Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Галузь знань 12 – Інформаційні технології

Спеціальність 123 –Комп'ютерна інженерія

на тему «Метод оптимізації завдань у багатопроцесорних системах»

КвРКІП. 2302183.23.02.19 ПЗ

Виконав: студент 2 курсу, група КІ2м-23-2 _____ Дмитро МАРТИНЮК
Підпис Ім'я, прізвище

Керівник к.е.н., доцент _____ Світлана САЧЕНКО
Науковий ступінь, вчене звання Підпис Ім'я, прізвище

До захисту допускаю:
Зав. кафедри КІІС, доктор філософії, доцент
Ольга ПАВЛОВА
29 04 2025 р.

Хмельницький, 2025

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Освітній рівень МАГІСТР

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма ОСВІТНЬО-НАУКОВА ПРОГРАМА «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ

Зав. кафедри Ольга ПАВЛОВА

" 01 " 09 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Дмитро МАРТИНЮК
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод оптимізації завдань у багатопроцесорних системах
Керівник проекту (роботи) Світлана САЧЕНКО, к.е.н., доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 08.01.2025 №8

2. Строк подання студентом проекту (роботи) на кафедру 01.05.2025 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

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

Аналіз відомих методів та засобів забезпечення метод оптимізації завдань у багатопроцесорних системах

Процес оптимізації багатопроцесорних вбудованих систем

Міграції процесів в багатозадачних вбудованих системах

Метод оптимізації завдань в багатопроцесорних вбудованих системах

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6. Консультанти розділів кваліфікаційної роботи магістра

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Сергій ЛИСЕНКО, професор кафедри КІІС		
Антиплагіат	Андрій НІЧЕПОРУК, доцент кафедри КІІС		

7. Дата видачі завдання «_01_»_____09_____2024р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) кваліфікаційної роботи магістра	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики КвРМ з керівником	01.09.2024	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	01.10.2024	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	01.11.2024	виконано
4	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	01.12.2024	виконано
5	Робота над науковою статтею	01.02.2025	виконано
6	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	15.02.2025	виконано
7	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	01.04.2025	виконано
8	Оформлення пояснювальної записки згідно вимог	18.04.2025	виконано
9	Попередній захист ДРМ	29.04.2025	виконано
10	Захист ДРМ на засіданні ЕК	До 15.05.2025	

Студент _____ Дмитро МАРТИНЮК
Підпис Ім'я, прізвище

Керівник роботи _____ Світлана САЧЕНКО
Підпис Ім'я, прізвище

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: «Метод оптимізації завдань у багатопроцесорних системах»

Автор роботи: Мартинюк Дмитро Васильович

Керівник роботи: Саченко С.І.

Пояснювальна записка: 71 с., 5 рис., 4 дод., 84 джерела.

ПЕРЕЛІК КЛЮЧОВИХ СЛІВ: багатопроцесорні системи, вбудовані системи, комп'ютерні системи, оптимізація, пристрої IoT.

Об'єктом дослідження є процес оптимізації завдань у багатопроцесорних системах.

Предметом дослідження є методи та засоби забезпечення оптимізації завдань у багатопроцесорних системах.

Метою кваліфікаційної роботи магістра є покращення ефективності оптимізації завдань у багатопроцесорних системах.

Для розв'язання поставлених задач використовувалися методи оптимізації, теорія комп'ютерних систем.

Наукова новизна отриманих результатів:

- розроблено новий метод оптимізації завдань у багатопроцесорних вбудованих системах, особливістю якого є використання реплікації та цільової функції при міграції завдань.

На основі проведених досліджень розроблено метод оптимізації завдань у багатопроцесорних вбудованих системах.

Практична значимість отриманих результатів полягає у розроблені схеми оптимізації завдань у багатопроцесорних вбудованих системах.

У вступі подано об'єкт та предмет дослідження, мету, наукову новизну та практичну цінність роботи, а також характеристику структури роботи.

У першому розділі проведено аналіз відомих рішень щодо оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

У другому розділі здійснено розроблення цільовихї функцій оптимізації завдань у багатопроцесорних вбудованих системах.

У третьому розділі розроблено розроблення алгоритми оптимізації завдань за різними параметрами у багатопроцесорних вбудованих системах.

У четвертому розділі здійснено розроблення методу оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах, розроблено програму, проведено експеримент та моделювання схеми оптимізації.

У висновках підведено підсумки досягнення результатів з розв’язання завдань дослідження.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 5

ВСТУП 6

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ТА ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ МЕТОД ОПТИМІЗАЦІЇ ЗАВДАНЬ У БАГАТОПРОЦЕСОРНИХ СИСТЕМАХ 8

1.1 Огляд та поняття схем оптимізації завдань у багатопроцесорних системах ……8

1.2 Методи оптимізації процесів в багатопроцесорних вбудованих системах14

1.3 Постановка задачі 21

1.4 Висновки до першого розділу 21

2 ПРОЦЕС ОПТИМІЗАЦІЇ БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМ 22

2.1 Моделі потоків даних багатопроцесорних вбудованих систем 22

2.2 Цільова функція оптимізації завдань у багатопроцесорних системах з міграцією 35

2.3 Висновки до другого розділу 42

3 МІГРАЦІЇ ПРОЦЕСІВ В БАГАТОЗАДАЧНИХ ВБУДОВАНИХ СИСТЕМАХ 44

3.1 Алгоритми міграції процесів в багатозадачних вбудованих системах 44

3.2 Схеми міграції процесів в багатозадачних вбудованих системах 54

3.3 Висновки до третього розділу 63

4 МЕТОД ОПТИМІЗАЦІЇ ЗАВДАНЬ В БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМАХ 64

4.1 Метод оптимізації завдань у вбудованих системах з багатьма процесорами 64

4.2 Дослідження ефективності методу оптимізації завдань в багатопроцесорних вбудованих системх 68

4.3 Висновки до четвертого розділу..... ..76

ВИСНОВКИ 77

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 78

ДОДАТКИ……88

ДОДАТОК А Презентація роботи…….88

ДОДАТОК Б Наукова праця здобувача……97

ДОДАТОК В Результати перевірки на плагіат……108

ДОДАТОК Г Програмний код……..109

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

IoT	Інтернет речей
IT	Інформаційні технології
ОС	Операційні системи

ВСТУП

Оптимізація завдань у багатопроцесорних вбудованих системах набуває особливого значення в контексті розроблення вбудованих пристроїв, що використовуються в різних сферах. Однак, попри значний прогрес, залишається низка дослідницьких проблем, які потребують глибокого аналізу та ефективних рішень.

