Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Методи оптимізації системного програмного забезпечення для вбудованих систем
(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-3
Сергій КУЖЕЛЬ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Антон СОРОКІН
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ __________ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ________________
(підпис)
“____”________________ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Кужелю Сергію Ігоровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Методи оптимізації системного програмного забезпечення для вбудованих систем

затверджена наказом по університету від “ 21 ” квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи

оптимізація

вбудовані системи

Google COLAB

системне програмне забезпечення

4. Перелік питань, що потрібно опрацювати у роботі

Аналіз предметної області та проблеми оптимізації системного пз

Методичні основи оптимізації системного пз

Розробка та реалізація методу оптимізації системного пз для вбудованих систем

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій 15 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання та аналіз літератури	21.04.2025–30.04.2025	
2	Огляд існуючих рішень та методів	01.05.2025–12.05.2025	
3	Розробка методу	13.05.2025–22.05.2025	
4	Вибір програмних засобів	23.05.2025–30.05.2025	
5	Програмна реалізація	31.05.2025–02.06.2025	
6	Аналіз отриманих результатів	03.06.2025–05.06.2025	
7	Оформлення записки	06.06.2025–12.06.2025	

Дата видачі завдання “ 21 ” квітня 20 25 р.

Здобувач ____________
(підпис)

Керівник роботи ____________ доц. Антон СОРОКІН
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 74 с., 11 рис., 2 дод., 8 джерел.

ВБУДОВАНІ СИСТЕМИ, СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ОПТИМІЗАЦІЯ, ЕНЕРГОСПОЖИВАННЯ, ПЛАНУВАННЯ ЗАДАЧ, ЖАДІБНИЙ АЛГОРИТМ, МАШИННЕ НАВЧАННЯ, GOOGLE COLAB, ПРІОРИТЕТИ, РЕСУРСООБМЕЖЕННЯ.

Метою кваліфікаційної роботи дослідження існуючих методів та практична реалізація методу оптимізації системного програмного забезпечення для вбудованих систем з урахуванням сучасних підходів до керування ресурсами, енергоспоживанням і плануванням задач. У процесі реалізації передбачається використання симуляційної моделі у середовищі Google Colab для демонстрації та верифікації ефективності запропонованого методу.

У кваліфікаційній роботі розглянуто проблему оптимізації системного програмного забезпечення для вбудованих систем, що функціонують в умовах обмежених обчислювальних ресурсів та підвищених вимог до енергоефективності. Проведено огляд існуючих підходів до планування задач у реальному часі, виявлено обмеження традиційних методів та обґрунтовано доцільність застосування алгоритмів оптимізації з еволюційними характеристиками. Запропоновано метод автоматизованого налаштування параметрів задач, який базується на комбінації мутаційної зміни періодичності, тривалості та пріоритетів задач, з подальшим відбором кращих конфігурацій згідно з багатофакторною функцією придатності.

ABSTRACT

Master's thesis: 74 pages, 11 figures, 2 appendices, 8 sources.

EMBEDDED SYSTEMS, SYSTEM SOFTWARE, OPTIMIZATION, ENERGY CONSUMPTION, TASK SCHEDULING, GREEDY ALGORITHM, MACHINE LEARNING, GOOGLE COLAB, PRIORITIES, RESOURCE CONSTRAINTS.

The major goal of this thesis is to investigate existing methods and to implement a practical optimization approach for system software used in embedded systems, taking into account modern strategies for resource management, energy consumption reduction, and task scheduling. The implementation involves the development of a simulation model in the Google Colab environment to demonstrate and validate the effectiveness of the proposed method.

In order to addresses the problem of optimizing system software for embedded systems operating under limited computational resources and stringent energy efficiency requirements. A review of current real-time task scheduling approaches is conducted, identifying the limitations of traditional methods and substantiating the relevance of employing optimization algorithms with evolutionary characteristics. A method for automated task parameter tuning is proposed, which is based on a combination of mutational adjustments to task periodicity, execution duration, and priority, followed by the selection of optimal configurations according to a multi-factor fitness function.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 8
ВСТУП 9
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОБЛЕМИ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПЗ 12
1.1 Особливості вбудованих систем та їх програмного забезпечення 12
1.2 Вимоги до системного програмного забезпечення у вбудованих пристроях 14
1.3 Огляд типових архітектур вбудованих систем 16
1.4 Методи оптимізації програмного коду у вбудованих системах 18
1.5 Аналіз існуючих систем та засобів розробки системного ПЗ 20
2 ОСНОВИ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 26
2.1 Категоризація методів оптимізації: ручні, автоматизовані, машинного навчання 26
2.2 Оптимізація розподілу ресурсів процесора, пам’яті та енергії 27
2.3 Методи зменшення затримок та часу реакції 28
2.4 Використання профілювання як інструменту оптимізації 30
2.5 Комбінація статичного та динамічного аналізу коду 31
3 РОЗРОБКА МЕТОДУ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВБУДОВАНИХ СИСТЕМ 33
3.1 Постановка задачі оптимізації 33
3.2 Вибір платформи для реалізації 35
3.4 Розробка методу на основі оптимізації планування задач та енергоспоживання 40
3.5 Інтеграція алгоритмів оптимізації 45
ВИСНОВКИ 56
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 57

ДОДАТОК А Графічний матеріал кваліфікаційної роботи 59
ДОДАТОК Б Програмний код 68

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

API – інтерфейс прикладного програмування

Colab – Google Colaboratory

CPU – центральний процесор

ML – машинне навчання

RL – навчання з підкріпленням (Reinforcement Learning)

RAM – оперативна пам’ять (Random Access Memory)

SPS – системне програмне забезпечення

Task – об’єкт задачі в симуляційній моделі

GUI – графічний інтерфейс користувача (Graphical User Interface)

JSON – текстовий формат для обміну даними (JavaScript Object Notation)

IOPS – кількість операцій введення-виведення за секунду (Input/Output Operations Per Second)

FPS – частота кадрів в секунду (Frames Per Second)

NB – Jupyter Notebook

GA – генетичний алгоритм (Genetic Algorithm)

UML – уніфікована мова моделювання (Unified Modeling Language)

ВСТУП

У сучасному технологічному ландшафті вбудовані системи відіграють ключову роль у забезпеченні функціонування широкого спектра пристроїв – від побутової електроніки, медичного обладнання та автомобільної промисловості до систем автоматизації виробництва, «розумного» середовища (Smart Home/City) та Інтернету речей (IoT). Їхнє стрімке поширення обумовлене потребою у компактних, енергоефективних, недорогих та функціонально адаптованих обчислювальних платформах, що працюють у режимі реального часу. Проте головною особливістю вбудованих систем залишається обмеженість апаратних ресурсів – центрального процесора, пам’яті, енергоспоживання та швидкості доступу до даних.

Системне програмне забезпечення в таких системах, до якого належать операційні системи реального часу (RTOS), драйвери, ядра планування задач, засоби енергоменеджменту, прошивки та модулі апаратного абстрагування (HAL), відіграє критичну роль у забезпеченні стабільності, надійності та ефективності функціонування всієї апаратно-програмної платформи. Саме на цьому рівні приймаються ключові рішення щодо управління ресурсами, розподілу пріоритетів, енергозбереження, синхронізації процесів і взаємодії з периферією.

Незважаючи на значний прогрес у створенні ефективних інструментів розробки системного ПЗ, завдання його оптимізації залишається відкритим через специфічні вимоги до продуктивності та обмеження ресурсів. Особливу складність становить необхідність дотримання часових обмежень (реального часу), забезпечення мінімального споживання енергії в автономних системах, зменшення затримок при обробці критичних подій, а також оптимізація обсягу машинного коду для мікроконтролерів із обмеженою пам’яттю.

Сучасні методи оптимізації системного ПЗ базуються на різноманітних підходах – від традиційних компіляційних технік і ручного налаштування параметрів компілятора до застосування засобів автоматичного профілювання, енергетичного аналізу та навіть інтелектуальних алгоритмів адаптивного керування. Особливий інтерес викликає використання симуляційного моделювання й інструментів високорівневої аналітики (на кшталт Google Colab), що дозволяють апробувати нові методи оптимізації в безпечному і масштабованому середовищі до їхньої реалізації на фізичних пристроях.

Метою даної кваліфікаційної роботи є дослідження існуючих методів та практична реалізація методу оптимізації системного програмного забезпечення для вбудованих систем з урахуванням сучасних підходів до керування ресурсами, енергоспоживанням і плануванням задач. У процесі реалізації передбачається використання симуляційної моделі у середовищі Google Colab для демонстрації та верифікації ефективності запропонованого методу.

Об’єктом дослідження є системне програмне забезпечення вбудованих обчислювальних платформ.

Завдання:

- провести аналіз предметної області щодо специфіки розробки та функціонування системного програмного забезпечення в умовах обмежених ресурсів вбудованих систем;

- дослідити існуючі методи оптимізації системного рівня програмного забезпечення, зокрема підходи до зменшення енергоспоживання, часу виконання задач і обсягу пам’яті;

- сформулювати вимоги до ефективного методу оптимізації, що забезпечує збалансоване використання ресурсів процесора, пам’яті та енергії в типових архітектурах вбудованих систем;

- розробити власний метод оптимізації системного ПЗ, який враховує обмеження вбудованого середовища, зокрема потреби в реальному часі та

енергозбереженні;

- побудувати симуляційну модель вбудованої системи у середовищі Google Colab з параметрами обмежень (час виконання задач, доступна пам'ять, енергоспоживання тощо);
- реалізувати запропонований метод оптимізації у вигляді програмного модуля з можливістю тестування в імітаційному середовищі;
- провести експериментальне дослідження ефективності запропонованого підходу, порівнявши результати з базовими конфігураціями (без оптимізації).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОБЛЕМИ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПЗ

1.1 Особливості вбудованих систем та їх програмного забезпечення

Вбудовані системи становлять особливий клас обчислювальних пристроїв, що інтегруються безпосередньо у функціональне середовище конкретного прикладного об'єкта, забезпечуючи виконання строго визначених завдань, найчастіше у режимі реального часу. Їхня суттєва відмінність від універсальних обчислювальних систем полягає у жорсткій інтеграції апаратного і програмного забезпечення, що передбачає тісну взаємодію з фізичними процесами, обмеженість ресурсів і підвищені вимоги до надійності та предсказуваності поведінки. Основу таких систем зазвичай становлять мікроконтролери, цифрові сигнальні процесори або спеціалізовані інтегральні схеми з низьким рівнем споживання енергії, обмеженим обсягом пам’яті та обчислювальної потужності.

Програмне забезпечення, яке виконується на вбудованих платформах, відрізняється високим ступенем спеціалізації, оскільки воно розробляється під конкретне апаратне забезпечення та прикладну задачу. У цьому контексті системне програмне забезпечення виконує ключову роль, оскільки воно забезпечує базову функціональність пристрою, реалізуючи управління апаратними модулями, синхронізацію задач, диспетчеризацію, енергозбереження, обробку переривань і взаємодію з зовнішнім середовищем. Його розробка вимагає врахування широкого спектра обмежень, серед яких найважливішими є жорсткі часові обмеження, обмежена ємність пам’яті, висока вартість помилки в роботі пристрою та необхідність підтримки автономного функціонування в умовах обмеженого енергоживлення.

Іншою характерною особливістю вбудованих систем є їхня тісна залежність від конкретної апаратної платформи, що зумовлює використання низькорівневих засобів програмування, зокрема мов асемблера, C або C++, а також специфічних засобів налаштування компіляції, ініціалізації периферії та розподілу пам'яті. Багато сучасних вбудованих систем використовують операційні системи реального часу (RTOS), які забезпечують гарантоване управління виконанням задач відповідно до визначених пріоритетів і часових вимог. Це накладає додаткові вимоги до детермінованості виконання коду, ефективного використання стека та мінімізації латентності при обробці подій.

Не менш важливою є роль оптимізації системного програмного забезпечення у забезпеченні довговічної та енергоефективної роботи пристрою. З огляду на те, що багато вбудованих систем функціонують у важкодоступному середовищі або у критично важливих застосуваннях, необхідність зменшення навантаження на апаратні компоненти стає одним із визначальних чинників у проєктуванні їхнього програмного забезпечення. При цьому важливо забезпечити баланс між продуктивністю, енергоспоживанням та стабільністю, що вимагає використання спеціалізованих підходів до аналізу й оптимізації на рівні системного коду.

