричинити значну неефективність, особливо під час роботи з великими наборами даних або коли пам'ять є дефіцитною.

Сукупний ефект неточних оцінок пам'яті та властивих хеш-колекціям накладних витрат може призвести до неоптимальної продуктивності програмного забезпечення. У критичних додатках це може призвести до збільшення операційних витрат і зниження

* Науковий керівник – д-р фіз.-мат. наук, професор Гук Н.А.

ефективності системи. Вирішення цих проблем вимагає глибшого розуміння технічних витрат, пов'язаних з хеш-колекціями, та розробки нових методологій для точної оцінки використання пам'яті в цих контекстах.

Це дослідження має на меті дослідити оптимізацію хеш-колекцій, зосередившись на точній оцінці використання пам'яті та зменшенні технічних накладних витрат. Шляхом розробки вдосконалених моделей оцінки пам'яті та визначення стратегій для зменшення витрат, пов'язаних із хеш-структурами даних, ця робота має на меті підвищити загальну ефективність програмних систем, забезпечуючи їх конкурентоспроможність та ефективне використання ресурсів у вимогливому технологічному середовищі.

Ключові слова: продуктивність, колекції, хеш-колекції, математична модель, знімок пам'яті, оптимізація, рекомендаційна система.

Постановка проблеми. У контексті сучасної розробки програмного забезпечення базові бібліотеки відіграють ключову роль, забезпечуючи розробників необхідними засобами для створення ефективних рішень. Ці бібліотеки, як фундаментальні елементи програмних систем, мають забезпечувати надійність [1] та сумісність у широкому спектрі застосувань. Проте жорсткі вимоги до незмінності інтерфейсів та поведінки базових компонентів можуть обмежувати можливості їх оптимізації, особливо у контексті шаблонів використання, які не були передбачені під час первісного проектування як пріоритетні.

Наявність таких обмежень вимагає від розробників прийняття компромісних рішень на початкових етапах проектування між швидкодією, обсягом функціоналу та сумісністю, що може призводити до підвищення витрат на ресурси обчислювальних систем. В результаті, нове програмне забезпечення, створене на базі стандартних бібліотек, часто виконує задані функції з вищими операційними витратами, що знижує його конкурентоспроможність. З огляду на значну роль ефективності ПЗ у сучасних економічних та технологічних реаліях, питання оптимізації базових колекцій вимагає детального вивчення та розробки нових підходів.

Цей виклик актуалізує потребу у розробці нових методологій та математичних моделей для оцінки та адаптації базових колекцій до змінних умов використання, що може забезпечити краще використання обчислювальних ресурсів і зниження загальних витрат на інфраструктуру. Здатність аналізувати та оптимізувати роботу базових бібліотек у контексті специфічних задач є важливим аспектом підтримання високого рівня продуктивності програмного забезпечення, що безпосередньо впливає на його стабільність та надійність.

Аналіз останніх досліджень і публікацій. Історична реалізація хеш колекції *Hashtable* надає інтерфейс для роботи з об'єктами до появи узагальнень (*Generics*), що вимагає операцій пакування при зберіганні чисел у якості ключа. Операція пакування (*boxing*) - це процес

перетворення значення структури (наприклад, *int*) у тип об'єкта, що додає накладні витрати на продуктивність, оскільки вимагає додаткових операцій копіювання даних та роботи з пам'яттю. Аналогічно, при зберіганні структур у якості значень також відбувається пакування, що значно знижує продуктивність при роботі з великими обсягами даних.

З появою узагальнень (*Generics*) стало можливим створення типобезпечних колекцій, таких як *Dictionary<TKey, TValue>*, які дозволяють зберігати значення без пакування. Узагальнення (*Generics*) - це механізм, який дозволяє визначати класи, методи та структури з використанням параметрів типу, що забезпечує більшу безпеку та продуктивність. Завдяки узагальненням, *Dictionary<TKey, TValue>* може зберігати значення будь-якого типу без необхідності перетворення їх у тип об'єкта, що усуває накладні витрати, пов'язані з пакуванням.

Обидві колекції, *Hashtable* та *Dictionary<TKey, TValue>*, базуються на хеш-алгоритмі, який використовує хеш-код ключа для доступу до кошика (*bucket*), де потенційно може знаходитися значення. Структура даних обох колекцій включає два масиви. Для розв'язання хеш-колізій, які виникають коли два різних ключі мають однаковий хеш-код, обидві колекції використовують направлений список (*linked list*). Хеш-колізія - це ситуація, коли хеш-функція повертає однакове значення для двох різних ключів. Внаслідок цього швидкодія операцій доступу до елементів може зменшуватися до лінійної складності.