Однією з ключових проблем є забезпечення надійності вбудованих систем, оскільки вони часто працюють в умовах, критичних для безпеки. Незважаючи на чітко визначений набір функціональних можливостей, які відомі на момент проектування, гарантування безперервної роботи в умовах реального часу залишається викликом. Необхідно забезпечити не лише правильність результатів, а й їхню своєчасність, що вимагає розробки нових підходів до управління ресурсами багатопроцесорних систем. Ще однією проблемою є необхідність підтримки жорстких гарантій у режимі реального часу. Це відрізняє вбудовані системи від систем загального призначення (наприклад, персональних комп'ютерів), які є гнучкішими щодо функціональності, але не забезпечують такого рівня надійності та передбачуваності. Оптимізація процесів планування завдань, що враховує ці особливості, потребує подальших досліджень. Важливо також відзначити, що вбудовані системи можна розділити на системи управління та потокові системи, які мають різні вимоги до обробки даних. Системи управління реагують на вхідні події з навколишнього середовища, що вимагає швидкого реагування та мінімізації затримок. Потокові системи, навпаки, працюють з безперервними потоками даних, що потребує високої пропускної здатності та ефективної обробки великих обсягів інформації. Розробка універсальних підходів, які могли б забезпечити оптимальну продуктивність для обох типів систем, є актуальним завданням для науковців та інженерів.

Отже, дослідження в області оптимізації завдань у багатопроцесорних системах має значний потенціал для розвитку, особливо у контексті вбудованих пристроїв. Подальші зусилля мають бути спрямовані на вирішення проблем надійності, гарантій реального часу та ефективного управління ресурсами, що дозволить створювати

безпечніші та продуктивніші системи.

Метою кваліфікаційної роботи магістра є покращення ефективності оптимізації завдань у багатопроцесорних системах.

Поставлена мета досягається розв’язанням таких основних завдань:

- проаналізувати відомі методи забезпечення функціонування систем з IoT та захищені протоколи для зв’язку між пристроями IoT;

- розробити удосконалення методу забезпечення функціонування систем з IoT із захищеним протоколом;

- здійснити реалізацію протоколу згідно розробленого методу забезпечення функціонування систем з IoT із захищеним протоколом;

- здійснити дослідження захищеного протоколу.

Об’єктом дослідження є процес оптимізації завдань у багатопроцесорних системах.

Предметом дослідження є методи та засоби забезпечення оптимізації завдань у багатопроцесорних системах.

Наукова новизна отриманих результатів:

- розроблено новий метод оптимізації завдань у багатопроцесорних вбудованих системах, особливістю якого є використання реплікації та цільової функції при міграції завдань.

На основі проведених досліджень розроблено метод оптимізації завдань у багатопроцесорних вбудованих системах.

Практична значимість отриманих результатів полягає у розроблені схеми оптимізації завдань у багатопроцесорних вбудованих системах.

Для розв’язання поставлених задач використовувалися методи оптимізації, теорія комп’ютерних систем.

За темою кваліфікаційної роботи опубліковано одну наукову статтю в науковому журналі категорії Б [84] (Dmytro Martiniuk, Oleksii Lyhun, Andriy Drozd, Oleksii Besedovskyi. TASK OPTIMISATION IN MULTIPROCESSOR EMBEDDED SYSTEMS. *Computer Systems and Information Technologies*, 2025. №1.).

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ТА ЗАСОБІВ ЗАБЕЗПЕЧЕННЯ МЕТОД ОПТИМІЗАЦІЇ ЗАВДАНЬ У БАГАТОПРОЦЕСОРНИХ СИСТЕМАХ

1.1 Огляд та поняття схем оптимізації завдань у багатопроцесорних системах

Оптимізація завдань у багатопроцесорних системах [1, 2], враховуючи застосування комп’ютерних систем в різних сферах, набула на сьогодні особливого значення в контексті розроблення вбудованих пристроїв. Більшість вбудованих систем мають такі характеристики: вони призначені для реалізації чітко визначеного набору функціональних можливостей, які відомі на момент проектування; вони повинні бути надійними, оскільки часто працюють в умовах, критичних для безпеки; вони повинні надавати жорсткі гарантії в режимі реального часу, тобто їх вихід повинен бути правильним, а також виробленим в певні терміни. Ці характеристики [3, 4] відрізняють вбудовані системи від систем загального призначення, таких як персональні комп'ютери, які демонструють набагато більшу гнучкість з точки зору функціональності та набагато менше фокусування на гарантії та надійності у режимі реального часу.

Вбудовані системи можна розділити на дві категорії [5, 6], в залежності від типу функціоналу, який вони забезпечують: системи управління чекають вхідних подій (або сигналів) з навколишнього середовища і реагують на ці події відповідним чином, тобто ці системи використовуються, наприклад, в промисловій автоматизації; потокові системи обробляють безперервний, можливо, нескінченний потік даних з навколишнього середовища, тобто ці системи знаходять застосування, наприклад, в обробці аудіо та відео.

Враховуючи складність [7] оптимізації завдань у багатопроцесорних системах будемо розглядати як об’єкт дослідження вбудовані потокові системи. Приклади програм для потокового передавання включають кодування та декодування аудіо/відео, оброблення сигналів, комп'ютерний зір, медичну візуалізацію, навігаційні системи, системи камер безпеки та багато інших.

Складні вбудовані системи [8, 9], такі як ті, що керують автономним водінням

автомобілів, насправді складаються з кількох підсистем, які взаємодіють одна з одною. У прикладі з автономним водінням автомобілів частина цих підсистем відноситься до категорії потокових систем, а інша частина - до категорії управління. Наприклад, автомобілі з автономним керуванням збирають велику кількість даних, які надходять у вигляді безперервних потоків з камер і лазерних датчиків, встановлених на самому автомобілі. Ці потоки даних повинні постійно уточнюватися та оброблятися для виконання планування руху, тобто визначення оптимального шляху та швидкості, за якою повинен слідувати транспортний засіб, та запобігання зіткнень, тобто виявлення та уникнення несподіваних перешкод, що наближаються. Ці рішення приймаються потоковими підсистемами [10] і передаються іншим підсистемам, типу управління, які змушують автомобіль фактично керувати, гальмувати або прискорюватися.

Системи, що керують автономними автомобілями, реалізують алгоритми планування руху та уникнення зіткнень [11], які мають надзвичайно високу взаємодію. Крім того, ці алгоритми повинні виробляти свої результати за короткий і передбачуваний час, щоб автомобіль міг швидко відреагувати на зовнішні події. Наприклад, людина раптово переходить вулицю перед автомобілем. Автомобіль повинен зупинитися якомога швидше. Висока складність реалізованих алгоритмів і вимога короткого часу виконання ставлять перед проектувальниками завдання домогтися високої продуктивності системи. Ця вимога є спільною для багатьох сучасних вбудованих систем. Фактично, протягом багатьох років вбудовані системи демонстрували постійну потребу в зростаючій продуктивності.