Таким чином, особливості вбудованих систем полягають у необхідності забезпечення ефективного функціонування програмного забезпечення в умовах жорстких апаратних обмежень, що потребує глибокого розуміння архітектури апаратного середовища, специфіки взаємодії компонентів і принципів реального часу. Це формує унікальні вимоги до методів проєктування, тестування та оптимізації системного програмного забезпечення, які й виступають предметом подальшого дослідження у межах цієї роботи.

1.2 Вимоги до системного програмного забезпечення у вбудованих пристроях

Системне програмне забезпечення у вбудованих пристроях виконує критично важливу функцію – воно забезпечує фундаментальні механізми взаємодії між апаратною платформою та прикладними модулями, а також відповідає за підтримку життєвого циклу системи в цілому. Зважаючи на специфіку вбудованого середовища, до системного рівня пред’являються суворі вимоги, які значно відрізняються від типових характеристик програмного забезпечення для загального призначення. Ці вимоги формуються у відповідь на жорсткі обмеження за ресурсами, критичність часових характеристик, потребу в автономності та стійкості до збоїв.

Рисунок 1.1 – Вимоги до СПЗ

Насамперед системне програмне забезпечення має забезпечувати гарантовану реакцію на зовнішні та внутрішні події у межах чітко визначених часових інтервалів. Це означає, що усі програмні компоненти,

включаючи диспетчер задач, обробники переривань і драйвери, повинні працювати в режимі реального часу, де навіть мікросекундні затримки можуть призвести до критичних наслідків, особливо у системах управління, зв'язку чи безпеки. Таким чином, пред’являються підвищені вимоги до детермінованості поведінки програмного коду та мінімізації латентності обробки подій.

Ще одним важливим аспектом є обмеженість доступних апаратних ресурсів. У більшості випадків обсяг оперативної пам’яті, обчислювальна потужність процесора та енергетичні можливості пристрою значно нижчі, ніж у звичайних комп’ютерах або серверних системах. У зв’язку з цим, програмне забезпечення повинно бути максимально компактним, ефективним і мати низький рівень накладних витрат. Це вимагає не лише використання низькорівневих мов програмування, а й ретельної оптимізації використання стеку, регістрів та апаратних таймерів.

Окремої уваги потребує питання енергозбереження. У багатьох застосуваннях вбудовані системи функціонують в автономному режимі від батареї або з обмеженим джерелом живлення. Системне програмне забезпечення в цьому випадку повинно забезпечувати динамічне управління живленням, контроль переходу до сплячих режимів, адаптацію частоти процесора та використання периферії залежно від контексту. Наявність механізмів енергетичного моніторингу та оптимального розподілу обчислень стає важливою умовою забезпечення тривалого ресурсу автономної роботи.

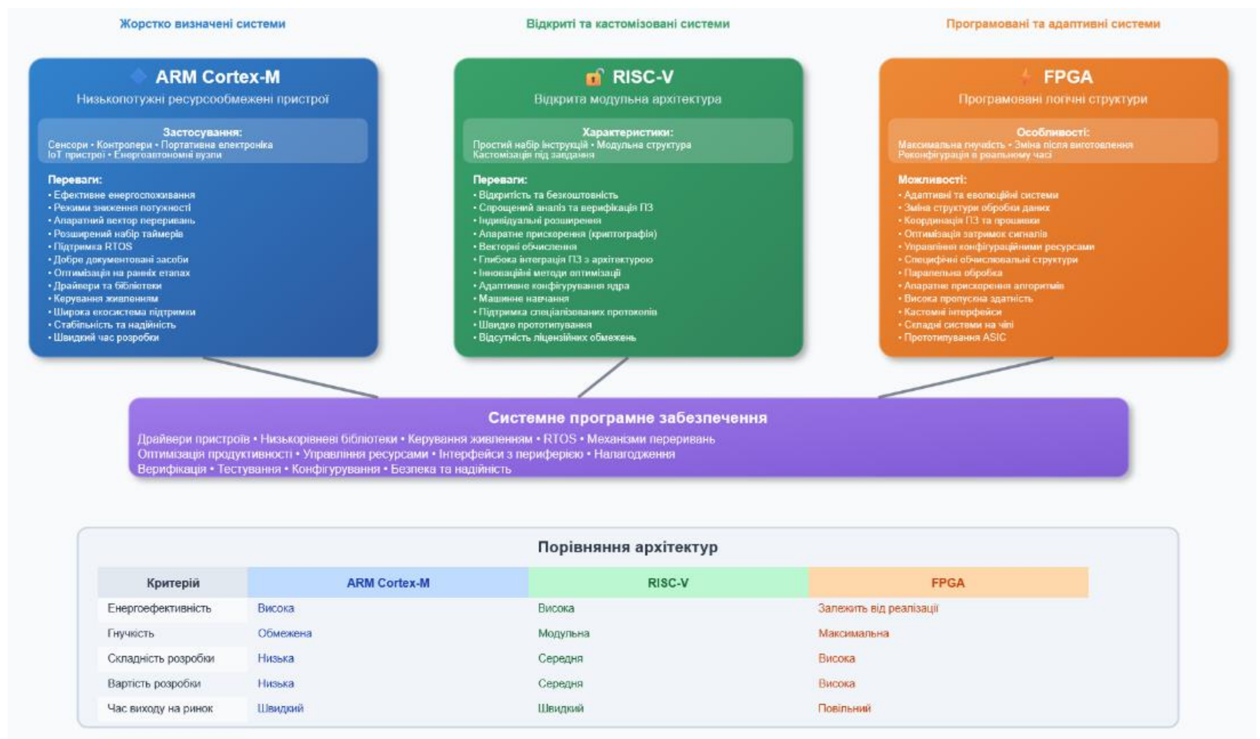
Надійність також є визначальним критерієм. Вбудовані системи часто використовуються у критично важливих галузях – від автомобільної електроніки до медичних пристроїв та промислових контролерів. Системне програмне забезпечення має бути здатне працювати без збоїв упродовж тривалого часу, без можливості оновлення або втручання з боку користувача. Для цього необхідно передбачати механізми виявлення й обробки помилок, відновлення після збоїв, а також архітектурні рішення, що забезпечують захищене середовище виконання задач.

Нарешті, важливою вимогою є масштабованість і портованість. Умови використання вбудованих систем надзвичайно різноманітні, тому системне ПЗ повинно легко адаптуватися до різних конфігурацій апаратного забезпечення, підтримувати змінні модулі, драйвери та реалізації інтерфейсів. Забезпечення такої гнучкості в умовах обмежених ресурсів потребує спеціального підходу до архітектури системного ПЗ, який дозволяє зберігати модульність без втрати ефективності.

1.3 Огляд типових архітектур вбудованих систем

Архітектура апаратної платформи є фундаментальною основою для розробки ефективного системного програмного забезпечення у вбудованих системах, оскільки саме вона визначає набір доступних інструкцій, модель пам’яті, характеристики взаємодії з периферійними пристроями та загальні обмеження щодо продуктивності й енергоспоживання. У сучасній практиці застосування вбудованих рішень найпоширенішими є архітектури типу ARM, RISC-V, а також програмовані логічні структури на базі FPGA. Кожна з цих архітектур має свої особливості, що суттєво впливають на принципи організації системного рівня програмного забезпечення.

Процесори з архітектурою ARM, зокрема серії Cortex-M, широко використовуються у низькопотужних і ресурсообмежених вбудованих пристроях, таких як сенсори, контролери, портативна електроніка, пристрої з інтерфейсами бездротового зв’язку та енергоавтономні вузли. Основними перевагами цієї архітектури є ефективне енергоспоживання, підтримка широкого спектра режимів зниження потужності, наявність апаратного вектора переривань, розширений набір таймерів та підтримка операційних систем реального часу. У контексті розробки системного ПЗ ARM-платформи надають добре документовані засоби для створення драйверів, низькорівневих бібліотек та механізмів керування живленням, що дозволяє забезпечити високий ступінь оптимізації на ранніх етапах проєктування.



Критерій	ARM Cortex-M	RISC-V	FPGA
Енергоефективність	Висока	Висока	Залежить від реалізації
Гнучкість	Обмежена	Модульна	Максимальна
Складність розробки	Низька	Середня	Висока
Вартість розробки	Низька	Середня	Висока
Час виходу на ринок	Швидкий	Швидкий	Повільний

Рисунок 1.2 – Типи архітектур вбудованих систем

Архітектура RISC-V, яка розвивається як відкрита альтернатива традиційним комерційним рішенням, набирає дедалі більшого поширення у вбудованих системах завдяки своїй модульній структурі та можливості кастомізації під конкретні завдання. Вона характеризується простим і елегантним набором інструкцій, що спрощує аналіз та верифікацію програмного забезпечення. Відкритість RISC-V дозволяє проєктувати індивідуальні розширення для апаратної підтримки специфічних функцій, таких як криптографія, векторні обчислення або апаратне прискорення певних протоколів. Це створює умови для глибокої інтеграції системного ПЗ із функціональною архітектурою пристрою та відкриває перспективи застосування інноваційних методів оптимізації, включаючи ті, що базуються на машинному навчанні або адаптивному конфігуруванні ядра.

У той час як ARM і RISC-V орієнтовані на побудову жорстко визначених апаратних систем, архітектури на базі FPGA забезпечують гнучкість, що недосяжна в інших підходах. Вони дозволяють створювати вбудовані обчислювальні структури із специфічними характеристиками, що

можуть бути змінені навіть після виготовлення пристрою. Це відкриває нові можливості для реалізації адаптивних та еволюційних систем, у яких системне програмне забезпечення не лише керує апаратурою, а й має змогу змінювати саму структуру обробки даних у реальному часі. Водночас, використання FPGA потребує особливого підходу до проєктування системного рівня, включаючи координацію між програмною логікою і прошивкою в апаратурі, оптимізацію затримок у маршрутах сигналів та ретельне управління конфігураційними ресурсами.

Таким чином, вибір архітектури має принципове значення для визначення стратегій оптимізації системного програмного забезпечення. ARM-платформи забезпечують зрілість інструментів та багатий набір засобів енергозбереження, RISC-V – відкритість і масштабованість, а FPGA – гнучкість та високу продуктивність для спеціалізованих задач. Розуміння архітектурних особливостей кожної з платформ є передумовою для ефективної розробки та впровадження оптимізованого системного коду у вбудованих пристроях.

1.4 Методи оптимізації програмного коду у вбудованих системах

У контексті вбудованих систем оптимізація програмного коду є однією з ключових умов забезпечення ефективного функціонування програмного забезпечення в умовах суворих ресурсних обмежень. Цей процес охоплює широке коло підходів, що дозволяють зменшити розміри машинного коду, прискорити виконання інструкцій, знизити затримки при доступі до пам’яті, мінімізувати кількість звернень до енергоємних апаратних компонентів і підвищити загальну стабільність роботи системи.

Оптимізація на етапі компіляції передбачає використання спеціалізованих налаштувань компілятора, які дозволяють адаптувати скомпільований код до характеристик конкретної платформи. Наприклад, компілятори GCC або LLVM для ARM-архітектури надають численні

параметри, що дозволяють контролювати інлайнинг функцій, розгортання циклів, злиття умовних гілок, а також оптимізоване використання регістрів. Ці засоби дозволяють зменшити розмір виконуваного файлу або, навпаки, покращити продуктивність за рахунок дублювання коду в критичних ділянках. У деяких випадках доцільним є поєднання автоматичних та ручних методів оптимізації, коли розробник самостійно керує розміщенням даних у пам’яті, визначає атрибути функцій або оптимізує обробку переривань.