Для забезпечення ефективності додавання елементів та мінімізації хеш-колізій, колекції створюють масиви для більшої кількості елементів, ніж потрібно на поточний момент. Це дозволяє зменшити кількість хеш-колізій, але водночас призводить до додаткових витрат пам'яті. Таким чином, для збереження декількох пар значень додатково створюються масиви з розмірами більше за необхідне. Така специфіка роботи додає додаткові витрати на пам'ять і впливає на загальну продуктивність системи зменшуючи її ефективність.

В результаті літературного дослідження не було знайдено матеріалів, які б досліджували або оцінювали вартість зберігання даних у реалізаціях хеш-колекцій у базових бібліотеках. Це вказує на потребу у подальшому дослідженні даної тематики для розробки більш ефективних методів управління пам'яттю у хеш-колекціях.

Формулювання цілей статті. Метою роботи є розробка методики оцінки операційної вартості використання хеш-колекцій у програмному забезпеченні за допомогою знімків пам’яті. Для цього необхідно буде розробити математичну модель оцінки вартості зберігання одиниці інформації для примітивних типів за допомогою знімків пам'яті та виявлення надмірного споживання пам'яті.

Основна частина.

Модель представлення знань про предметну область. Знімок пам'яті програми може бути зображеним у вигляді структурованого масиву

байтів в такий спосіб: $M_{time}=[b_1,b_2,...,b_N]$, де *time* – момент часу, у який відбувається зняття знімку пам'яті; $b_1,b_2,...,b_n$ – байти, з яких складається знімок; N – кількість байтів.

В середині знімку пам'яті відображається інформація про операції, що виконуються, стан потоків виконання, об'єкти, якими ці потоки керують. Відомості про об'єкти вміщують їх типи, стани та взаємозв'язки.

Введемо поняття середовища виконання Common Language Runtime (CLR), об'єкту *o* та типу *t*. CLR керує виконанням програм та забезпечує керування пам'яттю, безпекою типів, обробкою виняткових ситуацій (помилок). Тип (*t*) – це категорія даних, що визначає розмір, діапазон та операції над значеннями. У C# кожна змінна, константа, вираз, параметр та значення, що повертається, має тип. Типи можуть бути попередньо визначеними (наприклад, String, Integer тощо), або визначеними користувачем. Об'єктом *o* є екземпляр типу *t*, під який виділяється пам'ять в діапазоні пам'яті. Об'єкт *o* можна ідентифікувати та маніпулювати ним за допомогою посилання на його адресу (*a*).

На рис.1 зображено приклад об'єкту колекції o_x в знімку пам'яті, в рамці 1 позначено адресу $a \in [b_1, b_2, \dots b_N]$, за якою об'єкт o_x знаходиться у пам'яті, $a(o_x) = 00000203f2000928$; в рамці 2 позначено тип $t(o_x) = Dictionary < int, IndexedProduct >$ об'єкту *o*, що характеризується адресою, за якою визначений тип; лінією (3) позначено розмір (*s)* об'єкту *o в* байтах, *s(o)* = 80 байтів; поля *fl* об'єкту *o* визначаються типом *t* і містять поля для збереження пар ключ-значення. Таким чином, екземпляр об'єкту задається в вигляді $o_m=\langle a, t, s, fl\rangle$.

```
0:000> !do 00000203f2000928 1
Name:        System.Collections.Generic.Dictionary`2[[System.Int32, System.Private.CoreLib],[Model.Search.IndexedProduct,Engine]]
MethodTable: 00007ffae756b270
EEClass:     00007ffae5a4d838 2
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 00000203f2000978 _buckets
00007ffae615bb30  4002114       10 ...ivate.CoreLib]][]  0 instance 00000203f20009a0 _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 00000206fe93aec0 _values
```

Рис. 1. Приклад об'єкту у знімку пам'яті

Введемо поняття розміру одиниці інформації для зберігання s_{inf} для пари ключ-значення, яка визначається як сума розмірів об'єкту ключа та об'єкту значення: $s(k,v)_{inf} = s(k) + s(v)$.

Таким чином, загальна кількість пам'яті необхідної для зберігання N пар ключ-значення дорівнює математичній сумі розмірів окремих пар:

$$s([k,v]_N)_{inf} = \sum_{1}^{N} s(k,v)_{inf_N}.$$

Хеш-колекції мають реалізовувати пошук за хеш сумою об’єкта, логіку рішення колізій, додавання та видалення значень, надавати збалансовану швидкодію і прогнозоване використання пам’яті. Усі ці властивості потребують збереження додаткових метаданих. Позначимо реальний розмір, який займає реалізація хеш колекції з N елементами як $s([k,v]_N)_{fact}$ яка визначається як сума фактичних даних, та розміру допоміжних метаданих які забезпечують функціювання колекції: $s([k,v]_N)_{fact} = s([k,v]_N)_{inf} + s(hashImpl_N)$.