До середини 2000-х років більшість обчислювальних систем були реалізовані як однопроцесорні архітектури [12], і вищезазначена потреба в зростанні продуктивності задовольнялася за рахунок збільшення обчислювальної потужності самого одного процесора. Однак зростання продуктивності між наступними поколіннями однопроцесорів призвело до значного уповільнення на початку 2000-х, головним чином через [13, 14]: зменшення віддачі від нових рішень архітектури процесорів; дуже повільне збільшення тактової частоти між поколіннями процесора через проблеми з витоком живлення; зростаюча різниця у швидкостях між

процесором і пам'яттю. Тому, щоб ще більше підвищити продуктивність системи, виробники мікросхем з середини 2000-х років перенесли свої зусилля з досліджень і розробок на багатопроцесорні архітектури. Це технологічна тенденція [15, 16], яка вплинула як на загальне призначення, так і на вбудовані системи, є перспективною. Власне, все більше архітектури, запропонованої як науково-дослідними інститутами, так і промисловістю, показують все більшу кількість елементів обробки. Фактично, в даний час розробники вбудованих систем часто інтегрують в один чіп кілька процесорів, пам'яті, взаємозв'язків та інших апаратних периферійних пристроїв, утворюючи так звану багатопроцесорну систему на кристалі.

Однією з часто використовуваних схем для організації багатопроцесорних обчислень є схема [17, 18] з мікропроцесором загального призначення, який поєднується з декількома мультимедійними співпроцесорами.

Тому, розвиток та, відповідно, дослідження вбудованих систем і поява MPSoC [19] у вбудованій сфері є перспективним. Сучасні вбудовані MPSoC вимагають дедалі вищої продуктивності. Збільшення продуктивності, також, зросла зі складністю сучасних вбудованих MPSoC. На рис. 1.1 зображено архітектуру MPSoC [19].

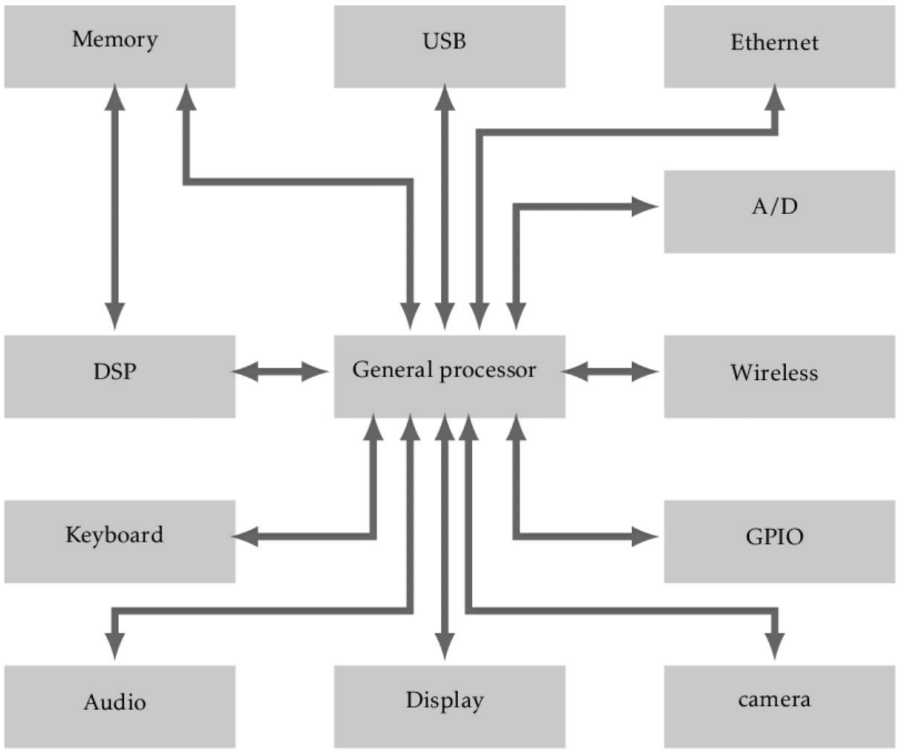


Рисунок 1.1 – Архітектура MPSoC [19]

З метою підвищення продуктивності програмного забезпечення комп’ютерні технології еволюціонують у напрямку багатоядерних платформ, що містять безліч процесорних ядер. Проте ефективне виконання паралельних програм на таких архітектурах залишається складним завданням [19]. Велика кількість доступних обчислювальних ресурсів потребує організації взаємодії між ядрами, синхронізації їхньої роботи та коректного виконання коду. Передача даних між процесорами може відбуватися автоматично завдяки програмному забезпеченню або бути контрольованою безпосередньо розробниками. Синхронізація передбачає визначення правильного моменту початку обчислень, щоб завдання виконувалися лише після завершення всіх необхідних попередніх операцій. Для ефективного виконання алгоритмів необхідно визначити, який процесор відповідатиме за конкретні завдання та в які моменти їх слід запускати, тобто здійснювати планування. Однак ці процеси є складними комбінаторними задачами, оскільки кількість можливих варіантів стрімко зростає в міру ускладнення системи.

Крім того, оцінювання рішень [19, 20] базується на низці критеріїв, які необхідно оптимізувати, зокрема споживанні пам’яті, часі виконання, витратах електроенергії та ефективності використання ресурсів. Ці аспекти формують багатокритеріальну оптимізаційну проблему [21, 22], для якої немає єдиного оптимального рішення. Замість цього існує набір компромісних варіантів [23, 24].

До класу застосунків, що працюють із безперервними потоками даних, належать так звані Streaming Applications [19]. Вони виконують однотипні обчислення над різними фрагментами інформації. Такі програми зручно моделювати за допомогою підходу, заснованого на потоках даних.

Для вирішення цієї проблеми формулюємо її у вигляді логічних обмежень та представляємо через теорію модулю задоволеності [25, 26]. Спеціальні розв’язувачі застосовують методи пошуку та поширення обмежень для знаходження відповідних варіантів виконання завдань відповідно до заданих умов. Тому, потрібно розробити підходи до зменшення обсягу обчислень, щоб врахувати симетрії у моделях, що повинно дозволити [27, 28] значно збільшити розмір задач, які можна ефективно розв’язати.