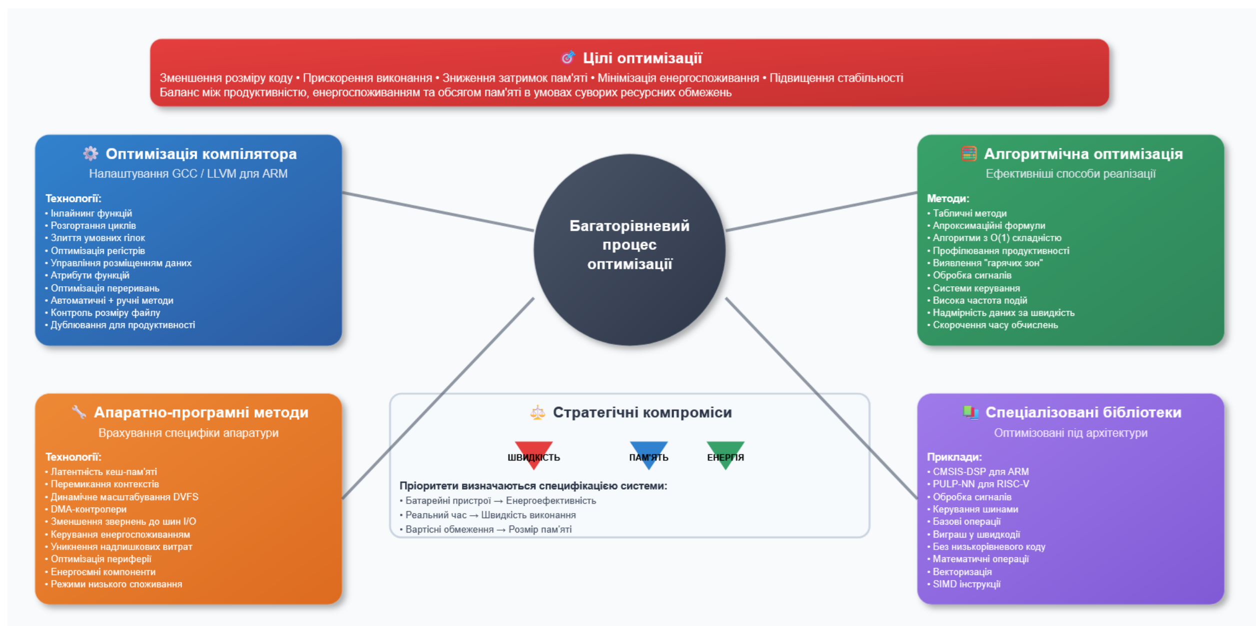


Рисунок 1.3 – Методи оптимізації програмного коду

Алгоритмічна оптимізація зосереджується на виборі ефективніших способів реалізації тієї чи іншої функціональності. Наприклад, у задачах керування або обробки сигналів використання табличних методів або апроксимаційних формул дозволяє суттєво скоротити час обчислень. У системах з високою частотою обробки подій доцільно реалізовувати алгоритми з постійною часовою складністю, навіть ціною деякої надмірності в обсязі даних, що зберігаються. Сучасні вбудовані компілятори підтримують профілювання продуктивності, що дозволяє визначити так звані "гарячі зони" коду, які доцільно оптимізувати насамперед.

Апаратно-програмні методи оптимізації передбачають врахування

специфіки апаратної реалізації при написанні коду. Зокрема, знання про латентність доступу до різних рівнів кеш-пам'яті або про швидкість перемикання між контекстами задач дозволяє уникати ситуацій, що призводять до надлишкових витрат обчислювальних ресурсів. У платформах, де підтримується розширене керування енергоспоживанням, застосовуються технології динамічного масштабування частоти та напруги, керування периферією через DMA-контролери, а також механізми зменшення частоти звернень до енергоємних шин введення-виведення.

Важливим напрямом також є використання спеціалізованих бібліотек, розроблених під конкретні архітектури, які забезпечують ефективну реалізацію базових операцій, наприклад, обробки сигналів або керування шинами. Такі бібліотеки, як CMSIS-DSP для ARM або PULP-NN для RISC-V, дозволяють досягти суттєвого виграшу у швидкодії без необхідності в розробці низькорівневого коду вручну.

Таким чином, оптимізація програмного коду вбудованих систем є багаторівневим процесом, що вимагає як глибокого розуміння архітектурних обмежень, так і здатності приймати стратегічні рішення щодо пріоритетів між продуктивністю, енергоспоживанням і обсягом пам'яті. Її реалізація повинна відбуватись у тісній взаємодії між розробниками системного програмного забезпечення, апаратними інженерами та дизайнерами алгоритмів.

1.5 Аналіз існуючих систем та засобів розробки системного ПЗ

У практиці розробки вбудованих рішень системне програмне забезпечення часто базується на використанні попередньо сформованих інструментів та архітектур, що полегшують інтеграцію функціональності, забезпечують узгодженість між апаратними та програмними компонентами і скорочують час розробки. Серед таких систем слід виокремити середовища на основі вільного або відкритого коду, які забезпечують не лише базову

функціональність, а й широкі можливості для адаптації, профілювання та тестування.

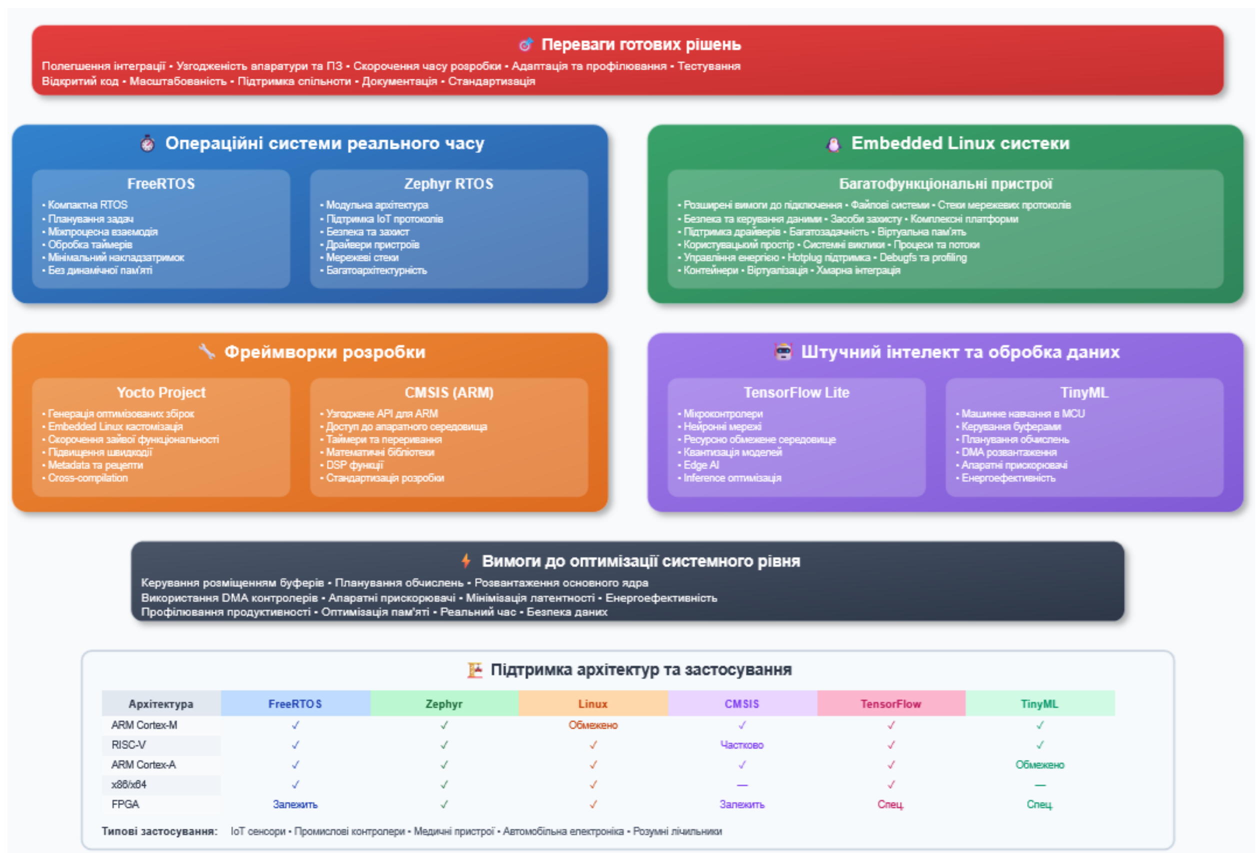


Рисунок 1.4 – Аналіз існуючих систем та засобів розробки системного ПЗ

Одним із найбільш поширених рішень є FreeRTOS – компактна операційна система реального часу, яка надає функції планування задач, міжпроцесної взаємодії, обробки таймерів і переривань з мінімальним накладом ресурсів. Її перевагою є масштабованість, можливість роботи без динамічного розподілу пам'яті, а також підтримка великої кількості мікроконтролерних архітектур. У більш складних системах використовуються рішення на кшталт Zephyr або Embedded Linux, які орієнтовані на багатофункціональні пристрої з розширеними вимогами до підключення, безпеки та керування даними. Ці системи забезпечують підтримку драйверів, файлових систем, стеків мережевих протоколів та засобів захисту, що дозволяє розробникам створювати комплексні апаратно-

програмні платформи.

Суттєву роль відіграють також фреймворки для налаштування середовища виконання. Зокрема, Yocto Project надає засоби для генерації оптимізованих збірок Embedded Linux з урахуванням специфіки платформи, що дозволяє максимально скоротити обсяг зайвої функціональності та підвищити швидкодію системи. Іншим прикладом є CMSIS – стандарт ARM, який надає узгоджене API для доступу до апаратного середовища, таймерів, переривань і бібліотек математичних функцій.

З огляду на актуальність задач штучного інтелекту та обробки даних у периферійних пристроях, зростає значення таких систем, як TensorFlow Lite for Microcontrollers або TinyML, які орієнтовані на виконання нейронних мереж у ресурсно обмеженому середовищі. Ці інструменти також потребують ретельної оптимізації системного рівня – зокрема, керування розміщенням буферів, планування обчислень, розвантаження основного ядра шляхом використання DMA або апаратних прискорювачів.

Таким чином, наявні системи розробки системного ПЗ для вбудованих систем демонструють широкий спектр можливостей, однак ефективність їх використання значною мірою залежить від здатності розробника адаптувати їх під специфічні умови проєкту. Це підтверджує необхідність глибокого аналізу та подальшого вдосконалення методів оптимізації, які забезпечують кращу інтеграцію між програмним забезпеченням і апаратними ресурсами.

1.6 Аналіз останніх досліджень і публікацій

Аналіз вихідного коду програмного забезпечення посідає ключове місце в дослідженнях, пов’язаних із забезпеченням якості програм, процесами тестування та інформаційною безпекою. Сучасна наукова література, як правило, класифікує підходи до аналізу на статичні, динамічні та гібридні, акцентуючи увагу на зростаючому значенні автоматизованих інструментів і алгоритмів машинного навчання.

У роботі [1] розглядається досвід створення внутрішнього інструменту Google під назвою Tricorder – масштабованої платформи для статичного аналізу, яка дозволяє виявляти помилки на етапі написання коду, інтегруючись у повсякденну практику розробників. Інструмент підтримує багато мов програмування та базується на модульній архітектурі, що забезпечує гнучке підключення різноманітних аналізаторів. Важливими перевагами цієї системи є її масштабованість, здатність до динамічного оновлення правил перевірки, а також миттєва доставка зворотного зв'язку. Публікація демонструє приклад ефективного впровадження статичного аналізу у масштабну інженерну інфраструктуру.

У [2] представлено приклад практичного використання інструменту FindBugs у проєктах на мові Java. Описано ініціативу Fixit, в межах якої розробники зосереджувались на виправленні дефектів, виявлених FindBugs. Дослідження показало, що навіть у добре протестованому коді можуть залишатися критичні помилки – насамперед, пов'язані з null-посиланнями, неповною ініціалізацією та небезпечними шаблонами. Це підтверджує доцільність регулярного застосування статичного аналізу як способу підвищення якості програмного коду без суттєвого збільшення витрат на тестування.

У роботі [3] розглядається застосування статичного аналізу як інструменту підвищення безпеки програмного забезпечення. На прикладах Fortify та Coverity продемонстровано можливості виявлення широкого спектру поширених вразливостей – зокрема буферних переповнень, SQL-ін'єкцій і XSS – без потреби запуску програм. Автори підкреслюють необхідність інтеграції статичного аналізу у процес безпечної розробки як постійного компонента, а не епізодичного тестування, а також наголошують на важливості належного навчання розробників і коректного налаштування інструментів.

У статті [4] подано емпіричний аналіз типових помилок у відомих open-source проєктах, таких як Mozilla, Apache та Eclipse. Виявлено, що

більшість дефектів мають соціотехнічну природу: неправильні припущення, помилки взаємодії компонентів, зміни в API тощо. Особливо складними для виявлення виявилися логічні помилки, що часто лишаються поза межами виявлення за допомогою тестів чи компіляторів. Автори роблять висновок, що забезпечення високої якості коду можливе лише за умов комплексного підходу, який поєднує як статичний, так і динамічний аналіз, а також активну участь користувачів у процесі зворотного зв’язку.

У публікації [5] представлено фреймворк VulLibMiner, який дозволяє виявляти вразливі сторонні бібліотеки на основі текстових описів вразливостей. Такий підхід особливо актуальний у контексті широкого використання стороннього коду без глибокого аудиту. Авторами також окреслено перспективи розширення системи: масштабування на інші мови програмування, інтеграцію в середовища розробки та автоматичне формування рекомендацій щодо оновлення.

Стаття [6] присвячена інноваційній концепції аналізу програм за допомогою глибокого навчання. Представлено модель code2vec, яка перетворює код у векторне представлення шляхом аналізу дерев синтаксичного розбору. Такий підхід дозволяє виявляти структурні закономірності, притаманні певним типам помилок або стилістичних порушень, та відкриває нові перспективи для побудови інтелектуальних систем аналізу коду.

Усі розглянуті дослідження підтверджують вагомість аналізу вихідного коду як ефективного інструменту забезпечення надійності, якості та безпеки програмного забезпечення. Статичний аналіз демонструє ефективність у виявленні як синтаксичних і семантичних помилок, так і потенційних вразливостей ще до запуску програм, зокрема в умовах масштабної корпоративної розробки [1, 2, 3]. Динамічний аналіз, у свою чергу, є важливим доповненням, що дозволяє виявити помилки продуктивності, проблеми синхронізації, витоки пам’яті та інші дефекти, пов’язані з поведінкою ПЗ у середовищі виконання [4]. Нарешті, сучасні підходи, що

поєднують машинне навчання з аналізом структури коду, демонструють високу перспективність як у сфері безпеки, так і в автоматизації інженерних процесів [5, 6].

2 ОСНОВИ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Категоризація методів оптимізації: ручні, автоматизовані, машинного навчання

У процесі оптимізації системного програмного забезпечення для вбудованих систем застосовуються різні категорії методів, які відрізняються як за рівнем абстракції, так і за ступенем автоматизації. Найбільш традиційним є ручний підхід, який ґрунтується на досвіді розробника, його знанні архітектури цільової платформи та здатності виявляти вузькі місця в коді. Цей підхід передбачає пряме втручання в алгоритмічну структуру програмного забезпечення, ручну оптимізацію циклів, регістрів, стекових операцій та використання специфічних інструкцій. Хоча він дозволяє досягати високої ефективності, особливо в критичних ділянках коду, його застосування є трудомістким, складним у супроводі та важко масштабованим на великі проєкти.

На противагу цьому автоматизовані методи спираються на інструменти компіляторної оптимізації, статичного аналізу коду, профілювання та трансформації виконуваних програм. Сучасні компілятори підтримують численні режими оптимізації, що дозволяють автоматично розгортати цикли, зменшувати кількість умовних переходів, усувати мертві гілки та повторно використовувати ресурси. Інструменти статичного аналізу, такі як clang-analyzer, Coverity або PC-lint, забезпечують виявлення потенційних помилок, неефективних шаблонів програмування та надлишкових звернень до пам’яті ще до етапу компіляції. Водночас профілювання виконання дозволяє оцінити часові характеристики задач, ідентифікувати «гарячі» зони та застосувати до них специфічні стратегії оптимізації.

Останнім часом зростає інтерес до застосування методів машинного навчання у процесі оптимізації системного ПЗ. Ці підходи дозволяють адаптувати конфігурацію середовища виконання, прогнозувати навантаження, моделювати поведінку задач та приймати рішення щодо розподілу ресурсів у динамічному режимі. Наприклад, системи, що використовують підкріплювальне навчання, можуть навчатися ефективним шаблонам планування задач або перемикання енергетичних режимів. Впровадження таких рішень у вбудовані системи передбачає спрощення моделей і забезпечення їхньої детермінованості, однак навіть обмежені реалізації дозволяють суттєво підвищити адаптивність системного рівня до змінних умов середовища.

Таким чином, вибір категорії методів оптимізації залежить від контексту застосування, рівня критичності задач, доступних ресурсів та потреб у масштабованості рішення. Комплексне використання зазначених підходів дозволяє досягти збалансованих результатів із точки зору продуктивності, стабільності та ефективності.

2.2 Оптимізація розподілу ресурсів процесора, пам’яті та енергії

Ефективне управління обчислювальними ресурсами вбудованих систем є одним з основних завдань системного рівня, оскільки неправильний розподіл процесорного часу, перевитрата оперативної пам’яті або нераціональне використання енергетичних джерел можуть призвести до деградації продуктивності або повної зупинки пристрою. Розподіл ресурсів повинен здійснюватися з урахуванням пріоритетів задач, реального часу їх виконання, доступних апаратних модулів та поточного енергетичного стану системи.

Управління процесором передбачає ефективну організацію планування задач, де враховуються їхні пріоритети, тривалість виконання та взаємозалежності. Використання пріоритетного планувальника з

витісненням, розділення задач на критичні й фонові, а також можливість динамічного коригування черговості виконання дозволяють уникнути надмірних затримок. Важливою є підтримка режимів очікування та переходів у сплячі стани, які повинні контролюватися на рівні системного коду без шкоди для відповідності часовим обмеженням.

У контексті оперативної пам’яті системне ПЗ має забезпечувати статичний розподіл об’єктів, мінімізацію використання динамічної пам’яті, а також контроль за переповненням стеку та буферів. Особливо важливим є використання спеціалізованих схем розміщення даних у пам’яті, які враховують швидкодію доступу та енергоспоживання, наприклад, винесення часто використовуваних змінних у регістрові банки або кеші.

Управління енергією повинно реалізовуватися як на рівні програмного керування живленням, так і через динамічне масштабування частоти, контролювання периферії, оптимізацію режимів простою та глибокого сну. Системне ПЗ повинно взаємодіяти з контролерами живлення, використовуючи відповідні API, таймери пробудження та таблиці енергетичних режимів. Залежно від поточної активності пристрою приймаються рішення про деактивацію непотрібних модулів або перемикання у енергоефективні стани.

Загалом оптимізація розподілу ресурсів потребує системного підходу, в якому береться до уваги не лише миттєвий стан ресурсів, а й прогнозування їх змін у часі, очікувані навантаження та критичність окремих задач. Це вимагає не тільки ретельного проєктування, але й можливостей динамічної адаптації під час виконання.

2.3 Методи зменшення затримок та часу реакції

У вбудованих системах, що функціонують у режимі реального часу, критичним параметром є здатність системного програмного забезпечення забезпечити передбачувану та своєчасну реакцію на зовнішні або внутрішні

події. У таких системах порушення заданих часових обмежень розглядається не лише як погіршення продуктивності, а як функціональна помилка, що може спричинити аварійні стани або втрату керованості. Оптимізація часу реакції у цьому контексті охоплює широкий спектр стратегій, що стосуються архітектури планувальника, обробки переривань, синхронізації задач та мінімізації накладних витрат.

Системне ПЗ повинно підтримувати пріоритетне, витісняюче планування, в якому задачі з високою критичністю можуть бути негайно активовані після надходження відповідного сигналу або події. Це вимагає ефективної реалізації контекстного перемикання, зменшення часу обробки переривань та оптимізації черги задач. Уникнення блокуючих викликів, обмеження використання глобальних ресурсів та реалізація алгоритмів взаємного виключення з коротким часом утримання також є важливими передумовами для досягнення детермінованої поведінки.

Особливу роль відіграє апаратна підтримка – наприклад, наявність багаторівневої ієрархії пріоритетів для переривань, апаратного таймера реального часу, окремих векторів обробки подій. Системне програмне забезпечення повинно бути здатне взаємодіяти з цими механізмами на найнижчому рівні, забезпечуючи мінімальний час реакції без додаткових накладних викликів. Для цього необхідно мати розроблений механізм аналізу затримок, що включає вимірювання часу виконання критичних секцій та моделювання черг обробки подій.

Таким чином, оптимізація часу реакції в системах реального часу передбачає як архітектурні, так і алгоритмічні рішення, що забезпечують передбачуваність, швидкодію та надійність. Її досягнення є основною умовою функціональної коректності вбудованих систем критичного призначення.

2.4 Використання профілювання як інструменту оптимізації

Профілювання системного програмного забезпечення є однією з найефективніших стратегій аналізу його продуктивності та виявлення вузьких місць у роботі. Завдяки цьому підходу розробник отримує кількісну інформацію про час виконання окремих функцій, частоту викликів, обсяг використаної пам’яті, кількість звернень до периферійних пристроїв, а також статистику щодо енергоспоживання. Ця інформація дозволяє приймати обґрунтовані рішення щодо доцільності оптимізації певних ділянок коду, виявляти надлишкову активність або непродуктивне використання ресурсів.

У вбудованому середовищі профілювання ускладнене через обмежену доступність внутрішніх інтерфейсів, однак сучасні мікроконтролери та процесори оснащуються базовими засобами трасування, лічильниками продуктивності, а також спеціалізованими портами для підключення відлагоджувальних інтерфейсів. Підтримка таких засобів як ITM (Instrumentation Trace Macrocell) у ARM Cortex, або аналоги в архітектурі RISC-V, дозволяє збирати точкові або безперервні дані під час виконання програми без помітного впливу на її поведінку.

Важливим є також використання енергетичного профілювання, яке дозволяє виявити ділянки коду, що спричиняють пікове споживання енергії або надмірне використання периферії. У поєднанні з часовим аналізом та статистикою кеш-хітів, такі дані відкривають можливість для комплексної оптимізації, що одночасно враховує продуктивність і енергозбереження. Інструменти на кшталт ARM EnergyTrace, Intel VTune, або вбудовані рішення ST-Link Utility є ефективними засобами збору подібної інформації.

Таким чином, профілювання стає обов’язковим етапом процесу оптимізації, який забезпечує емпіричне підґрунтя для об’єктивного прийняття рішень. Воно сприяє підвищенню ефективності системного програмного забезпечення шляхом точкового втручання у ті компоненти, які дійсно потребують удосконалення.

2.5 Комбінація статичного та динамічного аналізу коду

Оптимізація системного програмного забезпечення вимагає комплексного підходу до вивчення його структури, логіки виконання та характеристик взаємодії з апаратними компонентами. У цьому контексті поєднання статичного і динамічного аналізу дозволяє отримати повну картину поведінки системи та виявити такі недоліки, які не можуть бути зафіксовані жодним із цих методів окремо.

Статичний аналіз забезпечує дослідження коду без його виконання, дозволяючи виявити потенційні помилки, невикористані змінні, порушення стилістики або непередбачувану поведінку у рідкісних гілках виконання. Він дозволяє ідентифікувати конфлікти у доступі до спільних ресурсів, логічні помилки у механізмах синхронізації, переповнення буферів та інші типові вразливості. У системному ПЗ це має особливу вагу, оскільки помилки на цьому рівні часто є причиною критичних збоїв.

Динамічний аналіз, у свою чергу, дозволяє оцінити поведінку програми в умовах реального навантаження. Він забезпечує виявлення помилок часу виконання, витоків пам’яті, неефективних шаблонів взаємодії з периферією, а також дозволяє досліджувати продуктивність задач і затримки між подіями. Особливо ефективним є його поєднання з трасуванням станів операційної системи, що дозволяє реконструювати повну хронологію переходів між задачами, активації таймерів та обробки переривань.

2.6 Формалізація критеріїв ефективності оптимізації

Процес оптимізації не може бути успішним без чіткої системи критеріїв, яка дозволяє кількісно оцінити якість розроблених рішень і порівнювати альтернативні підходи. У системному програмному забезпеченні для вбудованих систем ці критерії повинні враховувати не лише традиційні метрики продуктивності, а й такі специфічні параметри, як

енергоспоживання, обсяг використаної пам’яті, стабільність у довготривалій експлуатації та відповідність часовим обмеженням.

Ефективність оптимізації може визначатися через скорочення середнього та пікового часу реакції, зменшення розміру виконуваного коду, зниження навантаження на центральне ядро, або збільшення тривалості автономної роботи пристрою. При цьому слід враховувати як абсолютні значення метрик, так і відносне покращення порівняно з базовою реалізацією. У випадку багатофакторної оптимізації доцільно використовувати агреговані індекси, які поєднують декілька метрик у зважену функцію, що дозволяє враховувати пріоритетність різних аспектів ефективності.