Для оцінки вартості зберігання інформації будемо застосовувати відношення: $StorageCost_N = \frac{s([k,v]_N)_{inf}}{s([k,v]_N)_{fact}}$.

Обсяг інформації яка міститься в одній парі ключ/значення на рис.1 є $s(k,v)_{inf} = s(k_{int}) + s(v_{obj}) = 4 + 8 = 12$ байт. Розмір об’єкту словник (без урахування вкладених полів) 80 байт, що є більше ніж в 6 разів від інформації яка зберігається. Програмні реалізації, які використовують невірну оцінку фактичного розміру, мають тільки відсоток від необхідних даних в підсистемі кешування замість очікуваної кількості при проектуванні.

Методика аналізу знімків пам’яті. Маємо множину об'єктів O_{time} *(*містить усі об’єкти *o)* та множину типів T_{time} *(*усі типи *t)*, які відображені у знімку пам’яті знятому у момент часу *time*.

Враховуючи, що кожен об’єкт *o* є екземпляром типу *t*, введемо функцію групування *G,* яка групує елементи вхідної множини об'єктів *O* за типами: $G(O): O \to T$.

В результаті операції групування отримано підмножини O_t, які містять усі об’єкти типу *t.* У рамках дослідження розглянемо реалізацію хеш колекції у середовищі .NET для ключів та значень типу число (Integer) : *Dictionary<int, object>*.

Оскільки кожен екземпляр об’єкту містить набір полів *fl(o),* будемо рахувати фактичний розмір об’єкта як суму розмірів усіх його полів без урахування розміру збережених значень. Це дозволить виявити фактичний обсяг пам’яті який необхідний для збереження даних.

На рисунку 2 представлені поля екземпляру об’єкту «словник», та розмір цього зокрема об’єкту – 80 байтів.

Рис. 2. Перелік полів у об’єкті типу словник

- *_buckets* – посилання на масив індексів для знаходження необхідного «кошику» в якому може знаходитись значення, займає 8 байт
- *_entries* – посилання на масив структур, які містять ключ, хеш-код, значення, індекс наступного елемента у випадку хеш-колізії, займає 8 байт
- *_fastModMultiplier* – множник для пошуку номера «кошика», 8 байтів.
- *_count* – кількість елементів яку містить колекція, 4 байти.
- *_freeList* – індекс першого вільного «кошика» для збереження значення, 4 байти, 4 байти.
- *_freeCount* –кількість «кошиків», які звільнились в результаті видалення даних з колекції, 4 байти.
- *_version* – версія колекції для запобігання одночасних операцій читання та запису, 4 байти.
- *_comparer* - посилання на довільну логіку розрахунку хеш-коду для ключа, 8 байтів.
- *_keys* - посилання на колекцію ключів, 8 байтів.
- *_values* - посилання на колекцію значень, 8 байтів.