Таким чином, актуальним є розроблення алгоритму дослідження простору архітектури з рис. 1.1, який наближує проблему рішення з різними компромісами щодо витрат [19]. Перевірку цих рішень можна виконати, якщо запускати їх на реальній багатоядерній платформі. Цю задачу розширюють [29, 30] як задачу планування на багатоядерні системи, які складаються з кластерів багатоядерних процесорів, з'єднаних через мережу. Може бути варіант рішення [31, 32], при якому виконується розподіл застосунків на таких платформах і моделюються їхні комунікаційні взаємодії. Крім того, при правильному моделюванні потоків можливо ефективно організувати обмін даними навіть у системах з обмеженою пам'яттю.

Загалом, дані застосунків [33] зазвичай не вміщуються в локальну пам'ять. Тому, аналізують клас паралельних застосунків, які мають типову схему доступу до даних і оперують значними обсягами інформації, рівномірно розподіленої між обчисленнями. Подібні сценарії часто зустрічаються у завданнях з обробки зображень.

Обробка даних передбачає їхнє завантаження з основної пам'яті у локальну пам'ять, подальше виконання обчислень і запис отриманих результатів назад в основну пам'ять. Ці операції виконуються пакетно, а вибір оптимального розміру таких пакетів є задачею оптимізації. Формалізуємо цю проблему для визначення оптимальної стратегії обміну даними, яка враховує характеристики як застосунку, так і апаратної платформи [34].

Крім того, виконання застосунків може динамічно починатися та зупинятися. Для ефективного функціонування всіх процесів та оптимізації глобальних показників, таких як споживання енергії чи час виконання, необхідно забезпечити налаштовуваність програм у процесі їх роботи [35]. Тому, перспективним є підхід, який дозволяє передбачуване та незалежне переміщення завдань під час виконання відповідно до обраної конфігурації, мінімізуючи їхній вплив на інші процеси.

Оскільки складність MPSoC постійно зростає, то в даний час проектування цих систем повинно виконуватися на правильному рівні абстракції. Зокрема, проектування на рівні воріт і регістра-передачі більше не є ефективним [36]. Більш високий рівень абстракції, а саме системний, необхідний для проектування сучасних

вбудованих MPSoC [37]. Як показано на рис. 1.1, на системному рівні проектувальники розробляють систему, задаючи модель платформи виконання, модель застосунку та відображення застосунку на платформу виконання [19]. Зокрема, модель платформи виконання описуватиме тип і кількість процесорів, доступних у системі, а також які типи пам'яті та взаємозв'язків присутні. Застосунок повинен [38, 39] моделюватись у вигляді набору завдань, які можуть бути розподілені між декількома процесорами. Крім того, модель застосування описуватиме, як ці завдання взаємодіють і синхронізуються [40] між собою. Нарешті, відображення визначатиме, як модель застосунку зіставляється з моделлю платформи виконання. Наприклад, відображення описує як завдання розподіляються між процесорами платформи виконання і як заплановано кілька завдань, відображених на одному процесорі, а також як примітиви зв'язку, що використовуються в прикладній моделі, перетворюються на відповідні примітиви платформи виконання. Потім, коли вказано програму, то платформу виконання та відображення, інструмент синтезу на рівні електронної системи генерує автоматизованим способом детальний опис апаратного забезпечення та програмне забезпечення, що працює на кожному процесорі системи.

В загальному випадку [41], для досягнення високої продуктивності моделі платформи виконання і застосунку повинні бути тісно пов'язані між собою. Наприклад, архітектура фон Неймана ідеально відповідає програмі, заданій за допомогою послідовної програми. Наприклад, з використанням мови програмування C. Якщо продуктивність системи не відповідає вимогам, то розробникам доводиться модифікувати платформу виконання, програмне забезпечення та/або специфікацію відображення, щоб покращити досягнутий рівень продуктивності.

Оскільки вбудовані системи [42] в даний час переходять від однопроцесорних до мультипроцесорних, то потрібно внести ряд змін в методологію проектування. Зокрема, дві основні проблеми, з якими стикаються розробники, таку: підходи до моделювання застосунків для того, щоб можна було використовувати численні ресурси обробки, доступні на сучасних платформах виконання; підключення процесорів в платформі виконання складно зробити, оскільки кількість процесорів у системах постійно збільшується.

Програмні або апаратні проблеми [43] у великих центрах обробки даних, як правило, призначаються різним експертам для вирішення. Дуже важливо розробити багатоцільні алгоритми планування, щоб максимізувати загальний ступінь відповідності та мінімізувати загальний час потоку.

Таким чином, розроблення нових підходів для схем оптимізації процесів в багатопроцесорних вбудованих системах є перспективним напрямом дослідження. Потребують розроблення алгоритми планування, щоб максимізувати загальний ступінь відповідності та мінімізувати загальний час потоку.

1.2 Методи оптимізації процесів в багатопроцесорних вбудованих системах

Більшість існуючих методів [44, 45] проблеми із призначенням розглядають лише єдину мету, тоді як деякі методи оптимізації не мають однакових цілей [46]. Методи багатоцільної оптимізації для проблеми планування завдань у розподілених системах подано в роботі [47]. Вивчення ефективності багатоцільної оптимізації можна ще більше покращити. Наприклад, в роботі [48] пропонується багатоцільний евристичний алгоритм для планування, який є поєднанням генетичного алгоритму та багатоцільового алгоритму [48]. Опитування інтелектуального управління сповіщеннями та інцидентами в IT -послугах подано в роботі [49]. В роботі [50] розглянуто системи онлайн-сервісу, до функціонування яких є вимоги щодо точних та швидких визначень та вирішень збоїв у своїх складних системах, і тому виникає рішення щодо локалізації несправностей у коді. Однак попередні моделі локалізації несправностей страждають від низької точності локалізації та поганої інтерпретації через складні залежності від характеристик несправностей у промисловій практиці. Щоб вирішити це питання, виклики, що виникають залежно від великих відстаней серед функцій несправностей та незбалансованого розподілу знань про несправності, а також покращення інтерпретації моделі, розглядають модель локалізації несправностей в онлайн-системах обслуговування, більш діючою та інтерпретаційною. Зокрема, вона складається з двох компонентів: компонента кодування функції; компонент локалізації несправностей. Компонент кодування

функції використовує графічні мережі уваги для зйомки складних просторово-часових залежностей у межах несправностей. Потім компонент локалізації несправностей застосовує триступеневий підхід, використовуючи механізм багатовії для виявлення та визначення пріоритетності найбільш релевантних функцій несправностей для точної локалізації. Крім того, модуль збалансування знань про несправності, який він містить, вводить зважену функцію втрати розбіжності, щоб переконатися, що модель приділяє належну увагу всім функціям несправностей, вирішенням питання незбалансованого розподілу знань про несправності та підвищення ефективності локалізації.