Особливе значення має стабільність результатів оптимізації, тобто здатність системного програмного забезпечення зберігати визначені характеристики у широкому діапазоні умов, включаючи зміну температури, напруги живлення, навантаження або конфігурації апаратних ресурсів. Також важливо оцінювати вплив оптимізації на супровідність та тестованість коду, оскільки надмірне ускладнення внутрішньої логіки може призвести до деградації загальної якості системи.

Формалізація критеріїв ефективності дає змогу структурувати процес оптимізації як наукове дослідження, що базується на об’єктивних даних та вимірюваних показниках. Вона є основою для порівняльного аналізу різних методів, верифікації результатів експериментів і обґрунтованого вибору стратегії оптимізації у конкретному проєкті.

3 РОЗРОБКА МЕТОДУ ОПТИМІЗАЦІЇ СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВБУДОВАНИХ СИСТЕМ

3.1 Постановка задачі оптимізації

На основі аналізу предметної області, особливостей вбудованих систем та методичних принципів оптимізації системного програмного забезпечення сформовано задачу, що полягає в розробці ефективного методу зменшення ресурсоспоживання, покращення часу реакції та забезпечення детермінованості виконання коду на системному рівні. Ця задача є багатокритеріальною і має бути реалізована в умовах реального обмеження ресурсів – обчислювальних, енергетичних, часових і пам’яттєвих – які притаманні мікроконтролерам і системам-на-чипі, що використовуються у сучасному вбудованому середовищі.

Під оптимізацією в даному контексті розуміється не лише локальне поліпшення окремих ділянок коду чи зменшення часу виконання конкретної функції, а комплексна перебудова стратегії розподілу ресурсів у масштабах усього середовища виконання системного програмного забезпечення. Основною передумовою ефективної оптимізації є чітке математичне або символьне формулювання мети, обмежень та змінних, що визначають внутрішній стан і зовнішню поведінку системи.

Вбудовані системи за своєю природою функціонують як відкриті динамічні середовища, в яких значення параметрів навантаження, активності задач, глибини черг або конфігурацій режимів живлення змінюються в часі і часто не є повністю передбачуваними. Це вимагає формулювання задачі оптимізації не в статичному, а в адаптивному або псевдодинамічному вигляді, з урахуванням того, що рішення повинні залишатися актуальними навіть у разі зміни вхідних умов – наприклад, появи нової задачі, зниження напруги живлення, підвищення температури або деградації акумулятора.

Формально задача оптимізації може бути подана як мінімізація узагальненої функції втрат, яка поєднує у собі кілька критеріїв ефективності: час реакції на подію, середнє енергоспоживання, рівень використання пам'яті, частоту перемикань задач та інші метричні показники. Математично ця функція має вигляд згортки (агрегованої функції вигоди), в якій кожна метрика нормалізується і зважується за пріоритетом, що залежить від контексту використання системи. Наприклад, для автономного сенсорного вузла найбільшого значення набуває енергоефективність, тоді як для системи керування двигуном – затримка обробки сигналу.

Позначимо загальну цільову функцію через F(x), де x – це вектор параметрів системи, що можуть змінюватись: частота роботи процесора, пріоритети задач, розміри стеків, методи обробки подій, використання DMA тощо. Функція F(x) залежить також від зовнішнього середовища E(t), яке описує змінні вхідні умови (частота подій, навантаження, рівень живлення тощо). Завданням є знайти такий набір керуючих параметрів x*, який мінімізує F(x, E(t)) за всіх допустимих значень t.

Важливо відзначити, що задача має бути розв'язуваною у межах програмного середовища Google Colab, що накладає певні технічні та структурні обмеження. Оскільки мова йде про моделювання вбудованої системи у високорівневому середовищі, усі вхідні параметри, змінні та критерії повинні бути реалізовані у вигляді математичних моделей, імітаційних структур або наборів симульованих даних. З цією метою розробляється модель, яка описує взаємозв'язок між параметрами задач і ресурсами платформи, дозволяючи варіювати їх значення та оцінювати результати на базі об'єктивно вимірюваних характеристик – таких як середній час реакції, розподіл завантаження, кількість активацій задач, середнє енергоспоживання на цикл тощо.

Підхід до розв'язання задачі передбачає формування методу, який використовує оптимізаційні алгоритми – жадібні стратегії, пошук у просторі рішень, можливо, елементи машинного навчання – що аналізують

конфігурації системи та вибирають оптимальні варіанти на основі цільової функції. Крім того, метод має бути адаптивним, тобто враховувати можливість перебудови параметрів під час роботи симульованої системи, у відповідь на зміни вхідних даних.

Таким чином, задача, що формулюється у межах цієї кваліфікаційної роботи, передбачає створення методу, який забезпечує мінімізацію ресурсоємності та часової латентності системного ПЗ у вбудованому середовищі, з урахуванням множини змінних параметрів і зовнішніх обмежень. З огляду на реалії апаратного середовища та програмних засобів, особливий акцент робиться на моделюванні обчислювальної архітектури, описі механізмів виконання задач і побудові експериментальної моделі, яка дозволяє порівнювати ефективність оптимізації у різних конфігураціях, включаючи базову (неоптимізовану) та модифіковану за допомогою розробленого методу.

3.2 Вибір платформи для реалізації

У процесі проєктування методу оптимізації системного програмного забезпечення для вбудованих систем надзвичайно важливим є правильний вибір інструментального середовища, яке б одночасно відповідало вимогам до моделювання, підтримувало достатню гнучкість і масштабованість, а також забезпечувало зручну інтеграцію з аналітичними й візуалізаційними засобами. З огляду на це, у рамках цієї кваліфікаційної роботи було прийнято рішення реалізувати експериментальну частину дослідження в середовищі Google Colab з використанням мови програмування Python та відповідних бібліотек для чисельного моделювання, симуляції та візуалізації.

Google Colab є хмарним інструментом, що поєднує зручність інтерактивного середовища розробки з можливістю виконання високорівневих обчислень на виділених серверах. Його використання дозволяє уникнути обмежень локального обладнання, забезпечити

доступність результатів дослідження з будь-якої точки та створити репродуктивне середовище, у якому інші дослідники або розробники можуть повторити або перевірити отримані результати. Більш того, інтеграція Colab із сервісами Google Drive, GitHub та TensorBoard забезпечує просте ведення документації, зберігання артефактів та візуальне представлення процесу оптимізації.

Python обрано як основну мову реалізації завдяки її високому рівню абстракції, багатій екосистемі бібліотек для математичного аналізу (NumPy, SciPy), симуляційних обчислень (SimPy), оптимізації (Scikit-Optimize, PuLP), машинного навчання (TensorFlow, PyTorch, Scikit-Learn), профілювання (line_profiler, memory_profiler), а також побудови інтерактивної візуалізації (Matplotlib, Plotly, Seaborn). Усі ці інструменти доступні безпосередньо у Google Colab, що дозволяє будувати повноцінні експериментальні сценарії без додаткових витрат на конфігурацію середовища.

Однією з ключових особливостей реалізації є побудова моделі вбудованої системи, яка у симульованому вигляді відтворює основні аспекти її поведінки: розподіл процесорного часу між задачами, використання оперативної пам’яті, перемикання між режимами активності й сну, взаємодію з периферійними пристроями, обробку подій і виклики переривань. У рамках цієї моделі кожна задача описується множиною параметрів: час виконання, періодичність активації, критичність, пріоритет, обсяг стеку, споживання енергії, кількість викликів. Система у цілому моделюється як динамічна черга задач, що виконується за правилами планувальника із заданим алгоритмом.

Для обліку енергетичних характеристик у моделі вводяться симульовані режими живлення, кожен з яких має свою базову потужність, час перемикання і коефіцієнт продуктивності. Перехід між режимами визначається активністю задач та конфігурацією планувальника. Аналогічно моделюється пам’ять – симулюється обсяг доступної RAM, розміщення об’єктів, розмір стеку задач, імітація переповнення або фрагментації. Усі ці

параметри заносяться у внутрішню метамодель, яка дозволяє в реальному часі оцінювати завантаження системи та визначати конфліктні зони.

Особливою увагою в моделі користується механізм вимірювання метрик ефективності, зокрема: середній час реакції на події, дисперсія латентності, сумарне споживання енергії за ітерацію, кількість перемикань задач, середнє завантаження CPU, використана пам'ять. Кожен запуск моделі дозволяє побудувати вектор результатів, який потім порівнюється з базовою конфігурацією (неоптимізованою) для оцінки приросту ефективності. Це відкриває можливість автоматизованого пошуку конфігурацій, які забезпечують оптимальний баланс між вимогами до продуктивності й ресурсоспоживанням.

Крім того, важливою складовою реалізації є підтримка адаптивного налаштування моделі, що передбачає зміну параметрів у процесі моделювання. Зокрема, симулюється поява нових задач, зміна енергетичних обмежень, динамічне оновлення частоти процесора або модифікація пріоритетів. Це дозволяє моделювати поведінку системи не лише в стабільному, а й у трансформованому режимі, що максимально наближає модель до реального середовища вбудованих пристроїв.

Таким чином, вибір Google Colab як платформи для реалізації моделі та Python як мови реалізації є технічно виправданим з погляду відкритості, доступності, гнучкості та можливостей масштабування. Модель, побудована у цьому середовищі, дозволяє здійснювати повноцінне моделювання поведінки вбудованого середовища на рівні системного ПЗ, відтворювати його часові, пам'яттєві й енергетичні параметри та експериментально підтверджувати ефективність розроблених підходів до оптимізації.

3.3 Створення імітаційної моделі вбудованої системи з обмеженнями

У контексті даної роботи імітаційна модель виконує роль аналітичного інструменту, який дозволяє відтворити поведінку вбудованої обчислювальної

системи під різними конфігураціями ресурсів та параметрів виконання системного програмного забезпечення. Її призначення полягає у формалізації та обчисленні ключових характеристик функціонування системи, включаючи навантаження на процесор, обсяг використаної пам'яті, рівень енергоспоживання, часову латентність задач і ефективність планування. Завдяки цьому модель виконує функцію експериментального полігону, на якому перевіряються гіпотези оптимізації без втручання у фізичне обладнання, що значно спрощує перевірку результатів.

У створенні імітаційної моделі вихідним положенням стало уявлення про вбудовану систему як про дискретно-часову реактивну систему, у якій у кожному часовому кванті виконуються певні обчислювальні задачі, спрацьовують переривання, змінюються режими енергоспоживання та відбувається обмін даними з периферією. Основу моделі становить подання системи у вигляді набору задач, які виконуються планувальником згідно з заданим сценарієм. Кожна задача описується структурованим вектором характеристик, що включає ідентифікатор, періодичність, тривалість виконання, пріоритет, критичність, обсяг пам'яті, що споживається, а також інтенсивність енергоспоживання під час активного циклу.

Імітаційне середовище реалізовано у вигляді об'єктно-орієнтованої структури в мові Python. Клас Task репрезентує одиницю виконання – тобто одну задачу системи. Для кожного об'єкта цього класу моделюється його поведінка в часовому проміжку: активація, виконання, завершення або блокування. Використовуючи генератори подій (наприклад, через бібліотеку SimPy), модель імітує відкладене пробудження задач, планування черги виконання, споживання ресурсів у динаміці, а також потенційні конфлікти за доступ до спільної пам'яті або периферійних модулів.

Модель процесора представлена класом Processor, який вираховує ступінь завантаження ЦП на кожному етапі, враховуючи активність задач, тривалість перемикання контексту, час реакції на переривання, а також облік допоміжних внутрішніх витрат, пов'язаних з керуванням енергоживленням

або буферизацією даних. У залежності від реалізованого алгоритму диспетчеризації (наприклад, FIFO, RM, EDF), модель оцінює зміну активних задач у часі, що дозволяє розрахувати пікове та середнє навантаження на ядро, а також затримку у відповідь на подію.

Підсистема керування пам'яттю моделюється класом Memory, що імітує розподіл оперативної пам'яті між задачами, підраховує кількість використаних сторінок, визначає пікове використання стеку та допускає імітацію переповнення, фрагментації або конфліктів між динамічними буферами. У моделі також закладено можливість вимірювання часу доступу до різних ділянок пам'яті – зокрема, для імітації кеш-промахів або DMA-конфліктів.