Враховуючи необхідність збереження логіки додавання нових елементів, масиви *_buckets* та *_entries* створюються з урахуванням додаткового буфера. На рисунку 3 показано, що розміри цих масивів становлять відповідно 36 та 96 байтів.

```
0:000> !DumpObj /d 00000203f2000978
Name:        System.Int32[]
MethodTable: 00007ffae51ebf38
EEClass:     00007ffae51ebeb8
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. Розмір допоміжних об’єктів всередині словника

Також словник містить створену колекцію значень, яка займає 24 байти, $s_{obj_{min}}$ (мінімальний розмір об’єкта в середовищі CLR) додатково.

Таким чином, фактичний розмір колекції для збереження однієї пари значень $s([k,v]_1)_{fact} = 80 + 36 + 96 + 24 = 236$ байтів.

Вартість зберігання інформації для колекції з одним елементом складає $\text{StorageCost}_1 = \frac{s([k,v]_1)_{inf}}{s([k,v]_1)_{fact}} = \frac{12}{236} = 0.05$.

Введемо функцію групування колекцій *GroupByCount(U)*, що розбиває множину хеш колекцій на підмножини за кількістю збережених елементів.

Введемо функцію *Count(U),* що підраховує кількість елементів в довільній множині *U*, і використаємо її для знаходження кількості елементів $GroupCount$:

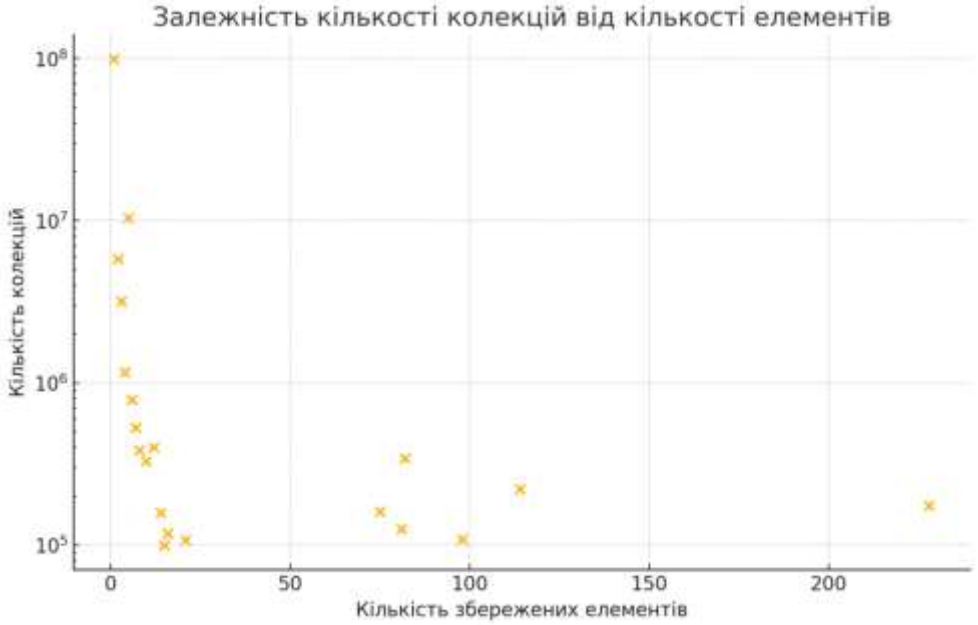

Рис. 4. Залежність кількості колекцій від кількості елементів

На рис.4 наведено розподіл колекцій за фактичною кількістю елементів які зберігаються в досліджуваній промисловій системі. За наведеним графіком відслідковується домінування колекцій з малою кількості елементів. У рамках цієї роботи розглянемо колекцію з однією парою ключ-значення. Завдяки наявній функції групування, розрахуємо загальну кількість пам'яті яку необхідно витрати для зберігання даних в колекції з одним елементом:

$$\mathrm{TechStorageCost}_1 = \mathrm{N} * (s([k,v]_1)_{fact} - s([k,v]_1)_{inf}).$$

Критерієм витоку пам'яті [2] є монотонне зростання кількості елементів у послідовно знятих знімках пам'яті, тож для типу t_{leak}, який спричиняє виток пам'яті, виконується умова: $Count\left(O_{t_{before}}\right) < Count\left(O_{t_{after}}\right)$, де t_{before} та t_{after} моменти часу належать часу функціювання програмного додатку, та *before* < *after*.

В досліджуваному випадку критерій витоку пам'яті не виконується, тож збільшене використання пам'яті (приблизно у 20 разів) пройде непоміченим повз існуючи системи моніторингу [3].

Для більш ефективного збереження даних [4] у пам'яті запропоновано реалізацію:

- Для колекцій, які не потребують власної хеш-функції, використати реалізацію без поля *comparer,* економія *3% або 8 байт.* У досліджуваній

системі жоден словник з 126 мільйонів екземплярів не використовує цей функціонал. Сумарний розмір полів складає близько 1 ГБ пам'яті.

- Динамічно створювати колекції ключів та значень замість відокремлення *_keys* та *_values* полів, економія *17% або 40 байт.* У досліджуваній системі 72.6 мільйона об'єктів мають цю колекцію, займаючи 2.8 ГБ пам'яті.
- Використання сталого значення *_ fastModMultiplier* для моделі пам'яті x64, економія *3% або 8 байт*. Сумарний розмір полів складає близько 1 ГБ пам'яті.
- Поєднання *_entries* та *_buckets* в один масив для створення лише одного екземпляру масиву заміть двох. Заміна збереження хеш-коду на індекс можлива для словників з ключем «число» через властивість хеш код числа є самим числом, економія *18.6% або 44 байт*
