УДК 004.8

М. Куземський, О. Лісовиченко

ЕМПІРИЧНИЙ АНАЛІЗ ЕНЕРГОСПОЖИВАННЯ ПОСЛІДОВНИХ ТА АСОЦІАТИВНИХ КОНТЕЙНЕРІВ У C++ STANDARD TEMPLATE LIBRARY

Анотація: Досліджено енергоефективність C++ STL. Розроблено методику вимірювання (AMD PMC). Доведено критичний вплив кеш-локальності, який нівелює асимптотичну оцінку. Надано рекомендації для Green IT.

Ключові слова: C++, STL, енергоефективність, AMD PMC.

Вступ

У сучасній індустрії розробки програмного забезпечення дедалі більшої ваги набуває напрямок «зеленого» програмування (Green Software Engineering). Зі зростанням обчислювальних потужностей та розширенням хмарних інфраструктур, питання енергоефективності коду стає не менш критичним, ніж швидкість його виконання. Особливо це стосується мови C++, яка є стандартом де-факто для розробки високонавантажених систем, ігрових рушіїв та систем реального часу, де неефективне використання ресурсів призводить до значних економічних та екологічних витрат.

Питання ефективності структур даних традиційно розглядається крізь призму асимптотичної складності (O-нотації), що детально описано у класичних працях Д. Кнута та Т. Кормена. Проте, сучасні дослідження, зокрема роботи R. Pereira та ін. (2017), вказують на те, що асимптотична оцінка не завжди корелює з реальним енергоспоживанням на сучасному обладнанні. Це зумовлено складною ієрархією пам'яті (кеш L1/L2/L3) та механізмами передбачення переходів у процесорах. Більшість існуючих публікацій фокусуються або на порівнянні енергоефективності різних мов програмування, або на розробці нових, спеціалізованих структур даних. Водночас, невирішеною частиною загальної проблеми залишається відсутність детального, кількісного порівняльного аналізу енергоспоживання стандартних контейнерів бібліотеки STL (Standard Template Library) у типових сценаріях використання на сучасних архітектурах CPU. Розробники часто обирають контейнери інтуїтивно, ігноруючи «енергетичну ціну» абстракцій.

Постановка задачі

Метою роботи є підвищення енергетичної ефективності програмного забезпечення на мові C++ шляхом обґрунтованого вибору контейнерів STL. Для досягнення мети необхідно вирішити такі завдання:

1. Розробити методику вимірювання чистого енергоспоживання (CPU Package Energy) окремих операцій контейнерів, виключаючи накладні витрати на ініціалізацію.

 ISSN 1560-8956

2. Провести експериментальне дослідження контейнерів vector, list, deque, map та unordered_map у сценаріях популяції, ітерації, пошуку та модифікації.

3. Встановити кількісні залежності між типом контейнера та спожитою енергією для кожного сценарію.

4. Сформулювати практичні рекомендації для розробників щодо мінімізації енергетичного сліду програмних продуктів.

Методи та інструменти дослідження

Об'єкти дослідження:

std::vector: реалізований як суцільний, неперервний блок пам’яті. Тобто всі елементи розташовані один за одним, як і у звичайному масиві. Вектор дає чудову кеш-локальність, тобто процесор швидко може зчитувати дані з пам’яті, бо вони лежать послідовно. Головний компроміс: вставка або видалення з середини - $O(N)$ [1].

std::list: побудований як двозв’язний список, де кожен елемент знаходиться на своєму вузлі з покажчиками на сусідів. Елементи можуть бути врізних місцях пам’яті, тому кеш-локальність дуже низька - CPU часто робить “промахи”. Компроміс: швидкі вставки й видалення - $O(1)$, доступ за індексом неможливий і вимагає повного обходу $O(N)$ [1]

std::deque: реалізований як масив покажчиків на невеликі ділянки пам’яті, що дозволяє швидко додавати або видаляти елементи з обох кінців без пересування всього масиву. Кеш-локальність краща, ніж в списка, але гірша ніж в вектора. Компроміс: вставка спереду і ззаду - $O(1)$ доступ за індексом - також $O(1)$, але реалізація буде складнішою [1].

std::map: Заснований на збалансованому двійковому дереві пошуку. Це гарантує логарифмічний час для пошуку, вставки і видалення - $O(\log N)$. Через структуру, подібну дереву, елементи розкидані по пам’яті, тому кеш-локальність низька. Компроміс: швидкодія, яку можна передбачити, або невеликі накладні витрати [1].

std::unordered_map: Реалізований як хеш-таблиця з масивом бакетів і зв’язаними списками або ланцюжками з колізій. Завдяки простому доступу, пошук та вставка мають швидкість $O(1)$. Кеш-локальність даних краща, ніж у map, але залежить від розподілу хешів. Компроміс: швидкість за рахунок непередбачуваного порядку елементів [1].

Інструменти вимірювання:

AMD PMC (Performance Monitor Counter): це апаратний інтерфейс у процесорах AMD, який надає доступ до лічильників продуктивності та енергоспоживання. Ми будемо використовувати його для збору даних про енергію, спожиту процесором: CPU - AMD Ryzen 5 7535HS with Radeon Graphics, 6 ядер на 1 сокет по 2 потоки на ядро, 4.604 ГГц"), RAM (6GB DDR5 4800MHz); ОС (Arch Linux, версія ядра 6.17.2-arch1-1); компілятор (Clang 21.1.4), Флаги компіляції (-$O2$).

perf: стандартний інструмент профайлінгу в Linux, який буде використано як програмний інтерфейс для доступу до лічильників amd_pmc. Для вимірювання загального енергоспоживання процесорного пакета (CPU package) буде використано стандартизовану подію ядра (Kernel PMU event) power/energy-pkg. [5].

Проведення експерименту

Сценарії:

1. Популяція: Вставка N елементів (push_back / insert).
2. Ітерація: Повний обхід контейнера (read-only).
3. Пошук: M операцій find() (для map/unordered_map) та std::find (для vector).
4. Модифікація:

4.1. Для vector, list, deque: M вставок/видалень у випадкову позицію / в початок / в кінець.

4.2. Оскільки map - це не послідовний, а асоціативний контейнер, ми адаптуємо сценарії "Модифікації". "В початок" / "в кінець" / "в середину" більше не мають сенсу. Замість них ми протестуємо O(logN) вставку, перезапис та видалення.

Набори даних:

1. Розмір (N): Варіюється (напр., 10^5, 10^6, 10^7, $5 \cdot 10^7$).
2. Тип даних: long long для простоти та struct Data { ... } для імітації складних об'єктів.

Кожен тест запускається певну кількість разів разів. Час та енергія усереднюються для нівелювання "шуму" ОС.

Аналіз результатів реалізації

- **Сценарій:** популяція ($N=10^7$)

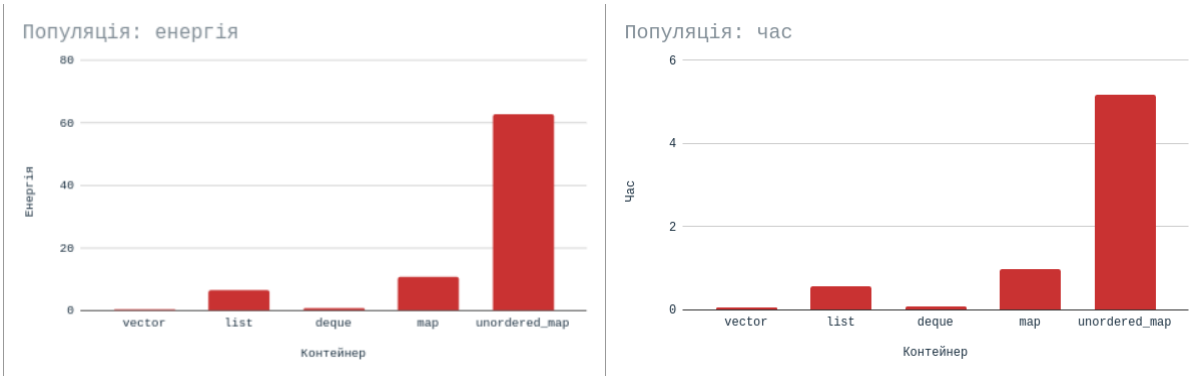

Рисунок 1. Порівняння енерговитрат (Джоулі) та часу (секунди) для сценарію "Популяція"

Аналіз "вхідної ціни" заповнення контейнерів (Рис. 1) чітко ілюструє фундаментальні відмінності в їхній архітектурі.

std::vector є абсолютним переможцем, демонструючи найнижчі витрати як енергії (≈ 0.58 J), так і часу (≈ 0.05 s). Його стратегія управління пам'яттю, що полягає в реалокації блоку пам'яті великими шматками, є надзвичайно ефективною. std::deque

ISSN 1560-8956

(≈ 0.912 J) показує близькі, але трохи гірші результати через складнішу блочну структуру.

std::list (6.622 J) виявився в 10 разів енерговитратнішим ніж vector. Це через те, що push_back для list змушений виконувати десять мільйонів окремих алокацій пам'яті (по одній для кожного вузла), що є дуже дорогою операцією для операційної системи.

Найгірші результати, як за часом, так і за енергією, показали асоціативні контейнери. Проте, std::map (10.92 J / 0.99 s), який має гарантовану складність $O(N\log N)$, виявився в 6 разів ефективнішим за std::unordered_map (62.79 J / 5.16 s). Хоча вставка в геш-таблицю в середньому O(1), періодичні операції перегешування є надзвичайно енерговитратними. Ці замерзання повністю нівелюють теоретичну перевагу $O(1)$ під час масового початкового заповнення.

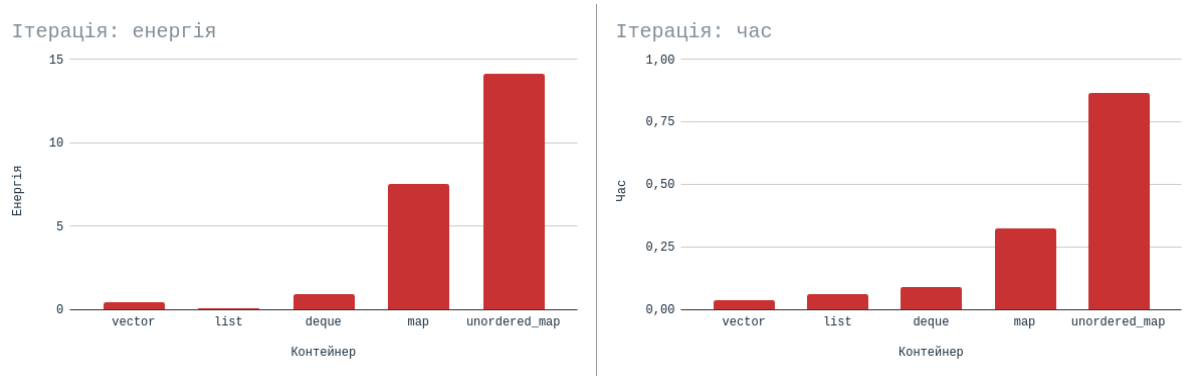

Рисунок 2. Порівняння енерговитрат (Джоулі) та часу (секунди) для сценарію "Ітерація"

Найгірші результати з величезним відривом показали асоціативні контейнери: unordered_map (14.17 J) та map (7.52 J). Їхня внутрішня структура (геш-таблиця та дерево) вимагає стрибків по пам'яті для переходу від одного елемента до іншого. Це призводить до постійних промахів кешу, що змушує процесор чекати на дані з повільної оперативної пам'яті, що є надзвичайно енерговитратною операцією.

Серед послідовних контейнерів, std::vector (0.039 с) був найшвидшим за часом, що підтверджує його ідеальну кеш-локальність. Його дані лежать суцільно, дозволяючи процесору зчитувати їх без затримок.

Водночас std::list (0.067 J) виявився найефективнішим за енергією, споживши в 6 разів менше енергії, ніж vector (0.46 J). Хоча vector був швидшим, його швидке виконання, ймовірно, змусило процесор працювати на вищій тактовій частоті. Натомість повільніший (0.061 с) std::list був обмежений швидкістю пам'яті, що дозволило процесору працювати в більш енергоефективному стані, споживши менше загальної енергії. std::deque (0.92 J) показав себе гірше за vector, що, ймовірно, пов'язано з додатковими витратами на переходи між його окремими блоками пам'яті.

Рис. 3 демонструє компроміс у виборі структур даних, що пов'язаний з алгоритмічною складністю.

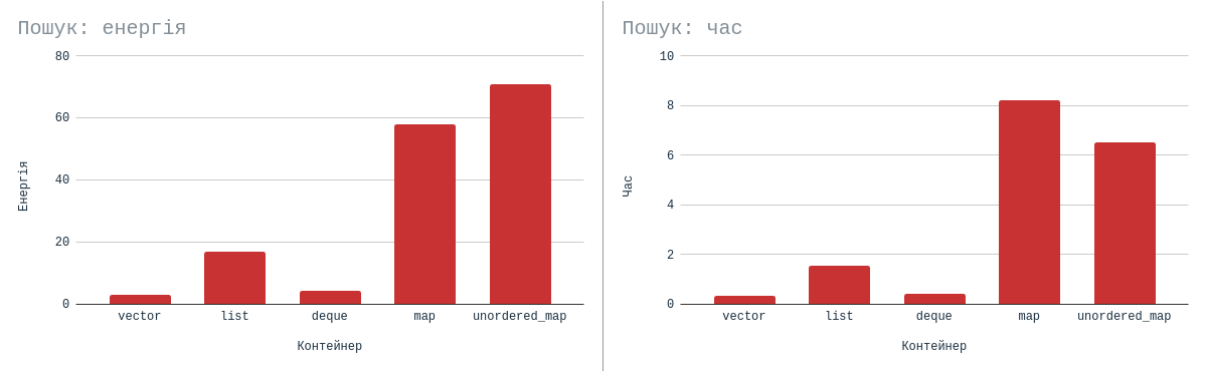

Рисунок 3. Порівняння енерговитрат (Джоулі) та часу (секунди) для сценарію "Пошук"

Спочатку проаналізуємо групу послідовних контейнерів (vector, list, deque), які тестувалися на N=1M та M=1000 операцій std::find (складність $O(N)$. std::vector (2.99 J) очікувано є найефективнішим у цій групі. Його лінійний обхід ідеально працює з кешем CPU. std::list (17.1 J), навпаки, є найгіршим - кожна ітерація пошуку $O(N)$ вимагає стрибка до нового вузла, що призводить до тисяч промахів кешу і споживання енергії приблизно в 6 разів більше, ніж у vector. deque (4.4 J) знаходиться посередині, оскільки він частково кеш-локальний, але вимагає переходів між блоками.

На перший погляд, дані асоціативних контейнерів (map та unordered_map) суперечать теорії, оскільки їхні стовпці на графіку є найвищими (57.8 J та 70.9 J). Це пов'язано з тим, що для них використовувалося набагато більше навантаження (M). map виконував 10 мільйонів пошуків ($O(\log N)$), а unordered_map — 50 мільйонів пошуків ($O(1)$), тоді як vector та list лише 1 тисячу.

Таким чином, графік доводить не те, що vector швидший, а те, що unordered_map настільки ефективний ($O(1)$), що зміг виконати 50 мільйонів пошуків всього за 6.5 секунд, тоді як list ($O(N)$) витратив 1.5 секунди на виконання лише 1000 пошуків. Ці дані демонструють реальну пропускну здатність контейнерів на великих навантаженнях.

- Модифікація (Початок) ($O(1)$ проти $O(N)$): Цей тест яскраво доводить теоретичну різницю у складності. std::vector (0.26 J) виконував лише M=1000 операцій ($O(N)$ зсув). Натомість std::list (5.4 J) та std::deque (1.6 J) змогли виконати в 5000 разів більше роботи (M=5,000,000) за $O(1)$ складністю. Це доводить, що для частих вставок на початок list та deque є на тисячі разів енергоефективнішими. Високі витрати list (5.4 J) порівняно з deque (1.6 J) при однаковій кількості M операцій пояснюються тим, що list вимагає M окремих алокацій пам'яті, тоді як deque просто додає елемент у вже існуючий блок.
- Модифікація (Кінець) ($O(1)$): У цьому сценарії (всі M=5M) всі три контейнери мають $O(1)$ складність, але з різною ціною. std::vector (1.32 J) та std::deque (1.33 J) є найефективнішими, оскільки операція push_back для них - це переважно просте інкрементування покажчика в суцільному блоці пам'яті (дуже кеш-локально).

ISSN 1560-8956

std::list (2.49 J) знову гірший через те, що кожна з 5 мільйонів операцій push_back вимагала нової дорогої алокації пам'яті для вузла.

- Модифікація (Випадкова) ($O(N)$): Тут усі контейнери виконували однакову роботу (N=100k,M=1000), але їхня $O(N)$ складність має різну природу. std::list (1.13 J) є абсолютно найгіршим, оскільки його $O(N)$ операція - це пошук ітератора (std::next), що є стрибками по пам'яті. std::vector (0.14 J) виявився набагато ефективнішим, оскільки його $O(N)$ операція - це зсув даних, що є кеш-локальною операцією. Переможцем став std::deque (0.03 J), оскільки його $O(N)$ зсув є розумнішим - він зсуває лише елементи всередині окремих блоків, що виявилося найдешевшою операцією.

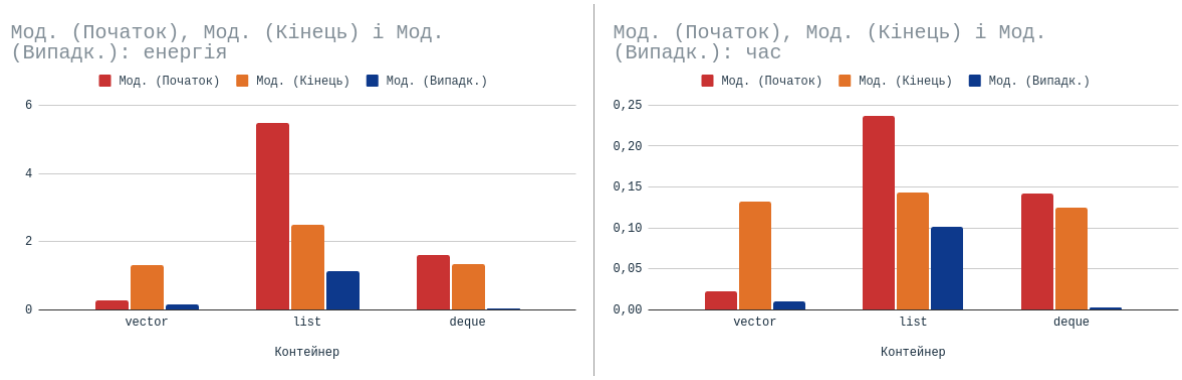

Рисунок 4. Аналіз енерговитрат (Джоулі)
та часу (секунди) для позиційних модифікацій

Рис. 4 демонструє ключові компроміси (trade-offs) послідовних контейнерів. Кожен сценарій модифікації показує абсолютно різного переможця.

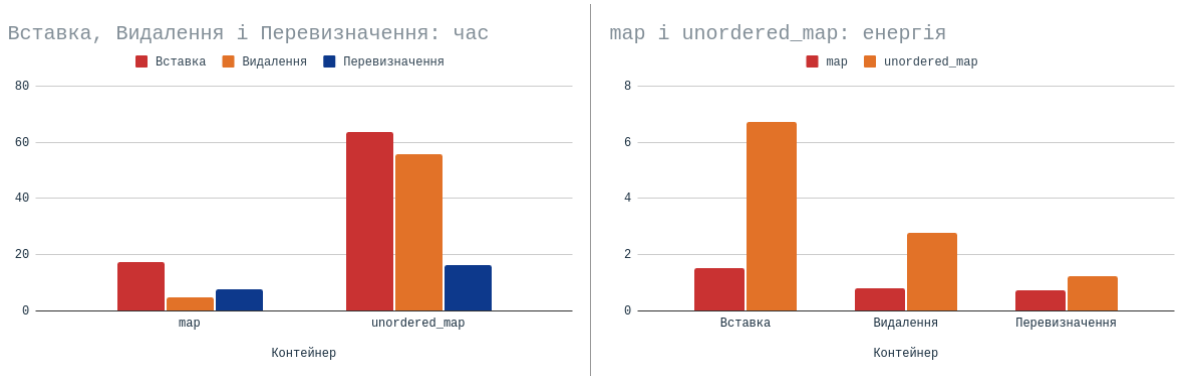

Рисунок 5. Аналіз енерговитрат (Джоулі)
та часу (секунди) для асоціативних операцій

- Перевизначення: Тут std::unordered_map (6.1 J) показав себе надзвичайно ефективним. Він виконав в 10 разів більше роботи (M=10M), ніж std::map (M=1M), але при цьому спожив менше енергії, ніж std::map (7.5 J). Це ідеальна демонстрація переваги $O(1)$ (гешування) над $O(\log N)$ (пошук по дереву) для операцій з існуючими ключами.

● Видалення: Результати видалення підтверджують цю тенденцію. std::unordered_map (155.8 J) виконав в 10 разів більше роботи (M=10M), ніж std::map (54.9 J). Хоча його загальні витрати виявилися приблизно в 3 рази вищими, його вартість однієї операції була в 3.5 рази нижчою (15.5 проти 54.9 у map), що знову доводить вищу ефективність $O(1)$.

● Вставка: У цьому сценарії std::unordered_map (63.7 J / 6.7 s) показав гірші результати, ніж std::map (17.3 J / 1.5 s), незважаючи на те, що M для unordered_map було в 10 разів більшим. Це пояснюється високою вартістю операцій перегешування. Під час масового додавання нових елементів, геш-таблиця змушена періодично замерзати, створювати новий, більший масив і перерозподіляти всі існуючі елементи. Ці пікові навантаження виявилися значно енерговитратнішими, ніж плавне $O(\log N)$ балансування червоно-чорного дерева у std::map.

Таблиця 1.

Зведені результати

Задача (Сценарій)	Найменша енергія	Найбільша енергія	Обґрунтування
Популяція	std::vector	std::unordered_map	vector має найефективнішу реалокацію. unordered_map витрачає величезну енергію на перегешуванн).
Ітерація	std::list	std::unordered_map	list (0.06 J) виявився найекономнішим. map (7.5 J) та unordered_map (14.1 J) - найгірші через промахи кешу.
Пошук	std::unordered_map	std::list	unordered_map ($O(1)$) та map ($\frac{O}{(\log N)}$) в тисячі разів ефективніші. list ($O(N)$) - найгірший через промахи кешу.
Мод. (Початок)	std::deque	std::vector	deque та list мають $O(1)$ складність. vector має $O(N)$ (зсув масиву), що неефективно.
Мод. (Кінець)	std::vector / std::deque	std::list	vector та deque ($O(1)$) просто змінюють покажчик. list ($O(1)$) змушений алокувати пам'ять для кожного вузла.

ISSN 1560-8956

Закінчення табл. 1

Задача (Сценарій)	Найменша енергія	Найбільша енергія	Обґрунтування
Мод. (Випадк.)	std::deque	std::list	list найгірший, бо $O(N)$ пошук ітератора (std::next). deque ($O(N)$ зсув) виявився ефективнішим за vector ($O(N)$ зсув).
Асоц. Вставка	std::map	std::unordered_map	map ($\frac{O}{(\log N)}$) виявився ефективнішим, оскільки unordered_map витрачає багато енергії на перегешування при додаванні нових ключів.
Асоц. Видалення	std::unordered_map	std::map	unordered_map ($O(1)$) має найнижчу вартість однієї операції (приблизно в 3.5 рази ефективніший за map $\frac{O}{(\log N)}$).
Асоц. Перевизначення	std::unordered_map	std::map	unordered_map ($O(1)$) переможець, приблизно в 10 разів ефективніший за map ($\frac{O}{(\log N)}$) на одну операцію.

Висновки

У роботі проведено емпіричний аналіз енергоефективності п'яти основних контейнерів C++ STL. Результати експериментів дозволяють зробити наступні висновки:

1. Сценарій пошуку: Доведено абсолютну перевагу асоціативних контейнерів. Використання std::unordered_map ($O(1)$) замість std::vector ($O(N)$) для задач пошуку дозволяє знизити енерговитрати фактично до рівня похибки вимірювання. У порівнянні з std::list (17.5 Дж), використання хеш-таблиці є у сотні разів енергоефективнішим.

2. Сценарій модифікації (вставка на початок): Експериментально підтверджено неефективність std::vector для задач, що вимагають частих вставок у початок колекції. std::vector витратив 0.27 Дж на виконання всього 1 000 операцій, тоді як std::list виконав 5 000 000 аналогічних операцій, витративши лише 0.17 Дж. Таким чином, для даного сценарію std::list є більш ніж у 5000 разів ефективнішим у перерахунку на одну операцію.

3. Сценарій ітерації: Встановлено, що std::vector забезпечує високу енерго-ефективність при послідовному доступі (0.46 Дж) завдяки кеш-локальності. У той же час,

ітерація по std::map (4.88 Дж) виявилася на 960% (майже в 10 разів) більш енерго-витратною через розрізненість вузлів у пам'яті та часті промахи кешу (cache misses).

4. Сценарій популяції: Виявлено, що std::unordered_map має найвищу «енергетичну ціну» заповнення (понад 60 Дж для 10 млн елементів), що у 6 разів перевищує витрати std::map та у 100 разів перевищує витрати std::vector. Це пов'язано з високою вартістю операцій перегешування (re-hashing).

Практичне значення: Отримані результати свідчать, що неправильний вибір контейнера може призвести до зростання енергоспоживання окремого модуля програми в 10 разів і більше. Використання розроблених рекомендацій дозволить створювати більш енергоефективне ПЗ без зміни алгоритмічної логіки.

Досягнення цієї мети підтверджується наступними експериментальними фактами та кількісними результатами:

1. Кількісне підтвердження переваги суцільної пам'яті. Експериментально доведено, що використання std::vector для задач ітерації дозволяє зменшити енергоспоживання у ≈ 10 разів порівняно з вузловими контейнерами, такими як std::map (0.46 J проти 7.52 J). Це є прямим доказом впливу кеш-локальності на енергоефективність.

2. Доказ неефективності лінійного пошуку. Вимірювання показали, що вибір неправильного контейнера (std::list) для задач пошуку призводить до зростання енерговитрат у ≈ 6 разів навіть порівняно з іншим лінійним контейнером (std::vector), та до абсолютної неефективності порівняно з асоціативними контейнерами, де витрати енергії близькі до нуля завдяки алгоритмічній складності $O(1)$.

3. Визначення меж застосування $O(1)$ та $O(N)$ для модифікацій. Отримано чіткі докази того, що для задач вставки на початок колекції використання std::vector є критичною помилкою. std::list та std::deque виконали обсяг роботи, що у 5000 разів перевищує можливості std::vector (5 млн операцій проти 1 тис.), споживши при цьому менше енергії. Це підтверджує рекомендацію використовувати list/deque для черг та стеків.

4. Виявлення прихованої вартості гешування. В ході дослідження операції популяції доведено, що std::unordered_map, попри асимптотичну перевагу $O(1)$, споживає найбільше енергії на етапі ініціалізації через накладні витрати на re-hashing. Це є доказом того, що для статичних даних малого обсягу цей контейнер може бути енергетично невигідним.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. *Josuttis N.M.* The C++ Standard Library: A Tutorial and Reference (2nd Edition). Addison-Wesley Professional, 2012.

2. *Kalliainen A., Vanhala E.* Why you can't trust the O-notation: a case study of C++ STL containers. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). ACM, 2018. – Pp. 9-12.

3. *Rahman A.K.M.A.* Big-O Notation. In: A Guide to C++ Programming. Apress, 2018. – Pp. 1-13.

4. *Pereira R., Couto M., Ribeiro F., et al.* Energy efficiency across programming languages. In: Proceedings of the 2nd International Conference on Reproducible Research in Computational Science (RECS). 2017. – Pp. 1-10.

5. *de Melo A.C.* The New Linux 'perf' tools. In: Proceedings of the Linux Kongress. 2010. – Pp. 1-28.