БАКАЛАВРСЬКА РОБОТА

БР. ІП - 08.00.00.000 ПЗ

Група ІП-22-3

Струк Сергій

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Струк Сергій Володимирович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Методології оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв
(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Струк С.В._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Процюк Галина Ярославівна, асистент
_____ (підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Струк Сергій Володимирович

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) "Методології оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв"

керівник проекту (роботи) Процюк Галина Ярославівна, асистент

затверджені наказом вищого навчального закладу від “ 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Аналіз вимог до програмного забезпечення, Аналіз вимог і постановка задачі, Проектування системи, Реалізація проекту. Тестування та впровадження

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Графік споживання енергії розвантаженням хмари (рис. 1.1, ст. 19)

2 Моделі оцінки OCV (рис. 1.3, ст. 26)

3 Етапи стільникової мережі HSPA, що беруть участь у передачі даних. (рис. 2.1 , ст.32)

4 Архітектура Sesame для автоматичного створення моделей (рис. 2.4, ст. 41)

5 Приклад перетворення програми (рис.3,5, ст.61)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	17.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	21.02.2026	виконано
3	Побудова моделі або алгоритму власного рішення	01.03.2026	виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	21.04.2026	виконано
5	Оформлення пояснювальної записки бакалаврської роботи	02.05.2026	виконано

Студент _____Струк С.В.
(підпис)

Керівник роботи _____Процюк Г.Я..
(підпис)

АНОТАЦІЯ

Бакалаврська робота робота містить 79 сторінок, 21 рисунок, 3 таблиці, список використаних джерел із 40 найменувань.

Метою роботи є дослідження методологій оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв.

Об'єкт дослідження: процеси енергоспоживання мобільного програмного забезпечення на сучасних мобільних пристроях.

Предмет дослідження: методи, моделі та інструменти оптимізації енергоспоживання мобільних застосунків, зокрема енергетичне профілювання, програмні механізми оптимізації та системи моніторингу ресурсів.

Результати дослідження: проведено аналіз сучасних методів енергетичного профілювання мобільних застосунків, досліджено підходи до моделювання енергоспоживання, розглянуто засоби моніторингу та оптимізації.

У першому розділі розглянуто теоретичні основи енергоспоживання мобільних пристроїв, технічні аспекти оптимізації програмного забезпечення та модельно-орієнтоване енергетичне профілювання.

У другому розділі досліджено методи та інструменти оптимізації енергоспоживання, класифікацію енергетичних профайлерів та засоби моніторингу енерговитрат мобільних застосунків.

У третьому розділі розглянуто реалізацію підходів до підвищення енергоефективності програмного забезпечення, методи зменшення енергоємності мобільних застосунків та результати тестування й бенчмаркінгу.

Висновок: застосування сучасних методологій оптимізації енергоспоживання дозволяє знизити навантаження на акумулятор мобільних пристроїв, підвищити продуктивність програмного забезпечення та покращити користувацький досвід при використанні мобільних застосунків.

КЛЮЧОВІ СЛОВА: МОБІЛЬНІ ЗАСТОСУНКИ, ЕНЕРГОСПОЖИВАННЯ, ЕНЕРГОЕФЕКТИВНІСТЬ, МОБІЛЬНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ОПТИМІЗАЦІЯ, ANDROID, IOS.

ANNOTATION

The thesis consists of 79 pages, 21 figures, 3 tables, and a reference list containing 40 sources.

The aim of the work is to study methodologies for optimizing energy consumption in software for mobile devices.

Research object: energy consumption processes of mobile software on modern mobile devices.

Research subject: methods, models, and tools for optimizing energy consumption of mobile applications, including energy profiling, software optimization mechanisms, and resource usage monitoring systems.

Research results: an analysis of modern methods of energy profiling for mobile applications was conducted, approaches to energy consumption modeling were studied, software tools for monitoring and optimizing energy costs were reviewed, and the effectiveness of various optimization methodologies was evaluated.

The first section describes the theoretical foundations of energy consumption in mobile devices, technical aspects of software optimization, and model-based energy profiling.

The second section analyzes methods and tools for energy consumption optimization, classification of energy profilers, and monitoring tools for mobile application energy usage.

The third section covers the implementation of approaches to improving software energy efficiency, methods for reducing the energy intensity of mobile applications, and the results of testing and benchmarking optimized solutions.

Conclusion: the application of modern energy optimization methodologies reduces battery consumption in mobile devices, improves software performance, and enhances user experience when using mobile applications.

KEY WORDS: MOBILE APPLICATIONS, ENERGY CONSUMPTION, ENERGY EFFICIENCY, ENERGY PROFILING, MOBILE SOFTWARE, OPTIMIZATION, BENCHMARKING, PROFILING, ANDROID, IOS.

.

ЗМІСТ

ВСТУП

У сучасному світі мобільні пристрої стали невід'ємною частиною повсякденного життя, забезпечуючи доступ до інформації, комунікації, фінансових сервісів, мультимедійних ресурсів та хмарних платформ. Постійне зростання функціональності мобільних застосунків супроводжується збільшенням навантаження на апаратні ресурси смартфонів і планшетів, що призводить до підвищеного енергоспоживання та швидкого розрядження акумуляторів. Сучасні виклики, пов'язані з обмеженим ресурсом батареї, необхідністю забезпечення високої продуктивності та підтримкою стабільної роботи мобільного програмного забезпечення, підкреслюють важливість створення ефективних методологій оптимізації енергоспоживання.

Актуальність теми зумовлена необхідністю підвищення енергоефективності мобільного програмного забезпечення шляхом застосування сучасних підходів до аналізу, профілювання та оптимізації використання ресурсів мобільних пристроїв. Існуючі мобільні застосунки часто характеризуються надмірним використанням процесорного часу, мережевих ресурсів, GPS-модулів та фонових сервісів, що негативно впливає на автономність пристроїв і комфорт користувачів. У контексті стрімкого розвитку мобільних технологій, Інтернету речей, хмарних сервісів та систем реального часу питання енергозбереження набуває особливого значення. Використання сучасних методів енергетичного профілювання, моделювання споживання ресурсів та оптимізації програмного коду дозволяє зменшити навантаження на батарею та підвищити загальну ефективність мобільних систем.

Метою роботи є дослідження методологій оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв, аналіз сучасних підходів до енергетичного профілювання та розробка рекомендацій щодо підвищення енергоефективності мобільних застосунків.

Завданнями дослідження є аналіз предметної області та існуючих підходів до оптимізації енергоспоживання мобільного програмного забезпечення, визначення основних факторів, що впливають на енерговитрати

мобільних пристроїв, аналіз інструментів моделювання енергоспоживання, реалізація підходів до оптимізації роботи мобільних застосунків, проведення тестування та бенчмаркінгу, а також оцінка ефективності запропонованих рішень.

Об'єктом дослідження є процеси енергоспоживання мобільного програмного забезпечення під час роботи на сучасних мобільних пристроях.

Предметом дослідження є методології, моделі, програмні засоби та інструменти оптимізації енергоспоживання мобільних застосунків, зокрема системи енергетичного профілювання, механізми моніторингу використання ресурсів, алгоритми оптимізації програмного коду та підходи до зменшення енергоємності мобільного програмного забезпечення.

Методи дослідження включають аналіз наукових джерел для вивчення сучасних підходів до енергоефективності мобільних систем, порівняльний аналіз інструментів енергетичного профілювання, моделювання процесів енергоспоживання, експериментальні методи тестування мобільних застосунків, а також статистичний аналіз результатів бенчмаркінгу.

У роботі передбачено дослідження методів оптимізації енергоспоживання шляхом зменшення навантаження на процесор, оптимізації використання мережевих ресурсів, зниження активності фонових процесів та використання сучасних механізмів управління ресурсами мобільних платформ Android та iOS. Особлива увага приділятиметься енергетичному профілюванню, аналізу продуктивності та оцінці ефективності оптимізаційних рішень, що сприятиме створенню більш енергоефективного мобільного програмного забезпечення. Очікується, що результати роботи дозволять підвищити автономність мобільних пристроїв, зменшити енергетичні витрати та покращити користувацький досвід.

Бакалаврська робота містить 79 сторінок, 21 рисунок, 3 таблиці, 3 розділи, список використаних джерел із 40 найменувань.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ЕНЕРГОСПОЖИВАННЯ МОБІЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основи енергоспоживання мобільних пристроїв та програмних систем

Оскільки мобільні додатки забезпечують дедалі складнішу функціональність, мобільні пристрої швидко витісняють персональні комп'ютери як основну обчислювальну платформу для зростаючої кількості користувачів [2]. Через обмеження акумулятора мобільних пристроїв, енергоефективність - підбір енергетичного бюджету та максимізація корисності додатків за заданих обмежень акумулятора - стала важливим фактором, що враховується при розробці та обслуговуванні програмного забезпечення [14]. Традиційно стурбований оптимізацією продуктивності та використання пам'яті, ідеальне обслуговування тепер має вирішувати проблеми оптимізації споживання енергії, оскільки існуючі методи ідеального обслуговування здебільшого не застосовуються.

Хоча сфера оптимізації енергоспоживання програмного забезпечення є широкою та різноманітною, існуючі рішення в основному зосереджені на апаратному, операційному та мережевому рівнях. На програмному рівні методом оптимізації енергоспоживання, який отримав значну увагу дослідницької спільноти, є *розвантаження хмари* [4], [1], [17], [8]. Ця оптимізація розділяє мобільні додатки таким чином, щоб їхня енергоємна функціональність виконувалася в хмарі, не розряджаючи акумулятор. Існуючі методи розвантаження хмари статично визначають енергоємну функціональність та відповідно розділяють мобільні додатки. Однак, щоб досягти максимальної користі, розвантаження хмари повинно враховувати апаратні можливості мобільного пристрою, на якому працює додаток, а також характеристики мобільної мережі, що з'єднує мобільний пристрій з хмарою. Іншими словами, рішення щодо розвантаження повинні прийматися динамічно під час виконання та постійно коригуватися у відповідь на коливання в мобільному середовищі виконання. Через брак такого рівня гнучкості

існуючі схеми розвантаження не застосовні як загальний підхід до оптимізації енергоспоживання для ідеального обслуговування.

У цій роботі ми представляємо новий підхід до розвантаження, який поєднує переваги попереднього рівня техніки як у розділенні мобільних додатків, так і в динамічній адаптації мобільних цілей виконання у відповідь на коливання мережевих умов. Наш підхід реалізовано як два основних технічних рішення: (1) перетворення програми розвантаження для кількох цілей, яке автоматично перезаписує централізовану програму в розподілену, локальний/віддалений розподіл якої визначається динамічно під час виконання; (2) система виконання, яка визначає необхідний локальний/віддалений розподіл результуючої розподіленої програми на основі поточного середовища виконання. Поєднання цих двох рішень може ефективно зменшити кількість енергії, що споживається мобільними додатками, без необхідності змінювати їхній вихідний код вручну, таким чином оптимізуючи їх за лаштунками.

З точки зору процесу обслуговування, наш підхід працює наступним чином [3]. Програміст обслуговування позначає методи, які підозрюються як гарячі точки споживання енергії. Потім серія методів аналізу програми перевіряє введені програмістом дані та автоматично переписує застосунок у розподілений застосунок, локальні та віддалені частини якого можна гнучко визначати під час виконання. Гнучкість у визначенні шаблонів розподілу під час виконання забезпечується завдяки складному механізму контрольних точок. Залежно від середовища виконання, різні частини стану програми можуть бути контрольно-пропускними та передані по мережі відповідно до вимог чинної стратегії розвантаження. Стратегія розвантаження визначається системою виконання, обов'язки якої включають: (1) керування з'єднанням між клієнтом і сервером, (2) постійну оцінку енергії, споживаної мобільним пристроєм, (3) розрахунок того, яку стратегію розвантаження слід дотримуватися, (4) синхронізацію контрольно-пропускного стану, що передається між сервером і клієнтом, та (5) забезпечення стійкості у разі збою через відключення мережі.

У наших експериментах ми застосували наш підхід для оптимізації

споживання енергії сторонніми, реальними Android-додатками. Усі досліджувані додатки не тільки зменшили споживання енергії, але й зберегли свої початкові характеристики продуктивності. Наш адаптивний, багатоцільовий підхід до розвантаження хмари може ефективно зменшити кількість споживаної енергії. Зокрема, наші бенчмарки та тематичні дослідження демонструють, що наш підхід може зменшити загальний енергетичний бюджет типового мобільного додатку на величину від 25% до 50%.

Виходячи з наших результатів, технічний внесок цієї роботи є наступним:

1) Трансформація програми розвантаження з кількома цільовими завданнями що дозволяє відкласти до початку виконання рішення про те, які частини програми слід виконувати локально чи віддалено.

2) Адаптивна система виконання розвантаження хмари, яка визначає - оптимальні стратегії розвантаження для розділених програм.

3) Емпірична оцінка багатоцільового розвантаження

що демонструє ефективну техніку зменшення споживання енергії реальними мобільними додатками сторонніх розробників.

Програмне забезпечення містить послідовність інструкцій, виконання яких вимагає від пристрою, на якому вона працює, споживання енергії. Сьогодні споживання енергії, нефункціональна властивість програми, рідко розглядається заздалегідь як нефункціональна вимога або, після цього, як властивість, яку потрібно вимірювати та контролювати. Однак споживання енергії може становити критичну проблему для кінцевих користувачів. У ноутбуках, планшетах і смартфонах споживання енергії явно впливає на час роботи від батареї і, отже, стає проблемою взаємодії з користувачем [4]. Для центрів обробки даних або майнерів біткойнів [2] споживання енергії безпосередньо впливає на рахунок за електроенергію. У літературі багато хто розглядав проблему вимірювання та зменшення споживання енергії, але зазвичай це було зроблено ситуативно [3].

Згідно з підходом до розробки програмного забезпечення на основі доказів (EBSE) [5], конкретне прийняття рішень повинно бути підкріплене емпіричними доказами, доступними в літературі. Такі докази повинні бути

достовірними, отриманими за допомогою документованого та повторюваного процесу, контекстуалізованими та пов'язаними з контекстом, де їх можна застосувати, ідентифікованими, відповідати на чітко визначене питання, оцінюваними та повідомляти про відомі обмеження результатів. Такі характеристики рідко присутні в більшості відповідної опублікованої літератури.

Якщо проблема споживання енергії вирішується на апаратному рівні, то завдання досягається шляхом зменшення споживання фізичними пристроями або створенням різних профілів використання (наприклад, процесори можуть знижувати частоту, коли вони використовуються менше). З іншого боку, якщо проблема споживання енергії вирішується на рівні операційної системи, то політики управління можуть використовувати різні апаратні профілі різних пристроїв (якщо вони доступні) або вимикати обладнання, коли воно не потрібне. Ми розглядаємо програмне забезпечення як фактор споживання енергії, оскільки воно вимагає виконання кількох дій базовим обладнанням, яке реагує на основі отриманих інструкцій. Вимірювання споживання енергії певним програмним забезпеченням передбачає вирішення двох основних питань :

- Ізоляція споживання енергії програмою, коли вона працює одночасно з іншими на тому ж пристрої
- Узагальнення отриманих результатів: нехай міра буде значущою для інших пристроїв

Під час збору даних про енергію за допомогою фізичних вимірювань на пристрої значення пов'язане з цільовим програмним забезпеченням та всіма іншими процесами, що одночасно працюють на пристрої. Фізичне вимірювання не дозволяє прямого узагальнення результатів, оскільки одне й те саме програмне забезпечення може поводитися по-різному залежно від апаратного забезпечення, на якому воно виконується, а також від іншого встановленого програмного забезпечення. Іншим варіантом є визначення моделей, які надають оцінку споживання енергії цільовим програмним забезпеченням, замість виконання фізичного вимірювання. Вхідні дані моделі складаються з показників використання ресурсів пристрою, зібраних під час виконання. Основна проблема, що впливає на цей

підхід, полягає в тому, що модель може бути репрезентативною для пристрою або сімейства пристроїв, що означає, що оцінка, обчислена моделлю, не завжди є достовірною. На жаль, отримати такий результат дуже важко, оскільки кожен виробник обладнання повинен надавати точні дані про споживання пристрою, і ці дані повинні бути доступні в режимі реального часу як інформація про стан пристрою через датчики та системні виклики з операційної системи. Тепер ми можемо легко виміряти споживання енергії програмою, вимірявши споживання енергії всім пристроєм, на якому вона виконується, проаналізувавши отримані дані та оцінивши споживання, мінімізуючи похибку [6]. Це вимагає точної методології для отримання найважливіших даних та їх аналізу на предмет корисної інформації для оцінки споживання енергії застосунком.

1.2 Аналіз проблем енергоефективності мобільних застосунків та технічні аспекти оптимізації

У цьому підрозділі ми надаємо технічну інформацію як про проблему, яку має вирішити наш підхід, так і про основні технології, що використовуються в нашому підході.

A. Визначення проблеми

Робота, представлена тут, мотивована висновками, які ми нещодавно отримали в результаті експериментального дослідження впливу механізмів розподіленого програмування на енергоефективність [9]. Виявивши, що наявні мережеві умови можуть суттєво впливати на те, скільки енергії витрачається на передачу тих самих даних, ми створили рекомендації для розробників мобільних додатків щодо передачі даних таким чином, щоб споживати найменшу кількість енергії для даної мобільної мережі. Оскільки передача тих самих даних через мережі WiFi, 3G або 4G споживає різну кількість енергії, оптимальний механізм розвантаження повинен бути адаптивним, передаючи різну кількість станів залежно від наявних мережевих умов.

На рисунку 1.1 показано графік споживання енергії для розвантаження

хмари (co-EcG), нову структуру даних для аналізу програм, яку ми ввели для моделювання споживання енергії за різних сценаріїв розвантаження [7]. Вузли CO-EcG представляють компоненти програми, інкапсульовані одиниці функціональності, які можна вивантажити в хмару. Кожен вузол позначено приблизною кількістю енергії, споживаної процесором для виконання функціональності компонента та його наступних компонентів на графіку. Ребра представляють зв'язок між компонентами, причому мітки показують, скільки енергії споживатиме мобільний пристрій для передачі даних між підключеними компонентами.