- Створення здатності зменшувати розмір до реальної кількості елементів, економія *20% або 48 байт*
- Відмова від перевірки цілісності при багато-потоковому доступі контролю, відмова від поля _version, *1.5 % або 4 байт*

Таким чином, реалізація з урахуванням оптимізацій $s'([k,v]_1)_{fact}$ дозволяє витратити на 152 байти (63 %) менше пам'яті ніж стандартна колекція, підвищуючи ефективність зберігання:

$$\text{StorageCost}_1{}' = \frac{s'([k,v]_1)_{inf}}{s'([k,v]_1)_{fact}} = \frac{12}{A(236-152)} = 0.136,$$

де *A(x)* – функція вирівнювання розміру об'єкта відповідно розрядності вказівника (8 байт).

Використовуючи наведені розрахунки, надамо кількісну оцінку покращення для промислової системи яка містить 99172580 об'єктів типу словник з одним елементом:

$$\begin{cases} TechStorageCost_1 = \text{N} * (s([k,v]_1)_{fact} - s([k,v]_1)_{inf}); \\ \text{TechStorageCost}_1^{`} = \text{N} * (s'([k,v]_1)_{fact} - s([k,v]_1)_{inf}); \\ StorageOptimization = TechStorageCost_1 - \text{TechStorageCost}_1^{`}. \end{cases}$$

Використовуючи наведені розрахунки, надамо кількісну оцінку покращення для промислової системи яка містить 99 мільйонів об'єктів (або 78.7%) типу словник з одним елементом:

$$StorageOptimization'_{1_{readonly}} =$$

$$N * (s([k,v]_1)_{fact} - s'([k,v]_1)_{fact}) = N * (236 - 88) = 148\,N = \quad 13.67 \text{ GB}.$$

Для колекцій які зберігаються в пам'яті у якості «незмінних», є можливість використання об'єкту з парою ключ/значення замість масиву *_entries.* Враховуючи стан незмінності, решта полів *_count, _freeList, _freeCount* втрачають необхідність. Це дозволяє надалі зменшити розмір колекції:

$$s'_{readonly}([k,v]_1)_{fact} = \text{A}\left(s'([k,v]_1)_{inf} + s_{obj_{min}}\right) = A(12 + 24 - 8) =$$

$$= A(28) = 32.$$

де $s_{obj_{min}}$ – мінімальний розмір об’єкта в CLR, дорівнює 24 байти, з яких 8 зарезервовано під дані об’єкту.

Така форма збереження однієї пари значень має максимальну ефективність зберігання:

$$\text{StorageCost}_1{}'_{readonly} = \frac{s([k,v]_1)_{inf}}{s'_{readonly}([k,v]_1)_{fact}} = \frac{12}{32} = 0.375.$$

Використовуючи наведені розрахунки, надамо кількісну оцінку покращення для промислової системи яка містить 99172580 об’єктів типу словник з одним елементом:

$$\begin{cases} TechStorageCost_1 = \text{N} * (s([k,v]_1)_{fact} - s([k,v]_1)_{inf}); \\ \text{TechStorageCost}`_{1\text{readonly}} = \text{N} * (s'_{readonly}([k,v]_1)_{fact} - s([k,v]_1)_{inf}); \\ StorageOptimization = TechStorageCost_1 - \text{TechStorageCost}`_{1\text{readonly}}. \end{cases}$$

Підставляючи значення розміру стандартного словника і оптимізованого отримаємо оцінку:

$StorageOptimization'_{1_{readonly}} =$
$N * (s([k,v]_1)_{fact} - s'_{readonly}([k,v]_1)_{fact}) = N * (236 - 32) = 204\,N =$ 18.84 ГБ.

Досліджувана промислова система використовує 115 ГБ оперативної пам’яті. Згідно таблиці 1, 21.8 ГБ використовується для представлення 99 мільйонів словників, які містять лише один елемент всередині. Інформація збережена у цих колекціях потребує лише 1.11 ГБ пам’яті.