Методи багатоцільної оптимізації для проблеми планування завдань у розподілених системах, також, можуть використовувати механізм попередження [51], який є важливим інструментом для виявлення аномалій та забезпечення плавної роботи онлайн-систем обслуговування, негайно повідомляючи інженерів про потенційні проблеми. Однак зростаюча масштаб і складність IT -інфраструктури часто призводить до суттєвих проблем під час системних збоїв з потоком часто корельованих сповіщень. Тому ефективна агрегація попередження має вирішальне значення для виділення першопричин та розростаючій роздільній здатності відмови. Існуючі підходи [52, 53], як правило, покладаються на семантичну схожість, або на статистичні методи, які мають значні обмеження, такі як ігнорування причинних відносин або боротьба з проблемами нечастого сповіщення. Щоб подолати ці недоліки [54], використовують підхід до агрегації двофазного сповіщення. Тобто, використовують часову просторову кластеризацію [55] для групування сповіщень на основі їх тимчасової близькості та просторових ознак. На другому етапі використовують великі мовні моделі для простеження каскадних наслідків збоїв у обслуговуванні та сукупних сповіщень, які мають однакову першопричину.

Для того, щоб вивчити проблему планування, дослідження спочатку повинні проводитись з точки зору планування одиного головного процесора [56]. Таким чином, буде встановлено модель для послідовності процесорів, яка спочатку заснована на кластері. Маршрут другої стратегії полягає в тому, щоб мінімізувати загальну тривалість завдання. Проблеми з розташуванням та розкладом, що беруть

участь у операціях, зазвичай вирішуються окремо від розкладу точок пропозиції до точок попиту, а також планування послідовності активності в точках попиту та пропозиції. Тому, автори роботи [57] оптимізували розташування процесорів, графік точок пропозиції в областях попиту та маршрутів на основі послідовності активності у зонах попиту на місці та запропонували модель бінарного цілого.

Дослідження в роботі [58] стосувалось динаміки і виявило, що це суттєво впливає на час, витрати, продуктивність та безпеку. Проблему досліджень не можна розглядати як лише односторонні фактори, тому було запропоновано внести ідею динамічного програмування в дослідження та запропоновано модель оптимізації для послідовності змінної амплітуди. Ця модель була порівняна з трьома звичайними стратегіями вдосконалення. Дослідження виявили, що моделі, які враховують динамічне програмування, ефективно скорочують загальний час та покращують швидкість використання. Можливості оптимізації безпосередньо впливають на продуктивність та час виконання завдань. Тому необхідно ефективно покращити швидкість використання процесорів з часом [59].

Проектування вбудованого MPSoC [60] - це процес, який включає в себе кілька етапів. Процес починається з визначення необхідної функціональності системи за допомогою моделі програми разом із специфікацією платформи виконання, на якій буде працювати програма. Після ряду етапів доопрацювання процес проектування завершується, коли отримується детальний опис апаратного забезпечення системи і програмного забезпечення, що працює на кожному процесорі. Тенденції проектування такі як широке впровадження багатопроцесорної архітектури і масштабованих взаємоз'єднань представляють нові методології проектування, спрямовані на досягнення високої продуктивності системи на багатопроцесорній архітектурі. Забезпечення високої продуктивності системи, однак, не є єдиною метою розробників вбудованих систем [61].

Під терміном «адаптивність системи» [62, 63] будемо визначати здатність системи пристосовуватися до мінливих умов, що нав'язуються навколишнім середовищем. Ці умови представлені параметрами, які можна розділити на два класи [64, 65]. Параметри, що належать застосунку впливають на спосіб виконання

програми [66, 67]. Наприклад, роздільна здатність програми для декодування відео зазвичай представлена двома параметрами, які визначають висоту та ширину кадрів. Параметри, що описують стан платформи виконання. Наприклад, параметр може визначати кількість активних процесорів у системі. Досягнення адаптивності системи у відповідь на зміну другого набору параметрів, тих, які описують стан платформи виконання є перспективним.

Система живиться від акумулятору, а заряд батареї закінчується. Користувач може вирішити відключити певну кількість елементів, щоб зменшити енергоспоживання системи. Це може призвести до зниження якості обслуговування програми. Наприклад, знижена швидкість кодування відео. Крім того, цей сценарій вимагає, щоб система перенесла завдання, які виконуються і які будуть вимкнені та які будуть залишатися активними, тобто змінити відображення завдань під час виконання.

Адаптивність системи може бути реалізована в обчислювальній системі по-різному. У випадку вбудованих мультипроцесорів програми як правило заплановані в системі операційною системою, представленою середнім шаром. ОС виступає в якості інтерфейсу між прикладним рівнем і апаратним шаром, який лежить в нижній частині стека [68, 69].

Розглянемо підходи, що забезпечують адаптивність системи на стику між прикладним рівнем і рівнем ОС. У цьому контексті в роботах [70, 71] вже існують підходи, що забезпечують адаптивність системи. Більшість з них дозволяють, в меншій або більшій мірі, змінювати відображення завдань програми під час виконання, тому вони дозволяють міграцію завдань. Міграція завдань є важливою вимогою для гарантії придатності системи. Однак існуючі рішення мають недоліки або в ступені підтримки адаптивності системи, або в масштабованості запропонованих підходів.

Ряд інших робіт [72, 73] націлені на адаптивність системи, враховуючи, що параметри застосунків можуть змінюватися. Вони дозволяють змінювати параметри безпосередньо на рівні застосунків. Для цього вони використовують адаптивні схеми з процесорами, які можуть моделювати можливість зміни параметрів застосунків під

час виконання. Зміна параметрів програми під час виконання також називається зміною режиму. В деяких випадках застосунки просто не виявляють властивої алгоритмічної адаптивності, тобто їх параметри фіксовані і не змінюються. Потрібно враховувати також випадки, коли потрібна адаптивність системи у відповідь на несправності, виявлені на платформі виконання [63, 66]. Ці випадки не можуть бути виражені за допомогою адаптивного процесору. Якщо для моделювання застосунків використовується адаптивний процесор, то методи можуть бути використані як спосіб зміни відображення завдань під час виконання, а отже, до зміни режиму.

