ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

Кваліфікаційна робота

Методи оптимізації системного програмного забезпечення для вбудованих систем

Виконав:
студент гр. СПм-23-3
Кужель С.І.

Керівник:
доц. каф. ЕОМ
Сорокін А.Р.

Мета роботи та завдання 2

Метою 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..

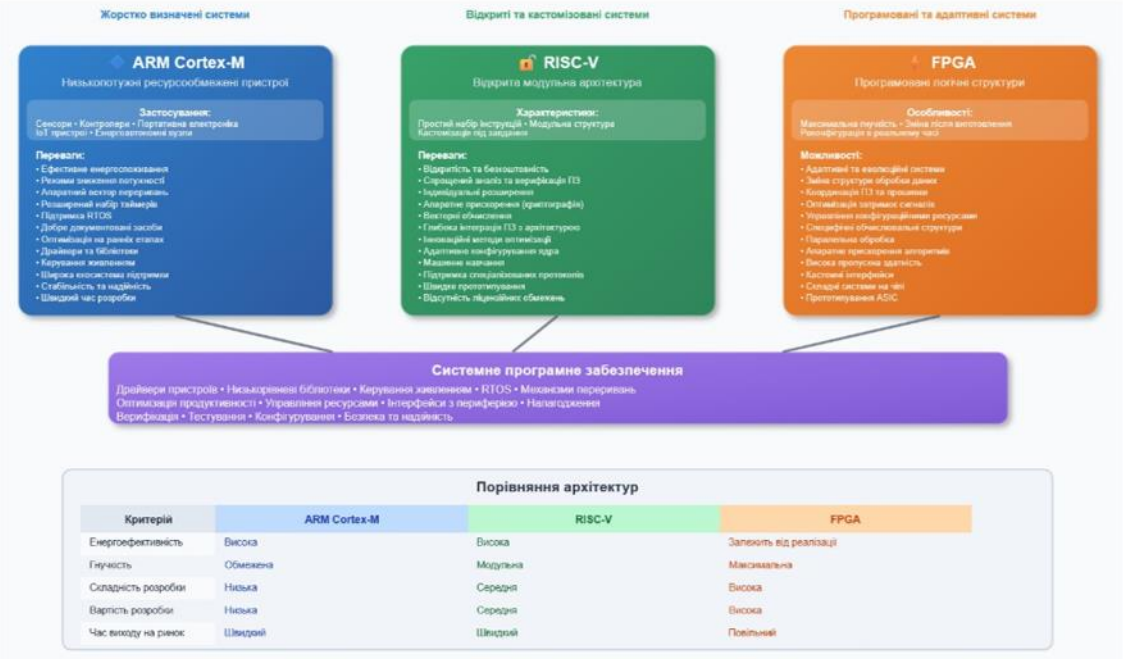
Об'єктом дослідження є системне програмне забезпечення вбудованих обчислювальних платформ.

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).