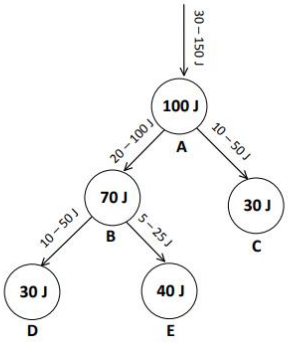


Рисунок 1.1 - Графік споживання енергії розвантаженням хмари.

У цій конкретній CO-ECG компонент A споживає приблизно 100 джоулів під час типового виконання, таким чином стаючи життєздатним кандидатом для вивантаження в хмару. Оскільки компонент A викликає компоненти C та B, які, у свою чергу, викликають компоненти D та E, його споживання енергії дорівнює сумі енергії, споживаної всіма наступними компонентами на графіку. Ми припускаємо, що енергія, витрачена на виконання вивантаженої функціональності в хмарі, є безкоштовною, оскільки вона не виснажує заряд акумулятора мобільного пристрою. Якщо компонент A вивантажено, то передача необхідних даних до нього через мережу, що дозволяє йому виконуватися віддалено, споживатиме від 30 до 150 джоулів залежно від типу мережі, доступної для передачі даних. Іншими словами, за певних мережевих умов вивантажене виконання зрештою споживатиме більше енергії, ніж виконання компонента A на мобільному пристрої [8]. Оскільки тип доступної мережі відомий лише під час виконання,

рішення щодо вивантаження повинні бути динамічними, щоб мати змогу оптимізувати кількість споживаної енергії за всіх умов виконання.

У цій роботі ми представляємо рішення, яке може вирішити проблему, обговорену вище. Ми називаємо наше рішення *адаптивним багатоцільовим розвантаженням хмари* (AMCO). Спеціальне перетворення програми створює розподілений клієнт/серверний застосунок, функціональність клієнта та сервера якого визначається динамічно під час виконання. Як конкретний приклад використання CO-ECG, описаного вище, під час роботи через мережу 3G компоненти C та D можуть бути розвантажені, тоді як під час роботи через мережу 4G може бути розвантажений лише компонент E. Нарешті, під час роботи через Wi-Fi можуть бути розвантажені компоненти A або B.

Точніше, позначення методу як гарячої точки енергії створює специфікацію розвантаження, в якій різні частини графа викликів, що кореняться в позначеному методі, можуть бути вивантажені відповідно до умов виконання. Оскільки для передачі даних через обмежені мережі потрібно більше енергії, оптимальна стратегія розвантаження повинна обмінювати енергію, потенційно зекономлену шляхом переміщення виконання в хмару, на енергію, споживану шляхом переміщення даних (тобто стану програми) для підтримки розвантаження. Наша трансформація програми дозволяє розвантажити будь-який підграф CO -ECG, тоді як наша система виконання запускає найвигідніше (тобто економить найбільше енергії батареї) розвантаження для середовища виконання.

B. Технічна довідка

Технічні концепції, що лежать в основі нашого підходу, включають розвантаження хмари, моделі споживання енергії та аналіз програм. Далі ми опишемо ці технології по черзі .

1) Розвантаження хмари: Розвантаження хмари – це метод оптимізації мобільних застосунків, який дозволяє виконувати енергоємні функції застосунку в хмарі, не розряджаючи акумулятор мобільного пристрою. Розвантаження хмари зазвичай реалізується як перетворення розділення програми, яке розділяє мобільний застосунок на дві частини: клієнт, що працює на мобільному пристрої,

та сервер, що працює в хмарі; весь зв'язок між частинами здійснюється через механізм проміжного програмного забезпечення, такий як XML-RPC. Таким чином, розвантаження хмари – це окремий випадок автоматизованого розділення програми – розподілу централізованої програми для запуску по мережі за допомогою інструменту на основі компілятора, перетворення централізованої програми [22] або міграції виконання між різними образами застосунків на рівні ОС [17], [2]. Перспективність розвантаження хмари демонструється поширенням конкуруючих підходів до цієї техніки в літературі. CloneCloud [1] розвантажує виконання на рівні потоків, тоді як Cloudlet [17] розвантажує на рівні віртуальної машини. MAUI [4] розвантажує за допомогою розділення застосунку на рівні методів. У нашій попередній роботі з розвантаження хмари [8] ми розділяли програми, не порушуючи їхню здатність виконуватися локально. Усі ці попередні методи розвантаження хмари мають спільну мету зменшення споживання енергії мобільними пристроями. Підхід, представлений у цій роботі, використовує багато з вищезазначених методів для автоматичного перетворення мобільних програм без будь-яких змін у їхньому вихідному коді та для синхронізації станів програм між розділами. Однак метою нашого підходу є підвищення ефективності попереднього методу розвантаження хмари шляхом відкладання рішень щодо розвантаження до часу виконання, коли механізм зворотного зв'язку зможе визначити, який обсяг розвантаження є оптимальним.

2) *Моделі споживання енергії в мобільних додатках:* Мережевий зв'язок є одним з найбільших джерел споживання енергії в мобільному додатку [15], [9]. Згідно з нещодавнім дослідженням, мережевий зв'язок споживає від 10 до 50% загального енергетичного бюджету типового мобільного додатку [10]. Зокрема, в нашому попередньому дослідженні [9] ми виміряли та проаналізували, як проміжне програмне забезпечення може суттєво впливати на споживання енергії мобільним додатком. Наші експерименти оцінювали споживання енергії для передачі різних обсягів даних через мережі з різними характеристиками затримки/пропускної здатності. Потім ми виділили, як мобільні додатки споживають енергію, щоб визначити їхні загальні моделі споживання енергії.

Експериментальні результати та систематичний аналіз, проведений в рамках цього дослідження, надихнули нас на розпочаття роботи, представленої в цій роботі.

3) Аналіз програми: Аналіз програми кодифікує набір методів для виведення різних фактів про вихідний код, які будуть використані для оптимізації та перетворення. Аналіз ієрархії класів (CHA) будує граф викликів в об'єктно-орієнтованих мовах. Аналіз потоку даних визначає, як змінні програми призначаються одна одній [18]. Аналіз без побічних ефектів [16] визначає, чи змінює метод купу програми. У цій роботі ми використовуємо CHA для обчислення функціональності для вивантаження та стану програми для передачі для заданого вивантаження. Щоб вибрати оптимальні стратегії вивантаження, ми поєднуємо аналіз потоку даних та побічних ефектів. На основі результатів, покращувач байт-коду потім переписує програму без зміни її вихідного коду. Ми використовували Soot [23] для реалізації нашого аналізу та перетворень програми.

1.3 Модельно-орієнтоване енергетичне профілювання програмного забезпечення

Для профілювання споживання енергії системою, її підкомпонентами або конкретними програмами необхідні моделі живлення. Профільувальник енергії на основі моделі надає певний профіль споживання енергії мобільним пристроєм, однією з його підсистем або програмним забезпеченням, що працює на ньому. Вимірювання потужності є невід'ємною частиною побудови моделі живлення і може бути реалізовано альтернативними способами [11]. У цьому підрозділі ми описуємо структуру профіліста енергії та пов'язані з нею дії, включаючи моделювання та вимірювання потужності.

Основна термінологія

Література багата на різні види рішень для профілювання енергії. Ці рішення зазвичай представлені як унікальні інструменти, хоча насправді вони є комбінаціями різних видів базових методів. Щоб глибше зрозуміти ці рішення,

важливо визначити ключові компоненти кожного профілера енергії та проаналізувати зв'язок між ними. У літературі термінологія не використовується послідовно. Наприклад, у деяких випадках межа між вимірюваннями потужності та профілюванням на основі моделі є тонкою, хоча певний підхід до оцінки споживання енергії можна описати за допомогою будь-якого з цих двох термінів. Отже, важливо чітко визначити термінологію, яку ми використовуємо:

Вимірювання потужності - це процес отримання значень споживання потужності (або струму) за допомогою спеціального обладнання. Він не передбачає жодних моделей, реалізованих у програмному забезпеченні, та калібрування. Найпоширенішим прикладом є вимірювання потужності за допомогою зовнішнього приладу, такого як Monsoon Power Monitor, підключеного до інтерфейсу акумулятора смартфона. Іншим прикладом є пряме вимірювання струму з інтелектуального інтерфейсу акумулятора смартфона, реалізованого зі спеціальною вбудованою електронікою. Наприклад, Nokia Energy Profiler (NEP) використовує цей підхід для отримання значень споживання струму.

— *Модель енергоспоживання* - це математичне представлення споживання енергії. Це функція змінних, яка кількісно визначає фактори, що впливають на споживання енергії, такі як використання апаратного підкомпонента та кількість пакетів, що передаються через бездротовий мережевий інтерфейс, з бажаним споживанням енергії на виході. Зазвичай значення цих змінних можна безпосередньо отримати з вимірювань, проведених на смартфоні або в мережі [12]. Модель енергоспоживання може характеризувати окрему підсистему, їх комбінацію або навіть цілий смартфон (модель системного рівня). Простим прикладом моделі енергоспоживання підсистеми є грубозерниста модель енергоспоживання дисплея, яка є функцією однієї змінної: рівня яскравості.

—*Оцінка енергоспоживання* повідомляє про споживання енергії смартфоном або його підсистемою на основі моделі(моделей) енергоспоживання. Точність оцінок енергоспоживання залежить від точності використовуваної моделі енергоспоживання.

— *Профілерівник потужності/енергії* - це система, яка характеризує

потужність та/або споживання енергії смартфоном. Ми відрізняємо профілювання від вимірювання потужності, вказуючи, що профілерівник за визначенням спирається на моделі потужності. Отже, профілерівник надає оцінки потужності або енергії, на відміну від вимірювань потужності, які можна отримати за допомогою монітора потужності. Крім того, різні профілерівники працюють на різних рівнях абстракції, таких як система, програма, процес або завдання, тоді як вимірювання потужності надає інформацію лише про споживання енергії вимірюваним обладнанням , найчастіше всім смартфоном.

Побудова енергетичного профілографа

На рисунку 1.2 показано процес побудови профілографа енергії на основі моделі. На рисунку процес поділено на чотири фази. Процес починається з вибору експертом методу моделювання, використовуючи свої знання з предметної області. Після цього йде етап фактичного моделювання потужності, на якому вибираються змінні та навчаються моделі за допомогою вимірювань потужності та системних журналів. Системні журнали стосуються фактичних змінних, що використовуються в моделях потужності, а навчання, по суті, полягає в обчисленні коефіцієнтів змінних моделі. На цих фазах зазвичай застосовуються методи статистичного навчання.

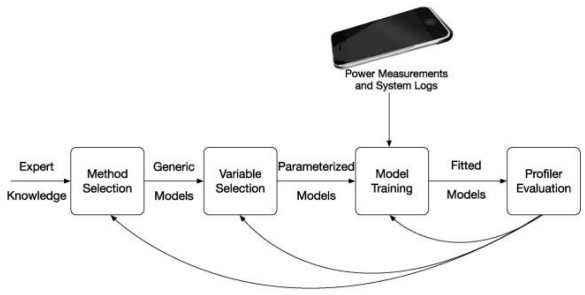


Рисунок 1.2 - Процес профілювання енергії або потужності включає моделювання, оцінку та цикл зворотного зв'язку через валідацію та верифікацію для уточнення та повторного калібрування моделей.

Потім моделі потужності об'єднуються у власне профайлер, який контролює змінні моделі та надає оцінки споживання енергії. Профайлер оцінюється за

допомогою додаткових вимірювань потужності, які порівнюються з оцінками потужності для характеристики їхньої точності. Цей етап надає цінний вхід експерту щодо процесу генерації та використання моделі [13]. Відповідно до вхідних даних, вибір підходу до моделювання та/або вибір змінних може бути переоцінений, тоді як модель може бути перенавчена за допомогою більш повного набору навчальних даних.

Спочатку ми обговоримо два альтернативні способи вимірювання загального енергоспоживання смартфона: за допомогою зовнішніх приладів та за допомогою самовимірювання. Зібрані вимірювання є частиною вхідних даних для тренувальних моделей енергоспоживання смартфона.

Вимірювання потужності за допомогою зовнішніх приладів

Зовнішні прилади, такі як Monsoon Power Monitor, можна використовувати для безпосереднього вимірювання струму, що споживається смартфоном. Інший спосіб використання зовнішнього приладу – підключити вольтметр до резистора, який послідовно з'єднаний з джерелом живлення смартфона, що дозволяє обчислювати струм за зміною виміряної напруги. Також можна виконувати ці вимірювання з оригінальним акумулятором як джерелом живлення або за допомогою зовнішнього джерела живлення. У першому випадку вплив акумулятора, такий як вплив стану заряду (SOC), враховується у вимірюванні. Зовнішні монітори живлення є золотим стандартом для аналізу живлення мобільних пристроїв завдяки своїй високій точності та належності. Вони обмежені вимогами лабораторних налаштувань і тому непридатні для масштабного розгортання.

Самовимірювання

Другий підхід називається *самовимірюванням,* що означає, що смартфон оснащений достатніми можливостями для визначення споживання енергії на рівні системи без допомоги зовнішніх приладів.

Моделі акумуляторів, напруга та стан заряду. Давайте спочатку коротко представимо три показники вимірювання, пов'язані з характеристиками акумулятора, перш ніж детальніше розглянемо різні підходи до само вимірювання [14]. До показників належать напруга на клемах , напруга розімкнутого кола (OCV) та

стан заряду (SOC). Напруга на клемах – це вимірювана напруга акумулятора на його клемах, тоді як OCV визначає напругу акумулятора без навантаження. Напруга на клемах ($Vt_{)}$ падає і, отже, відрізняється від OCV *(Vocv* $_{)}$, коли струм споживається з акумулятора, через його внутрішній омічний опір R, наступним чином:

$$V_t = V_{ocv} - I \cdot R \quad (1.1)$$

Попередня модель також відома як модель акумулятора Rint, а еквівалентна схема зображена на рисунку 1.3. Порівняно з цією моделлю, модель Тевеніна вводить паралельну RC-мережу послідовно поверх моделі Rint, як показано на рисунку 1.3. Відповідно, модель Тевеніна складається з OCV, внутрішніх опорів та еквівалентної ємності. Внутрішні опори включають омічний опір R та поляризаційний опір r . Ємність *CTh* описує перехідну характеристику акумулятора під час заряджання та розряджання. V_c – це напруга на C_{Th}.

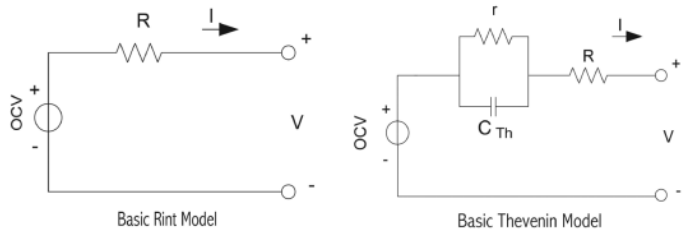


Рисунок 1.3 - Моделі оцінки OCV.

виражено як:

$$V_t = V_{ocv} - V_c - I \cdot R \quad (1.2)$$

Попередні моделі показують, що точне значення OCV навряд чи можна виміряти на увімкненому смартфоні; однак виміряна напруга на клемах близька до неї, коли всі апаратні компоненти перебувають у режимі низького енергоспоживання.

Стан заряду (SOC) визначає поточний стан заряду акумулятора або залишок ємності акумулятора у відсотках. Наприклад, 0% означає порожній акумулятор, а 100% – повністю заряджений. Альтернативний спосіб вираження залишку ємності – використання стану розряду (SOD), де 100% означає порожній акумулятор, а 0% – повністю заряджений. Однак, здатність визначати стан розряду (SOC) корисна для користувачів смартфонів, оскільки вони знають, коли акумулятор пристрою потрібно зарядити [15]. У контексті профілювання енергії SOC відіграє ще одну важливу роль, забезпечуючи засіб для оцінки середнього розсіювання струму при постійному навантаженні. Це робиться для того, щоб SOC записувався до та після застосування певного навантаження до смартфона. Зміна SOC, разом зі знанням ємності акумулятора, безпосередньо дає кількість енергії, споживаної телефоном протягом інтервалу вимірювання, з якого можна розрахувати середній струм, споживаний пристроєм під час перебування під певним навантаженням, оскільки відомі тривалість та OCV.

Оцінка стану заряду. Зазвичай, рівень заряду (SOC) не може бути виміряний безпосередньо. Натомість існує два підходи до його оцінки: (i) метод на основі напруги та (ii) кулонівський підрахунок. Перший метод використовує будь-яку з моделей акумуляторів, розглянутих у попередньому розділі, та перетворює напругу на клемах на рівень заряду (SOC) за допомогою таблиць пошуку OCV. Зазвичай використовується так звана крива розряду акумулятора для вираження зв'язку між OCV та залишковою ємністю, коли акумулятор розряджається з часом. Ця крива розряду завжди є суворо спадною функцією. За певного значення OCV можна визначити SOC. Недоліком є те, що відображення змінюється залежно від властивостей акумулятора, таких як модель та вік, а це означає, що для точної оцінки SOC для кожного пристрою необхідно створювати персоналізовану криву розряду та періодично оновлювати її.

Метод кулонівського підрахунку визначає стан заряду (SOC) шляхом накопичення струму, споживаного системою з часом. Цей метод вимагає можливості безпосередньо вимірювати струм, через що обговорювати його тут недоцільно. Причина полягає в тому, що пристрій, який має можливість вимірювати

струм, споживаний від акумулятора, може використовувати його лише для оцінки SOC і не надає миттєвий струм застосункам через ОС. Тому застосунок для профілювання енергії в такому випадку повинен покладатися на оцінку потужності на основі SOC. У цьому випадку SOC можна розрахувати як

$$\mathrm{SOC} = \mathrm{SOC_{init}} - \int \left(\frac{\mathrm{I_{bat}}}{\mathrm{C_{usable}}}\right) \mathrm{dt} \qquad (1.3)$$

де C можна використовувати залежить від зменшення ємності акумулятора через вік, температуру та цикли заряджання, а також втрат через бездіяльність акумулятора.Оскільки стан заряду (SOC) оцінюється, підходи на основі напруги на клемах, OCV та кулонівського підрахунку можуть мати похибки, а профайлери, що покладаються на SOC, також успадковують цю похибку.

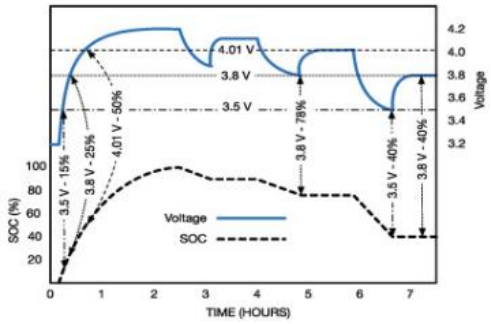


Рисунок 1.4 - Миттєве значення напруги не корелює зі ступенем заряду (SOC)

У випадку оцінки стану заряду (SOC) на основі напруги, похибка оцінки може бути значною, оскільки напруга акумулятора змінюється залежно від навантаження, температури та віку. Модель Rint для OCV не враховує перехідний характер літій-іонних акумуляторів, і тому похибка може бути значною при динамічному навантаженні системи.

Через динамічне навантаження, яке виникає в системі, залишається ймовірність помилки навіть за допомогою складної таблиці пошуку напруги та навантаження. Наприклад, на рисунку 3 ми бачимо, що напруга акумулятора 3,81 В

виникає на 25%, 78% та 40% від рівня заряду (SOC). На рисунку 4 показано аналогічний приклад оцінки SOC на основі OCV. Ми бачимо, що OCV представляє кілька SOC під час заряджання (15%) та розряджання (90%), і, отже, помилка SOC буде значною, якщо не ведуться окремі таблиці пошуку OCV для заряджання та розряджання.

У випадку кулонівського підрахунку існує два джерела помилки. По-перше, існує помилка накопичення струму зміщення [16]. Струм зміщення є результатом вимірювання струму та аналого-цифрового перетворення. Ця помилка накопичення збільшується з часом і сприяє неточності SOC, якщо її не компенсувати якимось методом релаксації напруги, таким як тривале перебування пристрою в режимі очікування. По-друге, існує помилка оцінки корисної ємності, яка пов'язана з віком акумулятора, температурою та швидкістю заряджання або розряджання. Традиційними підходами до оцінки корисної ємності є цикл заряджання та таблиці пошуку з урахуванням температур і швидкостей.

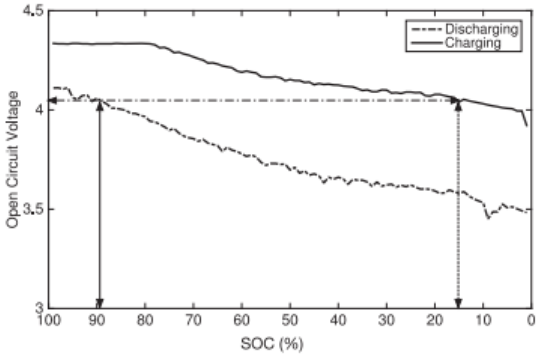


Рисунок 1.5 - Значення напруги відкритого ланцюга (OCV) смартфона Samsung Galaxy під час заряджання та розряджання не корелюють із рівнем заряду акумулятора (SOC)

API індикатора рівня палива та акумулятора. Стан заряду батареї (SOC) оцінюється апаратним компонентом мобільного пристрою, який називається *індикатором рівня палива.* Індикатори рівня палива на основі напруги зчитують напругу акумулятора та прості у впровадженні [17]. З іншого боку, індикатори рівня палива на основі кулонівських лічильників повинні бути оснащені

чутливим резистором у шляху заряду/розряду. Під час протікання струму аналого-цифровий перетворювач (АЦП) зчитує напругу на чутливому резисторі, а потім передає значення струму на лічильник. Інформація про час надається лічильником реального часу для інтегрування струму з кулонівськими лічильниками. Найновіші телефони, такі як Nexus 6/9, використовують індикатори рівня палива на основі кулонівських лічильників.

З точки зору програми профілювання енергії, інформація про самовимірювання надається через API батареї смартфона, що є способом для ОС надавати інформацію про стан батареї, наприклад, BatteryManager в Android. Точна інформація, що надається API батареї, залежить від моделі пристрою та типу індикатора рівня палива. У деяких випадках API може безпосередньо надавати інформацію про споживання струму, напругу батареї та температуру. API періодично оновлює цю інформацію та щоразу, коли відбувається зміна рівня заряду батареї залежно від індикатора палива. Частота оновлення варіюється від 0,25 Гц до 4 Гц. Пізніше ми побачимо, як профілери, що використовують підхід самовимірювання, використовують напругу, струм або рівень заряду батареї з цих оновлень.

1.4 Висновок по розділу

У першому розділі було розглянуто теоретичні основи оптимізації енергоспоживання мобільного програмного забезпечення. Проаналізовано особливості енергоспоживання мобільних пристроїв та фактори, що впливають на ефективність роботи програмних систем. Досліджено проблеми енергоефективності мобільних застосунків і визначено основні технічні аспекти їх оптимізації. Також розглянуто модельно-орієнтоване енергетичне профілювання програмного забезпечення як засіб аналізу та прогнозування енергоспоживання. Отримані результати формують теоретичну основу для подальшого дослідження методів і засобів оптимізації енергоефективності мобільних застосунків.

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ОПТИМІЗАЦІЇ ЕНЕРГОСПОЖИВАННЯ

2.1 Методології моделювання та оцінювання енергоспоживання мобільних застосунків

У цьому підрозділі ми детальніше розглянемо різні підходи до моделювання енергоспоживання смартфонів.

Типи моделей живлення

Підходи до моделювання енергоспоживання смартфона можна умовно розділити на три категорії залежно від типу вхідних змінних, які використовує модель: моделі на основі використання, моделі на основі подій та моделі на основі аналізу коду.

1. *Моделі на основі використання.* Моделі на основі використання враховують споживання енергії підкомпонентом шляхом кореляції споживання енергії цим компонентом з виміряним споживанням ресурсів. Для програми або процесу модель енергоспоживання включає змінні, що відображають споживання ресурсів усіма різними підкомпонентами, активними під час роботи цієї програми.

Гарним прикладом підходу, заснованого на використанні, є широко використовуваний метод моделювання споживання енергії обчислювальною підсистемою за допомогою апаратних лічильників продуктивності (HPC). Такий метод використовує той факт, що сучасні мікропроцесори виявляють свою внутрішню активність через кілька лічильників подій, таких як інструкції на цикл (IPC), лічильники вибірок, лічильники промахів/влучань та зупинки. Ідея полягає в тому, що кількість енергії, необхідної для виконання програмного забезпечення, пропорційна кількості активності, яка відбувається всередині мікропроцесора [18]. Опираючись на лічильники для моделювання споживання енергії як процесором, так і пам'яттю. Для моделювання споживання енергії пам'яттю автори розглядали лічильники промахів кешу та зупинок. Охарактеризовано поведінку

енергоспоживання комерційної ОС для широкого спектру застосувань з використанням IPC. Зпропонували модель енергоспоживання з використанням лічильників для процесора AMD Phenom. Дослідили використання лічильників для оцінки споживання енергії підсистемами поза процесором. Приклад моделювання енергоспоживання мобільних пристроїв на основі високопродуктивних обчислювальних систем.

Моделі на основі подій. Моделі потужності на основі використання добре фіксують лінійні залежності між споживанням ресурсів та енергії моделюваним обладнанням. Однак деякі апаратні компоненти смартфонів демонструють нелінійні характеристики споживання енергії. Така поведінка часто характеризується концепцією *хвостової енергії* , яка стосується того факту, що певний елемент обладнання залишається ввімкненим протягом деякого часу після того, як він більше не активно використовується. Наприклад, бездротові радіостанції зазвичай залишаються ввімкненими протягом певного часу, заданого таймером, після передачі або отримання останнього біта. З цієї причини було прийнято моделювання на основі подій [19]. Події дозволяють точнішу характеристику потужності в певних випадках, коли підхід, заснований на використанні, не працює належним чином. Очевидно, що також можна використовувати поєднання цих двох підходів.

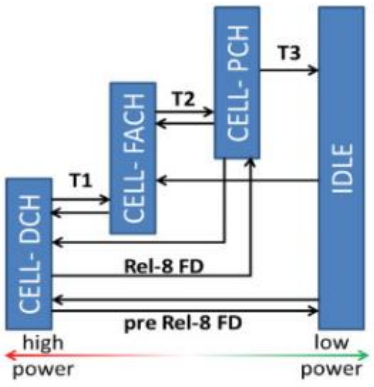


Рисунок 2.1 - Етапи стільникової мережі HSPA, що беруть участь у передачі даних.

Типовим прикладом підходу до моделювання на основі подій є той, який будує модель живлення на основі системних викликів. Відстеження системних викликів забезпечує чітке уявлення про те, які апаратні компоненти використовуються певною програмою або процесом і яким чином. Наприклад, завдання, пов'язані з вводом/виводом, доступні через системні виклики читання та запису.

Тривалість активного стану живлення відповідних апаратних підкомпонентів залежить від обсягу вводу/виводу, які задаються в параметрах цих системних викликів. Після завершення фактичних завдань вводу/виводу енергію хвоста можна оцінити за допомогою системних викликів, подібних до близьких. Розглянуто поведінку енергії хвоста різних апаратних компонентів.

Як інший приклад, стани протоколу керування радіоресурсами (RRC) у стільниковому зв'язку 3G та переходи між станами представлені на рисунку 2.1. На рисунку видно, що споживання енергії є найвищим у стані CELL_DCH. Час, необхідний для перебування в кожному стані, залежить від тривалості таймера та активності даних у даний момент [20]. Тривалість таймерів фактично залежить від того, як мережеве обладнання налаштоване оператором. Якщо підтримується швидкий стан спокою (FD), пристрій безпосередньо перемикатиметься зі стану CELL_DCH у стан CELL_PCH або IDLE після закінчення часу таймера FD.

2. *Моделі, засновані на аналізі коду.* Третя категорія моделей спирається на аналіз програмного коду, який має бути виконаний. Перевагою цього підходу є те, що він може оцінити споживання енергії, навіть не запускаючи програмне забезпечення на реальній системі. Цей підхід використовується рідше, оскільки споживання енергії часто залежить від контексту, що важко врахувати без фактичного запуску програмного коду на реальному пристрої. Наприклад, погана якість бездротового зв'язку, яка подовжує час передачі файлу та впливає на споживання енергії, може бути зафіксована за допомогою моделі на основі утиліт, яка відстежує передані біти як функцію часу, але не за допомогою аналізу коду. Прикладом підходу до аналізу коду є модель на рівні інструкцій, яка працює, пов'язуючи споживання енергії програмним забезпеченням з кожною виконуваною інструкцією та вимагає оцінки розсіювання потужності для кожної з

інструкцій розглянутого програмного забезпечення. Подібний метод можна застосувати для функції, процедури або підпрограми.

Моделювання білої скриньки проти чорної скриньки

Підходи до моделювання можна додатково розрізнити за обсягом доступної апріорної інформації про апаратний компонент, що моделюється. Чисто метод чорної скриньки, як випливає з назви, не має апріорної інформації про поведінку споживання енергії апаратним компонентом, тоді як у випадку моделювання білої скриньки ця поведінка добре вивчена.

Моделювання енергоспоживання "білого ящика" зазвичай фіксує поведінку енергоспоживання апаратного забезпечення за допомогою кінцевих автоматів (FSM), які описують стани енергоспоживання системи та переходи між ними. Цей підхід вимагає точного розуміння поведінки енергоспоживання апаратного забезпечення. Зокрема , події, які запускають перехід з одного стану енергоспоживання в інший, повинні бути відомі та добре зрозумілі. Навчання моделі не вимагається, окрім простого вимірювання абсолютного енергоспоживання апаратного забезпечення в різних станах енергоспоживання. Цей підхід добре працює при моделюванні енергоспоживання підсистем бездротового зв'язку смартфона. Енергоспоживання радіоприймача можна досить точно абстрагувати, використовуючи простий набір станів енергоспоживання, які відповідають частці часу, коли радіоприймач повністю увімкнений. Переходи між станами енергоспоживання залежать від використовуваних протоколів канального рівня, але зазвичай вони ініціюються таймерами неактивності та порогами, пов'язаними зі швидкістю передачі даних.

На відміну від цього, моделювання чорної скриньки завжди вимагає навчання моделі [21]. Основним методом є регресійний аналіз, і, зокрема, лінійна регресія. Модель, заснована на регресійному аналізі, фіксує зв'язок між вхідним робочим навантаженням, описаним за допомогою регресійних змінних, та споживанням енергії. Як правило, доступні деякі апріорні знання, які допомагають, наприклад, вибрати регресійні змінні. Лінійна регресія є природним вибором при побудові моделі енергоспоживання на основі комунальних послуг, де

використання процесора, яскравість екрана та мережева активність можуть бути фіксовані за допомогою регресійних змінних. Вона проста та ефективна, але також обмежена припущенням лінійності, але перетворення іноді можуть подолати це обмеження [22]. Прикладами профайлерів, які використовують лінійну регресію, є DevScope та V-edge. Існують деякі профайлери, які виконують моделювання як білої, так і чорної скриньки, такі як ті, що запропоновані

2.2 Класифікація та аналіз енергетичних профайлерів на основі моделей

Як пояснювалося профілемір енергії – це програмне забезпечення, яке вимірює та контролює енергію, споживану підкомпонентом мобільного пристрою, усім пристроєм або програмою. У таблиці 2.1 підсумовано аспекти профілемірів потужності на основі моделей, які ми обговорювали в попередніх розділах. Потужність або енергію можна моделювати як лінійну або нелінійну функцію кількох змінних. Обчислення змінних можна проводити на настільному комп'ютері, у хмарі або навіть на самому мобільному пристрої. Дані, що визначають змінні в моделі, можуть включати статистику використання апаратних ресурсів, трасування системних викликів та робочі стани або стани живлення різних апаратних компонентів [23]. Методологія моделювання може бути білою або чорною скринькою, або їх комбінацією. Модель потужності може залежати від виміряних значень потужності, які можна виміряти за допомогою інтерфейсу інтелектуального акумулятора або за допомогою фізичних інструментів вимірювання потужності, таких як вимірювач потужності.

У таблиці 2.2 ілюструється класифікація існуючих потужних профайлерів. Дві основні осі, за якими ми розрізняємо профайлери, це те, чи відбувається побудова моделі та навчання на смартфоні чи ні, та чи працює профайлер на телефоні чи ні. Як результат, ми визначаємо три основні категорії профайлерів наступним чином:

Таблиця 2.1

Орієнтовні питання для класифікації профілів

Критерій (Criteria)	Варіанти вибору (Choices)
Який рівень деталізації профілювання?	Весь пристрій (система), підкомпонент, застосунок або API.
Де навчати моделі енергоспоживання?	На самому пристрої (потребує функції самовимірювання) або поза ним (наприклад, на ПК чи в хмарі).
Де брати предиктори (параметри) моделі?	Метрики використання апаратних ресурсів (наприклад, HPC); траси виконання ПЗ (системні виклики, трафік); режими роботи заліза/ПЗ.
Яку методологію моделювання використовувати?	"Біла скринька" (White box); "Чорна скринька" (Black box, напр. лінійна регресія); Комбінація обох методів.
Як отримати значення потужності для навчання моделі?	Використання фізичних вимірювачів (напр. Monsoon Power Monitor); виконання самовимірювання пристроєм.

Внутрішньопристроєві профайлери з побудовою внутрішньопристроєвих моделей : ці профайлери не потребують офлайн-налаштування, вимірювання чи попереднього навчання моделей енергоспоживання. Натомість вони генерують моделі енергоспоживання під час виконання на основі інформації, зібраної з системи, отже, покладаючись на самовимірювання. Внутрішньопристроєві моделі важливі з двох причин. По-перше, апаратний компонент мобільного пристрою може змінюватися. Наприклад, карта пам'яті підключена до телефону або сам пристрій може змінюватися, а споживання енергії підкомпонентами може відрізнятися між пристроями однієї моделі або різних моделей.

По-друге, використання системи або програм може бути різним для різних користувачів, а поведінка користувачів змінюється з часом. Таким чином, моделі на пристроях дозволяють не лише уникнути складної інструментарії, але й забезпечують профілювання енергії незалежно від пристрою та використання.

— Профайлери на пристрої з побудовою моделей поза пристроєм : ці профайлери також працюють на мобільних пристроях, але спираються на моделі живлення, попередньо навчені в лабораторних умовах [24]. Вони використовують ці моделі з інформацією з пристрою для характеристики споживання енергії . Наприклад, споживання енергії різними підкомпонентами моделюється

заздалегідь за допомогою зовнішнього інструменту вимірювання живлення, а пізніше статистика використання на пристрої використовується для оцінки споживання енергії. На відміну від попередньої категорії, моделі цих профайлерів є специфічними для кожного пристрою, тому їхня точність може значно відрізнятися залежно від пристрою.

Таблиця 2.2

Класифікація приладів для вимірювання енергетичного профілю

Гранулярність профілювання	**Тип моделі**	**Побудова моделі (Construction)**	**Тип розгортання (Deployment)**
Системний рівень	Не застосовується	—	—
Системний, підкомпонентний та прикладний рівні	Не застосовується	—	—
Системний рівень	Лінійна регресія	На пристрої	На пристрої
Системний та підкомпонентний рівні	Лінійна регресія, використання (utilization)	На пристрої	На пристрої
Системний та підкомпонентний рівні	FSM (кінцеві автомати), лінійна регресія	На пристрої	На пристрої
Прикладний рівень	Моделі DevScope	На пристрої	На пристрої
Системний та підкомпонентний рівні	Лінійна регресія	Поза пристроєм	На пристрої
Системний, підкомпонентний та прикладний рівні	Використання (utilization)	Поза пристроєм	На пристрої
Системний, підкомпонентний та прикладний рівні	FSM	Поза пристроєм	На пристрої
Рівень API	Генетична модель	Поза пристроєм	На пристрої
Рівні процесів та процедур	Аналіз коду	Поза пристроєм	Поза пристроєм
Рівень потоків (Thread level)	Використання (HPC - апаратні лічильники)	Поза пристроєм	Поза пристроєм
Системний, підкомпонентний та прикладний рівні	Трасування системних викликів та FSM	Поза пристроєм	Поза пристроєм
Системний, підкомпонентний та прикладний рівні	Трасування системних викликів, лінійна регресія	Поза пристроєм	Поза пристроєм
Системний та підкомпонентний рівні	Використання, лінійна регресія	Поза пристроєм	Поза пристроєм

— *Профайлери поза пристроєм:* ці профайлери оцінюють споживання енергії програмами або апаратними компонентами мобільних пристроїв на настільному комп'ютері або в хмарі, а не на мобільному пристрої. Їхні моделі профілювання розробляються в лабораторії за допомогою деяких зовнішніх інструментів моніторингу живлення, таких як Monsoon Power Monitor або будь-який інструмент моделювання [25]. Таким чином, ці профайлери часто можуть точніше або з більшою деталізацією характеризувати споживання енергії пристроєм, програмами та підкомпонентами, наприклад, для кожного виклику методу, а не для всієї програми. Загалом, такі профайлери здебільшого корисні для розробників програм та системних архітекторів, але не для звичайних користувачів.