Оптимізація для одного елемента дозволяє зменшити споживання пам’яті на 18.8 ГБ, але потребує чіткого розділення операцій наповнення словнику, та його використання тільки для читання[5]. Для випадків коли цього чіткого розділення не існує, оптимізована версія словника для ключ типу «число» дозволяє зберегти 13.67 ГБ за рахунок відмови від частини функціоналу і використання властивості хеш коду для числа рівного самому числу.

При наявності чіткого розподілення при побудові даних (наприклад при завантажені інформації в додаток), збереження значень напряму в об’єкт (замість використання масиву) було досліджено до 7 елементів включно.

Згідно рис. 4, в пам’яті системи знаходиться значно більше колекцій з кількістю елементів до 8. Схожий розподіл було ідентифіковано в 95% досліджуваних систем. Для зменшення обсягів пам’яті, що використовується, запропоновану реалізацію словників з фіксованою кількістю елементів. Це дозволяє відійти від використання масивів всередині колекції і значно зменшити вартість зберігання інформації.

Таблиця 1

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	Різниця відносно базового	Розмір даних	Додатковий технічний розмір	Оптимізація відносно базового
Інформаційна цінність	12	0	0	1,11	0,00	0,00
Стандартний словник	236	224	0	21,80	20,69	0,00
Оптимізований для чисел	88	76	148	8,13	7,02	13,67
Оптимізований для одного значення	32	20	204	2,96	1,85	18,84

Розроблена методика виявлення колекцій зі знімку пам’яті була реалізована у вигляді програмного додатку за допомогою .NET та бібліотеки ClrMD.

Аналіз результатів. Для аналізу ефективності запропонованого підходу було проведено серії експериментів за допомогою бібліотеки Benchmark.NET [7].

Згідно рис. 5, реалізація словника за замовченням (stock) створюється у 8 разів довше ніж запропонований варіант для випадку колекції з одним елементом.

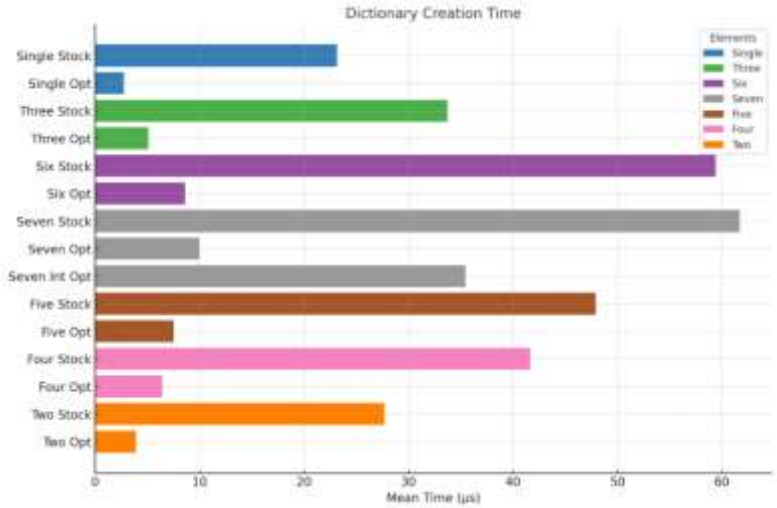

Рис. 5. Час на створення екземпляру колекції

Реалізація за замовченням з сьома елементами потребує у 6 разів більше часу ніж оптимізований варіант з фіксованою кількістю елементів і вдвічі більше ніж оптимізована версія з довільною кількістю елементів. Згідно рис.6 створення словника за замовченням для збереження одного елемента потребує у 7.5 разів більше пам’яті ніж запропонований варіант. Для зберігання семи елементів, використовується в 3 рази менше пам’яті.

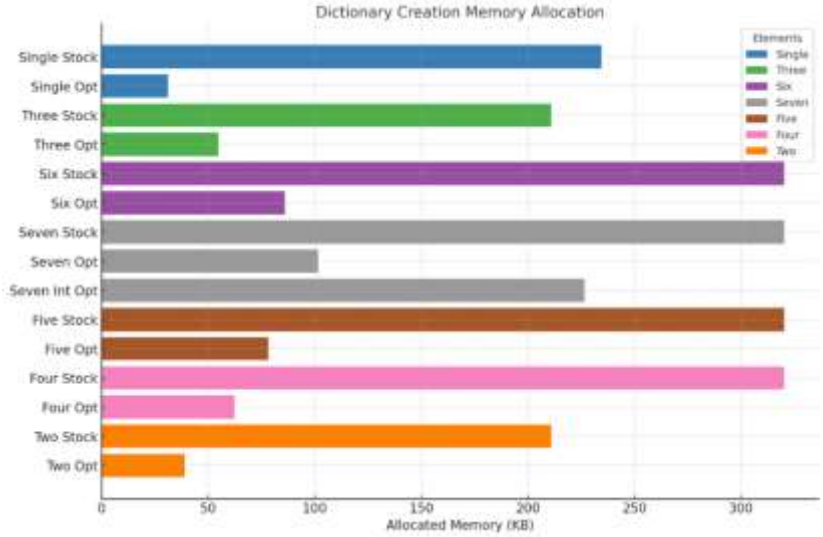

Рис. 6. Використання пам’яті для створення екземпляру колекції

Значного пришвидшення отримали операції пошуку елементу у колекції. Згідно рис.7 пошук існуючого елемента у колекції з одним елементом відбувається швидше у 17 разів. У 15.5 разів швидше відбувається пошук існуючого елемента у колекції з шести елементів. Для колекції з семи елементів швидкість збільшена у два рази.

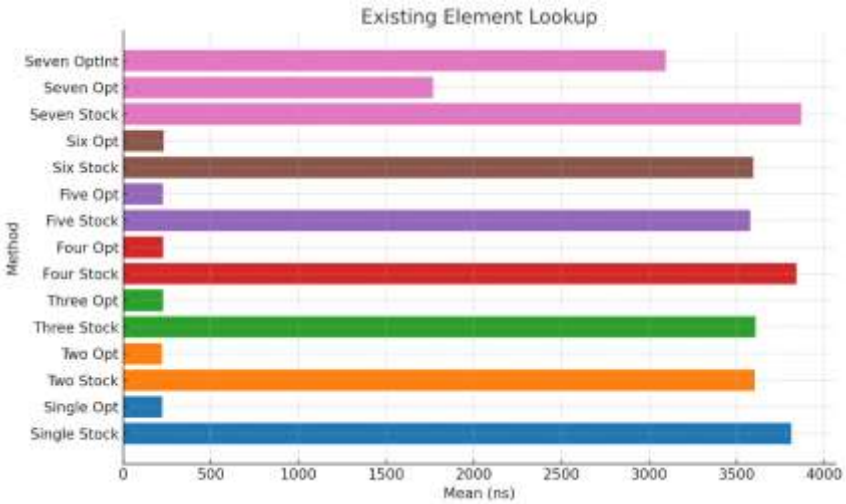

Рис. 7. Час на пошук існуючого елемента в колекції

Згідно рис.8 схожі пришвидшення отримано при пошуку неіснуючого елемента в колекції.

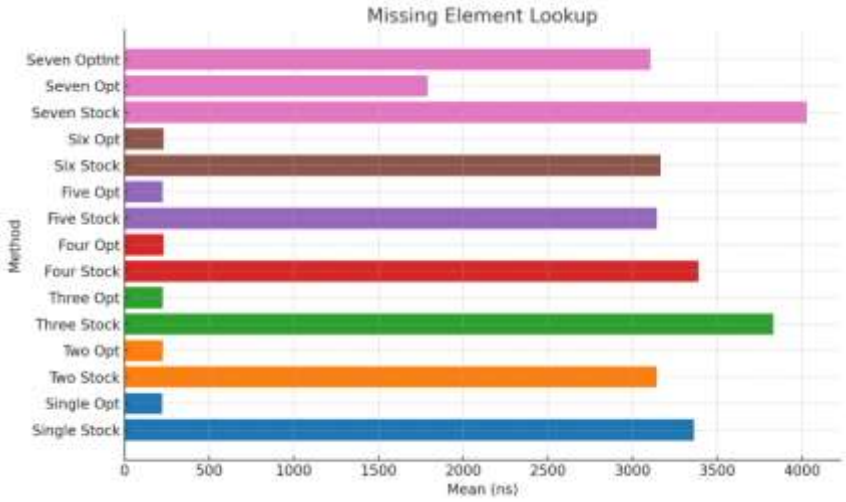

Рис. 8. Час на пошук неіснуючого елемента в колекції

Висновки. В роботі здійснено аналіз використання пам’яті для збереження інформації в процесі роботи програмного додатку. Створено математичну модель для оцінки вартості зберігання інформації, проаналізовано існуючи реалізації хеш колекцій. На основі знімків пам’яті [8] надано оцінку неоптимального використання пам’яті для кожної досліджуваної промислової системи. Запропоновано реалізації колекцій для невеликої кількості елементів, доведено значне покращення швидкодії в розрахункових експериментах. Великий приріст швидкодії зумовлено покращенням “data locality”, а саме відмови від використання екземплярів масивів, які можуть зберігатись в інших частинах оперативної пам’яті, та виконанням «дешевих» операцій порівняння чисел на невеликих обсягах даних. Згідно обчислювального експерименту, при збільшенні кількості елементів у колекції з 6 до 7, кількість часу для пошуку елементів збільшилась у декілька разів. Хеш алгоритми дозволяють пришвидшити пошук при збільшені кількості елементів всередині. Враховуючи знайдену особливість існування великої кількості екземплярів з кількістю елементів до восьми, запропонований підхід було використано в промислових системах для значного зменшення кількості об’єктів та зменшеного використання пам’яті. Завдяки впровадженню наданих рекомендацій, досліджувані промислові системи зменшили витрати пам’яті у середньому на 16 % без зменшення кількості функціоналу.

Література

1. Dodson, Thomas J. Veasey and Stephen J. Anomaly Detection in Application Performance. *International Journal of Machine Learning and Computing*, 2014, Т. 4. Режим доступу: http://www.ijmlc.org/papers/398-LC018.pdf.
2. Gene Novark, Emery D. Berger, Benjamin G. Zorn. Efficiently and precisely

locating memory leaks and bloat. New York : Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009. DOI: 10.1145/1542476.1542521.
3. Microsoft. Smart detection in Application Insights. (Дата звернення 25 04 2023 р.) Режим доступу: https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/proactive-diagnostics.
4. Bentley, Jon Louis. Writing efficient programs. Pittsburgh, Pennsylvania : Prentince-hall, 1982. ISBN:978-0-13-970251-8.
5. Gregg, Brendan. Systems Performance, Enterprise and the Cloud, second edition. 2021. ISBN-13: 978-0-13-682015-4.
6. Code specialization for memory efficient hash tries GPCE 2014: Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, Pages 11 – 14, https://doi.org/10.1145/2658761.26587633.
7. Selecting C# Profiling tools. Osipov, Aleksandr. 2019 p. Режим доступу: https://urn.fi/URN:NBN:fi:amk-2019061016294.
8. ADVANCED .NET DEBUGGING. Hewardt, Mario.: Pearson Education, Inc., 2010 p. ISBN 978-0-321-57889-1.

ENHANCING COLLECTION PERFORMANCE IN SOFTWARE THROUGH MEMORY SNAPSHOT ANALYSIS AND MATHEMATICAL MODELING

Nikolay Mitikov, Natalia Guk

In the realm of modern software development, foundational libraries are critical for enabling developers to build efficient and reliable systems. These libraries, serving as the bedrock of software architecture, must deliver consistent performance and compatibility across diverse applications. However, the rigid requirements for interface stability and predictable behavior can restrict opportunities for optimizing these core components, particularly in usage scenarios that were not initially prioritized during design.

One significant challenge arises from the inaccurate estimation of memory usage within hash collections, a common feature in many foundational libraries. When caches or hash-based collections are employed, the actual memory footprint can be underestimated, leading to inefficient use of resources. This misjudgment not only impacts memory management but also inflates the operational costs associated with running software, particularly in environments where resource efficiency is paramount.

Additionally, the technical overhead of hash collections contributes to these costs. Hash tables and dictionaries, for example, rely on hash algorithms that distribute keys across a series of buckets. To minimize hash collisions—situations where multiple keys map to the same bucket—these collections often allocate more memory than immediately necessary, creating a buffer to enhance

performance under heavy load. However, this strategy results in underutilized memory and can lead to significant inefficiencies, particularly when dealing with large datasets or when memory is at a premium.

The compounded effect of inaccurate memory estimates and the inherent overhead of hash collections can lead to suboptimal software performance. In critical applications, this can translate into increased operational expenses and reduced system efficiency. Addressing these issues requires a deeper understanding of the technical costs associated with hash collections and the development of new methodologies for accurately estimating memory usage in these contexts.

This research aims to explore the optimization of hash collections by focusing on the accurate assessment of memory usage and the reduction of technical overhead. By developing improved models for memory estimation and identifying strategies to mitigate the costs associated with hash-based data structures, this work seeks to enhance the overall efficiency of software systems, ensuring that they remain competitive and resource-efficient in a demanding technological landscape.

Keywords: performance, collections, hash collections, mathematical modelling, memory snapshot, optimization, recommendation system.

Referenses

1. Dodson, Thomas J. Veasey and Stephen J. 2 (2014) Anomaly Detection in Application Performance. *International Journal of Machine Learning and Computing*, Vol. 4. Retrieved from: http://www.ijmlc.org/papers/398-LC018.pdf.
2. Gene Novark, Emery D. Berger, Benjamin G. Zorn. (2009) Efficiently and precisely locating memory leaks and bloat. New York : Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. Retrieved from: https://doi.org/10.1145/1542476.1542521.
3. Microsoft. Smart detection in Application Insights. (2023). Retrieved from https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/proactive-diagnostics.
4. Bentley, Jon Louis. (1982) Writing efficient programs. Pittsburgh, Pennsylvania : Prentince-hall, 1982. ISBN:978-0-13-970251-8.
5. Gregg, Brendan. (2021) Systems Performance, Enterprise and the Cloud, second edition. ISBN-13: 978-0-13-682015-4.
6. Code specialization for memory efficient hash tries. (2014) GPCE 2014: Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences, Pages 11 – 14, Retrieved from: https://doi.org/10.1145/2658761.26587633.
7. Osipov, Aleksandr. (2019) Selecting C# Profiling tools. Retrieved from: https://urn.fi/URN:NBN:fi:amk-2019061016294.
8. ADVANCED .NET DEBUGGING. (2010) Hewardt, Mario.: Pearson Education, Inc. ISBN 978-0-321-57889-1.