Вимоги до системного програмного забезпечення у вбудованих пристроях 3

Типові архітектури вбудованих систем 4



Порівняння архітектур

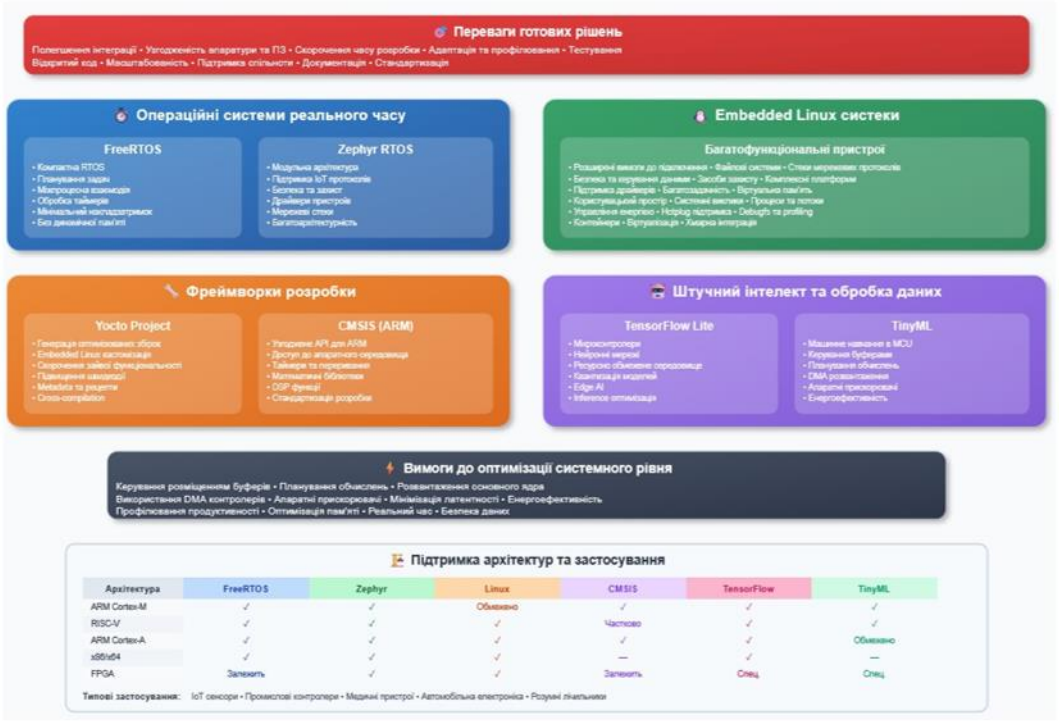
Критерій	ARM Cortex-M	RISC-V	FPGA
Енергоефективність	Висока	Висока	Залежить від реалізації
Гнучкість	Обмежена	Модульна	Максимальна
Складність розробки	Низька	Середня	Висока
Вартість розробки	Низька	Середня	Висока
Час виходу на ринок	Швидкий	Швидкий	Повільний

Методи оптимізації програмного коду у вбудованих системах

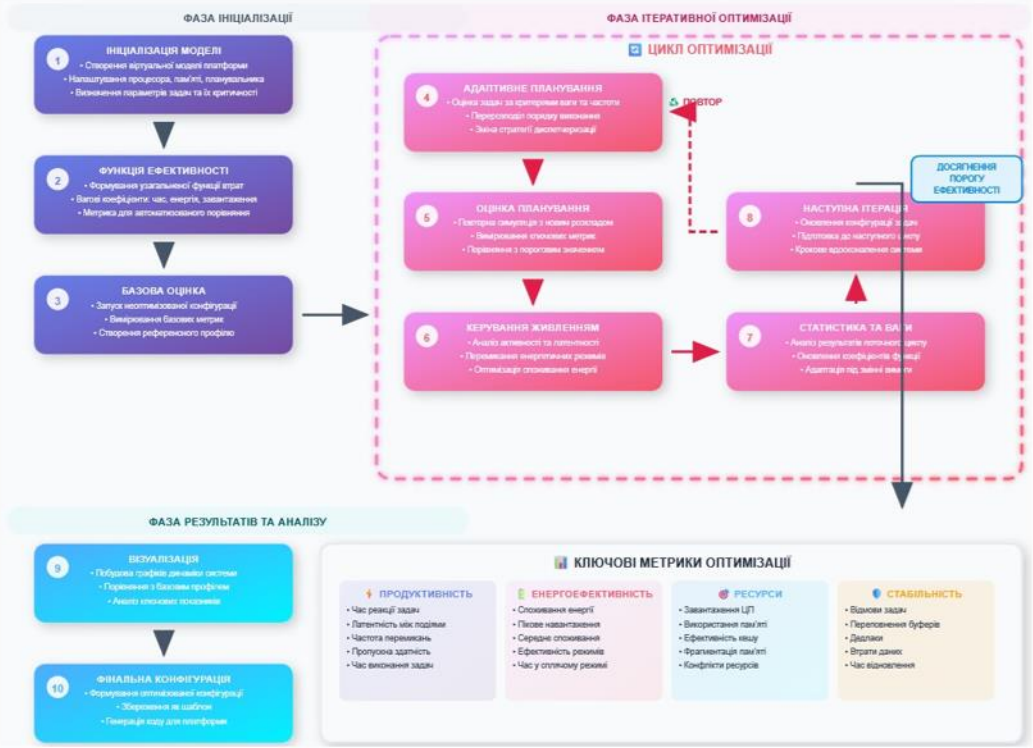
5

Аналіз існуючих систем та засобів розробки системного ПЗ

6



Розробка методу на основі оптимізації планування задач та енергоспоживання 7



Опис методу на основі оптимізації планування задач та енергоспоживання 8

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 альтернатив.

Опис методу на основі оптимізації планування задач та енергоспоживання 9

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.