Існує також четверта категорія, а саме позапристроєві профайлери з побудовою моделі на пристрої, але ми її виключаємо, оскільки нам невідома жодна робота, присвячена цим профайлерам. У більшості випадків, якщо побудова моделі може бути виконана на пристрої, має сенс виконувати профілювання також на пристрої, замість того, щоб передавати вхідні дані, необхідні моделі, з пристрою до зовнішнього профайлера.

Ступінь деталізації профілювання різних профайлерів різниться. У цьому контексті гранулярність стосується здатності профайлера аналізувати загальне споживання енергії смартфоном. Профілювання на рівні підкомпонентів та програм є складнішим, ніж профілювання на рівні системи, просто тому, що потрібна детальніша інформація про поведінку системи та виконання програм. Наприклад, PowerTutor використовує моделі живлення на рівні компонентів, що надаються PowerBooter, для оцінки споживання енергії, специфічного для програми, шляхом приписування споживання енергії, специфічного для підкомпонентів, програмі.

У наступних розділах ми представимо різні енергетичні профілографи відповідно до трьох категорій, згаданих раніше. Опис профілографів також відповідає хронологічному порядку, як показано в таблиці 2.2.

Профілографи цієї категорії не потребують офлайн-підтримки для

вимірювань або калібрування моделі. Вони долають ці обмеження, замінюючи інструментальні вимірювання потужності самовимірюванням. Прикладами є NEP, Trepn Profiler, PowerBooter, Sesame, DevScope, AppScope та V-edge. У цьому розділі ми спочатку опишемо ці енергетичні профілографи , а потім дослідимо їх подібності та відмінності.

NEP - це окреме програмне забезпечення для вимірювання енергоспоживання пристроїв Nokia на базі Symbian. Це один із піонерів сучасної тенденції профілювання енергоспоживання на пристроях. Однак NEP більше не має практичного значення, оскільки пристрої Symbian більше не випускаються на ринку. Він відображає споживання енергії системою під час роботи у ватах та амперах шляхом моніторингу системної та мережевої активності.

Одночасно NEP може контролювати силу мережевого сигналу та стан підключення мобільних пристроїв до стільникової мережі. Він також відображає криву розряду напруги [26]. Більшість інформації відображається у вигляді часових графіків і може бути експортована у файли C-V. Інтерфейс розумного акумулятора пристроїв Nokia Symbian надає показники датчиків як напруги, так і струму, і вимірювання потужності NEP реалізовано на основі цих двох показників.

Цей інструмент використовується користувачем вручну. Після активації він починає профілювання системи. Ідея полягає в тому, що користувач запускає кілька тестових програм, а потім відвідує інтерфейс NEP, щоб перевірити споживання енергії цільовими програмами. Користувач також може перевірити загальне споживання енергії та використання мережі програмами. Частоту дискретизації NEP обмежено до 4 Гц, щоб зменшити споживання ресурсів самим профайлером.

Профілерівник Trepn схожий на NEP для профілювання використання обладнання та споживання енергії. Trepn розроблено спільнотою Qualcomm та працює на пристроях Android на базі чіпсета Snapdragon. Це програма для користувацького простору, яка може профілювати використання та споживання енергії процесором, графічним процесором, Wi-Fi, блокуванням після сну, пам'яттю,

SD-картою та аудіо, а також споживання енергії всім пристроєм під час роботи. На відміну від NEP, він може надавати детальні дані про споживання енергії для окремих підкомпонентів. Однак Trepn вимагає додаткового апаратного забезпечення в пристрої, яке називається *Mobile Development Platform* (MDP). MDP працює на вбудованому SOC для керування живленням, який збирає показники з сенсорних резисторів та перетворює їх на струм для окремих апаратних компонентів, таких як процесор, графічний процесор та Wi-Fi. Trepn також залежить від спеціального чіпа індикатора рівня палива з інтегрованою мікросхемою керування живленням, яка контролює розподіл енергії від акумулятора [27]. Для статистики використання різних апаратних компонентів Trepn залежить від процесу та інших системних файлів. Хоча Trepn вибірково перевіряє інформацію кожні 100 мс, він може адаптувати частоту вибірки до системного навантаження, щоб уникнути накладних витрат на вибірку.

Подібно до NEP, він також пропонує різні режими візуалізації інформації. Trepn також забезпечує накладання різних графіків та діаграм на передньому плані, щоб розробники додатків могли пов'язувати продуктивність додатків з використанням ресурсів та споживанням енергії під час виконання. Водночас, він дозволяє експортувати необроблені дані в режимі реального часу для офлайн-аналізу. Trepn також дозволяє налагоджувати продуктивність додатків, фіксуючи наміри Android та реєструючи стани додатків разом з іншими точками даних. Нарешті, Trepn можна керувати із зовнішньої програми, що сприяє автоматизованому профілюванню.

PowerBooter автоматично генерує моделі споживання енергії, корелюючи споживання енергії окремими підкомпонентами зі станом акумулятора за допомогою регресії. На рисунку 2.2 показано ключові кроки механізму генерації моделі: отримання кривих розряду акумулятора для окремих апаратних компонентів, вимірювання споживання енергії компонентами та генерація моделей.

Першим кроком є побудова кривих розряду для кожного окремого компонента. Для цього PowerBooter використовує оновлення інтерфейсу батареї. На рисунку 2.2 показано один приклад кривої, яка є монотонно спадною та

відображає зв'язок між напругою батареї та напруженістю заряду (SOD). Крутизна цієї кривої залежить від струму розряду, температури приміщення та віку батареї. Крім того, різні батареї можуть мати різні криві. Отже, PowerBooter характеризує окремі батареї, повністю розряджаючи повністю заряджену батарею з постійним струм та підтримує лінійну залежність між напруженістю розряду (SOD) та часом розряду. Зауважте, що для представлення залежності між SOD застосовуються кусково-лінійні функції, а вихідна напруга акумулятора – залежність.

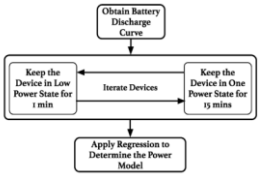


Рисунок 2.2 - Загальний огляд процесу створення моделі PowerBooter для декількох пристроїв.

Вимірювання споживання енергії проводяться на другому етапі. На цьому етапі налаштовуються стани окремого компонента, водночас інші компоненти залишаються в станах зниженого енергоспоживання або в деяких статичних конфігураціях. Наприклад, під час визначення споживання енергії процесором на різних частотах дисплей, Wi-Fi , GPS та стільниковий інтерфейс вимикаються [28]. Акумулятор розряджається протягом 15 хвилин, а пристрій залишається в режимі очікування протягом 1 хвилини до та після інтервалу розрядки. Під час вимірювань також враховуються взаємозалежності між різними підкомпонентами , такі як взаємозалежність між споживанням енергії Wi-Fi та процесором, шляхом моніторингу станів живлення інших компонентів під час роботи певного компонента.

На третьому етапі споживання енергії компонентами визначається на основі вимірювань, проведених на другому етапі. Активність протягом цього інтервалу відображається на зміні SOD, яка, у свою чергу, перетворюється на споживання енергії. Споживання енергії протягом інтервалу розраховується як $P \times (\ t^2$

—*t\) = E* x *(SOD(V 2) — SOD(Vi))*, де *P* – середнє споживання енергії за часовий інтервал *[t 1* , t 2], *E* – енергія акумулятора, пов'язана з ємністю акумулятора, а SOD(Vi), *SOD(V 2)* – стани розряду при напрузі V 1 та V 2 відповідно, як показано на рисунку 2.3. Нарешті, регресія застосовується для створення моделей потужності.

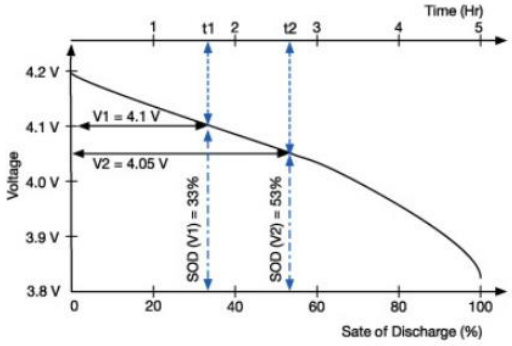


Рисунок 2.3 - Крива розряду акумулятора: динаміка напруги акумулятора у міру розряду з плином часу.

Sesame також використовує самовимірювання для автоматичного створення моделі живлення. На відміну від PowerBooter, Sesame покладається на отримання миттєвих показників струму безпосередньо без необхідності вдаватися до оцінки на основі SOC під час створення моделей живлення. Це конструктивне рішення обмежує використання Sesame телефонами, які мають мікросхеми датчика рівня палива на основі OCV. Головною новизною порівняно з PowerBooter є те, що Sesame використовує статистичне навчання для створення моделей живлення, які мають вищу точність і швидкість порівняно з інтерфейсом батареї. На рисунку 2.4 ілюструється архітектура Sesame. Вона має три підкомпоненти: колектор даних, конструктор моделей і генератор моделей.

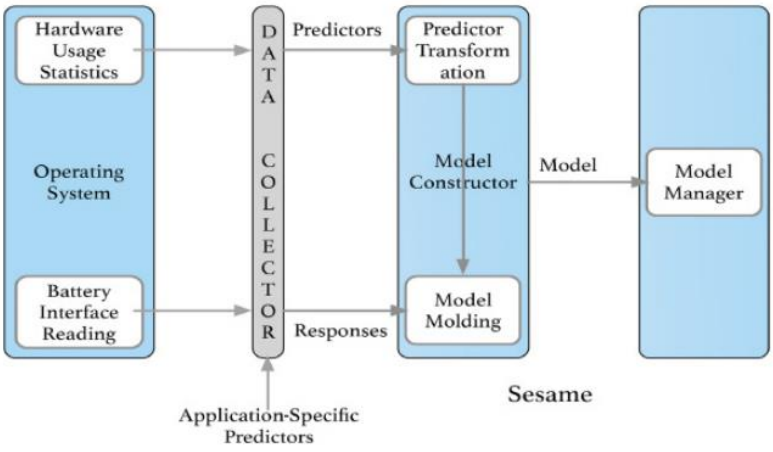


Рисунок 2.4 - Архітектура Sesame

Окрім інтерфейсу батареї, колектор даних збирає дані про використання різних підкомпонентів з кількох джерел, таких як системні файли proc та dev. Sesame також враховує стан живлення різних апаратних компонентів. Однак різні джерела мають різну частоту оновлення, яка також може залежати від інших дій. Тому зчитування з цих джерел з вищою або нижчою швидкістю, ніж фактична частота оновлення джерел призведе до помилки [29]. Наприклад, резидентність P-стану процесора (P0-P5) оновлюється з частотою 250 Гц, а зчитування цього P-стану з частотою 100 Гц призводить до 20% помилки, оскільки збирач даних може пропустити проміжні зміни стану. Крім того, може виникнути затримка між оновленням значення предиктора та фактичним виконанням завдання. Доступ до деяких джерел даних, таких як зчитування інтерфейсу батареї, також створює накладні витрати. Sesame зменшує ці накладні витрати, адаптуючи швидкість доступу до швидкості оновлення джерела. Якщо швидкість оновлення джерела вища за швидкість оновлення Sesame, то виконується опитування. В іншому випадку Sesame очікує змін у залежних джерелах. Крім того, Sesame вводить новий системний виклик, який може зчитувати з ОС кілька структур даних підкомпонентів одночасно.

Конструктор моделей – це серце Sesame. Щоб створити модель з покращеною точністю та частотою оновлення, Sesame застосовує формування моделі. Формування моделі працює у два кроки. Перший крок називається

розтягуванням, який включає побудову моделі з найвищою точністю та нижчою частотою оновлення, ніж цільова частота. Ця точна модель будується шляхом усереднення кількох показників, і обґрунтування базується на спостереженні, що точність вища, коли показники інтерфейсу батареї усереднюються протягом тривалішого періоду часу. Наприклад, якщо частота оновлення батареї становить 0,5 Гц, то модель з частотою 0,01 Гц буде побудована шляхом усереднення послідовних 50 показників інтерфейсу батареї, і точність буде вищою. На другому кроці модель з низькою частотою точного стискається для побудови моделі з високою частотою, що досягається шляхом застосування коефіцієнтів лінійної регресії при розрахунку споживання енергії протягом потрібного інтервалу часу [30].

Оскільки моделі споживання енергії залежать від використання різних підкомпонентів або предикторів, таких як частота процесора та стан живлення, кілька таких предикторів можуть бути великими. Тому знайти фактичні предиктори є складним завданням. Для цього Sesame застосовує перетворення до предикторів за допомогою аналізу головних компонентів (PCA). Формування моделі та PCA разом підвищують точність моделі. Однак конструктор моделей може генерувати моделі для трьох категорій конфігурацій системи: (i) інформація про апаратне забезпечення та виробника; (ii) налаштування програмного забезпечення, такі як увімкнення або вимкнення DVS; та (iii) налаштування, керовані користувачем, такі як яскравість та гучність. Нарешті, менеджер моделей адаптує модель до конфігурації системи під час виконання. Він порівнює показники енергії з активної моделі з низькошвидкісною версією моделі, отриманою з інтерфейсу батареї.

Зелена лінія представляє стани живлення підкомпонента на графіку живлення, а чорна лінія представляє значення потужності для станів живлення, що реалізуються DevScope на основі періодичного оновлення інтерфейсу батареї.

Зелена лінія представляє стани живлення підкомпонента на графіку живлення, а чорна лінія представляє значення потужності для станів живлення, що реалізуються DevScope на основі періодичного оновлення інтерфейсу батареї.

Після перевищення певного порогу конструктор моделі генерує нову

модель з новими предикторами. Таким чином, менеджер моделі адаптується до використання системи.

DevScope також використовує оновлення інтерфейсу батареї для подолання практичних вимог до вимірювань [31]. Згодом він стикається з аналогічною проблемою, що й Sesame, - низькою швидкістю оновлення інтерфейсу батареї. DevScope має чотири компоненти: блок моніторингу подій оновлення батареї, контролер синхронізації, контролер компонентів та генератор моделі живлення. DevScope спочатку знаходить апаратні компоненти, доступні в системі, та їх конфігурації. Монітор подій батареї збирає інформацію про струм розрядки. Блок контролера синхронізації оцінює швидкість оновлення інтерфейсу батареї. Він також інформує контролер компонентів, коли починати та завершувати тест. Контролер компонентів генерує та відповідно виконує сценарії тестування, специфічні для компонентів. Нарешті, генератор моделі живлення аналізує результати тестування та генерує коефіцієнти моделі живлення.

Оскільки частота оновлення інтерфейсу батареї може змінюватися і не може контролюватися, DevScope використовує подієвий підхід, який розглядає оновлення батареї як подію. Блок моніторингу батареї зберігає запис позначок часу для кожної такої події, а контролер синхронізації знаходить частоту оновлення на основі цих позначок часу. Після обчислення частоти оновлення контролер синхронізації запускає тестовий сценарій; таким чином, тестові сценарії синхронізуються відповідно до оновлення інтерфейсу батареї.

DevScope розглядає ще одну проблему розпізнавання станів живлення підкомпонентів, таких як інтерфейси Wi-Fi та стільникової мережі [32]. Однак розпізнавання переходів станів є складним і вимагає знання про тривалість різних станів живлення. У деяких випадках ці переходи визначаються робочим навантаженням та умовами експлуатації. Отже, DevScope неодноразово використовує різні робочі навантаження, щоб визначити ¬поріг, який викликає перехід стану живлення та оновлення в інтерфейсі батареї. На рисунку 2.5 показано фактичні стани живлення підкомпонента та вимірювання, здійснені DevScope.

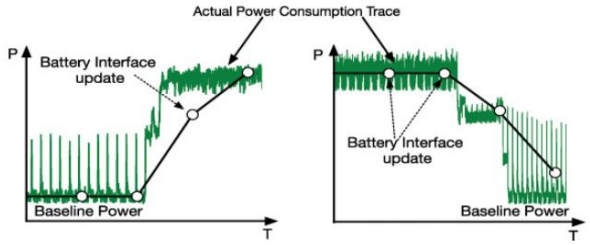


Рисунок 2.5 - Ідентифікація станів живлення підкомпонентів за допомогою DevScope.

Автори DevScope запропонували прикладну платформу для профілювання енергії програм у мобільних пристроях: AppScope. Це залежить від моделей живлення DevScope. Профілювання живлення AppScope працює у три фази. На першій фазі AppScope запитує виявлені апаратні компоненти. Другий крок включає аналіз використання цих підкомпонентів та змін їхнього стану.

Правий рисунок ілюструє, як V-front виявляє струм розряду за миттєвою зміною напруги.

Зрештою, споживання енергії застосунком оцінюється шляхом підсумовування статистики використання підкомпонентів, що використовуються застосунками.

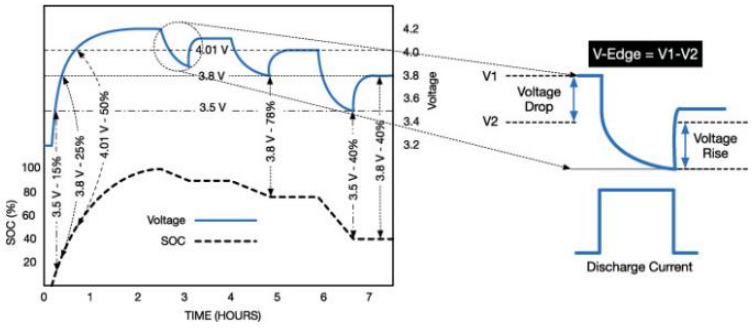


Рисунок 2.6 - Графіки стану заряду (SOC) та миттєвої напруги на лівому рисунку показують, що одна й та сама напруга може представляти кілька SOC

Одним із поширених обмежень описаних раніше підходів є те, що вони не можуть ізолювати використання спільних ресурсів, таких як використання дисплея кожною програмою. AppScope не покладається на системні файли для

статистики використання таких спільних підкомпонентів. Натомість він використовує Android RPC з протоколом зв'язування RPC та іншими інструментами налагодження, такими як Kprobe. Kprobe відстежує події, пов'язані із запитами апаратних компонентів, та аналізує статистику використання, необхідну для застосування моделей живлення [33]. Для збору даних використовуються функції, специфічні для підкомпонентів. Наприклад, інтерфейс Linux CPU Governor використовується для збору інформації про використання та частоту процесора. Використання Wi-Fi визначається з викликів функцій нижчого рівня в ядрі Linux. Потім стан живлення визначається за допомогою швидкості передачі пакетів, а споживання енергії обчислюється на основі тривалості активного часу інтерфейсу. У випадку 3G, зв'язування RPC використовується для виявлення роботи обладнання. Зміни в мережевому з'єднанні виявляються на основі даних міжпроцесного зв'язку на рівні радіоінтерфейсу. У випадку дисплея, AppScope використовує Android ActivityManager, щоб знайти програму, що працює на передньому плані, та відстежує цю активність, доки на передній план не буде виведена інша активність або дисплей не буде вимкнено. Нарешті, використання GPS відстежується за допомогою викликів до LocationManager. AppScope підраховує такі виклики, коли GPS активовано, та розподіляє споживання енергії відповідно до кількості викликів різними програмами.

Подібно до раніше описаних вбудованих профілерів живлення, V-edge має на меті генерувати моделі живлення за допомогою самовимірювання. Його принцип роботи близький до PowerBooter. Основна відмінність PowerBooter від інших підходів на основі SOC полягає в тому, що V-edge використовує зміни миттєвої напруги на клемах акумулятора для визначення споживання струму, тоді як методи на основі SOC уникають такого миттєвого падіння напруги, щоб зменшити похибку оцінки SOC. На рисунку 2.6 показано, як миттєва напруга може спотворити оцінку SOC і як V-edge використовує це. Однак основним ефектом цієї різниці є пришвидшення генерації моделі живлення порівняно з PowerBooter. Щоб зрозуміти, чому, згадаємо, що PowerBooter повинен утримувати телефон у певному постійному стані живлення достатньо довго, щоб

значення SOC змінилося, тоді як V-edge може вимірювати струм майже миттєво. Тому будь-яку модель акумулятора на основі OCV, таку як модель Rint або Thevenin, можна підключити для визначення струму розряду. V-крап ефективно використовує струм, що споживається внутрішнім резистором R. Рівняння моделі Тевеніна можна записати як:

$$V_t = OCV - V_c - \Delta I \times R \quad (2.1)$$

$$V_t = OCV - V_c - V_{edge} \quad (2.2)$$

$$V_{edge} = R \times \Delta I = R \times I - R \times I_0 \quad (2.3)$$

$$I = \frac{1}{R} \times V_{edge} + I_0 \quad (2.4)$$

Коли спостерігається помітна зміна струму, OCV та V_c залишаються незмінними протягом короткого періоду часу. На рисунку 2.6 ми бачимо, що відбувається різка зміна напруги при споживанні струму. Це миттєве падіння напруги спричинене внутрішнім опором. Після цього напруга повільно падає через розряд струму акумулятора. Миттєве падіння напруги *(R x* AI) визначається як V-фронт у попередньому рівнянні. I_0 стосується базового споживання струму, якого можна досягти, розпочавши все навчання з однієї базової лінії під час створення моделей потужності.

Щоб знайти такі V-фронти, застосували підхід, подібний до DevScope. Спочатку мобільний пристрій залишається в режимі очікування протягом тривалішого періоду, ніж інтервал оновлення інтерфейсу батареї. Потім на деякий час збільшується використання процесора, і напруга вибірково вимірюється з частотою 1 Гц. Процесор залишається в режимі очікування, коли відбувається падіння напруги. Вибірка зупиняється, коли відбувається оновлення напруги. Інтервал між цими двома інцидентами падіння напруги та оновлення полегшує виявлення

V-фронту і, таким чином, вимірювання струму, як показано на рисунку.

Архітектура системи V-edge складається з колектора даних, генератора моделей та профілера живлення [34]. Колектор даних збирає інформацію про напругу акумулятора для створення моделі живлення та статистику використання різних підкомпонентів для оцінки споживання енергії програмами, такими як процесор, екран, Wi-Fi та GPS. Моделі живлення для конкретних компонентів будуються на основі V-edge шляхом запуску набору відповідних навчальних програм. Навчання починається з початкового стану живлення підкомпонентів, щоб забезпечити узгодженість їхніх значень V-edge. Нарешті, оцінка живлення виконується за допомогою зібраної статистики використання ресурсів.

2.3 Інструменти профілювання та системи моніторингу енергоспоживання мобільних пристроїв

На відміну від повністю вбудованих профайлерів, ці програми вимагають фаз автономного калібрування та вимірювання потужності. Іноді ці програми розробляються постачальниками мобільних пристроїв і є невід'ємною частиною мобільних систем. У цьому підрозділі ми описуємо профайлери потужності, що належать до цієї категорії, а в кінці підсумовуємо їхні ключові подібності та відмінності.

ОС Android має власні моделі енергоспоживання та профілери для оцінки споживання енергії різними програмами та компонентами. Повна система профілювання базується на трьох підкомпонентах: BatteryStats, списку значень споживання енергії апаратними компонентами (профіль живлення) та моделях живлення. BatteryStats залежить від профілів живлення та моделей для оцінки споживання енергії. Ми детально розглянемо цей сервіс у наступних розділах.

1. *BatteryStats.* BatteryStats відповідає за відстеження використання апаратних компонентів, таких як процесор, GSP, Wi-Fi, блокування після сну, сканування GSM-радіостанцій, телефон (дзвінки) та Bluetooth. Інформація про використання цих компонентів записується разом із міткою часу. BatteryStats

безпосередньо не вимірює споживання енергії з акумулятора. Натомість він обчислює загальне використання різного обладнання на основі позначок часу та оцінює споживання енергії. BatteryStats збирає статистику двома різними способами: різні служби передають зміни стану компонентів до BatteryStats, або ж періодично збирає інформацію про процесор та інші компоненти, що використовуються програмами, із системних файлів процесу. BatteryStats зберігає статистику протягом 30 хвилин, щоб дані не втрачалися під час раптового перезавантаження або збою системи. Більшість інших профайлерів живлення, таких як PowerBooter та AppScope, отримують цю статистику від BatteryStats або безпосередньо витягують її із системних файлів процесу. BatteryStats також надає цю статистику іншим службам, що запитують її. Тому реєстрація за допомогою BatteryStats є безпечнішою, оскільки розташування файлів статистики може відрізнятися на різних пристроях. Нещодавня версія Android Lollipop надає інструмент для вилучення BatteryStats з мобільних пристроїв для аналізу поза пристроєм за допомогою Battery Historian.

2. *Значення профілю живлення.* Для оцінки споживання енергії або потужності BatteryStats залежить від попередньо виміряних значень потужності різних апаратних компонентів. Це, по суті, попередньо навчені коефіцієнти моделі живлення. Значення постачаються з програмою Android framework та зберігаються у файлі XML. Файл містить інформацію про стани живлення, що підтримуються процесором, GSM-радіомовленням, Wi-Fi, дисплеєм, Bluetooth та аудіо- та відеоцифровим процесором сигнальної обробки (DSP). Файл також містить струм, споживаний цими компонентами в міліамперах у цих станах, які дуже специфічні для відповідних моделей пристроїв, наданих виробниками. Наприклад, файл містить тактові частоти, що підтримуються процесором, та струм, споживаний на кожній тактовій частоті. Android Power Profiler припускає, що всі ядра мають однорідні характеристики частоти та споживання енергії.

3. *Моделі живлення.* Після того, як відома статистика використання підкомпонентів та базова потужність, що споживається ними, BatteryStats може легко обчислити споживану енергію за допомогою деяких базових моделей. У

таблиці IV ми представляємо список моделей живлення, що використовуються BatteryStats на пристроях Android, які ми витягли з вихідного коду Android framework. Ми бачимо, що це прості моделі на основі використання. BatteryStats спочатку обчислює часовий проміжок використання апаратних ресурсів, а потім обчислює споживання енергії відповідно до станів живлення. Для бездротового зв'язку спочатку обчислюється швидкість передачі або прийому в байтах за секунду, а потім обчислюється енергія на байт. Нарешті, система приписує споживання енергії програмі, просто підсумовуючи енергію, споживану компонентами, що використовуються програмою [35].

Деякі ресурси можуть одночасно використовуватися кількома програмами, і BatteryStats докладає додаткових зусиль для розподілу витрат між цими програмами. У цьому випадку корисні блокування після пробудження, і блокування після пробудження для компонента може бути встановлено кількома програмами. Після цього ті програми, які встановили блокування після пробудження, розподіляють витрати на живлення. Час, протягом якого програми пов'язані з блокуванням після пробудження, визначає їх часткові витрати.

PowerTutor — це програма для профілювання живлення, розроблена на основі моделі PowerBooter для мобільних пристроїв Android. Однак модель залежить від живлення від вимкненого пристрою.

Програма оцінює споживання енергії різними апаратними компонентами та програмами. PowerTutor ілюструє частку енергії, що споживається дисплеєм, процесором, Wi-Fi, GPS та 3G, за допомогою кругової діаграми. Він також використовує лінійні графіки для опису споживання енергії цими компонентами під час виконання в джоулях. Для цього PowerTutor спирається на точні вимірювання потужності компонентів у різних станах живлення та на статистику використання, зібрану з системних файлів proc та
Android BatteryStats.

Подібно до Android Power Profiler, PowerTutor також оцінює споживання енергії окремою програмою. Це дозволяє розробникам програм візуалізувати споживання енергії їхніми програмами та, таким чином, забезпечує подальшу

оптимізацію. Водночас користувачі можуть зрозуміти вплив їхньої взаємодії на час роботи акумулятора своїх мобільних пристроїв. Однак, складно оцінити споживання енергії цими програмами, коли одночасно працює кілька програм, які використовують або спільно використовують спільні ресурси [36]. У таких ситуаціях незрозуміло, як розподілити споживання енергії апаратним забезпеченням. PowerTutor вирішує цю проблему, оцінюючи вартість для однієї програми, вимагаючи, щоб вона була єдиною програмою, що працює в цей час.

Для побудови моделей потужності потрібен набір навчальних вимірювань. Навчання можна розпочати, коли пристрій не використовується, або навіть під час процесу встановлення програми.

Архітектура системи PowerProf полягає у зборі даних вимірювань через API NEP. Потім генетичний алгоритм, розміщений на окремому комп'ютері, обробляє ці дані для створення моделей потужності. Кроки процесу такі. Спочатку до інтерфейсу акумулятора надсилається запит на надання вимірювань потужності з часовими мітками. Виконуються тести на конкретні функції телефону, і відповідні часові позначки реєструються для початку та кінця тесту. Потім визначаються деякі очевидні характеристики, такі як фонове споживання енергії. Далі застосовується генетичний алгоритм для знаходження додаткових параметрів, необхідних для моделей потужності. Функція придатності алгоритму обчислює відстань між споживанням енергії, виміряним API, та моделлю. Нарешті, значення параметрів, які мінімізують функцію придатності, використовуються в кінцевій моделі. Кінцева модель потужності є умовною функцією чотирьох часових параметрів, що нагадує чотири стани потужності окремого компонента.

Ми виявили, що профайлер енергоспоживання системи Android та PowerTutor залежать від попередньо виміряних значень споживання енергії. У випадку Android Power Profiler значення енергоспоживання для конкретних компонентів надходять від постачальників. Однак у файлі профілю енергоспоживання можуть бути неправильні значення вимірювань енергоспоживання, що може давати оманливі оцінки енергії, споживаної програмами або пристроями. Наприклад, ми помічаємо, що у файлах профілю енергоспоживання можуть бути

заявлені однакові розміри батарей двох пристроїв, хоча вони відрізняються один від одного.

Інший профайлер, PowerTutor, має ширше покриття компонентів та програм, ніж Android Power Profiler [37]. Щодо споживання енергії окремими апаратними компонентами та значень їх таймерів стану, таких як Wi-Fi, 3G та інші, PowerTutor залежить від попередньо виміряних константних значень. Тому точність PowerTutor становить 97,5%. Спочатку цільовими пристроями для моделей живлення були пристрої HTC G1, G2 та Nexus One. Однак він також працює на інших пристроях з приблизними оцінками, оскільки споживання енергії апаратними компонентами варіюється. PowerProf виділяється з-поміж усіх вбудованих профайлерів тим, що застосовує генетичні алгоритми для побудови моделей живлення.

2.4 Висновок по розділу

У другому розділі було досліджено методи та інструменти оптимізації енергоспоживання мобільного програмного забезпечення. Розглянуто методології моделювання та оцінювання енергоспоживання мобільних застосунків, що дозволяють визначати найбільш енергоємні компоненти системи. Проаналізовано класифікацію та особливості енергетичних профайлерів на основі моделей, які використовуються для контролю та аналізу використання ресурсів пристрою. Також досліджено сучасні інструменти профілювання та системи моніторингу енергоспоживання мобільних пристроїв. У результаті визначено ефективні підходи до оцінювання та оптимізації енергетичних характеристик програмного забезпечення.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МЕТОДІВ ОПТИМІЗАЦІЇ

3.1 Реалізація підходу до підвищення енергоефективності мобільних застосунків

eLens аналізує реалізацію мобільного застосунку та надає оцінки на рівні коду енергії, яку він споживатиме під час виконання. Результати цього аналізу узагальнюються з урахуванням деталізації всієї програми, шляху, методу та рядка вихідного коду, щоб допомогти розробнику приймати обґрунтовані рішення щодо впровадження для зменшення споживання енергії. Вхідними даними для підходу (Рисунок 3.1) є: (1) артефакт програмного забезпечення; (2) робоче навантаження, яке описує спосіб використання програмного забезпечення під час виконання; та (3) системні профілі, які використовують моделі енергії на інструкцію для визначення характеристик потужності платформ, для яких розробник орієнтується на реалізацію. В eLens є три компоненти: Генератор робочого навантаження перетворює робоче навантаження на набори шляхів через артефакт програмного забезпечення; Аналізатор використовує шляхи та системні профілі для обчислення оцінки енергії; та Анотатор вихідного коду поєднує шляхи та оцінку енергії для створення анотованої версії вихідного коду, яка надається розробнику. Вихідними даними eLens є візуалізація , яка показує приблизне споживання енергії програмним забезпеченням на рівні шляху, методу, вихідного коду та всієї програми.

A. Генерування робочого навантаження

Генератор робочого навантаження відповідає за перетворення дій на рівні користувача, для яких розробник хоче отримати оцінку, в інформацію про шлях, що використовується аналізатором та анотатором вихідного коду. Тривіально, цю інформацію можна отримати, припустивши, що кожен шлях в артефакті виконується *n* кількість разів. Однак це не буде точним відображенням того, як програма працюватиме під час виконання, і оскільки споживання енергії не є

рівномірним для різних шляхів, отримана оцінка не буде такою корисною. Наприклад, у програмі для перегляду відео шляхи, пройдені для перегляду відео, споживатимуть більше енергії, ніж шляхи, пройдені для запуску або виходу з програми.

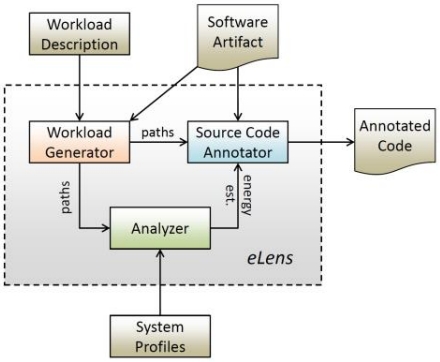


Рисунок 3.1 - Огляд eLens

Вхідними даними для генератора робочого навантаження є опис робочого навантаження *W* та реалізація програми S. *Опис робочого навантаження* – це специфікація поведінки програми, для якої розробник хоче оцінити споживання енергії, і може бути представлений як послідовність варіантів використання *(u1 , u2 ,..., un*). Генератор робочого навантаження використовує *S* для створення версії *S0 , яка записуватиме шляхи ,* пройдені під час виконання. Далі генератор робочого навантаження запускає кожне *ui* . 2 *W* на S^0 . Шляхи, пройдені для u_i позначаються як P_i . Множина всіх P_i , P є виходом генератора робочого навантаження.

Опис робочого навантаження може бути заданий неформально, коли розробник просто взаємодіє з інструментованим додатком, або формально, коли послідовність дій явно перелічена та може бути виконана автоматизованими інструментами тестування Android, такими як MonkeyRunner [1] та Robotium [2]. Ми вимагаємо лише, щоб механізм специфікації міг виконувати інструментовану версію додатка, тому шляхи, пройдені додатком, записуються. Немає критерію адекватності для опису робочого навантаження, окрім того, що він представляє набір дій, які цікавлять розробника. Наприклад, неформальний опис

робочого навантаження відеоплеєра може вказувати, що користувач виконує такі дії: (a) запустити додаток, (b) знайти відео за допомогою терміна *«ebens* at ICSE», (c) відтворити перше знайдене відео, (d) повторити відео 100 разів та (e) вийти з додатка.

Інструментарій, вставлений генератором робочого навантаження, записує шлях, пройдений кожним методом програми [38]. Цей запис базується на ефективній техніці профілювання шляху. Підхід призначає ваги ребрам графа потоку керування (CFG) методу таким чином, що сума ваг ребер вздовж кожного унікального шляху через CFG призводить до унікального ідентифікатора шляху; тоді достатньо однієї інструментальної змінної на метод для запису пройденого шляху для одного виклику методу. Отже, кожен *P i* складається з послідовності кортежів підшляхів, кожен з яких позначено як *(m, id*), де *m* – це метод, а *id* – ідентифікатор пройденого шляху. Наша реалізація розширює підхід Балла-Ларуса для обробки вкладених викликів методів, паралельності та винятків, як описано в розділі VA.

Щоб проілюструвати результат роботи генератора робочого навантаження, розглянемо програмний артефакт із трьома методами *a, b* та c. Для цього артефакту можливим P є *{((a,* 1), (b, 1), *(a,* 2)), *((a,* 2), (c, 3))}*, який містить два шляхи. Перший відповідає випадку використання, в якому виконується шлях 1 методу *a, за яким йде шлях 1 методу b* , а потім шлях 2 методу *a* . Другий відповідає випадку використання, в якому виконується шлях 2 методу *a, за яким йде шлях 3 методу c* .

Вхідні дані: *H* : набір апаратних компонентів, *l* : лінія джерела, m: метод та P: шлях.

Вихідні дані: Оцінка енергії в джоулях 1

eLens відстежує стани живлення апаратних компонентів, а також певні типи даних, пов'язаних з траєкторією. Ця інформація включена як один або декілька аргументів до функцій вартості, визначених у SEEP. Детальна інформація та приклади інформації, що залежить від траєкторії, наведені нижче.

```
1: cost ← 0
2: ⟨Θ, M, L⟩ ← regenerate(P)
3: D ← propagateDT(Θ)
4: for all h ∈ H do
5:     for all i ∈ Θ do
6:         f_h ← powerstate(i, h)
7:         if L(i) = l ∧ M(i) = m then
8:             cost(h) += C(i, h, f_h, D(i))
9: return ∑_{h∈H} cost(h)
```

Лічтинг 3.1 - Оцінка споживання енергії

Перша операція, яку виконує аналізатор, полягає у повторній генерації послідовності інструкцій, представленої *P* (рядок 2 алгоритму 1). Як визначено алгоритмом Болла-Ларуса, для заданого підшляху *(m, id*) можна повторно створити послідовність інструкцій, що визначають кожен підшлях. *Повторна генерація* об'єднує кожен підшлях у *P* , щоб визначити повний шлях (від входу до виходу), пройдений під час виконання. Для вкладених викликів методів, де метод *A* викликає метод *B, повторна генерація* обчислює послідовність інструкцій a1 , a2 ,...,*a2*,..., $_{a2m}$, пройдених у *A*, та *b1* , b2 ,...,*bm* , пройдених у B. Якщо *K* є викликом B, то кінцева послідовність буде $_{a1}$, a2 . ,...,a_K , b_1 , b_2 ,...,b_m , a_{k+1} ,...,a_n . Визначити, які підшляхи в *P* були викликані іншими підшляхами, є простою, оскільки *P* є послідовністю, і кожен *(m, id)* додається до *P* після завершення *m* . Кінцевим результатом *regenerate* є кортеж (0, *M, L),* де 0 - повний шлях, *M* відображає кожну інструкцію в 0 на метод, що її містить, а *L* відображає кожну інструкцію в 0 на номер рядка її вихідного коду.

Вартість енергії певних інструкцій базується на інформації, що залежить від шляху. Наприклад, вартість інструкції надсилання по мережі залежить від обсягу надісланих даних, а вартість відкриття вхідного потоку залежатиме від типу потоку. Рядок 3 алгоритму поширює дані аргументу та типу вздовж 0, які можна використовувати для ідентифікації цього типу інформації, та ініціалізує функцію *D* , яка пов'язує цю інформацію з кожною інструкцією. Функція *propagateDT* реалізує цю функціональність, імітуючи потік даних вздовж шляху 0. Інструментарій, представлений генератором робочого навантаження, записує інформацію,

таку як розмір даних, над якими працює інструкція, або клас, що реалізує виклик API у певних точках виконання. Потім *propagateDT* імітує вміст стеку вздовж 0, щоб відстежувати типи та відповідні атрибути даних до точки, де вони потрібні функціям вартості енергії. Моделювання стеку добре працює в цьому контексті, оскільки воно працює лише на одному повному шляху (0), і потрібно імітувати лише підмножину всіх інструкцій на шляху. У випадках, коли значення маніпулюють неінструментованою функцією (наприклад, викликом бібліотеки), для функцій вартості енергії використовується середнє значення.

Багато API, що залежать від шляху, вимагають спеціалізації загального підходу, описаного вище. Через обмеження простору ми наводимо два репрезентативні приклади аналізу, що використовується функцією *propagateDT*.

(1) Точна ідентифікація класу, який розширює InputStream, є складною, оскільки в додатках Android вони часто походять від фабричного класу. Щоб зібрати цю інформацію, генератор робочого навантаження вставляє зонд у S' після викликів певних фабричних методів для запису типу класу, що реалізує. Ця інформація потім використовується для анотації елемента даних у симуляції стеку, щоб під час виклику API InputStream можна було ідентифікувати клас, що реалізує, у стеку, та визначити відповідні функції вартості.

(2) Інструкції розподілу виділяють місце в пам'яті для масиву. Наприклад, мережеві буфери можуть виділити байтовий масив і визначити його розмір, використовуючи жорстко закодовану константу 1024. Функція *propagateDT* відстежує кількість констант у стеку, щоб під час їх вилучення та використання як аргументів можна було ідентифікувати значення константи. У наведеному вище прикладі аналізатор зможе визначити, що 1024 – це значення константи у стеку, яка використовується для передачі аргументу оператору розподілу.

Як описано вище, багато компонентів мають кілька станів живлення. Наприклад, сучасні процесори економлять енергію, змінюючи частоту процесора у відповідь на високе або низьке використання. Тому споживання енергії інструкцією може залежати від частоти процесора під час її виконання. Функція powerstate, що використовується в рядку 6, відображає кожну інструкцію i ∈ 0 на

стан живлення f_n компонента h (де h може бути процесором, Wi-Fi або іншим компонентом з кількома станами живлення), коли і виконувалося . eLens обчислює f_n , відстежуючи під час генерації робочого навантаження, коли відбуваються зміни стану живлення компонента. Функції вартості для кожної інструкції приймають f_n та h як аргументи. Наприклад, якщо припустити , що процесор має два рівні частоти, високий і низький, функція вартості процесора для інструкції ldc поверне два різні значення вартості енергії e_n або e_n , залежно від того, чи повідомила f_n частоту як високу чи низьку для цього екземпляра інструкції.

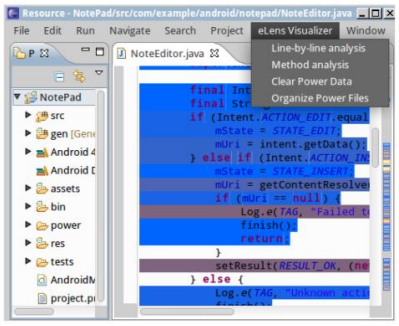


Рисунок 3.2 - Візуалізація лінії джерела, надана eLens

Кожна інструкція, яка задовольняє обмеження методу та номера рядка, додається до загальних витрат енергії (рядки 7-8). Аналізатор розраховує витрати енергії інструкції для кожного компонента платформи як функцію його типу, даних, що залежать від шляху, та стану живлення компонента. Аналізатор можна налаштувати для дослідження різних наборів апаратних компонентів через вхід *H*.

C. Енергетичні анотації

Анотатор вихідного коду перетворює інформацію про шлях та оцінку енергоспоживання у графічне представлення, яке дозволяє розробникам візуалізувати споживання енергії їхнім програмним забезпеченням. Представлення пов'язують оцінені показники енергоспоживання зі структурою реалізації програмного артефакту. Можливість візуалізації споживання енергії на рівні рядка вихідного коду є унікальною особливістю *eLens* . Зворотній зв'язок на цьому рівні

дозволяє розробникам ітеративно вдосконалювати деталі реалізації, такі як оператори та цикли, для покращення загального споживання енергії програмним забезпеченням. Ми реалізували візуалізацію як плагін Eclipse, який може забезпечити візуальне представлення споживання енергії на чотирьох різних рівнях гранулярності: для всього програмного забезпечення, для кожного методу, для кожного рядка вихідного коду та для кожного шляху. Зауважте, що для кожного рівня гранулярності споживання енергії може бути відображено для деяких або всіх випадків використання в робочому навантаженні. Механізм для двох із цих представлень обговорюється нижче; механізми для анотацій шляхів та всього програмного забезпечення випливають з них.

Для кожного рядка вихідного коду: Для заданого вихідного файлу анотатор ранжує кожен рядок вихідного коду відповідно до його енергетичної вартості за всіма *P* eP. Потім ранжування зіставляються з колірним спектром, наприклад, від синього до червоного, і кожен рядок вихідного коду забарвлюється на основі його положення в спектрі. Це призводить до візуалізації енергоспоживання програмного забезпечення, подібної до SeeSoft [7]. На рисунку 3.2 показано скріншот вихідного файлу Java з розфарбовуванням на основі енергії.

Представлення методу: Для заданого вихідного файлу анотатор вихідного коду генерує граф викликів (CG) програмного артефакту. Методи в CG ранжуються відповідно до споживання енергії за всіма P eP, а потім їм призначається колір у спектрі на основі їх відносного використання енергії. Призначений методу колір та значення енергії потім використовуються для анотації відповідного вузла в CG.

3.2 Методи зменшення енергоємності та оптимізація функціональності програмного забезпечення

У цьому підрозділі ми представляємо адаптивне багатоцільове розвантаження хмари (AMCO). Спочатку ми даємо огляд підходу , а потім по черзі описуємо його основні частини. На завершення ми обговорюємо застосовність та

обмеження цього підходу.

A. Огляд підходу

На рисунку 3.3 показано робочий процес AMCO. Підхід складається з двох основних частин – перетворення розділення на основі аналізу програми та адаптивної системи виконання.

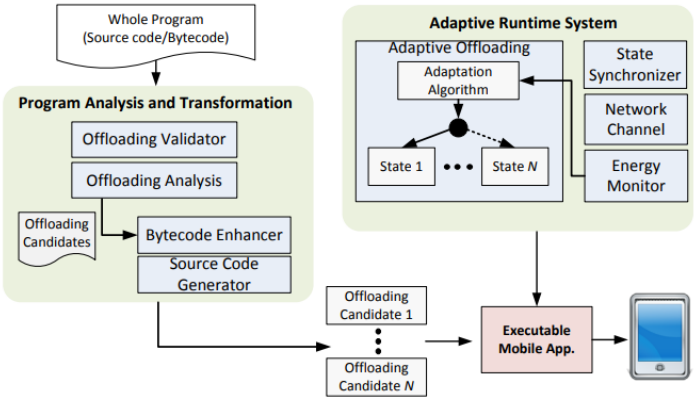


Рисунок 3.3 - Загальна процедура адаптивного розвантаження хмари для багатьох цільових груп.

Модель програмування AMCO є простою: програміст позначає компоненти, які підозрюються як енергетичні гарячі точки. В AMCO компоненти можуть бути визначені на будь-якому рівні деталізації програми, причому найменшими є окремі методи, а найбільшими – колекція пакетів. Для позначення компонентів гарячих точок AMCO надає спеціальну анотацію Java @Off loadingCandidate ; цю інформацію також можна вказати через файл конфігурації XML . На основі цих вхідних даних механізм аналізу спочатку перевіряє, чи можна вивантажити вказаний компонент , а також будь-які його підкомпоненти (тобто наступники в графі викликів). Механізм також обчислює стан програми, який потрібно передати між віддаленим та локальним розділами, який необхідно передати для розвантаження виконання як усього компонента, так і будь-якого з його підкомпонентів. Потім підсилювач байт-коду генерує контрольні точки, які зберігають та відновлюють обчислений стан для всіх компонентів гарячої точки, а також для кожного з його підкомпонентів. Під час виконання адаптивна система виконання постійно контролює енергію, споживану кожним компонентом-кандидатом на

розвантаження, з розбивкою за кожним з його складових підкомпонентів. На основі оціненого споживання енергії середовище виконання розвантажує ті підкомпоненти, хмарне виконання яких заощадить найбільшу кількість енергії за наявних мережевих умов [39]. Середовище виконання також синхронізує віддалені та локальні стани (зберігаючи при цьому псевдоніми як у локальних, так і у віддалених купах). Ще одним обов'язком системи виконання є відмовостійкість - обробка тимчасових відключень мережі та нестабільності.

В. Аналіз та трансформація програми

Щоб створити евристику аналізу програми, яка може розраховувати перетворення програми, що дозволяють розвантаження кількох цілей, ми розширили нашу попередню евристику для статичного розвантаження [8].

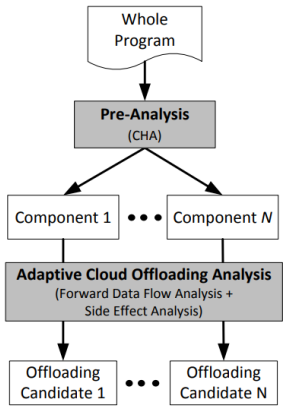


Рисунок 3.4 - Аналіз програми для адаптивного розвантаження хмари для багатьох цільових ресурсів.

На рисунку 3.4 показано процедуру аналізу, яка включає побудову графа викликів за допомогою аналізу ієрархії класів та визначення стану, який потрібно передати або синхронізувати, за допомогою аналізу прямого потоку даних та побічних ефектів. На етапі попереднього аналізу збираються всі компоненти та підкомпоненти, позначені @OffloadingCandidate, а потім основний аналіз визначає стан, який потрібно передати для кожного сценарію розвантаження [40]. Однією з технічних переваг основного аналізу є те, що він зменшує обсяг стану, який потрібно передати по мережі. Необхідність передачі великих обсягів даних по

мережі може швидко звести нанівець переваги споживання енергії, що забезпечуються віддаленим розвантаженням. Щоб уникнути необхідності передавати весь стан, наш підхід використовує ці аналізи прямого потоку даних та побічних ефектів для зменшення розміру переданого стану, що робить передачу стану практичною для оптимізації енергоспоживання. Алгоритм перевіряє оператори присвоєння та виклику, щоб визначити, чи змінився поточний стан під час розвантаження хмари.

Після визначення кандидатів на розвантаження, підсилювач байт-коду трансформує їх, щоб мати змогу запускати їхні компоненти та підкомпоненти гарячої точки на клієнті та сервері за потреби для реалізації заданого сценарію розвантаження. Потім адаптивна система виконання контролює середовище виконання та визначає оптимального кандидата на розвантаження. На рисунку 3.5 показано, як трансформується вихідний код централізованого мобільного застосунку.

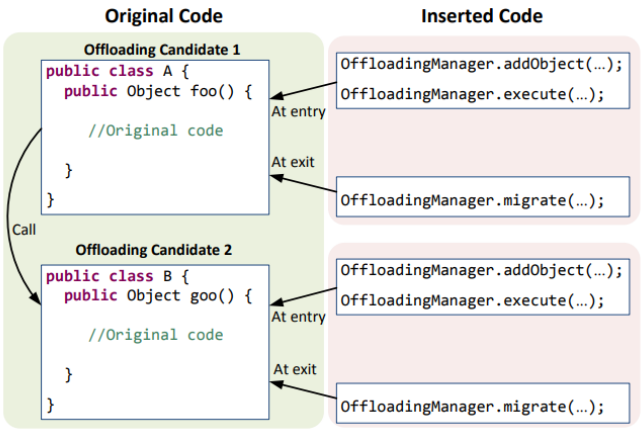


Рисунок 3.5 - Приклад перетворення програми.

Підсилювач байт-коду вставляє код , який може контролювати та відновлювати необхідний стан програми на вході та виході потенційно вивантажених методів відповідно. Ці методи включають кандидатів на розвантаження та їхніх наступників у CO-ECG.

A. Адаптивна система виконання

На рисунку 3.6 показано конструкцію адаптивної системи виконання AMCO, яка складається з трьох основних компонентів: (1) прогнозування та керування розвантаженням хмари, (2) оцінка споживання енергії та (3) обробка розвантаження хмари. Шляхом постійного моніторингу середовища виконання, система виконання інтелектуально співвідносить зібрану інформацію, щоб запропонувати оптимальні стратегії розвантаження.

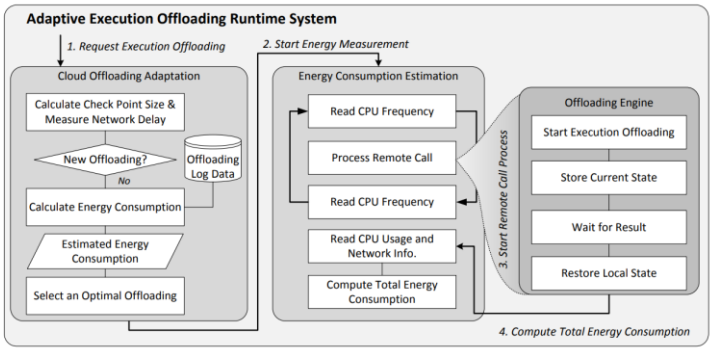


Рисунок 3.6 - Деталі процесу адаптивного перенесення виконання.

1) Прогнозування та керування розвантаженням хмари: На рисунку 3.7 показано процедуру прогнозування та керування розвантаженням хмари для кількох цілей. Модуль прогнозує майбутнє споживання енергії, аналізуючи зібрані дані про виконання середовище, а потім вибирає стратегію розвантаження, яка забезпечить найнижче майбутнє споживання енергії. Спочатку майбутнє споживання енергії обчислюється за допомогою затримки, яка вимірюється шляхом надсилання зондуючого пакета на сервер розвантаження. Потім поточна затримка перераховується шляхом надання ваги нещодавно отриманому значенню (тобто *затримка = затримка ха + затримка х* (1 - а)), тим самим уникаючи піку затримки, а також частково відображаючи останнє значення затримки. Нарешті, модуль обчислює стратегію розвантаження, яку пропонує, вибираючи найменше прогнозоване майбутнє споживання енергії з усіх доступних кандидатів на розвантаження (тобто підграфів CO-ECG).

```
adaptiveOffloading() {
  delay ⟵ delay × α + currentDelay() × (1 − α);

  while(∀S) {
    estimation ⟵ computeEnergyConsumption(delay, S_n);

    if (estimation is the lowest) {
      if (estimation > local execution) {
        return ExecuteLocal();
      } else {
        newState ⟵ execute(cp);
        stateMigration(newState, state);
        updateHistory(state, delay, estimation);
        return state;
    } } }
}
```

Лістинг 3.2 - Обчислення стратегії розвантаження

2) Оцінка енергоспоживання: Модуль оцінки енергоспоживання розраховує енергоспоживання до та після операції перенесення на хмарні ресурси. Для оцінки ефективності перенесення на хмарні ресурси ми обчислюємо лише енергоспоживання процесора та мережевих комунікацій таким чином:

$$E = \left\{P_{cpu_{freq}^{active}} \times \left(T_{cpu_{time}^{user}} + T_{cpu_{time}^{sys}}\right) + \left(P_{net}^{active} \times T_{net}^{active}\right) + \left(P_{net}^{idle} \times T_{net}^{idle}\right)\right\} \times V \quad (3.1)$$

де P_c“$_p$^{c}U7$_r^e$$_{eq}$– це потужність, що споживається процесором. Сучасні процесори мають функцію зміни швидкості (speed-step), яка дозволяє операційній системі динамічно змінювати тактову частоту процесора, з різними рівнями споживання енергії на кожній тактовій частоті. Наприклад, точка доступу Samsung Galaxy S III забезпечує п'ять кроків, від 302,4 МГц до 1512 МГц, а кількість споживаної енергії на кожній швидкості коливається від 55 мА до 577 мА. T $_c$“ ^{s}„ ^{er} - ^{time} та T $_c$ ^{s} pUS- ^{time} – це час користувача та час системи, виміряні операцією розвантаження, і вони отримуються шляхом звернення до статистики, наданої операційною системою. V – це поточна напруга, яка отримується за допомогою API, пов'язаних з батареєю (наприклад, **клас** BatteryStat в ОС Android). P^Cf^ та P^Cf^ – це кількість енергії, яку мережевий процесор споживає відповідно під час активної та простою фаз. Наприклад, коефіцієнт споживання енергії в активному та простої режимі для Samsung Galaxy S III становить 96. мА/0,3 мА під час зв'язку через WiFi та 250 мА/3,4 мА під час мобільного зв'язку (наприклад, 4G).

Нарешті, T – це періоди активності та простою під час операції розвантаження хмари, виміряні під час виконання. Ці значення, специфічні для пристрою та виконання, використовуються для обчислення кількості енергії, споживаної під час кожної оптимізації розвантаження.

Ще одним важливим обов'язком системи виконання є прогнозувати майбутнє споживання енергії. Для прогнозування енергії, яка ймовірно буде споживатися під час оптимізації розвантаження, здійснюється співвіднесення попередньо виміряного споживання енергії з поточним середовищем виконання. Отримавши поточні значення затримки мережі, типу підключення, частоти процесора та напруги, майбутнє споживання енергії обчислюється наступним чином:

$$E_{est} = \left\{E_{cpu}^{avg} + \left(P_{net}^{active} \times T_{net_{active}^{est}}\right) + \left(P_{net}^{idle} \times T_{net_{idle}^{est}}\right)\right\} \times V \qquad (3.2)$$

де $_{Eest}$ – прогнозоване майбутнє споживання енергії, – середнє споживання енергії для даного віддаленого виклику, а TnSb“ ^{ctive} – розрахунковий час зв'язку, який обчислюється з використанням відповідно розміру вивантажених даних та затримки . Зрештою, на основі розрахункового часу зв'язку та попередніх виконань, система виконання прогнозує майбутнє споживання енергії. Обчислене споживання енергії використовується для визначення оптимальної стратегії розвантаження.

3) Механізм розвантаження хмари: Механізм розвантаження хмари керує з'єднанням між сервером розвантаження та мобільним клієнтом, синхронізує стан контрольних точок та забезпечує стійкість у разі збою через нестабільність мережі. Стан контрольних точок синхронізується за допомогою копіювання-відновлення, розширеної семантики, впровадженої в проміжне програмне забезпечення виклику віддалених методів з метою передачі як параметрів структур даних (наприклад, зв'язаних списків, дерев та карт) [21]. Копіювання-відновлення копіює весь доступний стан на сервер, а потім перезаписує стан клієнта зміненими даними сервера. Щоб адаптуватися до роботи через стільникові мережі з

обмеженою пропускною здатністю, ми модифікували оригінальну реалізацію копіювання-відновлення, щоб використовувати розріджені масиви, які ефективно кодують простір нульових значень. Наша реалізація використовує нульові значення для позначення частини переданого стану, яка не була змінена під час операцій розвантаження.

На рисунку 3.7 показано, як система виконання синхронізує контрольований стан.

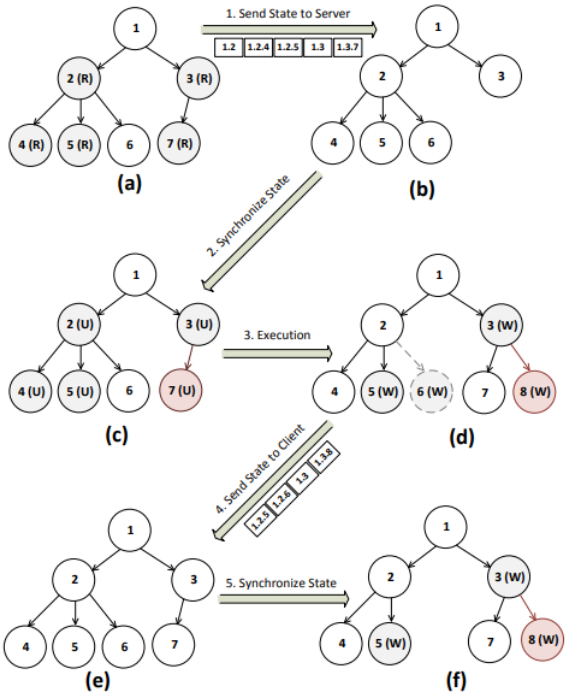


Рисунок 3.7 - Процедура синхронізації двох різних станів.

Графік (a) зображує стан мобільного пристрою, який буде передано на сервер. Система виконання передає ¬лише ті вузли, які аналіз визначив як такі, що використовуються сервером (вузли 2, 3, 4 та 7). Графік (b) зображує стан сервера до його синхронізації з переданим станом. Вузли 2, 4 та 5 оновлюються новими значеннями; вузол 3 перепризначається вузлу 3.7. Графік (c) показує синхронізований стан сервера. У цьому прикладі виконання розвантаженого сервера призначає новий екземпляр, вузол 8, вузлу 3, змінює вузли 3 та 5 та призначає значення null вузлу 6, а графік (d) зображує результати. Потім змінений стан - передається клієнту та синхронізується зі своїм станом, зображеним на графіку

(e). Зокрема, вузол 6 видаляється, вузол 3 перепризначається вузлу 8, а вузли 3 та 5 перезаписуються новими значеннями. Графік (f) показує стан клієнта після синхронізації.

A. Обговорення

У цьому розділі ми обговорюємо переваги та обмеження нашого підходу – адаптивного багатоцільового розвантаження хмари (AMCO).

1) *Переваги:* AMCO працює зі стандартним, немодифікованим апаратним/програмним стеком; він використовує байт -кодову інженерію для перетворення програм та легку систему виконання для динамічного керування та адаптації розвантаження. AMCO дозволяє зберегти версію вихідного коду мобільного застосунку недоторканою, оскільки перетворюється лише версія байт-коду. AMCO вимагає мінімальних зусиль програмування, обмежених позначенням методів як енергетичних точок. AMCO приймає рішення щодо розвантаження під час виконання, моніторячи середовище виконання, таким чином виявляючи оптимальні стратегії розвантаження. Нарешті, перетворення розвантаження AMCO не перешкоджають локальному виконанню мобільного застосунку у разі відключення мережі

2) *Обмеження:* Незважаючи на свої переваги, AMCO не може бути застосований для оптимізації енергоспоживання всіх мобільних додатків. Підхід AMCO є автоматизованим, а не автоматичним; він покладається на програміста для визначення енергоємних методів. Розвантажуючи на межах методів, AMCO спирається на досліджувані додатки, дотримуючись загальноприйнятих принципів якісного проектування програмного забезпечення (тобто інкапсуляції , модульності, слабкого зв'язку), так що розвантажені методи інкапсулюють окрему функціональність, слабко пов'язану з рештою додатка. Модель розвантаження AMCO також вимагає, щоб час життя всіх потоків у розвантажених методах збігався з межами методів. Тобто, розвантажені методи можуть вільно використовувати кілька потоків, але Очікується, що всі потоки завершаться, коли методи припиняють виконуватися. Як і у всіх системах розділення, що залежать від байт-кодової інженерії, AMCO може розділяти лише невласний (тобто виражений

виключно в байт-коді) стан [20]. Зрештою, розвантаження може збільшити затримки виконання, тим самим погіршуючи взаємодію користувача з високоінтерактивними програмами.

3.3 Аналіз результатів оптимізації за допомогою бенчмаркінгу та тестування

Ми оцінили наш підхід за допомогою мікро-бенчмарку та більшого тематичного дослідження. Результати показують, що наша система виконання повідомляє корисну інформацію про навколишнє середовище, не створюючи надмірних витрат на продуктивність та енергоспоживання. Крім того, наш загальний підхід може ефективно зменшити кількість споживаної енергії для добре спроектованих програм, оскільки впроваджені трансформації програм та виконання ніколи не призводять до перевищення вдосконалених рівнів споживання енергії вдосконаленими програмами.

А. Мікро-бенчмарк

Мета цього мікробенчмарку - зрозуміти накладні витрати, що накладаються системою виконання, обов'язки якої включають моніторинг відповідних коливань у середовищі, оцінку потенційної економії енергії завдяки можливим крокам розвантаження та синхронізацію куп під час розвантаження.

Ми базували наш набір тестів на бенчмарках, спочатку запропонованих проектом JavaParty [7], який широко використовується для бенчмаркінгу реалізацій проміжного програмного забезпечення. Ці бенчмарки включають віддалені виклики з різними розмірами та типами параметрів. Аналогічно, наш набір тестів припускає, що клієнту потрібно виконати деякі серверні методи, кожен з яких приймає параметри, що відрізняються за розміром та типом. Оскільки виконувані серверні методи мають порожні тіла, можна обґрунтовано віднести енергію, споживану під час їх виклику, до базової системи виконання.

1) Експериментальна установка: Експериментальна установка складалася з Motorola Droid (процесор 600 МГц, 256 МБ оперативної пам'яті, 802.11g,

3G) та Samsung Galaxy III (двоядерний процесор 1,5 ГГц, 2 ГБ оперативної пам'яті, 802.11n, 4G) як мобільного пристрою, а також ПК Dell (чотириядерний процесор 3,0 ГГц, 8 ГБ оперативної пам'яті) як розвантаженого сервера. Мобільний пристрій зв'язувався із сервером через WiFi, мережу 3G та мережу 4G. Для WiFi-з'єднання ми експериментували з двома емульованими мережевими умовами, які мають такі відповідні характеристики часу передачі даних (RTT) та пропускної здатності: 2 мс та 50 Мбіт/с, типові для мобільної мережі високого класу, та 50 мс та 1 Мбіт/с, типові для мобільної мережі середнього класу. Для мобільного з'єднання ми використовували мережу 3G для Motorola Droid та мережу 4G для Samsung Galaxy. У таблиці I наведено енергетичні профілі цих мобільних пристроїв. Ці значення, специфічні для пристрою, параметризують систему виконання.

Значною мірою саме апаратні характеристики мобільного пристрою визначають його профіль споживання енергії. Наприклад, відомо, що нанотехнологічний процес зменшує кількість енергії, споживаної сучасними процесорами. Аналогічно, новітні мережеві та радіочіпсети покращили свою енергоефективність. Тому характеристики, характерні для кожного пристрою, відіграють ключову роль в оцінці енергії, споживаної певним мобільним пристроєм.

Таблиця 3.1 -

Енергетичні характеристики, надані виробником

Компонент	**Samsung Galaxy S III**	**Motorola Droid**
Процесор (CPU)	1512.0 МГц: 577 мА	800.0 МГц: 280 мА
	1209.6 МГц: 408 мА	685.7 МГц: 236 мА
	907.2 МГц: 249 мА	571.4 МГц: 207 мА
	604.8 МГц: 148 мА	342.8 МГц: 165 мА
	302.4 МГц: 55 мА	228.5 МГц: 87 мА
	Н/Д	114.2 МГц: 66 мА
WiFi	96 мА	130 мА
	0.3 мА	4 мА

2) Оцінка споживання енергії: Спочатку ми оцінили, наскільки точно система виконання може передбачити, скільки енергії буде спожито за заданий часовий інтервал. На рисунку 3.8 порівнюється споживання енергії, передбачене системою виконання, та споживання енергії, оцінене нашою моделлю на основі фактичного використання ресурсів. Середня похибка становить 10,6%, а стандартне відхилення – 21,3%. Якщо врахувати лише 90% даних без викидів, середня похибка становить 8,5%, а стандартне відхилення – 6,8%. Ці результати свідчать про те, що система виконання може достатньо точно передбачити майбутнє споживання енергії, при цьому розбіжності становлять в середньому 6-7%.

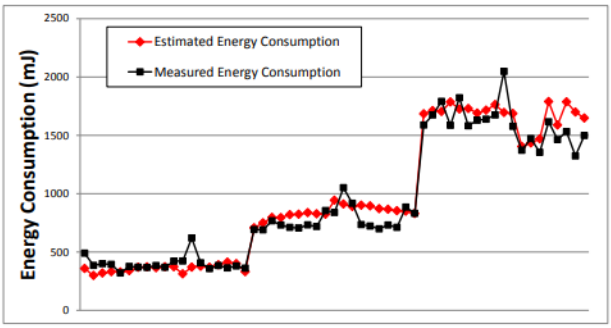
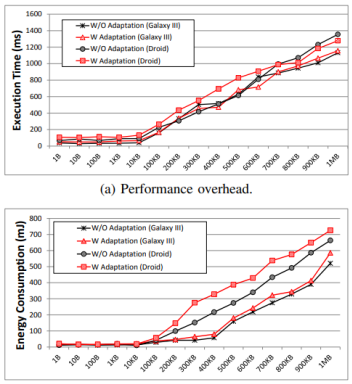


Рисунок 3.8 - Оцінка споживання енергії.

3) Накладні витрати: У цьому бенчмарку ми порівняли загальний час виконання та споживання енергії базових версій бенчмарків зі споживанням енергії за наявності системи виконання AMCO. Перший графік на рисунку 3.9 показує загальний час виконання, виміряний на двох пристроях . Як бачимо, накладні витрати на продуктивність досить незначні. Зокрема, накладні витрати для обох пристроїв ніколи не перевищують 100 мс і залишаються постійними для всіх виміряних розмірів передачі даних. Другий графік показує енергію, споживану кожним пристроєм. Як і очікувалося, пристрій високого класу (Samsung Galaxy) споживає менше енергії, ніж пристрій низького класу (Motorola Droid). Незважаючи на те, що пристрій низького класу має більші накладні витрати, ніж пристрій високого класу, фактичне число становило 100 мДж, що є незначним значенням порівняно із загальною енергією, яку зазвичай споживає мобільний додаток.



(a) Performance overhead.

Рисунок 3.9 - Витрати на продуктивність та споживання енергії.

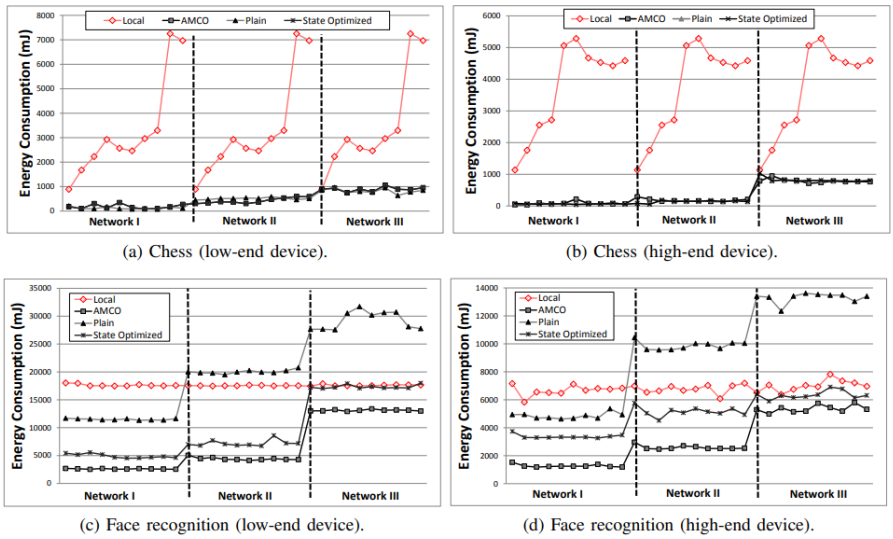
Тематичне дослідження

Щоб визначити, чи застосовний наш підхід до реальних мобільних застосунків, ми експериментували з проектами з відкритим кодом як об'єктами експерименту. Кишенькові шахи– це мобільна шахова гра, рушій штучного інтелекту якої, що міститься в класі SimpleEngine, був позначений як @OffloadCandidate. JJIL – це програма для розпізнавання облич, функціональність розпізнавання якої, що міститься в класі DetectHaarParam , була позначена як @OffloadCandidate .

На рисунку 3.10 показано, як підхід AMCO зменшив кількість енергії, споживаної суб'єктами дослідження. Для кожного суб'єкта дослідження ми представляємо чотири графіки, що показують кількість енергії, споживаної типовими, простими варіантами використання. Зокрема, для шахової програми ми перемістили одну випадково вибрану шахову фігуру; для програми розпізнавання облич ми перевірили один файл зображення на наявність людських облич. Варіанти використання були виконані за такими чотирма режимами оптимізації: (1) оригінальне централізоване виконання (базовий рівень), (2) розвантаження простої хмари (розвантаження всього дерева викликів, що корениться в методі, позначеному @Off loadCandidate) , (3) те саме, що й (2), але з мінімізацією переданого стану купи за допомогою аналізу програми, та (4) наш підхід, AMCO.

Оптимізовані версії досліджуваних програм споживали менше енергії, ніж їхні оригінальні та прості версії з розвантаженням у хмарі. Типова стратегія розвантаження перевантажує в хмару важку обробку процесора, необхідну для

розрахунку наступного ходу, і передає назад лише нову позицію для руху фігури.



(a) Chess (low-end device). (b) Chess (high-end device).

(c) Face recognition (low-end device). (d) Face recognition (high-end device).

Рисунок 3.10 - Споживання енергії досліджуваними застосуваннями.

Через цю оптимальну архітектурну властивість шахової програми, її оптимізації споживають від 10% до 90% енергії, що споживається оригінальною програмою. У міру проходження гри оптимізовані версії демонструють постійну швидкість споживання енергії, тоді як оригінальна версія споживає все більшу кількість енергії, оскільки необхідна обробка штучним інтелектом інтенсифікації. Як і очікувалося, навіть найпростіша оптимізація енергоспоживання дає значну економію енергії, без будь-яких відчутних переваг, які дає використання адаптивної системи виконання AMCO.

У випадку програми розпізнавання облич усі три стратегії оптимізації продемонстрували різні рівні ефективності. По-перше, простий підхід до розвантаження хмари може економити енергію лише за найсприятливішого мережевого середовища (Wi-Fi). По-друге, оптимізація обсягу переданого стану демонструє постійне покращення обсягу споживаної енергії. По-третє, коли середовищу виконання наказано не контролювати середовище, обсяг споживаної енергії у розвантаженій версії фактично перевищує обсяг споживаної енергії в оригінальній централізованій версії . Нарешті, адаптивна здатність розвантаження нашого підходу AMCO, здається, є ключовою для постійної економії обсягу

споживаної енергії. Зокрема, наш підхід AMCO оптимізує програму для споживання на 10%-80% менше енергії, ніж оригінальна локальна версія, а також на 25%-50% менше енергії, ніж сучасний статичний підхід до розвантаження хмари [8].

C. Загрози валідності

Наведені вище експериментальні результати піддаються як внутрішнім, так і зовнішнім загрозам валідності. Внутрішня валідність знаходиться під загрозою через спосіб запуску досліджуваних програм. Оскільки досліджувані програми передбачають взаємодію з користувачем, їхня поведінка та споживання енергії залежать від дій користувача (наприклад, вибір певної шахової фігури для ходу або зйомка певного зображення). Ці взаємодії з користувачем можуть суттєво впливати на те, скільки енергії буде споживатися.

Зовнішня валідність знаходиться під загрозою через механізм, який використовується системою виконання AMCO для вимірювання споживання енергії. Замість того, щоб вимірювати фізично спожиту енергію безпосередньо, вона оцінює споживання енергії на основі фактичної інформації про використання ресурсів. Як результат, розрахункове споживання енергії, ймовірно, буде менш точним, ніж те, що було б виміряно за допомогою спеціалізованого обладнання. Крім того, енергетичні профілі, які ми використовували в системі виконання, надаються виробниками, тому точність наших вимірювань залежить від точності наданих енергетичних профілів.

Під час створення підходу AMCO ми спостерігали позитивну кореляцію між бажаними властивостями програмної інженерії (тобто сильною модульністю, високою когезійністю, низьким рівнем зв'язку тощо) застосунку та його придатністю для оптимізації розвантаження хмари. На основі цих спостережень ми далі представляємо три ідеальні рекомендації щодо обслуговування, які програміст може слідувати інструкціям, щоб зробити свої мобільні додатки краще придатними для ефективної оптимізації розвантаження хмари.

a) *Застосування рефакторингу методу розділення:* Метод є однією з найперших абстракцій для розділення завдань. Добре розроблений метод в ідеалі

повинен містити одну цілісну одиницю функціональності. Загалом, чим більший метод, тим менш він цілісний. Як виявляється, великі методи можуть бути громіздкими як одиниці розвантаження хмари, особливо у випадку дрібнозернистого, адаптивного розвантаження, як в AMCO. З огляду на це спостереження, ми рекомендуємо перевірити енергоємні методи, позначені @Off loadCandidate , на їхню цілісність, а потім за необхідності рефакторувати. Рефакторинг методів розділення розділення розділяє функціональність одного методу на окремі методи, що викликають один одного. Менші методи інкапсулюють концептуально цілісні одиниці функціональності. Як результат, дрібнозернисті, добре модульовані мобільні додатки є не тільки бажаними з точки зору розуміння програмного забезпечення, але й добре піддаються ідеальному обслуговуванню за допомогою розвантаження хмари.

b) *Інкапсуляція нативного стану:* Хоча нативний код зазвичай робить навколишній байт-код незмінним [20], програміст може керувати інструментами розвантаження, надаючи API байт-коду, який отримує доступ та синхронізує частину стану, реалізовану в нативному коді. Оскільки нативний код може бути неможливо проаналізувати, програміст несе відповідальність за те, щоб API обгортки правильно синхронізував нативний стан без будь-яких шкідливих побічних ефектів.

Усунення спільного використання хибних станів: В об'єктно-орієнтованих мовах програмування всі поля-члени класу можуть бути доступні всіма його методами. Однак метою полів-членів є визначення стану, призначення якого залишається незмінним протягом усього життя об'єктів класу. Поширеним недоліком проектування є використання одного й того ж поля-члена в кількох методах класу в різних можливостях. Тобто, замість того, щоб оголошувати поле локально в кожному методі, кілька методів використовують одне й те саме поле-член, умову, яку ми називаємо спільним використанням хибних станів. Спільне використання хибних станів є серйознішим недоліком проектування, ніж може здатися на перший погляд, оскільки проблеми, які він викликає, подібні до тих, що викликаються глобальними змінними в процедурних мовах. Що стосується

розвантаження хмари, спільне використання хибних станів ускладнює аналіз програми та синхронізацію станів, тим самим обмежуючи обсяг функціональності, який можна ефективно розвантажити. Тому ми рекомендуємо, щоб в рамках ідеального обслуговування для оптимізації енергоспоживання програмісти максимально усунули спільне використання хибних станів шляхом рефакторингу.

Збільшення терміну служби батареї шляхом зменшення споживання енергії мобільними додатками було предметом багатьох додаткових досліджень, таких як більш енергоефективні операційні системи (наприклад, енергоефективне планування процесора [26], управління живленням дисків [24] та міграція процесів [1], мережеві протоколи [25]), методи розвантаження на рівні віртуальних машин [17] та на рівні додатків [4]. Хоча більшість цих зусиль зосереджена на одному конкретному системному рівні (тобто, головним чином, мережі), техніка, яка називається міжшаровим підходом, ефективно контролює споживання енергії, використовуючи інформацію, що надається кількома системними рівнями [5], [11]. Кілька програмних абстракцій дозволяють ефективно адаптуватися, що використовують кілька стратегій оптимізації. Платформа Odyssey [5] адаптує дані або якість обчислень для економії споживання енергії, щоб не перевищувати доступні системні ресурси. Ці енергозалежні адаптації можуть виявляти можливі компроміси між споживанням енергії та якістю додатків, вибираючи стратегію управління енергією на основі умов виконання. DYNAMO [11] – це ще одна платформа проміжного програмного забезпечення, яка адаптує стратегії оптимізації енергоспоживання на різних системних рівнях, включаючи додатки, проміжне програмне забезпечення, ОС, мережу та апаратне забезпечення, для оптимізації як продуктивності, так і енергоспоживання .

Хоча багато досліджень зосереджено на рішеннях системного рівня, підходи на рівні програмування (наприклад, алгоритми [12], шаблони проектування [10], моделі програмного забезпечення [19]) отримали мало уваги в літературі. Нещодавнім мовним підходом до енергооцінювального програмування є ET [3], нова об'єктно-орієнтована мова програмування, яка дозволяє програмісту писати енергооцінювальний код.

На противагу цьому, АМСО зосереджується на ідеальному обслуговуванні, хоча й запозичує методи як з програмних, так і з системних підходів до оптимізації енергоспоживання. Новизна АМСО полягає в поєднанні трансформації програми на основі аналізу та адаптації під час виконання.

Представлено новий підхід до ідеального обслуговування, спрямований на зменшення енергоспоживання мобільних додатків, адаптивне багатоцільове розвантаження хмари (АМСО). Наш підхід зменшує енергоспоживання мобільних додатків без зміни їхнього вихідного коду, використовуючи потужний аналіз та перетворення програм, а також адаптивну систему виконання, яка визначає оптимальну стратегію розвантаження під час виконання. Ми оцінили наш підхід, зменшуючи енергоспоживання мікробенчмарками та сторонніми додатками в різних середовищах виконання. Ці результати свідчать про те, що наш підхід є перспективним напрямком у розробці прагматичних та систематичних інструментів для ідеального
обслуговування мобільних додатків.

3.4 Висновок по розділу

У третьому розділі було розглянуто реалізацію та оцінку ефективності методів оптимізації енергоспоживання мобільних застосунків. Проведено реалізацію підходів до підвищення енергоефективності програмного забезпечення з урахуванням особливостей мобільних платформ. Досліджено методи зменшення енергоємності та оптимізації функціональності застосунків для скорочення використання апаратних ресурсів і підвищення автономності пристроїв. Також виконано аналіз результатів оптимізації за допомогою бенчмаркінгу та тестування, що дозволило оцінити ефективність застосованих рішень. Отримані результати підтверджують доцільність використання сучасних методологій оптимізації для створення енергоефективного мобільного програмного забезпечення.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено методології оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв та проаналізовано сучасні підходи до підвищення енергоефективності мобільних застосунків. У роботі розглянуто особливості функціонування мобільних платформ, досліджено вплив програмних компонентів на споживання енергії акумулятора та визначено основні фактори, що впливають на продуктивність і автономність мобільних систем. Процес дослідження охопив аналіз теоретичних основ енергоспоживання, вивчення інструментів енергетичного профілювання, моделювання енергоефективних рішень, реалізацію підходів до оптимізації та оцінку їх ефективності за допомогою бенчмарків і тестування.

У першому розділі було розглянуто теоретичні основи енергоспоживання мобільного програмного забезпечення, визначено основні проблеми оптимізації використання енергії та досліджено технічні аспекти функціонування мобільних пристроїв. Значну увагу приділено аналізу процесорних ресурсів, роботи пам’яті, мережевих модулів, сенсорів та фонових процесів, які суттєво впливають на рівень енергоспоживання. Також було досліджено модельно-орієнтовані підходи до енергетичного профілювання, що дозволяють оцінювати витрати енергії під час виконання окремих програмних операцій.

У другому розділі досліджено методи та інструменти оптимізації енергоспоживання в мобільних застосунках. Було проаналізовано сучасні методології моделювання енергетичних характеристик програмного забезпечення, розглянуто таксономію енергетичних профілювальників та інструментів моніторингу енергоспоживання. Особливу увагу приділено підходам до побудови моделей профілювання як безпосередньо на мобільному пристрої, так і з використанням зовнішніх засобів аналізу. У роботі також розглянуто засоби автоматизованого тестування, профілювання продуктивності та оптимізації ресурсів мобільних платформ Android і iOS.

У третьому розділі було реалізовано підхід до стимулювання енергоефек-

тивності мобільних застосунків шляхом оптимізації обчислювальних процесів, мережевої взаємодії та роботи з фоновими службами. Проведено аналіз механізмів розвантаження енергоємних задач, використання асинхронних операцій, кешування даних та оптимізації частоти оновлення інтерфейсу. Оцінка ефективності реалізованих підходів виконувалася за допомогою бенчмарків і тестових сценаріїв, що дозволило визначити рівень зменшення енергоспоживання та підвищення продуктивності мобільного програмного забезпечення.

У процесі дослідження було встановлено, що застосування сучасних методологій оптимізації дозволяє значно знизити навантаження на апаратні ресурси мобільних пристроїв і підвищити тривалість автономної роботи без втрати функціональності програмного забезпечення. Використання енергетичного профілювання, модельного аналізу та адаптивних механізмів управління ресурсами сприяє підвищенню ефективності роботи мобільних застосунків і забезпечує кращий користувацький досвід.

Загалом результати роботи підтверджують, що оптимізація енергоспоживання є важливою складовою сучасної розробки програмного забезпечення для мобільних платформ. Використання енергоефективних алгоритмів, адаптивних механізмів керування ресурсами та інструментів моніторингу дозволяє створювати більш продуктивні, стабільні та економічні мобільні застосунки. У перспективі результати дослідження можуть бути використані для вдосконалення систем автоматичного управління енергоспоживанням, інтеграції технологій штучного інтелекту для прогнозування енергетичних витрат та створення інтелектуальних мобільних платформ нового покоління.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Android Developers. (2025). Optimize battery use for task scheduling APIs. developer.android.com. https://developer.android.com/develop/background-work/background-tasks/optimize-battery

2. Android Studio Profiler Documentation. (2025). developer.android.com. https://developer.android.com/studio/profile

3. Apple Developer Documentation. (2025). Energy Efficiency Guide for iOS Apps. developer.apple.com. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/

4. Azure DevOps. (2025). CI/CD Pipeline. learn.microsoft.com. https://learn.microsoft.com/en-us/azure/devops/pipelines/

5. Basoglu, C., & Andrew, P. (2024). Energy-aware software optimization techniques for mobile applications. Journal of Systems and Software, 210, 111954. https://doi.org/10.1016/j.jss.2024.111954

6. Bootstrap. (2025). Documentation. getbootstrap.com. https://getbootstrap.com/docs/

7. Carroll, A., & Heiser, G. (2010). An analysis of power consumption in a smartphone. Proceedings of the USENIX Annual Technical Conference, 21–21. https://www.usenix.org/conference/usenix-atc-10/analysis-power-consumption-smartphone

8. Chen, X., & Wang, Y. (2023). Adaptive energy optimization for mobile computing systems. IEEE Access, 11, 56782–56795. https://doi.org/10.1109/ACCESS.2023.3284561

9. Clean Code Principles in .NET. (2023). devblogs.microsoft.com. https://devblogs.microsoft.com/dotnet/clean-code-principles/

10. Cypress End-to-End Testing. (2025). docs.cypress.io. https://docs.cypress.io/guides/overview/why-cypress

11. Deloitte. (2024). Sustainable software engineering for mobile platforms. deloitte.com. https://www2.deloitte.com/global/en/pages/technology/articles/sustainable-software-engineering.html

12. Energy Profiler for Android Applications. (2024). source.android.com. https://source.android.com/docs/core/power

13. Fowler, M. (2002). Patterns of Enterprise Application Architecture (pp. 5–10). martinfowler.com. https://martinfowler.com/eaaCatalog/

14. Freeman, E., & Robson, E. (2021). Head First Design Patterns. oreilly.com. https://www.oreilly.com/library/view/head-first-design/9781492077992/

15. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. pearson.com. https://www.pearson.com/store/p/design-patterns-elements-of-reusable-object-oriented/P100000252058

16. Google Android Battery Historian. (2025). developer.android.com. https://developer.android.com/topic/performance/power/setup-battery-historian

17. Green Software Foundation. (2024). Software Carbon Intensity Specification. greensoftware.foundation. https://greensoftware.foundation/projects/software-carbon-intensity-specification

18. Gupta, R., & Singh, P. (2025). Machine learning approaches for energy-efficient mobile applications. ACM Transactions on Embedded Computing Systems, 24(2), 1–24. https://doi.org/10.1145/3701458

19. Hassan, S., & Ahmed, K. (2023). Dynamic voltage and frequency scaling techniques in mobile systems. IEEE Transactions on Mobile Computing, 22(9), 4983–4996. https://doi.org/10.1109/TMC.2023.3245127

20. IBM Cloud Documentation. (2025). Mobile application performance optimization. ibm.com. https://www.ibm.com/docs/en/cloud

21. ImpactQA. (2024). Mobile application testing and performance optimization. impactqa.com. https://www.impactqa.com/mobile-application-testing-services/

22. iOS Instruments Documentation. (2025). developer.apple.com. https://developer.apple.com/documentation/xcode/analyzing-your-app-s-performance

23. Kumar, S., & Sharma, R. (2024). Profiling and benchmarking energy-efficient mobile applications. Journal of Software Engineering Research and Development, 12(1), 34–49. https://doi.org/10.1186/s40411-024-00201-3

24. Li, H., & Zhou, T. (2024). Energy-aware task scheduling in mobile operating systems. Future Generation Computer Systems, 152, 88–101. https://doi.org/10.1016/j.future.2023.11.019

25. Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. pearson.com. https://www.pearson.com/store/p/clean-architecture-a-craftmans-guide-to-software-structure-and-design/P100001465201

26. Microsoft .NET Documentation. (2025). Performance and energy optimization. learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/core/performance/

27. MongoDB Documentation. (2025). mongodb.com. https://www.mongodb.com/docs/

28. .NET MAUI Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/dotnet/maui/

29. Osherove, R. (2019). The Art of Unit Testing. manning.com. https://www.manning.com/books/the-art-of-unit-testing-third-edition

30. OWASP Mobile Application Security. (2024). owasp.org. https://owasp.org/www-project-mobile-top-10/

31. Postman API Testing Guide. (2025). learning.postman.com. https://learning.postman.com/docs/

32. React Native Documentation. (2025). reactnative.dev. https://reactnative.dev/docs/getting-started

33. Samsung Developer Documentation. (2024). Galaxy battery optimization techniques. developer.samsung.com. https://developer.samsung.com/

34. SASS Basics. (2024). sass-lang.com. https://sass-lang.com/guide

35. Selenium Documentation. (2025). selenium.dev. https://www.selenium.dev/documentation/

36. Silva, D., & Costa, M. (2025). Benchmarking methodologies for mobile energy consumption analysis. Sustainable Computing: Informatics and Systems, 41, 100942. https://doi.org/10.1016/j.suscom.2024.100942

37. Wang, L., & Zhang, Y. (2025). Real-time energy analytics for mobile software systems. ACM Transactions on Software Engineering and Methodology, 34(1), 1–22. https://doi.org/10.1145/3698541

38. Xamarin Documentation. (2025). learn.microsoft.com. https://learn.microsoft.com/en-us/xamarin/

39. Zhang, J., & Lee, K. (2024). Cloud-assisted energy optimization for mobile devices. IEEE Transactions on Sustainable Computing, 9(3), 612–624. https://doi.org/10.1109/TSUSC.2024.3367812

40. Zhou, M., & Chen, Y. (2023). Energy-efficient software architectures for mobile computing environments. Journal of Parallel and Distributed Computing, 178, 15–29. https://doi.org/10.1016/j.jpdc.2023.03.012

БІБЛІОГРАФІЧНА ДОВІДКА

Методології оптимізації енергоспоживання в програмному забезпеченні для мобільних пристроїв"

Обсяг пояснювальної записки: 78 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____