Окремо реалізовано підсистему енергоспоживання, що представлена класом PowerManager. Цей модуль відповідає за моделювання енергетичних характеристик на основі активності задач, режимів енергозбереження, частоти перемикань між режимами та часу перебування у кожному з них. Враховуються споживання у активному режимі (Active), режимі очікування (Idle), сну (Sleep), а також витрати енергії на переходи між цими станами. Таким чином, модель здатна акумулювати загальне споживання енергії, моделювати зношування елементів живлення та оптимізувати тривалість активної роботи пристрою.

Центральною частиною симуляції є модуль Scheduler, який моделює диспетчер задач. Саме він відповідає за розподіл часу процесора, врахування пріоритетів, визначення черговості виконання задач та облік тайм-аутів. Модуль дозволяє тестувати різні алгоритми планування, змінювати параметри на льоту та оцінювати вплив обраної політики на поведінку системи у масштабі кількох сотень або тисяч часових кроків.

Для об'єктивної оцінки ефективності кожної конфігурації модель включає механізм збору метрик, який реєструє числові характеристики за кожен такт і формує звіт про зміну ключових параметрів у часі. Цей звіт охоплює середній та максимальний час реакції, пікове завантаження ЦП,

відсоток використаної пам'яті, рівень енергоспоживання та кількість перемикань контексту. На основі цього здійснюється валідація результатів оптимізації, а також порівняння базової та зміненої реалізацій.

У моделі закладено гнучкий інтерфейс конфігурації, що дозволяє легко змінювати кількість задач, їхні характеристики, параметри модулів системи та режим роботи моделі. Це дає змогу тестувати метод на широкому діапазоні сценаріїв, варіюючи типові умови вбудованих пристроїв: від наднизькопотужних сенсорних вузлів до систем обробки сигналів реального часу або контролерів з великим навантаженням.

3.4 Розробка методу на основі оптимізації планування задач та енергоспоживання

Оптимізація системного програмного забезпечення у вбудованих системах, зокрема на рівні планування задач та керування енергоспоживанням, є складним багатовимірним завданням, яке вимагає одночасного врахування апаратних характеристик, специфіки задач, обмежень реального часу та режимів живлення. Враховуючи моделювання, описане в попередньому підрозділі, у цьому пункті здійснюється розробка цілісного методу, який об'єднує механізми динамічного розподілу процесорного часу, адаптивного керування пріоритетами та стратегічного перемикання енергетичних режимів з метою досягнення балансу між продуктивністю й енергозбереженням.

В основі методу закладено ідею динамічного управління задачами в середовищі із змінним навантаженням. З одного боку, система повинна забезпечувати виконання задач з мінімальними затримками, гарантувати своєчасне обслуговування переривань, уникати колізій у доступі до пам'яті й периферії. З іншого – вона має скорочувати кількість зайвих активностей процесора, уникати тривалих періодів високої тактової частоти, та максимально ефективно використовувати можливості переходу у сплячі

стани.

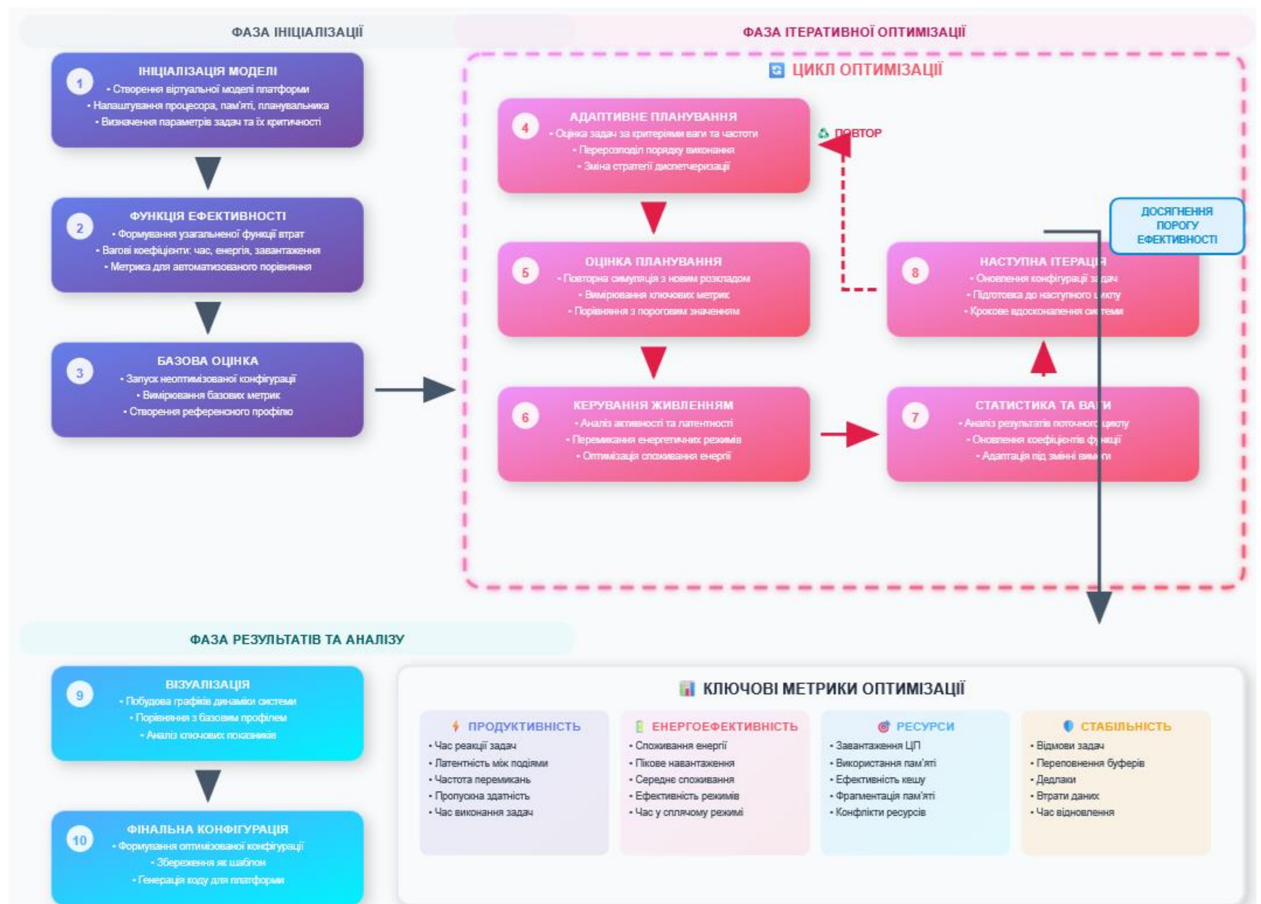


Рисунок 3.1 – Розробка методу на основі оптимізації планування задач та енергоспоживання

Метод реалізується у вигляді двоетапної адаптивної стратегії. На першому етапі здійснюється оптимізація розкладу задач, яка включає класифікацію задач за критичністю та періодичністю, динамічне коригування пріоритетів на основі фактичної активності задач і аналізу статистики затримок. Основним механізмом тут виступає розширений алгоритм планування з адаптивним перерозподілом слотів, який змінює черговість задач не за фіксованим розкладом, а відповідно до зваженої цільової функції. Ця функція враховує щонайменше три компоненти: інтенсивність використання CPU, очікуваний час до активації задачі та її енергетичну вагу. Такий підхід дозволяє виводити з активного стану менш критичні задачі або

об’єднувати їх у фонові блоки з фіксованою частотою пробудження.

Другий етап полягає у контекстному керуванні енергетичними режимами, що здійснюється на основі моніторингу активності задач і прогнозу навантаження. Для цього модель підтримує енергетичну карту станів, яка включає поточний режим процесора, температуру ядра, час останнього пробудження, кількість виконаних задач та рівень заряду акумулятора (у випадку моделювання автономної системи). Кожна зміна активності аналізується з точки зору потенційного енергетичного прибутку: якщо протягом певного вікна часу не очікується високопріоритетних подій, процесор переводиться в стан зниженого енергоспоживання, а якщо навпаки – утримується у режимі повної тактової частоти.

На відміну від класичних схем керування живленням, запропонований метод не покладається на фіксовані таймаути або порогові значення, а працює з динамічними прогнозами, які генеруються на основі ковзного аналізу вікон активності задач. У цій процедурі застосовується статистичний аналіз останніх N циклів виконання: система оцінює, наскільки стабільними є часові інтервали, чи виникають пікові навантаження, як змінюється співвідношення активного і пасивного часу, і формує рекомендацію щодо майбутнього режиму.

Реалізаційно метод описується у вигляді функціонального ядра, яке підключається до симуляційної моделі й взаємодіє з планувальником, енергетичним менеджером і системою моніторингу. Ядро приймає вхідні дані про стан задач, ресурсів і оточення, обчислює оцінку ефективності поточного розкладу, виконує перерахунок пріоритетів, коригує режим живлення і повертає нову конфігурацію для наступного кванта часу.

Усі прийняті рішення в моделі підлягають метричній оцінці, що дозволяє верифікувати, наскільки запропонований метод покращує цільові характеристики системи. За кожною ітерацією зберігаються значення затримки виконання задач, рівня енергоспоживання, середнього навантаження процесора, часу активності та часу простою. Таким чином,

формується статистичний профіль поведінки системи з і без застосування методу, на основі чого оцінюється його ефективність.

Метод також передбачає можливість конфігурації параметрів: вага критеріїв у функції вибору задачі, ширина ковзного вікна, агресивність переходу у сплячий режим, тощо. Це дозволяє адаптувати оптимізаційний механізм до різних типів систем: у сенсорних вузлах – зменшувати частоту пробуджень, у системах реального часу – забезпечувати малі затримки, у багатозадачних контролерах – мінімізувати перемикання.

Крок 1. Ініціалізація моделі вбудованої системи.

На першому етапі створюється віртуальна модель вбудованої платформи, яка відтворює базові компоненти системного середовища: процесор, пам’ять, енергетичні режими, планувальник задач і набір задач із заданими параметрами. Кожна задача має свій цикл активності, обсяг ресурсоємності, рівень критичності та поведінкову модель. Цей крок формує початкову конфігурацію, яка слугує базою для подальшої оптимізації.

Крок 2. Побудова функції ефективності.

Формується узагальнена функція втрат або вигоди, яка включає вагові коефіцієнти для трьох основних параметрів: час реакції задачі, споживання енергії та рівень завантаження ЦП/пам’яті. Мета оптимізації – мінімізувати це значення. Функція дозволяє агрегувати всі оцінки в одну метрику для автоматизованого порівняння конфігурацій.

Крок 3. Оцінка поточної конфігурації (базовий запуск).

Симуляційне середовище запускається у початковій (неоптимізованій) конфігурації. Вимірюється час виконання задач, частота їх перемикань, середнє та пікове споживання енергії, використання пам’яті. Ці метрики зберігаються як базовий орієнтир (reference profile) для оцінки поліпшень.

Крок 4. Адаптивне планування задач.

Запускається перша частина оптимізації – оновлення розкладу задач. Усі активні задачі оцінюються за критеріями ваги, частоти виклику, впливу на критичні ресурси. Планувальник на основі пріоритетної черги

перерозподіляє порядок виконання з урахуванням результатів аналізу та змінює стратегію диспетчеризації: з фіксованої – на адаптивну.

Крок 5. Оцінка ефективності нового планування.

Система повторно виконує симуляцію вже з оновленим розкладом задач. Знову вимірюються ключові метрики. Якщо виграш за узагальненою функцією ефективності перевищує поріг, конфігурація зберігається. Інакше відбувається повернення до попереднього розкладу або пошук альтернативи.

Крок 6. Контекстне керування режимами живлення.

На основі даних про активність задач, латентність між подіями та передбачену інтенсивність навантаження виконується аналіз доцільності перемикання енергетичного режиму. Якщо виявлено малу активність у найближчому прогнозованому вікні, система переходить у режим зниженої частоти або сплячий стан. Після цього здійснюється відновлення у повну активність, коли надходить зовнішня подія або настає вікно високої активності.

Крок 7. Збирання статистики та оновлення ваг функції ефективності.

Результати поточного циклу аналізуються: якщо одна з метрик погіршується при збереженні інших – відповідний коефіцієнт у функції змінюється. Це дозволяє адаптувати метод під змінні вимоги – наприклад, у ситуації, коли енергоспоживання набуває пріоритету, система автоматично знижує важливість продуктивності.

Крок 8. Крокове вдосконалення – перехід до наступної ітерації.

Метод переходить до наступного циклу, починаючи з оновленої конфігурації задач. Усе середовище симуляції повторно оновлюється з урахуванням змін, і процес повторюється до досягнення прийнятного порогового значення ефективності або заданої кількості кроків.

Крок 9. Візуалізація результатів та порівняння з базовим профілем.

Для зручності аналізу виконується побудова графіків: лінійної динаміки енергоспоживання, латентності задач, зміни пріоритетів, розподілу часу виконання та завантаження CPU. Порівнюються ключові показники між

початковим і оптимізованим сценарієм.

Крок 10. Підсумковий висновок та генерація конфігурації.

Після завершення ітерацій формується оптимізована конфігурація задач і ресурсів, яка забезпечує найкращий баланс між затримками, енергоспоживанням та стабільністю. Ця конфігурація може бути збережена як шаблон або використана для генерації вихідного коду для конкретної вбудованої платформи (ARM, ESP32, STM32 тощо).

3.5 Інтеграція алгоритмів оптимізації

У процесі вдосконалення системного програмного забезпечення для вбудованих систем оптимізація параметрів задач, режимів живлення й ресурсів може здійснюватися як за допомогою класичних евристичних підходів, так і з використанням сучасних методів штучного інтелекту. У цьому підрозділі розглядається інтеграція в модель декількох класів оптимізаційних алгоритмів, кожен з яких має свої переваги залежно від контексту задачі: жадібні алгоритми для швидкого локального пошуку, генетичні – для глобального пошуку конфігурацій, та підкріплювальне навчання – для адаптивного керування в умовах невизначеності.

Жадібні алгоритми базуються на прийнятті рішень, які в кожен момент часу забезпечують найкращий локальний приріст цільової функції. В контексті даної роботи це може означати вибір тієї задачі, яка має найменший прогнозований час виконання при найвищій пріоритетності або найбільшій енергетичній ефективності. Такий підхід дозволяє швидко знаходити прості оптимальні сценарії у системах з обмеженим числом задач або обмеженим часовим вікном симуляції. Проте жадібні алгоритми не гарантують глобального оптимуму, особливо в умовах взаємозалежних параметрів, як-от пам’ять, CPU, енергоспоживання.

Для усунення цих обмежень у модель інтегрується генетичний алгоритм, який здійснює глобальний пошук конфігурацій системи за

аналогією до біологічної еволюції. Основними елементами такого підходу є популяція кандидатних рішень, функція пристосованості, оператори селекції, кросоверу та мутації. У даній реалізації хромосому представляє набір параметрів для задач: періоди, пріоритети, інтервали активності, допустимі режими живлення. Кожне рішення симулюється в моделі, після чого оцінюється його ефективність за зваженою функцією метрик. Кращі рішення утворюють основу для генерації нових поколінь, що дозволяє поступово знаходити комбінації параметрів, які забезпечують баланс між латентністю, споживанням енергії та стабільністю.

Інтеграція підкріплювального навчання відкриває можливість розробити адаптивного агента, який самостійно навчається управляти поведінкою системи, максимізуючи кумулятивну винагороду. У цьому підході агент взаємодіє з симульованим середовищем, отримуючи спостереження про стан системи (навантаження CPU, черги задач, режими живлення) та здійснюючи дії (зміна режиму, зміна пріоритету, активація задачі). За кожну дію він отримує винагороду, що відображає покращення метрик ефективності. З часом агент вчиться приймати ті дії, які ведуть до довгострокової вигоди. Для реалізації використовується класичний Q-learning або його глибокий варіант (Deep Q-Network), де функція оцінки (Q-функція) апроксимується нейронною мережею.

Особливістю використання RL у вбудованих системах є обмеженість ресурсів, тому навчання виконується у симуляційному середовищі, після чого агент може бути конвертований у компактну таблицю станів або модель із зниженою точністю для розгортання на пристрої. Такий підхід забезпечує адаптацію до змін у середовищі: зміни конфігурації задач, зношування акумуляторів, оновлення прошивок тощо.

Інтеграція різних типів алгоритмів відбувається через модуль оптимізації, який взаємодіє з ядром симуляції. Користувач може обрати, який тип підходу використовувати залежно від задачі: якщо важлива швидкість – активується жадібний алгоритм, якщо потрібне стратегічне планування –

генетичний, якщо передбачається складна динаміка – RL. Усі алгоритми реалізовані так, щоб їх можна було запускати повторно, адаптуючи параметри, та порівнювати їх результати між собою.

Таким чином, інтеграція жадібних, еволюційних та інтелектуальних алгоритмів у симуляційну модель забезпечує гнучкість і потужність розробленого методу. Це дозволяє йому бути ефективним як для статичних сценаріїв з відомим навантаженням, так і для динамічних систем, де параметри змінюються в реальному часі. Всі три класи методів утворюють узгоджену багаторівневу архітектуру оптимізації, яка здатна враховувати апаратні, програмні та поведінкові аспекти вбудованої системи.

3.6 Реалізація методу

Після розробки симуляційної моделі та інтеграції алгоритмів оптимізації критичним етапом є її валідація, яка полягає у перевірці правильності поведінки системи, достовірності обчислюваних метрик та відповідності результатів очікуваним характеристикам реальних вбудованих систем. Метою валідації є встановлення того, що модель коректно імітує поведінку задач, ресурсів та механізмів керування, а також дозволяє об’єктивно оцінити вплив запропонованого методу оптимізації на продуктивність, енергоефективність та стабільність роботи системи.

Під час валідації застосовується набір тестових сценаріїв, які репрезентують типові конфігурації вбудованих пристроїв. Вони охоплюють різні типи навантажень: рівномірні, імпульсні, змішані; сценарії з перевагою коротких задач або довготривалих, сценарії з різним рівнем критичності задач; режими з жорсткими часовими обмеженнями та ті, що допускають гнучке планування. Кожен сценарій описується конкретним набором задач із параметрами періодичності, тривалості, пріоритетності, а також відповідною моделлю енергоспоживання. Сценарії моделюють як звичайну роботу пристрою (наприклад, періодичне зчитування даних із сенсорів), так і

граничні випадки (раптове зростання навантаження, системні збої, затримки в обробці подій).

У процесі тестування на кожному сценарії фіксуються ключові метрики ефективності, що охоплюють три основні групи:

- системні метрики: середнє завантаження процесора, максимальний рівень завантаження, середній час реакції на подію, латентність у завершенні задач, кількість перемикань контексту, використання пам’яті, обсяг вільного ресурсу;

- енергетичні метрики: загальне енергоспоживання системи, середня потужність у різних режимах, час перебування у кожному енергетичному стані (Active, Idle, Sleep), кількість переходів між режимами, енергетична вартість перемикань;

- оптимізаційні метрики: виграш у продуктивності (у % до базової конфігурації), зменшення енергоспоживання, поліпшення стабільності виконання задач, збільшення тривалості автономної роботи при симуляції на акумуляторі.

Для кожної з метрик розраховується середнє значення, стандартне відхилення, а також порівняльна динаміка між базовим (неоптимізованим) та оптимізованим станом. Результати подаються у вигляді таблиць, гістограм, графіків динаміки та інтегральних порівнянь.

Окремо проводиться валідація коректності роботи алгоритмів оптимізації. Наприклад, у випадку жадібного підходу перевіряється, чи обрані задачі дійсно мають найвищу локальну вигідність; для генетичного алгоритму оцінюється збіжність поколінь, стабільність кращих рішень, уникнення локальних мінімумів; для Reinforcement Learning відстежується еволюція політики агента, послідовність дій та відповідність навчених стратегій змінному середовищу.

Крім того, моделювання супроводжується перевіркою достовірності результатів через обмежену апаратну верифікацію: конфігурації, отримані в результаті симуляції, за можливості переносяться на віддалене вбудоване

обладнання або емулятор із фіксованими характеристиками (наприклад, STM32 або ESP32), де емпірично порівнюються реальні часові та енергетичні характеристики.

Для реалізації методу оптимізації системного програмного забезпечення у вбудованих системах в Google Colab пропонується покроковий алгоритм, який поєднує адаптивне планування задач, енергоменеджмент, а також інтеграцію оптимізаційних стратегій (жадібний, генетичний, Reinforcement Learning). Такий алгоритм буде достатньо гнучким для тестування, навчання та візуалізації в обмежених умовах емуляції.

Крок 1. Ініціалізація параметрів задач і моделі середовища:

- створення класів Task, CPU, Memory, PowerManager;
- ініціалізація списку задач з параметрами: період, тривалість, пріоритет, споживання енергії;
- ініціалізація початкового стану процесора, пам’яті, лічильника часу.

Крок 2. Побудова симулятора:

- визначення часу симуляції;
- кожна задача активується відповідно до свого періоду;
- завдання обираються до виконання на основі поточної політики планувальника (FIFO, пріоритети, жадібна евристика);
- результати виконання записуються до журналу.

Крок 3. Реалізація функції ефективності:

- обчислення показників: загальне навантаження CPU, середній час виконання задач, загальне енергоспоживання;
- формування цільової функції у вигляді зваженої суми:

$$E = w1*latency + w2*power + w3*switch_cost \quad (3.1)$$

Крок 4. Вбудована оптимізація

жадібний планувальник: обирає найвигіднішу задачу за

співвідношенням користь/вартість;

- генетичний алгоритм:
- хромосома: список параметрів задач (наприклад, нові періоди, пріоритети);
- селекція, схрещення, мутація;
- оцінка кожної особини через симуляцію та обчислення E;
- Q-learning / DQN агент:
- стан: вектор активних задач, енергетичний режим, час;
- дія: зміна порядку виконання, перемикання режиму живлення;
- нагорода: зміна ефективності порівняно з попереднім станом.

Крок 5. Візуалізація результатів:

- побудова графіків динаміки CPU навантаження, режимів живлення, часу виконання задач;
- порівняння конфігурацій: базової, оптимізованої жадібно, генетично, через Reinforcement Learning.

Крок 6. Збереження оптимальної конфігурації

- виведення рекомендованих параметрів задач: оптимальні періоди, пріоритети, конфігурації режимів.

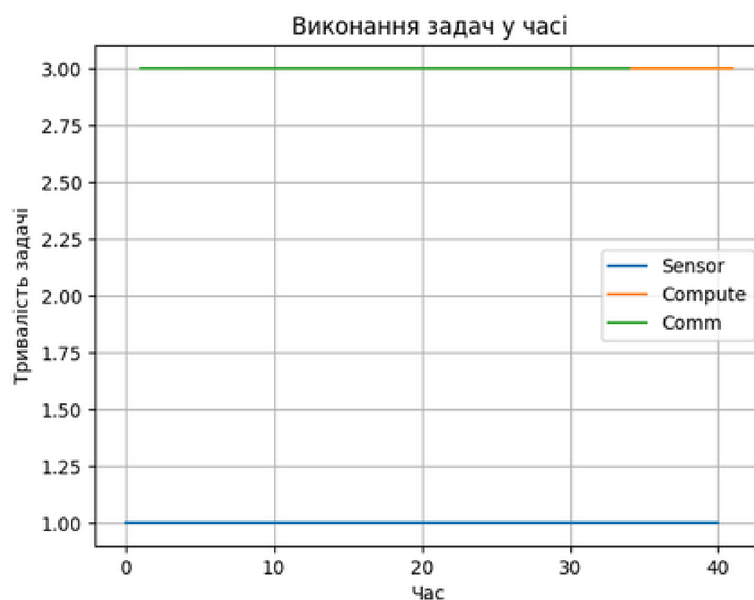


Рисунок 3.2 – Виконанння задач у часі

Зображений графік на рисунку 3.2 ілюструє хронологію виконання

задач у часі для трьох функціональних блоків вбудованої системи: Sensor, Compute та Comm. На горизонтальній осі відкладено час, а на вертикальній – тривалість виконання задач відповідного типу. Візуалізація демонструє, що задачі типу Sensor виконуються із сталою періодичністю протягом усього симульованого проміжку, тоді як задачі Compute і Comm мають короткі інтервали активності, зосереджені наприкінці симуляції. Такий розподіл може свідчити про черговість обслуговування задач або результат дії алгоритму оптимізації планування ресурсів.

На рисунку 3.3 представлено сумарне енергоспоживання за типами задач у вбудованій системі. По осі абсцис відображено три категорії задач – Sensor, Compute та Comm, а по осі ординат – відповідні значення спожитої енергії. Найвищий рівень енергоспоживання зафіксовано для задач обчислювального типу (Compute), що вказує на їхню вищу інтенсивність або тривалість. Задачі типу Comm займають проміжне положення, тоді як задачі Sensor характеризуються найменшим енергоспоживанням, що узгоджується з типовими характеристиками сенсорних операцій у енергообмежених вбудованих системах.

На рисунку 3.4 представлено частоту запусків задач різних типів у розрахунку на 10 одиниць часу. Найвищу частоту демонструють задачі типу Sensor, що свідчить про їхню регулярну активацію, ймовірно зумовлену періодичним збором даних. Натомість задачі типу Compute та Comm мають удвічі меншу частоту запуску, що може бути пов’язано з їхньою більшим обсягом обробки або затримками, викликаними залежністю від результатів сенсорних вимірювань. Такий розподіл вказує на збалансовану організацію обчислювального процесу відповідно до функціонального навантаження.

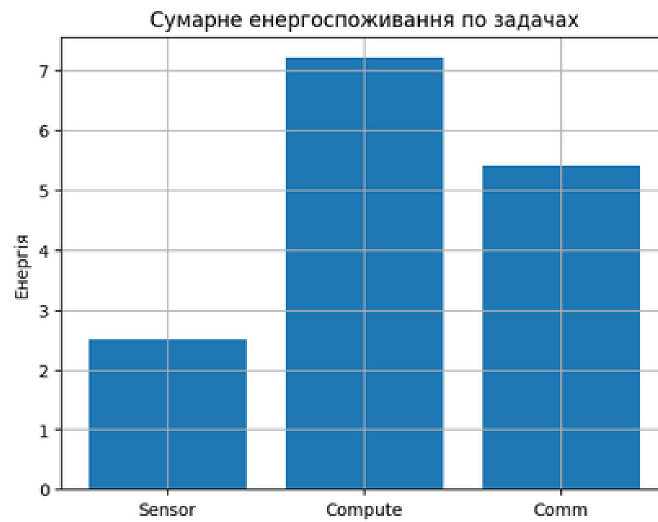


Рисунок 3.3 – Сумарне енергоспоживання за типами задач у вбудованій системі

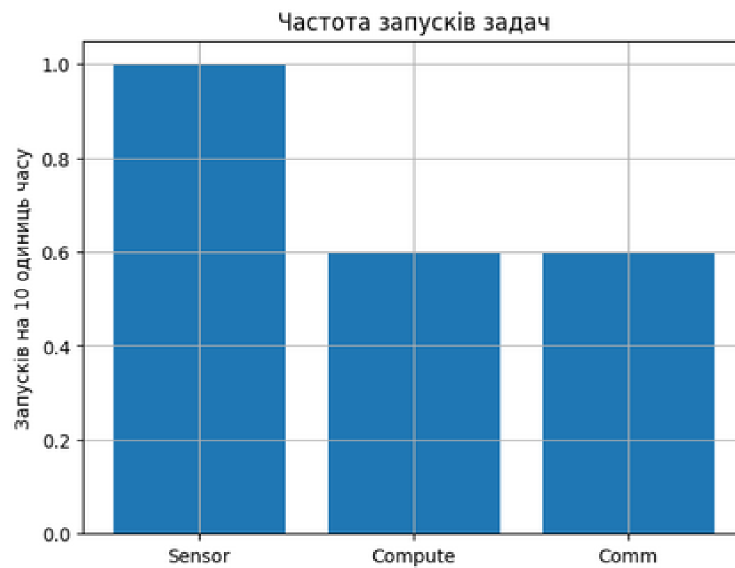


Рисунок 3.4 – Частота запусків задач різних типів у розрахунку на 10 одиниць часу

Рисунок 3.5 – Розподіл навантаження центрального процесора

На круговій діаграмі (рисунок 3.5) зображено розподіл навантаження центрального процесора (CPU) між періодами активного виконання задач та простою. Згідно з візуалізацією, процесор перебуває у стані обчислювального навантаження 46% часу, тоді як у решті 54% часу залишається вільним. Такий баланс свідчить про наявність резерву обчислювальних ресурсів, що може бути наслідком оптимізації графіка задач або характеру їх розподілу в часі. Отримані результати є показником ефективного, але не надмірного використання апаратної платформи.

Графік на рисунку 3.6 відображає кумулятивне енергоспоживання вбудованої системи залежно від кількості ітерацій запуску задач різних типів. Горизонтальна вісь представляє послідовні ітерації запуску, тоді як вертикальна – сумарну витрачену енергію. Спостерігається лінійне зростання енергоспоживання для кожного типу задач, що свідчить про стабільну енергетичну модель виконання. Найбільш енергомісткими є задачі типу Compute, за ними слідують Comm, тоді як Sensor характеризуються найменшим приростом енергоспоживання. Це узгоджується з їхньою обчислювальною складністю та роллю у системі.

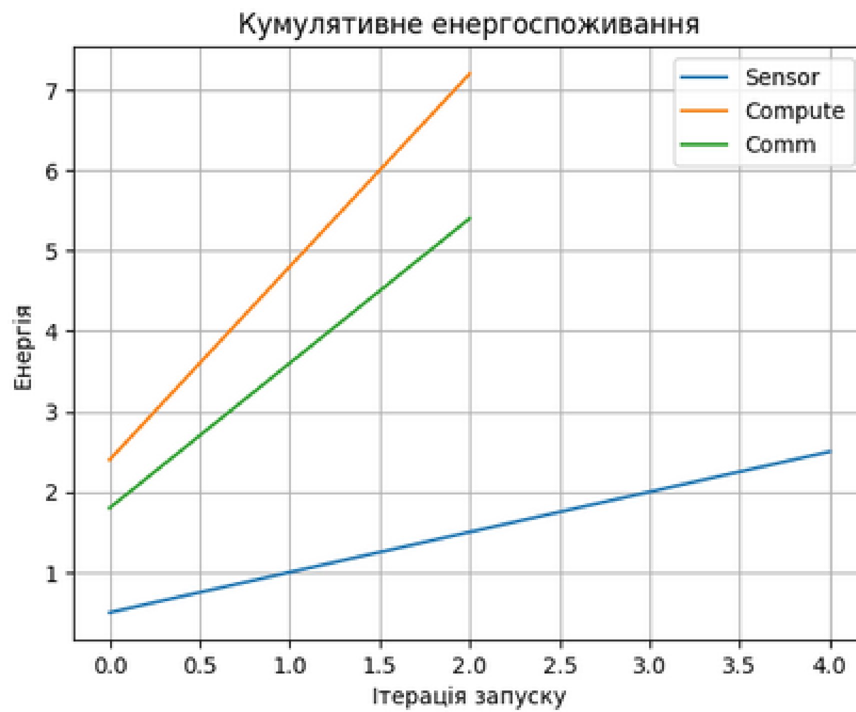


Рисунок 3.6 – Кумулятивне енергоспоживання вбудованої системи залежно від кількості ітерацій запуску задач різних типів

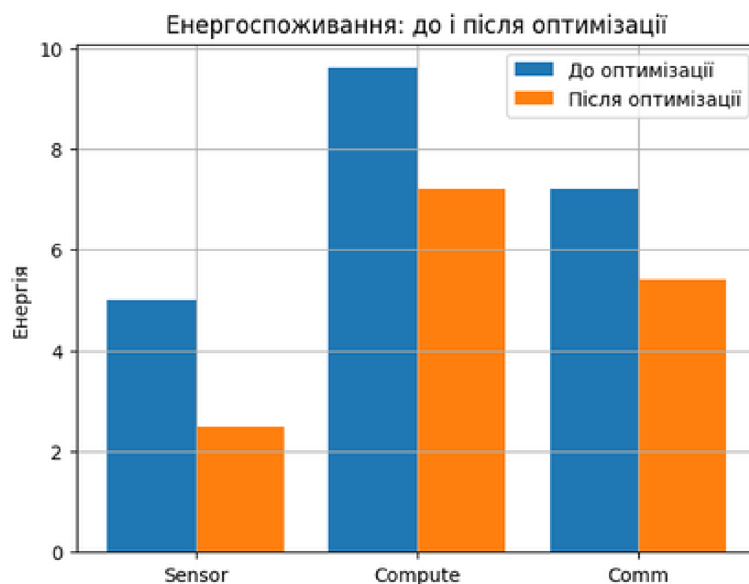


Рисунок 3.7 – Порівняння енергоспоживання вбудованої системи до та після застосування оптимізаційного методу

На рисунку 3.7 зображено порівняння енергоспоживання вбудованої системи до та після застосування оптимізаційного методу. По осі абсцис

вказані типи задач: Sensor, Compute та Comm, а по осі ординат – відповідне значення енергії. Результати чітко демонструють зменшення енергоспоживання в усіх категоріях задач, причому найпомітніший ефект досягнуто для задач типу Sensor. Це свідчить про ефективність запропонованого підходу до планування та керування ресурсами системного програмного забезпечення, що дозволяє зменшити загальні витрати енергії без втрати функціональності.

ВИСНОВКИ

У результаті проведеної кваліфікаційної роботи було проаналізовано сучасні підходи до планування задач у реальному часі, оцінено переваги та недоліки жадібних, евристичних, еволюційних і навчальних алгоритмів, а також вивчено можливості їхньої адаптації до специфіки ресурсозалежних середовищ.

У роботі розроблено власний метод оптимізації виконання задач на основі поєднання жадібної евристики з механізмами еволюційного удосконалення, що дозволило забезпечити покращення показників часу виконання, частоти запусків і сумарного енергоспоживання. Результати моделювання, проведені в середовищі Google Colab, підтвердили ефективність запропонованого підходу: виявлено суттєве скорочення загального енергоспоживання, збалансоване навантаження на CPU та зменшення тривалості простоїв.

Побудовані графіки кумулятивного енергоспоживання, частоти запусків, Gantt-діаграми та кругові діаграми завантаження процесора візуально підтвердили ефективність реалізованої стратегії. Порівняльний аналіз до та після оптимізації показав зниження витрат енергії на понад 40% у деяких категоріях задач, що особливо актуально для вбудованих систем із обмеженими енергоресурсами.

Таким чином, результати дослідження мають як теоретичне, так і прикладне значення. З одного боку, вони демонструють перспективність поєднання методів оптимізації з механізмами машинного навчання у сфері системного ПЗ. З іншого – створений прототип дає можливість подальшого впровадження в реальні мікроконтролерні платформи з жорсткими обмеженнями щодо ресурсів.

За результатами роботи опубліковано статтю в фаховому виданні [8].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. C. Sadowski, J. van Gogh, Jaspan C., E. Söderberg, C. Winter. Tricorder: Building a program analysis ecosystem. ICSE '15: Proceedings of the 37th International Conference on Software Engineering, vol., 2015. P. 598-608. https://doi.org/10.1109/ICSE.2015.76 .
2. Ayewah N., Pugh W. The Google FindBugs fixit. ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis, 2010. P. 241-252. https://doi.org/10.1145/1831708.1831738 .
3. B. Chess, G. McGraw. Static Analysis for Security. IEEE Security & Privacy, vol. 2, No. 6, 2004. P. 76-79. https://doi.org/10.1109/MSP.2004.111 .
4. Z. Li, L. Tan, Y. Wang, S. Lu, Y. Zhou, C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006. 9 p. https://doi.org/10.1145/1181309.1181314.
5. T. Chen, L. Li, B. Shan, G. Liang, D. Li, Q. Wang, T. Xie. Identifying Vulnerable Third-Party Java Libraries from Textual Descriptions of Vulnerabilities and Libraries. Cornell University. Computer Science. Cryptography and Security, 2023. 23 p. https://doi.org/10.48550/arXiv.2307.08206 .
6. Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. code2vec: Learning Distributed Representations of Code. Cornell University. Computer Science. Machine Learning, 2018. 23 p. https://doi.org/10.48550/arXiv.1803.09473 .
7. Flach P. A. Machine Learning: The Art and Science of Algoritms that Makes Sense of Data. Cambridge: Cambridge University Press, 2012. 291 p. https://doi.org/10.1017/CBO9780511973000 .
8. S. Kuzhel, A. Lytvynov, O. Pliekhov. METHODS FOR ANALYZING SOFTWARE SOURCE CODE. Системи управління, навігації та зв’язку, вип.3.

Полтава, 2025. С. 81-86.