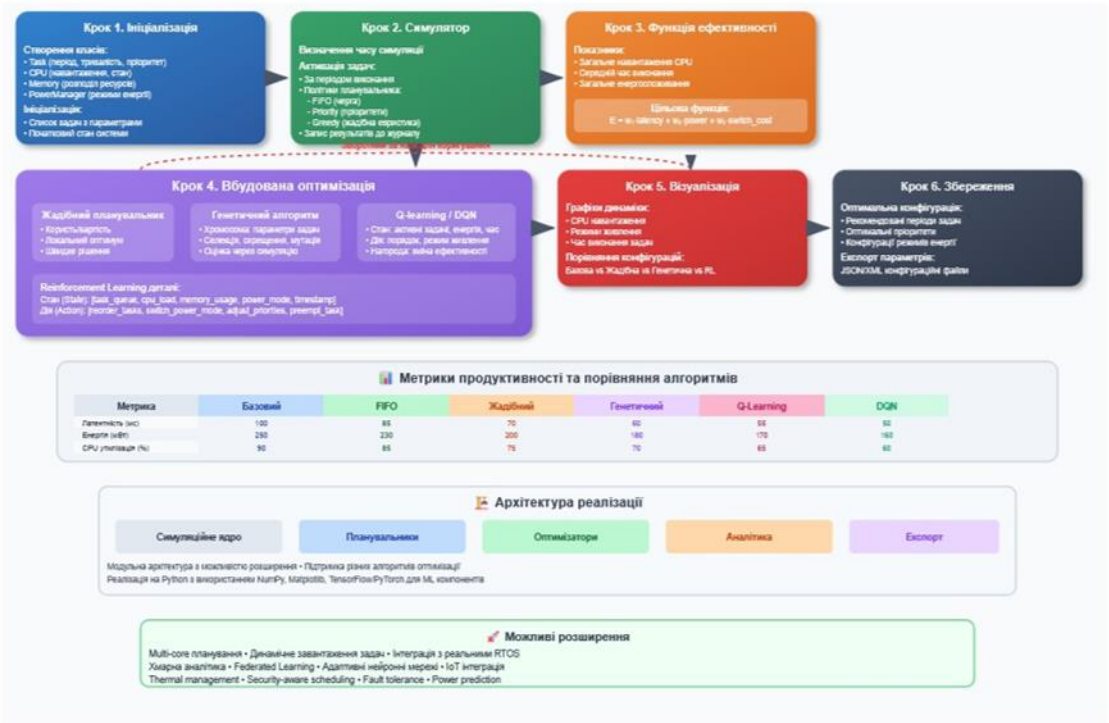
Крок 9. Візуалізація результатів та порівняння з базовим профілем.

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.

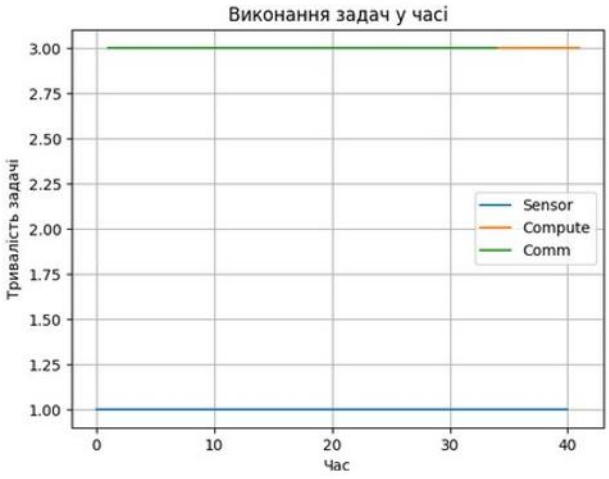
Після завершення ітерацій формується оптимізована конфігурація задач і ресурсів, яка забезпечує найкращий баланс між затримками, енергоспоживанням та стабільністю. Ця конфігурація може бути збережена як шаблон або використана для генерації вихідного коду для конкретної вбудованої платформи

Реалізація методу оптимізації системного програмного забезпечення у вбудованих системах в Google Colab 10



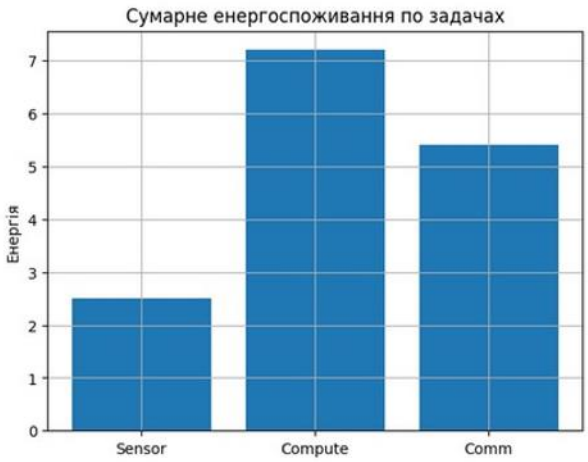
11

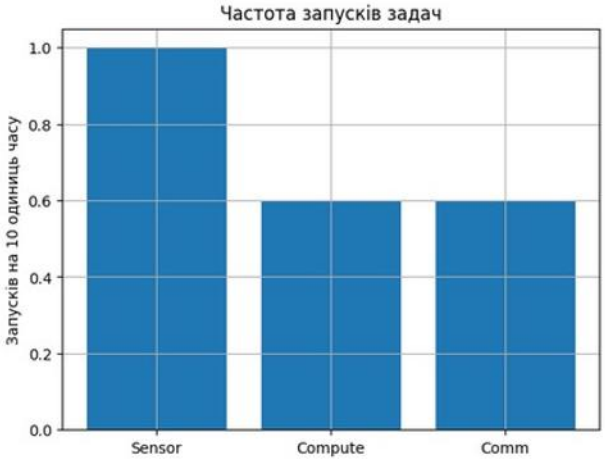
Аналіз результатів



12

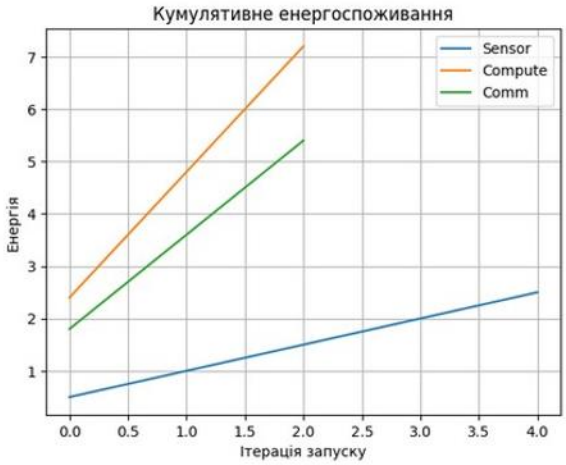
Аналіз результатів

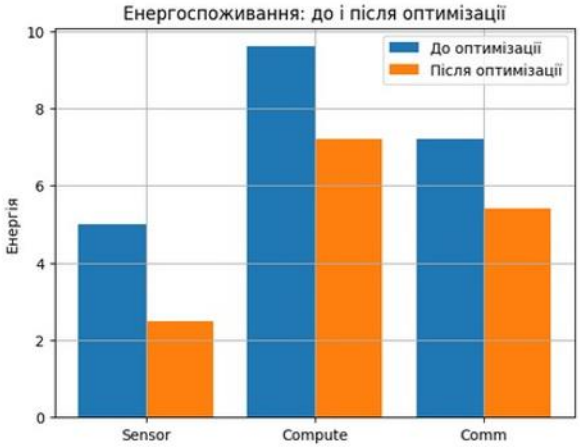




Аналіз результатів

13





Апробація результатів

14

METHODS FOR ANALYZING SOFTWARE SOURCE CODE

Abstract. Relevance. The study of methods for analyzing source code is driven by several current trends in the field of software engineering. On the one hand, the increasing complexity and scale of software solutions necessitate the enhancement of software quality and stability. On the other hand, the growing frequency of cyber threats demands greater attention to the security of source code. Modern software systems are involved in critical areas such as financial operations, healthcare, industrial automation, and infrastructure management. Errors and deficiencies in such systems can lead to serious consequences, including significant economic losses, failures in the operation of essential services, and even threats to human life. In an environment of intensifying competition and rapid digitalization, software quality has become a key factor in the success of companies in the market. The implementation of effective source code analysis methods is becoming essential not only for large enterprises, but also for small and medium-sized businesses seeking to develop reliable and secure software solutions. Emphasis should be placed on the promising potential of integrating modern technologies of artificial intelligence and machine learning into the process of code analysis. Such approaches enable the automatic detection and classification of errors, which significantly reduces the time required for their identification and resolution, while also improving the accuracy and efficiency of the analysis. Thus, the development and deepening of research in the field of source code analysis methods is a vital task of modern software engineering, as it enables the resolution of urgent challenges related to software quality assurance and system security. **The object of research** the source code of software is considered a structure subject to formal, semantic, and behavioral analysis, aimed at identifying errors, vulnerabilities, architectural flaws, and violations of coding standards. **Purpose of the article.** The study focuses on existing methods of source code analysis, particularly static and dynamic approaches, their tool support, and the prospects of applying modern technologies, especially artificial intelligence, to enhance the efficiency of error detection, and to ensure the quality, reliability, and security of software code throughout all stages of the software development lifecycle. **Research results.** The classification of software source code analysis methods has been systematized, including static, dynamic, hybrid, and intelligent approaches. A comparative analysis of static and dynamic techniques has been conducted based on key criteria such as efficiency, error coverage, resource intensity, and applicability at various stages of the software development lifecycle. Typical categories of errors detectable through dynamic analysis have been identified, including memory leaks, resource access errors, and performance issues. The potential of intelligent tools, particularly neural network-based models such as code2vec and VulLibMiner, has been examined for automated analysis and vulnerability prediction. The feasibility of a comprehensive approach that integrates both static and dynamic analysis has been substantiated as the most effective strategy for ensuring the quality and security of software systems. **Conclusions.** Static analysis is effective for early error detection and ensuring code compliance with established standards. Dynamic analysis is essential for identifying runtime errors such as memory leaks and race conditions. Neither method is universal; the best results are achieved through their combination. Intelligent approaches (AI/ML) significantly enhance the automation and accuracy of code analysis. The comprehensive implementation of code analysis contributes to the development of secure, high-quality, and maintainable software.