Розглянемо використання міграції завдань для досягнення адаптивності системи в системах з найкращими зусиллями [74, 75]. Для реалізації схеми потрібен стек з проміжним рівнем, який називається проміжне програмне забезпечення (middleware). Цей рівень залишається між програмами, які вказано як процеси і базовою ОС. Шар проміжного програмного забезпечення спрямований на забезпечення адаптивності в системах. Стек розгортається на кожному процесорі апаратної платформи. Рівень проміжного програмного забезпечення складається з двох компонентів, які є взаємозалежними. Перший компонент розглядає проблему зв'язку на платформах на з розподіленою пам'яттю. Точніше, він перетворює примітиви зв'язку у відповідні примітиви платформи виконання. Є кілька підходів для ефективного впровадження комунікації. Запропоновані підходи розрізняються за ступенем необхідної синхронізації. Однак всі ці підходи дозволяють процесу обмінюватися даними незалежно від фактичного просторового відображення процесів на процесори, тобто вони є незалежними від відображення. Це фундаментальна вимога для підтримки функціональності системи у разі міграції завдань/процесів. Другий компонент рівня проміжного програмного забезпечення реалізує механізм міграції. Механізм міграції процесів відповідає наступним вимогам. Перша вимога полягає в тому, що міграція процесів, одного разу запущена, повинна бути завершена протягом певного відомого періоду часу. Її називають властивістю передбачуваності. Друга вимога полягає в тому, що міграція завдань може бути запущена в системі в будь-який момент. Третя вимога полягає в тому, що код, необхідний для міграції завдань, повинен бути згенерований автоматизованим

способом, без ручного втручання розробника.

Використання напіврозділених підходів у жорсткому плануванні в реальному часі обов’язково повинно бути включено при розроблені методу оптимізації [76].

В роботі [77] обгрунтовано потребу в розширенні структури таким чином, щоб алгоритми планування м'якого реального часу могли бути використані для планування завдань програми. Згідно з планувальниками, завдання можуть пропускати свої терміни на обмежене значення, яке називається запізненням. Незважаючи на це, підхід все ще може надати жорсткі гарантії в режимі реального часу для інтерфейсів вводу/виводу програми з навколишнім середовищем. Для цього напіврозділеного підходу пропонують [78, 79] евристику розподілу завдань, яка спрямована на зменшення мінімальної кількості процесорів, необхідної для планування роботи застосунків, порівняно з чисто розділеним алгоритмом планування, збереження низьких витрат на пам'ять і затримок, спричинених планувальником порівняно з чисто розділеним алгоритмом планування.

Підхід в роботі [80] може призвести до значних переваг за рахунок зменшення кількості процесорів, необхідних для планування певної програми, порівняно з підходом із розділеним підходом, досягаючи при цьому тієї ж пропускної здатності. Однак, це скорочення кількості необхідних процесорів відбувається за рахунок підвищених вимог до пам'яті та затримки програм.

Напіврозділені підходи дозволяють досягти більш високої енергоефективності порівняно з розділеними [81].

Алгоритм в [82] підтримується глобально над розглянутим набором процесорів, тобто не для кожного ядра, з дискретним набором режимів робочої напруги/частоти. Виводяться умови, що забезпечують коректне планування завдань застосунків в двох випадках: коли використовується найнижча частота, яка гарантує можливість планування та підтримується системою; коли використовується схема періодичного перемикання частот, яка зберігає можливість планування та дозволяє досягти більш високої економії енергії. В цілому, запропонований алгоритм дозволяє рівномірно розподілити утилізацію завдань між доступними процесорами. У свою чергу, це дозволяє процесорам працювати на більш низькій частоті, що призводить до меншого

споживання енергії. Порівнянні з чисто розділеним підходом до планування, ця техніка досягає однакової пропускної здатності програми з значною економією енергії у застосуванні до реальних програм потокового передавання. Ця економія енергії, однак, досягається за рахунок більш високих вимог до пам'яті та затримки застосунків [83].

Таким чином, ключовою особливістю більшості сучасних методів управління ресурсами є використання механізму міграції завдань. Цей підхід дозволяє гнучко розподіляти обчислювальні навантаження між процесорами, що, у свою чергу, сприяє підвищенню ефективності роботи системи. Відомі методики застосовують міграцію завдань у різний спосіб залежно від специфіки досліджуваної архітектури.

У випадку деяких систем методи, які використовують міграцію завдань, допускають її виконання динамічно, тобто у будь-який момент часу. Така міграція може ініціюватися як користувачем, так і самим середовищем виконання, наприклад, у відповідь на непередбачувані обставини, такі як збій обладнання, перевантаження окремих процесорів або необхідність енергозбереження. У таких системах підхід до балансування навантаження є адаптивним, оскільки реакція на зміну умов відбувається в реальному часі, що підвищує стійкість системи до можливих відмов та забезпечує безперервність обчислень.

На противагу цьому, інші методи, призначені для платформ із жорстким алгоритмом планування, реалізують міграцію завдань відповідно до чітко визначеної просторово-часової схеми. Це означає, що кожне завдання виконується згідно з попередньо розрахованим розподілом ресурсів, а переміщення завдань між процесорами підпорядковується певним правилам, встановленим у рамках алгоритму напіврозділеного поділу. У таких системах міграція є контрольованою та передбачуваною, що дозволяє уникнути невизначеностей під час виконання обчислень та підтримувати високий рівень детермінізму.

Наприклад, у певних архітектурах може бути дозволено періодичне чергування завдань між двома процесорами. У такому випадку одне й те саме завдання по черзі виконується то на першому, то на другому процесорі, що дає змогу рівномірно розподілити навантаження між ними. Такий підхід є корисним для зменшення ризику

перегріву окремих обчислювальних вузлів та підвищення загальної продуктивності системи. Крім того, чергова міграція може бути використана для зниження енергоспоживання або покращення балансування обчислювальних потоків у системах реального часу.

Залежно від характеру завдань і вимог до продуктивності може бути обраний той чи інший підхід до організації міграції. Гнучкі системи з адаптивним управлінням добре підходять для сценаріїв, де важлива стійкість до змін та несподіваних збоїв, тоді як детерміновані схеми корисні для жорстко регламентованих середовищ, таких як системи керування польотами або критично важливі промислові застосування. У будь-якому випадку вибір методу залежить від вимог до ефективності, прогнозованості та загальної архітектури апаратної платформи.

1.3 Постановка задачі

Поставлена мета досягається розв’язанням таких основних завдань:

- проаналізувати відомі методи та засоби оптимізації багатопроцесорних систем;
- розробити метод оптимізації багатопроцесорних систем;
- розробити цільову функцію оптимізації багатопроцесорних систем;
- здійснити реалізацію прототипу системи та дослідження розробленого методу оптимізації багатопроцесорних систем.

1.4 Висновки до першого розділу

Проаналізовано відомі методи та засоби оптимізації багатопроцесорних систем, а також визначено стратегію для покращення ефективності цього процесу. Для забезпечення розробленого методу оптимізації багатопроцесорних систем визначено архітектуру вбудованих систем, для якої будуть розроблятись механізми оптимізації.

2 ПРОЦЕС ОПТИМІЗАЦІЇ БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМ

2.1 Моделі потоків даних багатопроцесорних вбудованих систем

Вбудовані багатопроцесорні системи є спеціалізованими обчислювальними комплексами, що містять два або більше процесорів, які працюють разом для виконання певних завдань. Вони використовуються у пристроях, де важливими є енергоефективність, продуктивність, компактність і надійність. На відміну від традиційних комп’ютерів, такі системи проєктуються для виконання конкретних функцій, зокрема обробки сигналів, керування пристроями або реалізації алгоритмів штучного інтелекту. Вони можуть працювати в реальному часі та мають жорсткі обмеження щодо енергоспоживання, обсягу пам’яті та швидкості обробки даних.

Архітектура вбудованих багатопроцесорних систем може відрізнятися залежно від того, як організована взаємодія між процесорами. Симетрична багатопроцесорна архітектура передбачає рівний доступ усіх процесорів до пам’яті та ресурсів системи, що забезпечує ефективний обмін даними. В асиметричній архітектурі процесори можуть виконувати різні завдання, де один або кілька є головними, а інші – підлеглими. Гетерогенна архітектура використовує процесори різного типу, наприклад, поєднання центрального процесора (CPU) та графічного процесора (GPU), що дозволяє ефективно розподіляти навантаження. Системи-на-чіпі поєднують кілька процесорних ядер із периферійними компонентами, такими як контролери, пам’ять та інтерфейси, що робить їх популярними у смартфонах, маршрутизаторах та автомобільних системах.

Функціонування таких систем вимагає чіткої організації синхронізації процесорів, комунікації між ними та розподілу ресурсів. Для цього використовуються механізми блокування, семафори, бар’єри та черги повідомлень, що дозволяють уникати простоїв і гарантувати мінімальні затримки у виконанні критичних завдань. Обмін даними між процесорами може здійснюватися через спільну пам’ять або міжпроцесорні комунікації, а у високопродуктивних системах застосовуються мережі-на-чіпі для ефективного розподілу інформаційних потоків. Розподіл ресурсів

здійснюється операційною системою або мікроконтролером, які керують завданнями, забезпечуючи оптимальне використання продуктивності та енергоспоживання.

Такі системи мають широке застосування в автомобільній промисловості, телекомунікаціях, медичному обладнанні, мультимедіа та штучному інтелекті. В автомобілях вони використовуються в системах допомоги водієві, автоматизованих системах керування та бортових комп'ютерах. У телекомунікаціях вони забезпечують швидку обробку даних у маршрутизаторах, комутаторах і базових станціях зв'язку. Медичні пристрої, зокрема МРТ-сканери та кардіомонітори, також використовують ці технології для аналізу сигналів. Обробка відео та мультимедіа включає кодування, розпізнавання облич та відеоаналітику, а у сфері штучного інтелекту вони застосовуються для реалізації нейронних мереж та машинного навчання в IoT-пристроях.

Незважаючи на значні переваги, вбудовані багатопроцесорні системи стикаються з низкою викликів. Однією з основних проблем є оптимізація програмного забезпечення для ефективного використання багатоядерних ресурсів. Крім того, мінімізація енергоспоживання є критично важливою для мобільних і автономних пристроїв, які працюють у режимі реального часу. Забезпечення надійності та безпеки також відіграє ключову роль, оскільки системи мають бути захищені від збоїв, кіберзагроз і апаратних несправностей. У майбутньому розвиток таких систем буде спрямований на покращення інтеграції штучного інтелекту, удосконалення мереж-на-чіпі та впровадження нових обчислювальних технологій.

Моделі потоків даних у багатопроцесорних вбудованих системах є ефективним підходом до організації обчислень, особливо у випадках, коли потрібно обробляти великі обсяги інформації в реальному часі. Такі моделі ґрунтуються на представленні програм у вигляді графів потоку даних, де вузли відповідають обчислювальним операціям, а ребра позначають передачу даних між ними. Це дозволяє розділяти обчислення на незалежні або частково залежні завдання, які можуть виконуватися паралельно на різних процесорах, забезпечуючи оптимальне використання доступних ресурсів.

В основі моделей потоків даних лежить концепція детермінованого виконання,

що означає, що обчислення виконуються лише тоді, коли всі необхідні вхідні дані доступні. Це дозволяє уникнути станів змагань, які характерні для традиційних моделей програмування, що базуються на потоках. Завдяки такій організації, потокові моделі дозволяють значно покращити продуктивність багатопроцесорних вбудованих систем, зменшити застосункові витрати на синхронізацію процесів та забезпечити ефективну передачу даних. Крім того, подібні моделі забезпечують добру масштабованість, оскільки збільшення кількості обчислювальних вузлів дозволяє розширювати систему без суттєвої модифікації програмного забезпечення.

У вбудованих системах потокові моделі часто застосовуються в обробці сигналів, аналізі зображень, відеоструму, телекомунікаційних мережах та інших завданнях, де велика кількість вхідних даних обробляється послідовно або паралельно. Вони дозволяють розбити обчислення на окремі етапи, кожен із яких може бути призначений конкретному процесору або групі процесорів. Це, у свою чергу, сприяє зменшенню затримок і підвищенню загальної продуктивності системи.

Існують різні типи моделей потоків даних, які відрізняються способом передачі даних і управління виконанням. Наприклад, статичні моделі потоків даних характеризуються тим, що всі залежності між операціями визначаються заздалегідь, що дозволяє здійснювати точне планування обчислень і мінімізувати комунікаційні затримки. Динамічні моделі, навпаки, передбачають можливість адаптації під час виконання програми, що забезпечує більшу гнучкість, але потребує складніших механізмів керування і може збільшувати застосункові витрати.

У реальних системах часто використовується змішаний підхід, коли деякі частини алгоритму виконуються за статичною схемою, а інші можуть адаптуватися до змін у даних або ресурсах. Це особливо актуально для складних багатоядерних платформ, де необхідно враховувати не лише продуктивність процесорів, а й енергоспоживання, завантаженість комунікаційних каналів та можливі апаратні обмеження.

Ефективна реалізація моделей потоків даних у багатопроцесорних вбудованих системах потребує ретельного картографування завдань на апаратні ресурси. Важливу роль тут відіграє алгоритмічне планування, яке визначає, коли і на якому

процесорі має виконуватися певна операція. Це дозволяє мінімізувати простої процесорів і покращити загальну продуктивність системи. Крім того, розподіл завдань повинен враховувати обмеження на пам’ять, оскільки у вбудованих системах часто використовується обмежений обсяг локальної пам’яті, що може впливати на ефективність передачі даних між вузлами обчислень.

Ще одним важливим аспектом є комунікація між процесорами, яка може здійснюватися через спільну пам’ять або через спеціальні високошвидкісні мережі-на-чіпі. Використання спільної пам’яті дозволяє швидко обмінюватися даними між процесорами, проте може призводити до конфліктів доступу і збільшення затримок через необхідність синхронізації. Мережі-на-чіпі, навпаки, забезпечують більш ефективну взаємодію у великих багатопроцесорних системах, дозволяючи процесорам обмінюватися даними безпосередньо один з одним за допомогою маршрутизованих комунікацій.

У контексті оптимізації роботи багатопроцесорних вбудованих систем з потоковими моделями важливим є баланс між продуктивністю та енергоефективністю. Деякі сучасні платформи використовують динамічне масштабування напруги і частоти процесорів, що дозволяє знижувати енергоспоживання, коли система не працює на максимальній потужності. Крім того, програмне забезпечення може адаптуватися до змін у навантаженні, розподіляючи завдання таким чином, щоб уникати перевантаження окремих процесорів і рівномірно розподіляти обчислювальні ресурси.

Таким чином, моделі потоків даних відіграють ключову роль у підвищенні ефективності багатопроцесорних вбудованих систем, дозволяючи розподіляти обчислення між численними ядрами, зменшувати затримки та оптимізувати використання ресурсів. Вони знаходять застосування в багатьох галузях, де критично важливо забезпечити обробку великих обсягів даних у реальному часі, зберігаючи при цьому високу продуктивність і мінімізуючи енергоспоживання. Розвиток таких моделей у майбутньому буде спрямований на ще глибшу інтеграцію адаптивних алгоритмів, покращення ефективності управління ресурсами та впровадження нових методів комунікації між процесорами, що дозволить досягти ще вищої

продуктивності та надійності вбудованих багатопроцесорних платформ. Типова багатопроцесорна вбудована система зображена на рис. 2.1.

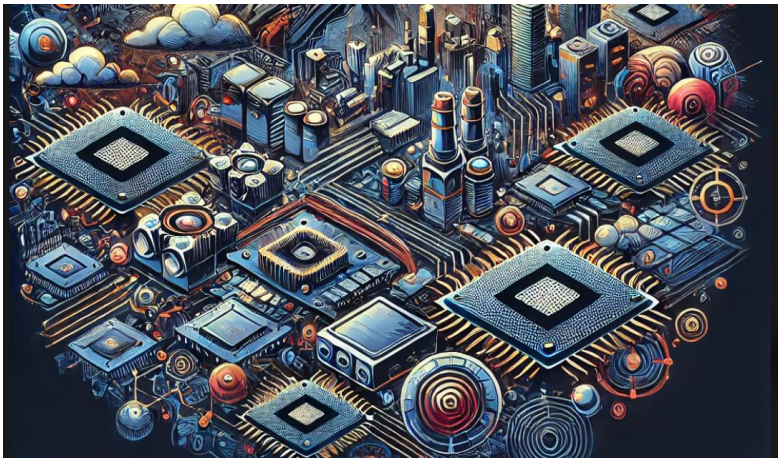


Рисунок 2.1 – Багатопроцесорна вбудована система

Розглянемо моделі потоків даних обчислень. Модель потоку даних добре підходять для подання потокових програм, оскільки вони дозволяють природним чином виразити паралелізм, доступний у таких програмах. У проектах вони задаються у вигляді орієнтованих графів, в яких вузли графа представляють завдання (активні сутності) програми, а ребра графа представляють залежності між завданнями даних. У вузли графа застосунків заповнимо процесами.

Розглянемо однорідні синхронні потоки даних. Багатопроцесорна система синхронного потоку даних моделюється у вигляді напрямленого мультиграфа. Вершини представляють завдання програми і ребра представляють залежності даних між завданнями. Вершини спілкуються через ребра, генеруючи потік даних, який ділиться на атомарні об'єкти даних, які називаються токенами. Ребро представляє буфер першого типу і визначається кортежем. Цей кортеж означає, що ребро спрямоване від вершини, яку називають джерелом, до вершини, яку називають пунктом призначення. Визначимо вхідні та вихідні актори графа G наступним чином. Вхідна вершина графа отримує вхідний потік застосунку з оточення. Вихідна вершина графа виробляє вихідний потік програми в середовище. Виклик може почати своє виконання тільки в тому випадку, якщо на всіх його вхідних ребрах присутня

достатня кількість вхідних даних. Під час одного виклику вхідна вершина споживає вхідні дані з усіх своїх вхідних ребер та обробляє ці дані відповідно до заданої функції та записує вихідні дані на свої вихідні краї. Кількість зчитаних/записаних даних кожого ребра вводу/виводу є фіксованою та відомою під час компіляції і називається швидкістю споживання/виробництва. Можна визначити множину вершин-попередників і наступників. Припускаємо, що будь-яка вхідна вершина не має попередників, а будь-яка вихідна вершина не має наступників. Крім того, можна визначити для кожної вершини набір вхідних і вихідних ребер. Окремим випадком буде той, в якому всі норми виробництва/споживання всіх суб'єктів дорівнюють одиниці.

Оскільки потокові програми обробляють безперервні потоки даних, то визначення розкладу графіка, який може тривати нескінченно, використовуючи кінцевий обсяг пам'яті для реалізації каналів, що відповідають ребрам графа. Такий граф може бути отриманий під час компіляції, якщо граф є послідовним і не має циклів. Граф є несуперечливим, якщо його рівняння балансу має додатний цілочисельний розв'язок.

Матриця топології будується так, що вектор повторення з найменшою нормою називається базовим вектором повторення. Будемо використовувати базовий вектор повторення графа для виконання аналізу. Будь-який вектор отриманий шляхом множення основного вектора повторення на додатне натуральне число, також є вектором повторення, тобто задовольняє рівнянню. Існування додатного цілого розв'язку рівняння є лише необхідною умовою для виконання графа нескінченно з періодом.. Ще однією умовою, яка повинна бути виконана, є відсутність тупикових ситуацій, що можна перевірити побудовою періодично допустимого графа. Граф, який не має циклу та тупіка, називається вільним від дедлоку. Важливою властивістю багатопроцесорних систем є те, що стабільність і вільність від дедлоку графа можуть бути перевірені під час компіляції. Розглядатимемо тільки послідовні і вільні від дедлоку графи.

Циклостатичний потік даних є узагальненням поняттям, подібно до графу і також складається з набору вершин та набору ребер. Однак, на відміну від

поперднього графу, поведінка його суб'єктів є циклічною.

Кожна вершина має певну кількість фаз. