DOI 10.34185/1562-9945-6-155-2024-23
УДК. 004.41:519.85

Н.А. Гук, М.Ю. Мітіков

МАТЕМАТИЧНА МОДЕЛЬ ОПТИМІЗАЦІЇ ПОШУКУ ДУБЛІКАТІВ ОБ'ЄКТІВ ТИПА STRING У ЗНІМКАХ ПАМ'ЯТІ

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

© Гук Н.А., Мітіков М.Ю., 2024

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

1. Вступ. Розвиток інформаційних технологій поширив всебічне застосування програмних додатків в багатьох сферах сьогодення. Необхідність оброблювати все більші обсяги інформації може компенсуватись збільшенням розрахункової потужності [1] або оптимізацією існуючих програмних реалізацій [2].

Враховуючи лінійне зростання вартості одиниці оперативної пам'яті у хмарному хостингу [3], складається хибне враження можливості лінійного масштабування сукупних експлуатаційних витрат. Існуючі пропозиції віртуальних машин пропонуються з розмірами оперативної пам'яті кратними ступеню двійки (4ГБ – 8ГБ – 16ГБ – 32ГБ -..). При потребі одного додаткового гігабайта понад наявної пам'яті виникає необхідність масштабування на наступну пропозицію яка мінімум у 2 рази дорожче за попередню (з урахуванням кількості процесорної потужності).

Оптимізації використання пам'яті програмним додатком потребують глибокого розуміння функціювання системи та доступу до процесу розробки, що також робить цей метод економічно необґрунтованим при наявності доцільнішої можливості масштабування. Масштабування розрахункових потужностей є вигідним при малих розмірах, але стає дедалі менш економічно-обґрунтованим при кожному збільшенні розміру віртуальної машини.

Варто зазначити, що сучасні методології розробки базуються на наявності декількох середовищ для тестування перед випуском версії у публічний доступ [4] і наявності вторинної репліки/регіону (5) для підвищення доступності. Таким чином, перехід на вищий рівень суттєво збільшує загальну вартість експлуатаційних витрат.

Існуючі публікації в більшості [6], [7] фокусуються на дослідженні проблеми витоків пам'яті, приділяючи менше уваги надмірному використанню пам'яті. Відомо, що витік пам'яті – це стан, коли програма помилково не може вивільнити більше непотрібну пам'ять для повторного використання, тож продовжує виділяти нові блоки.

На рисунку 1 представлено стрімке збільшення використання пам'яті програмним додатком, що призводить до зниження продуктивності і необхідності перезапуску системою моніторинга кожні декілька годин. Високонавантажені системи, які працюють з великими обсягами даних особливо вразливі до цієї проблеми.

Для знаходження першопричини витоку використовуються знімки пам'яті [8] – запис вмісту оперативної пам'яті у певний момент часу.

Знімки пам'яті також використовуються для пошуку шкідливого програмного коду [9], особливо при ін'єкції блоку на виконання в довірений процес. При наявному фокусі на ідентифікацію витоків, існуючі дослідження обділяють увагою проблему надмірного використання пам'яті – коли програмний додаток використовує більше пам'яті ніж того потребує розв'язувана задача. Зазначена проблема може виникнути з різних причин, але одна з найпоширеніших це дублювання незмінних об'єктів. Якщо програма створює декілька об'єктів з однаковими значеннями замість одного з властивістю незмінності, то це призводить до необхідності зберігати в пам'яті дублюючу інформацію і збільшує вимоги щодо обсягу пам'яті.

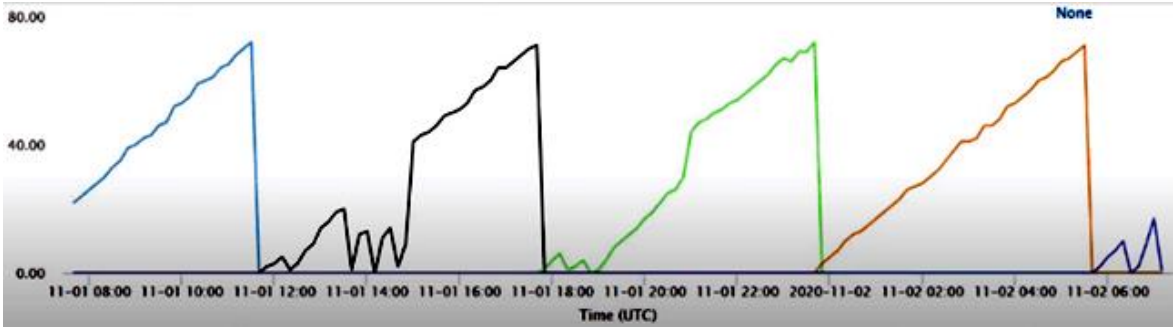

Рисунок 1 - Динаміка використання пам'яті

Однією з поширених проблем функціонування сучасних систем є дублювання об'єктів типу String (строка), що є серйозним викликом, який призводить до збільшення загальної вартості експлуатаційних витрат і постає непомітнім для існуючих діагностичних систем бо саме критерій витоку пам’яті не виконується.

Для знаходження дублікатів необхідно виконати порівняння усіх строк між собою, що призведе до надвеликої кількості операцій і впливатиме на швидкодію додатку у негативному ключі.

Метою роботи є розробка нової математичної моделі виявлення дублікатів об'єктів типу «строка» на основі аналізу знімків пам’яті.

2. Основні результати. Візуалізація вартості користування хмарними обчислювальними машинами [3] в залежності від розміру оперативної пам’яті відображає збільшення вартості мінімум вдвічі при вичерпанні обсягу пам’яті віртуальної машини і необхідності використовувати наступний рівень.

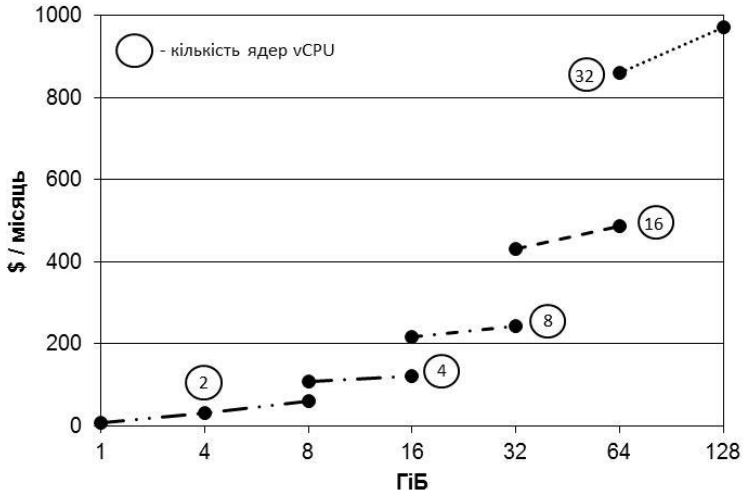

Рисунок 2 - Залежність вартості хостингу від об'єму оперативної пам'яті

Дослідники Adriaan Labuschagne та Laura Inozemtseva [4] у роботі по дослідженню регресійного тестування наочно продемонстрували методи для підвищення якості продукту. Тестування продукту у схожих до реального середовищах дозволяє суттєво зменшити ризики появи помилок у функціоналі [10], але вимагає наявності середовищ і збільшує кількість розрахункових вузлів (віртуальних машин).

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Велика увага приділяється механізмам знаходженням витоків пам’яті в роботах Gene Novark, Emery D. Berger, Benjamin G. [6] та Markus Weninger [7].

Jon Louis Bentley [2] описує підходи до зменшення використання пам’яті, але механізми по знаходженню кандидатів для оптимізації знаходяться поза рамками його робіт. Ioannis T. Christou, Sofoklis Efremidis [11] описують використання патерну проектування «пул об’єктів». За результатами їх досліджень, використання шаблону проектування може значно зменшити час відгуку високопродуктивних багатопотокових застосунків, особливо у середовищах з обмеженою пам'яттю. Механізм знаходження належних використань не описується.

Автори цих досліджень сходяться у думці, що знаходження застосувань цього шаблону залежить від конкретного програмного продукту і даними якими він оперує. Надмірне використання пам’яті може бути серйозною проблемою для високонавантажених систем. Однак, з розумінням причин неоптимального розподілу пам’яті та використанням відповідних стратегій для їх уникнення, можна підвищити стабільність та ефективність роботи системи.

Такий сценарій може бути важко виявити, оскільки він не проявляється явно – критерій витоку пам’яті не виконується.

Для виявлення дублювання даних необхідно проаналізувати пам’ять програми і виділити об’єкти які мають властивість незмінності. Властивість незмінності може задаватись як на рівні мови програмування [12], так і проявлятись для існуючого коду через відсутність викликів методів які можуть змінювати стан об’єкту.

Знаходження властивості незмінності є ключем до зменшення витрат пам’яті через повторне використання однакових об’єктів [13], і потребує наявності повної кодової бази виконуваної програми.

Знімок пам'яті програми може бути зображеним у вигляді структурованого масиву байтів в такий спосіб:

$$M_{time} = [b_1, b_2, ..., b_C],$$

де $time$ – момент часу, у який відбувається зняття знімку пам’яті; $b_1, b_2, ..., b_C$ – байти, з яких складається знімок; C – кількість байтів.

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

Знімки пам’яті широко використовуються для аналізу та виявлення шкідливого коду який може бути введений в пам’ять довіреного процесу. Традиційні статичні та динамічні методи неефективні у виявленні шкідливих програм, що проживають у пам'яті [14]. Крім того, існуючі рішення з використанням форензичного аналізу пам'яті показують незадовільну ефективність у плані швидкості виявлення та залежать від масивних знань експертів у аналізі пам'яті. Рішенням цієї проблеми є зняття знімків пам’яті здорових процесів для використання у якості навчальної бази для алгоритмів машинного навчання. При виявленні невідповідності, система безпеки має змогу швидко відреагувати попри відсутність фізичного файлу-загрози.

Враховуючи наявність великої кількості об’єктів в пам’яті промислових програмних додатків, ручний пошук дублікатів поміж тисяч різних типів не є доцільним. Необхідно обмежити пошук до типів які займають найбільшу кількість місця в пам’яті, у більшості випадків це об’єкти типу «строка».

Крім того, створення, модифікація та видалення об'єктів типу «строка» вимагає певних ресурсів. На рисунку 3 наведений приклад дублювання об’єктів з досліджуваної промислової системи, яка споживає більше 20ГБ пам’яті. Цей програмний додаток не відповідає ознакам витоку пам’яті, а саме монотонному зростанню використаної пам’яті, тож існуючі системи моніторингу не знаходять проблем з використанням пам’яті. Завдяки знімку пам’яті, були знайдені дублікати об’єктів з властивістю незмінності, тож ідентифіковане збільшене використання оперативної пам’яті.

```
00000181696ce700 (Model.CountryToCountrySetting)
    <ObjectState>k__BackingField:0x0 (Undefined) (Data.CacheObjectStates)
    <SourceCountryId>k__BackingField:0xe2 (System.Int64)
    <TargetCountryId>k__BackingField:0x20 (System.Int64)
    <SettingName>k__BackingField:00000181696ce740 (System.String) Length=34, String="AddShippingCostToCommercialInvoice"
    <SettingValue>k__BackingField:00000181696ce7a0 (System.String) Length=6, String="5;;USD"
    <SettingDescription>k__BackingField:00000181696ce7c8 (System.String) Length=111, String="Value for Commercial and 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"
    <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"
    <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,
00000181696cec40 (Model.CountryToCountrySetting)
    <ObjectState>k__BackingField:0x0 (Undefined) (Data.CacheObjectStates)
    <SourceCountryId>k__BackingField:0xe2 (System.Int64)
    <TargetCountryId>k__BackingField:0x23 (System.Int64)
    <SettingName>k__BackingField:00000181696cec80 (System.String) Length=34, String="AddShippingCostToCommercialInvoice"
    <SettingValue>k__BackingField:00000181696cece0 (System.String) Length=6, String="5;;USD"
    <SettingDescription>k__BackingField:00000181696ced08 (System.String) Length=111, String="Value for Commercial and Parcel Invoice,
```

Рисунок 3 - Приклад дублювання строк в об'єктах

Сканування знімків пам'яті – це важливий процес для розуміння механізму роботи програмного додатку та діагностування проблем [2]. Автоматизація процесу сканування знімків пам’яті - це ефективний метод, який допомагає швидко проводити аналіз роботи високонавантажених систем. Тому автоматизація сканування може принести значні переваги, зокрема зменшення часу на аналіз, підвищення точності виявлення проблем та покращення загальної продуктивності системи [3].

Ідентифікуємо об'єкти у знімку пам'яті за їх властивостями, і визначимо це як метрику. Метрика об'єктів вводиться для визначення «відстані» між об’єктами та їх схожості. Це можуть бути властивості об'єктів як то час, «вага», «довжина» і т.ін. Якщо таких властивостей нема, тоді створюємо метрику штучно.

Автоматизована система сканування утворить множину об'єктів O [1]. Застосуємо функцію групування за типами і знайдемо підмножину об’єктів, які мають незмінні типи O_t. У цій підмножині виділимо підмножину $STRN$ всіх об'єктів типу «строка» і підмножину STR об'єктів, які мають властивість незмінності. Таким чином, виділення підмножини «строк»:

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

$$STR \subset STRN \subset O_{t_i} \subset O \ . \tag{1}$$

Об'єкт типу «строка» має нульовий байт $b(0)$, довжину об'єкта у байтах і інформацію у цих байтах. Таким чином, об'єкт типу «строка» можна записати:

$$String\left(b(0), b\right).$$

Введемо додаткову властивість s об’єкту, як арифметичну суму байтів, що виділяються для збереження змісту об’єкта. Що дозволяє вираз для об'єкту типу «строка» записати у такий спосіб

$$String\left(b(0), b, s\right).$$

Кожен екземпляр об'єкту $String$ належить множині:

$$STRN = \left\{String_1\left(b_1(0), b_1, s_1\right), String_2\left(b_2(0), b_2, s_2\right), ..., String_N\left(b_N(0), b_N, s_N\right)\right\}. \tag{2}$$

де N – кількість екземплярів об'єктів типу «строка». У цю множину ввійдуть всі об'єкти типу «строка», в тому числі дублікати.

Кожен екземпляр об'єкта «строка» ідентифікується по параметром s, який утворює множину

$$S = \left\{s_1, s_2, ..., s_U\right\}.$$

де U – кількість унікальних розмірів об'єктів типу «строка», $U \le N$.

У множині S знайдемо мінімальне a і максимальне b значення. Відрізок $[a,b]$ розіб'ємо на m напівінтервалів довжиною

$$\Delta = \frac{b-a}{m}, \ m \le U \ . \tag{3}$$

У кожен напівінтервал може ввійти кілька значень параметра s об'єктів типу «строка», а може і жодного. Розподіл по інтервалам параметра s проводиться з підпорядкуванням підмножин:

$$k = \prod_{i=1}^{N}\left(\frac{s_i}{\Delta}\right), \ k \le U \ , \tag{4}$$

де k – номер інтервалу.

Таким чином, відбувається часткове упорядкування множини S по напівінтервалам. Зауважимо, що отримані напівінтервали можуть містити строки з однаковим значенням s_i, але бути різними за вмістом. Наочним прикладом є строки, в яких змінений порядок слів, але алгебраїчна сума байтів що відводиться під їх збереження, буде однаковою. Функція підсумовування було обрано як швидку хеш-функцію. Серед множини напівінтервалів знайдуться такі, що будуть містити кілька неупорядкованих значень. В таких напівінтервалах слід провести додаткове упорядкування. Тому ставиться задача здійснити мінімальну кількість упорядкувань. Позначимо функцією $G(m,L)$ загальну кількість упорядкувань. Таким чином, виникає задача знайти мінімум $G(m,L)$. Введемо припущення, у кожен інтервал попадає однакова кількість чисел L, в такому разі загальна кількість упорядкувань дорівнює

$$G(m,L)=mL\frac{L-1}{2}. \quad (5)$$

Треба встановити значення m і L при яких функція $G(m,L)$ набуде найменше значення. Оскільки $m=\frac{U}{L}$, то після підстановки загальна кількість упорядкувань

$$G(U,L)=U\frac{L-1}{2}. \quad (6)$$

Мінімум функції $G(m,L)$ залежить від значення параметра L. Тривіальний варіант $L=1$ не розглядаємо. Під час виконання операції сортування потрібно щонайменше два значення, тому мінімальне значення функція $G(m,L)$ набуде при $L=2$. В такому разі:

$$minG(U,L)=\frac{U}{2}. \quad (7)$$

За такого значення L кількість напівінтервалів дорівнює $m=\frac{U}{2}$.

Оскільки заздалегідь невідомо, на яку кількість нерівних напівінтервалів слід робити множину S так, щоб ці напівінтервали містили однакову кількість чисел, вводимо оцінку:

$$inf\,G(U,L)=\frac{U}{2}. \quad (8)$$

Верхню границю значень для функції G знайдемо з рівняння (5) при $L=U$ і $m=1$:

$$supG(U,L)=U\frac{U-1}{2}. \quad (9)$$

Побудуємо розподіл на основі даних зі знімку пам’яті однієї з досліджуваних промислових систем.

3. Приклади. На рисунку 4 показано розподіл об'єктів типу «строка» для типу «Model.CountryToCountrySetting» по параметру S. На графіку присутні три піки які демонструють значне збільшення елементів з однаковим значенням контрольної суми s_i, і відповідають відповідають трьом об'єктам «5;;USD», «AddShippingCostToCommercialInvoice», «Value for Commercial and Parcel Invoce». Відповідно схемі побудові напівінтервалів і сортуванню кожна вершина графіку відповідає одному значенню s_i та повністю ідентифікує об'єкт. Що дозволяє швидко проводити аналіз і ідентифікацію.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

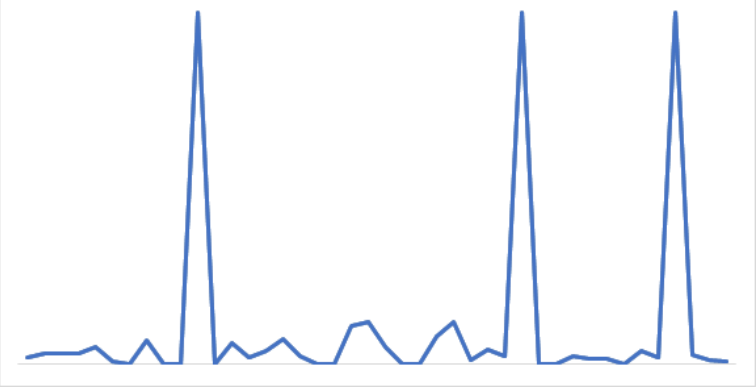

Рисунок 4 - Розподіл по множині S

Проведене групування за параметром s_i підтвердило наявність 220 тисяч унікальних значень і 14.5 мільйона дублікатів. Знайдено найбільш дубльовані строки, таким чином рисунок 5 підтверджує початкове припущення про наявність аномального дублювання у знімку пам'яті. Утворимо множину s^1 параметрів s_i цих об'єктів, $s^1 = \{s_{k1}, s_{k2}, s_{k3}\}$. Множина s^1 входить у множину S, тобто $s^1 \subset S$. Видаливши множину s^1 з множини S, отримаємо нову множину $S^1 = S \setminus s^1$. Знову будуємо розподіл по множині S^1.

Рисунок 5 - Розподіл по множині S^1

Отримуємо новий розподіл. По якому розподілу були знайдені об'єкти (таблиця 1).

Таблиця 1

Назва таблиці

«Строка»	Кількість дублікатів	Об'єм пам'яті
True	635181	2.4 MB
HideStandardShippingMethod	390073	9.7 MB
false	318273	1.5 MB
Restricted due to ticket	257600	8.4 MB
Visa	241913	945 KB

Визначимо ефективність цього метода по кількості операцій, порівняно з простим перебором [4]. При простому переборі буде виконано $U\frac{U-1}{2}$ операцій порівнянь.

4. Доведення. Визначимо коефіцієнт ефективності K_{ef} для $inf\,G(U,L)$ та $sup G(U,L)$ з додаванням U операцій розподілу по напівінтервалам та U операцій для визначення значень a і b

$$K_{ef}^{inf}=\frac{U\frac{U-1}{2}}{\frac{U}{2}+2U}=\frac{U-1}{5},$$

$$K_{ef}^{sup}=\frac{U\frac{U-1}{2}}{U\frac{U-1}{2}+2U}=\frac{U-1}{U+3},$$

Таким чином коефіцієнт ефективності буде знаходитись у межах

$$\frac{U-1}{U+3}\le K_{ef}\le\frac{U-1}{5}.$$

Наступним кроком оптимізації процесу пошуку дублікатів є зменшення початкового відрізку Δ (формула 3) з урахуванням розподілу строк, а саме великої кількості дублікатів до 150 символів. Таким чином було досягнуто більш рівного розподілу значень по напівінтервалам.

5. Висновки. Більшість публікацій за тематикою дослідження фокусуються на проблемі витоків пам’яті, обходячи надмірне споживання через неможливість його діагностування. Дослідження по застосуванню шаблонів проектування для зменшення використання пам’яті наводять механізми впровадження та ефект, але залишають стороною критерії та механіки пошуку кандидатів через велику варіативність сценаріїв та програмних продуктів.

Дана публікація фокусується саме на алгоритмі пошуку кандидатів за допомогою знімків пам’яті. Хоча знімки пам’яті використовуються для пошуку витоків пам’яті, або введеного шкідливого коду, їх використання для пошуку надмірного споживання пам’яті є новітнім впровадженням.

Для об'єктів типу «строка» була створена метрика. Побудовано покращений алгоритм пошуку дублікатів об'єктів типу «строка» у знімках пам'яті. Проведено тестування алгоритму на реальній промисловій системі який пришвидшує аналіз з 7464 до 1894 мс – майже в чотири рази.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Таблиця 2

Результати роботи оптимізованого алгоритму

Кількість підмножин	Максимальна кількість елементів в 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

При збільшенні кількості підмножин, час виконання розбиття зменшується непропорційно зростанню використаної пам’яті процесом групування.

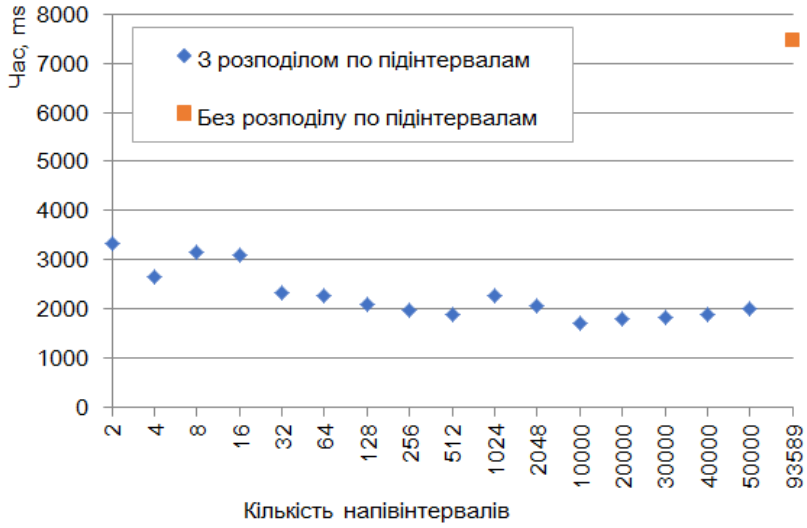

Рисунок 6 - Залежність часу виконання від кількості підмножин

Зменшення максимальної кількості елементів в підмножинах також спадає непропорційно збільшенню кількості підмножин.

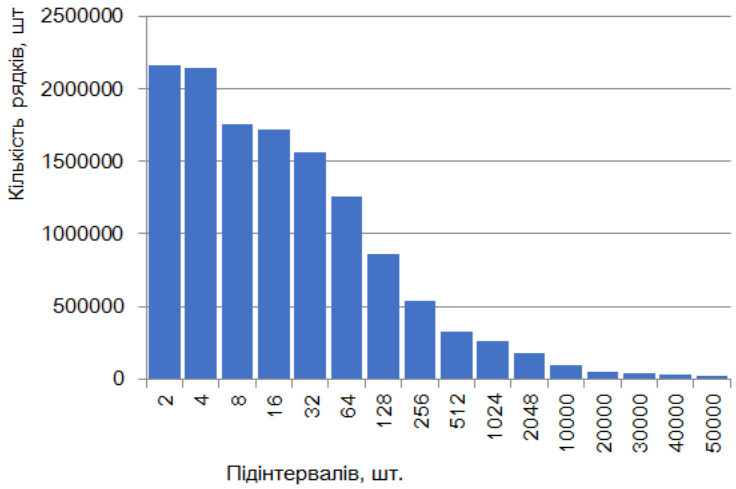

Рисунок 7 - Залежність максимальної кількість елементів в підмножині від її обсягу

Згідно зробленого припущення, найкраща швидкодія розбиття досягається при схожій кількості елементів в підмножинах для зменшення частки строк з однаковою контрольною сумою, але різних за змістом. Це припущення було підтверджене серією експериментів зі знімками пам'яті досліджуваних промислових систем, кількість підмножин для пришвидшення розбиття знаходиться у діапазоні (460 – 520). Подальше збільшення кількості підмножин не дає вагомого пришвидшення операції, але призводить до необхідності тримати в пам'яті значно більше число груп, тож збільшує вимоги до кількості пам'яті необхідної для проведення аналізу.

Внесок авторів.

Наталія Гук – концептуалізація, методика.

Миколай Мітіков – концептуалізація, математична модель, програмне забезпечення, проведення обчислювального експерименту, збір і перевірка емпіричних даних, аналіз джерел, підготовка огляду літератури або теоретичних основ дослідження.

ЛІТЕРАТУРА

1. Gregg, Brendan. 2.7.3 Scaling solutions. Systems Performance, Second Edition. Boston : Addison-Wesley, 2021, с. 929.

2. Bentley, Jon Louis. Writing efficient programs. Englewood Cliffs, N.J. : Prentice-Hall, 1982. ISBN-13/ 978-0139702440.

3. Microsoft. Windows Virtual Machines Pricing. Cloud Computing Services | Microsoft Azure. [Онлайновий] Microsoft . [Цитовано: 8 February 2024 р.] https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/.

4. 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.

5. Kleppmann, Martin. CHAPTER 5: Replication. Designing Data-Intensive Applications. місце видання невідоме : O'Reilly Media, 2017.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

6. Efficiently and precisely locating memory leaks and bloat. Gene Novark, Emery D. Berger, Benjamin G. Zorn. Dublin : Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009. 978-1-60558-392-1.
7. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. Weninger, Markus. Mumbai : Association for Computing Machinery, 2019. 978-1-4503-6239-9.
8. Hewardt, Mario. Advanced .NET Debugging. Addison-Wesley Professional, 2009. 978-0-321-57889-1.
9. Hamad Naeem, Shi Dong, Olorunjube James Falana, Farhan Ullah. Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification. Expert Systems with Applications. 2023 p., Т. 223.
10. Roy Osherove, Vladimir Khorikov. The Art of Unit Testing. місце видання невідоме : Manning, 2024. ISBN 9781617297489.
11. Ioannis T. Christou, Sofoklis Efremidis. To Pool or Not To Pool? Revisiting an Old Pattern. Marousi : Athens Information Technology, 2018. arXiv:1801.03763.
12. Immutable Objects for a Java-Like Language. C. Haack, E. Poll, J. Schäfer, A. Schubert. Berlin : Springer, 2007. 978-3-540-71316-6.
13. М.Ю. Мітіков, Н.А. Гук Огляд методів виявлення та аналізу проблем продуктивності в програмному забезпеченні: підходи, виклики та перспективи // Питання прикладної математики і математичного моделювання [Текст]: зб. наук. пр. / редкол.: О.М. Кісельова (відп. ред.) [та ін.]. – Дніпро, 2023. – Вип. 23. – с. 171 – 178. DOI: 10.15421/322318
14. 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., Т. 6. 10.1186/s42400-023-00157-w.

REFERENCES

1. Gregg, Brendan. 2.7.3 Scaling solutions. Systems Performance, Second Edition. Boston : Addison-Wesley, 2021, с. 929.
2. Bentley, Jon Louis. Writing efficient programs. Englewood Cliffs, N.J. : Prentice-Hall, 1982. ISBN-13/ 978-0139702440.
3. 3. Microsoft. Windows Virtual Machines Pricing. Cloud Computing Services | Microsoft Azure. [Online] Microsoft . [Cited: 8 February 2024 p.] https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/..
4. 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.
5. Kleppmann, Martin. CHAPTER 5: Replication. Designing Data-Intensive Applications. місце видання невідоме : O'Reilly Media, 2017.
6. Efficiently and precisely locating memory leaks and bloat. Gene Novark, Emery D. Berger, Benjamin G. Zorn. Dublin : Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009. 978-1-60558-392-1.
7. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. Weninger, Markus. Mumbai : Association for Computing Machinery, 2019. 978-1-4503-6239-9.

8. Hewardt, Mario. Advanced .NET Debugging. Addison-Wesley Professional, 2009. 978-0-321-57889-1.
9. Hamad Naeem, Shi Dong, Olorunjube James Falana, Farhan Ullah. Development of a deep stacked ensemble with process based volatile memory forensics for platform independent malware detection and classification. Expert Systems with Applications. 2023 p., T. 223.
10. Roy Osherove, Vladimir Khorikov. The Art of Unit Testing. Manning, 2024. ISBN 9781617297489.
11. Ioannis T. Christou, Sofoklis Efremidis. To Pool or Not To Pool? Revisiting an Old Pattern. Marousi : Athens Information Technology, 2018. arXiv:1801.03763.
12. Immutable Objects for a Java-Like Language. C. Haack, E. Poll, J. Schäfer, A. Schubert. Berlin : Springer, 2007. 978-3-540-71316-6.
13. M. Yu. Mitikov, N.A. Huk Overview of methods for identifying and analyzing performance problems in software: approaches, challenges and prospects // Issues of applied mathematics and mathematical modeling [Text]: coll. of science pr. / editor: O.M. Kiselyova (corresponding editor) [etc.]. – Dnipro, 2023. – Issue 23. - p. 171 – 178. DOI: 10.15421/322318
14. 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.

Received 02.12.2024.
Accepted 12.12.2024.

Mathematical optimisation model for searching duplicate string objects in the memory snapshot

The purpose of this paper is to identify the increased memory usage of software applications. The modern software development cycle focuses on functionality and often ignores aspects of optimal resource usage. Limited physical scaling sets an upper limit on the system's capacity to process requests. The presence of unchanged objects with the same information is a sign of increased memory consumption. Avoiding duplicate objects in memory allows for a more rational use of the existing resource and an increase in the amount of information processed. Existing scientific publications focus on the study of memory leakage problems, and limit their attention to excessive memory usage due to the lack of a unified model for finding excessive memory usage. It is worth noting that existing programming templates contain the 'object pool' template, but leave the conclusion about the feasibility of its implementation to engineers without providing a mathematical basis. The paper presents the development of a mathematical model for the process of detecting duplicate objects with the immutability property of the String type in a memory snapshot. Industrial systems that require hundreds of GB of RAM to operate and contain millions of objects in RAM are analysed. Given this scale of data, there is a need to optimise the duplicate detection process. The research method is to analyse memory snapshots of highly loaded systems using the software code developed on .NET technology and the ClrMD library. The memory snapshot reflects the state of the process under study at a given time, contains all objects, threads and operations performed. The ClrMD library allows you to programmatically examine objects, their types, get field values, and build graphs of relationships between objects. Based on the results of the study, an opti-

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

misation was proposed that allows to speed up the process of finding duplicates several times. The scientific contribution of the study is the creation of a mathematically sound approach that significantly reduces the use of memory resources and optimises computing processes. The practical usefulness of the model is confirmed by the optimisation results achieved through the recommendations, reduced hosting costs (which provides greater cost-effectiveness in the deployment and use of software systems in industrial environments), and increased data processing.

Keywords: mathematical model; optimisation; algorithm; performance; memory snapshot; duplication; string

Гук Наталія Анатоліївна – завідувачка кафедри комп'ютерних технологій, доктор фізико-математичних наук, професорка, Дніпровський національний університет імені Олеся Гончара, huk_n@fpm.dnu.edu.ua, ORCID ID: 0000-0001-7937-1039
Мітіков Микола Юрійович – аспірант кафедри прикладної математики, Дніпровський національний університет імені Олеся Гончара, mitikov.m22@fpm.dnu.edu.ua, ORCID ID: 0009-0002-1297-5676

Huk Nataliia - Head of the Department of Computer Technologies, Doctor of Physical and Mathematical Sciences, Professor, Oles Honchar Dnipro National University, huk_n@fpm.dnu.edu.ua, ORCID ID: 0000-0001-7937-1039
Mitikov Nikolay - PhD student, Department of Applied Mathematics, Oles Honchar Dnipro National University, mitikov.m22@fpm.dnu.edu.ua, ORCID ID: 0009-0002-1297-5676