Keywords: source code analysis, static analysis, dynamic analysis, software defects, code security, software quality, analysis tools, artificial intelligence, machine learning, automated verification, code2vec, VulLibMiner, CI/CD.

Introduction

Modern information technologies have fundamentally transformed approaches to software development. The complexity of software products continues to increase, which in turn raises the risks of errors, vulnerabilities, and deficiencies in the source code. For this reason, the effective application of source code analysis methods has become critically important. Their main objective is to ensure code quality, security, reliability, and compliance with established standards. These analysis techniques assist developers in quickly identifying and eliminating errors, which ultimately helps reduce the time and resources required for software development and testing.

Depending on the verification approach, analysis dynamic analysis allows for the observation of program behavior under real operating conditions, identifying resource leaks, memory handling issues, multithreading errors, and other problems that are difficult to predict using static methods alone.

In recent years, the integration of artificial intelligence into source code analysis methods has significantly increased. The use of machine learning enables the automation of the analysis process, the classification of errors, and the prediction of their occurrence – all of which substantially enhance the efficiency and quality of software development.

Review of Recent Studies and Publication. The issue of source code analysis is actively explored in the context of software quality assurance, software testing, and information security. Most academic publications

lyzers. A distinctive feature of the platform is its scalability, the ability to continuously update verification rules, and a focus on automated, real-time feedback delivery. The article demonstrates how static analysis can be organically integrated into large-scale production workflows to improve overall software quality in high-intensity development environments.

Article [2] is dedicated to Google's experience with the use of the static analysis tool FindBugs in Java projects. The authors describe the Fixit initiative, a focused effort in which developers dedicated time exclusively to resolving issues identified by FindBugs. During the experiment, thousands of defects were analyzed, most of which fell into categories such as null reference errors, incomplete initialization, and the use of potentially dangerous programming patterns. Notably, numerous defects were discovered even in well-tested code that had previously gone unnoticed. The authors conclude that the regular application of static analysis significantly improves code quality without requiring additional testing resources. This publication is particularly valuable as an example of how source code analysis can be effectively adopted at industrial scale.

Study [3] explores the importance of static source code analysis from a security perspective. The authors emphasize that many common vulnerabilities, such as buffer overflows, SQL injections, XSS, and other standard security issues, can be detected at early development stages using specialized analyzers. Using tools like Fortify and Coverity as examples, the authors demonstrate the capability to automatically detect a wide range of security flaws without executing the code. The paper highlights that static analysis should be a systematic part of the secure development process, not a one-time testing effort. The importance of proper developer training and tool configuration for achieving effective results is also underlined. This work remains foundational in the field of secure programming and retains its relevance amid growing cyber threats.

Paper [4] presents an empirical study of bugs in modern open-source software. The authors analyzed bugs reported in the repositories of several major open-source projects, including Mozilla, Apache, and Eclipse, and classified them by type, origin, and consequences. It was found that most errors stem from human factors – incorrect assumptions, API changes, or complex component interactions. Special attention is given to the difficulty of identifying logical errors, which are not always caught by tests or compilers. The authors conough security audits. The authors also discuss the future development prospects of the system, including its scalability to other programming languages, integration into IDEs, and automatic generation of recommendations for updating insecure libraries.

Publication [6] presents an innovative approach to code analysis using deep learning techniques. The study introduces the code2vec system, which transforms source code into vector representations, enabling the use of neural network models for tasks such as classification, recommendation, and bug prediction. The method is based on representing code snippets as paths in the abstract syntax tree, which are then fed into a neural network. As a result, the system can «understand» the structure of code and learning to recognize patterns associated with certain types of errors or stylistic deviations. code2vec exemplifies a new generation of code analysis tools that combine classical syntactic analysis methods with artificial intelligence capabilities. This work is of fundamental importance to the development of intelligent code analysis systems.

The reviewed publications demonstrate the high scientific and practical relevance of source code analysis methods as one of the key tools for ensuring software quality, security, and reliability. The research shows that static analysis is extremely effective for the early detection of syntactic, semantic, and logical errors – particularly at scale in corporate environments (e.g., Google and SAP) – and for identifying vulnerabilities before program execution. Industrial practice indicates that static analysis integrated into CI/CD pipelines significantly reduces technical debt and improves release quality.

Meanwhile, dynamic analysis is considered a complementary yet essential stage that enables the detection of issues related to performance, memory leaks, concurrency, and real-world execution behavior. Publications describing tools such as VulLibMiner highlight the growing importance of analyzing third-party libraries and dependencies, which constitute a critical part of the modern software landscape.

The purpose of this work is to investigate and critically analyze modern methods of source code analysis, including static, dynamic, and intelligent approaches, to evaluate their effectiveness, applicability at various stages of the software development lifecycle, and the potential for integrating machine learning tools to enhance the quality, security, and reliability of software code amid the increasing complexity of software

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.

Висновки

15

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.

ДОДАТОК Б

Програмний код

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "id": "9f7af830",
   "metadata": {},
   "source": [
    "# Покращений метод оптимізації системного ПЗ для вбудованих
систем\n",
    ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "855b526f",
   "metadata": {},
   "outputs": [],
   "source": [
    "class Task:\n",
    "    def __init__(self, name, period, duration, priority,
power, memory):\n",
    "        self.name = name\n",
    "        self.period = period\n",
    "        self.duration = duration\n",
    "        self.priority = priority\n",
    "        self.power = power\n",
    "        self.memory = memory\n",
    "        self.next_activation = 0\n",
    "        self.executed_time = 0\n",
    "        self.total_energy = 0\n",
    "        self.log = []\n",
    "\n",
    "    def reset(self):\n",
    "        self.next_activation = 0\n",
    "        self.executed_time = 0\n",
    "        self.total_energy = 0\n",
    "        self.log.clear()\n",
    "\n",
    "class CPU:\n",
    "    def __init__(self):\n",
    "        self.time = 0\n",
    "        self.load = 0\n",
    "        self.history = []\n",
    "\n",
    "    def run(self, task, duration):\n",
    "        self.load += duration\n",
```

```
    "        task.executed_time += duration\n",
    "        task.total_energy += task.power * duration\n",
    "        self.history.append((self.time, task.name,
duration))\n",
    "        self.time += duration\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "8931dfb9",
   "metadata": {},
   "outputs": [],
   "source": [
    "class GreedyScheduler:\n",
    "    def __init__(self, tasks):\n",
    "        self.tasks = tasks\n",
    "\n",
    "    def schedule(self, cpu, max_time):\n",
    "        time = 0\n",
    "        for task in self.tasks:\n",
    "            task.reset()\n",
    "        while time < max_time:\n",
    "            available = [t for t in self.tasks if time >=
t.next_activation]\n",
    "            if available:\n",
    "                best = min(available, key=lambda t:
(t.duration / (t.priority + 1)) * t.power)\n",
    "                cpu.run(best, best.duration)\n",
    "                best.log.append((time, best.duration))\n",
    "                best.next_activation += best.period\n",
    "                time += best.duration\n",
    "            else:\n",
    "                time += 1\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "6f72add0",
   "metadata": {},
   "outputs": [],
   "source": [
    "import random\n",
    "import copy\n",
    "\n",
    "def mutate(task):\n",
    "    choice = random.choice(['period', 'priority',
'duration'])\n",
    "    if choice == 'period':\n",
    "        task.period = max(5, task.period + random.randint(-
2, 2))\n",
    "    elif choice == 'priority':\n",
```

```
    "        task.priority = max(1, min(5, task.priority +
random.choice([-1, 1])))\n",
    "    elif choice == 'duration':\n",
    "        task.duration = max(1, task.duration +
random.choice([-1, 1]))\n",
    "\n",
    "def fitness(tasks, cpu, max_time):\n",
    "    total_energy = sum(t.total_energy for t in tasks)\n",
    "    utilization = cpu.load / max_time\n",
    "    fairness = 1 / (1 + max(abs(t.executed_time - (max_time
/ t.period) * t.duration) for t in tasks))\n",
    "    return -total_energy + 2 * utilization + fairness\n",
    "\n",
    "def optimize_system(base_tasks, generations=15,
pop_size=10, max_time=50):\n",
    "    best_tasks = None\n",
    "    best_score = float('-inf')\n",
    "    for _ in range(generations):\n",
    "        population = []\n",
    "        for _ in range(pop_size):\n",
    "            new_tasks = copy.deepcopy(base_tasks)\n",
    "            for task in new_tasks:\n",
    "                mutate(task)\n",
    "            population.append(new_tasks)\n",
    "        for tasks in population:\n",
    "            cpu = CPU()\n",
    "            scheduler = GreedyScheduler(tasks)\n",
    "            scheduler.schedule(cpu, max_time)\n",
    "            score = fitness(tasks, cpu, max_time)\n",
    "            if score > best_score:\n",
    "                best_score = score\n",
    "                best_tasks = copy.deepcopy(tasks)\n",
    "    return best_tasks\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "2653db37",
   "metadata": {},
   "outputs": [],
   "source": [
    "import matplotlib.pyplot as plt\n",
    "\n",
    "def plot_execution_timeline(tasks):\n",
    "    for task in tasks:\n",
    "        times = [t[0] for t in task.log]\n",
    "        durations = [t[1] for t in task.log]\n",
    "        plt.step(times, durations, where='post',
label=task.name)\n",
    "    plt.xlabel(\"Час\")\n",
    "    plt.ylabel(\"Тривалість задачі\")\n",
    "    plt.title(\"Виконання задач у часі\")\n",
```

```
    "    plt.legend()\n",
    "    plt.grid(True)\n",
    "    plt.show()\n",
    "\n",
    "def plot_energy(tasks):\n",
    "    names = [t.name for t in tasks]\n",
    "    energies = [t.total_energy for t in tasks]\n",
    "    plt.bar(names, energies)\n",
    "    plt.title(\"Сумарне енергоспоживання по задачах\")\n",
    "    plt.ylabel(\"Енергія\")\n",
    "    plt.grid(True)\n",
    "    plt.show()\n",
    "\n",
    "def plot_frequencies(tasks, max_time):\n",
    "    names = [t.name for t in tasks]\n",
    "    frequencies = [len(t.log) / (max_time / 10) for t in
tasks]\n",
    "    plt.bar(names, frequencies)\n",
    "    plt.title(\"Частота запусків задач\")\n",
    "    plt.ylabel(\"Запусків на 10 одиниць часу\")\n",
    "    plt.grid(True)\n",
    "    plt.show()\n",
    "\n",
    "def plot_cpu_utilization(cpu, max_time):\n",
    "    busy_time = cpu.load\n",
    "    idle_time = max_time - busy_time\n",
    "    labels = ['Зайнятий', 'Вільний']\n",
    "    sizes = [busy_time, idle_time]\n",
    "    plt.pie(sizes, labels=labels, autopct='%1.1f%%',
startangle=90)\n",
    "    plt.title(\"Завантаження CPU\")\n",
    "    plt.axis('equal')\n",
    "    plt.show()\n",
    "\n",
    "def plot_cumulative_energy(tasks):\n",
    "    for task in tasks:\n",
    "        cumulative = []\n",
    "        total = 0\n",
    "        for t in task.log:\n",
    "            total += task.power * t[1]\n",
    "            cumulative.append(total)\n",
    "        plt.plot(cumulative, label=task.name)\n",
    "    plt.title(\"Кумулятивне енергоспоживання\")\n",
    "    plt.ylabel(\"Енергія\")\n",
    "    plt.xlabel(\"Ітерація запуску\")\n",
    "    plt.legend()\n",
    "    plt.grid(True)\n",
    "    plt.show()\n",
    "\n",
    "def plot_task_histogram(tasks):\n",
    "    all_times = []\n",
    "    for task in tasks:\n",
    "        all_times.extend([t[0] for t in task.log])\n",
```

```
    "    plt.hist(all_times, bins=20, edgecolor='black')\n",
    "    plt.title(\"Гістограма запусків задач\")\n",
    "    plt.xlabel(\"Час\")\n",
    "    plt.ylabel(\"Кількість запусків\")\n",
    "    plt.grid(True)\n",
    "    plt.show()\n",
    "\n",
    "def plot_priority_distribution(tasks):\n",
    "    names = [t.name for t in tasks]\n",
    "    priorities = [t.priority for t in tasks]\n",
    "    plt.bar(names, priorities, color='orange')\n",
    "    plt.title(\"Розподіл пріоритетів задач\")\n",
    "    plt.ylabel(\"Пріоритет\")\n",
    "    plt.grid(True)\n",
    "    plt.show()\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "78fb2d9c",
   "metadata": {},
   "outputs": [],
   "source": [
    "tasks = [\n",
    "    Task(\"Sensor\", period=10, duration=2, priority=1,
power=0.5, memory=128),\n",
    "    Task(\"Compute\", period=20, duration=4, priority=2,
power=0.8, memory=256),\n",
    "    Task(\"Comm\", period=15, duration=3, priority=3,
power=0.6, memory=192)\n",
    "]\n",
    "\n",
    "original_tasks = copy.deepcopy(tasks)\n",
    "\n",
    "# До оптимізації\n",
    "cpu_orig = CPU()\n",
    "GreedyScheduler(original_tasks).schedule(cpu_orig,
max_time=50)\n",
    "\n",
    "# Оптимізація\n",
    "optimized_tasks = optimize_system(tasks)\n",
    "cpu = CPU()\n",
    "GreedyScheduler(optimized_tasks).schedule(cpu,
max_time=50)\n",
    "\n",
    "# Вивід результатів\n",
    "for t in optimized_tasks:\n",
    "    print(f\"{t.name}: пріоритет = {t.priority}, період =
{t.period}, енергія = {t.total_energy:.2f}\")\n"
   ]
  },
  {
```

```
   "cell_type": "code",
   "execution_count": null,
   "id": "21b5097e",
   "metadata": {},
   "outputs": [],
   "source": [
    "def plot_energy_comparison(before_tasks, after_tasks):\n",
    "    names = [t.name for t in before_tasks]\n",
    "    before = [t.total_energy for t in before_tasks]\n",
    "    after = [t.total_energy for t in after_tasks]\n",
    "\n",
    "    x = range(len(names))\n",
    "    plt.figure()\n",
    "    plt.bar([i - 0.2 for i in x], before, width=0.4,
label='До оптимізації')\n",
    "    plt.bar([i + 0.2 for i in x], after, width=0.4,
label='Після оптимізації')\n",
    "    plt.xticks(x, names)\n",
    "    plt.title(\"Енергоспоживання: до і після
оптимізації\")\n",
    "    plt.ylabel(\"Енергія\")\n",
    "    plt.legend()\n",
    "    plt.grid(True)\n",
    "    plt.show()\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "f329e3be",
   "metadata": {},
   "outputs": [],
   "source": [
    "# Порівняльний аналіз\n",
    "plot_energy_comparison(original_tasks, optimized_tasks)\n"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "id": "ee21b482",
   "metadata": {},
   "outputs": [],
   "source": [
    "plot_execution_timeline(optimized_tasks)\n",
    "plot_energy(optimized_tasks)\n",
    "plot_frequencies(optimized_tasks, max_time=50)\n",
    "plot_cpu_utilization(cpu, max_time=50)\n",
    "plot_cumulative_energy(optimized_tasks)\n",
    "plot_task_histogram(optimized_tasks)\n",
    "plot_priority_distribution(optimized_tasks)\n"
   ]
  }
```

```
 ],
 "metadata": {},
 "nbformat": 4,
 "nbformat_minor": 5
}
```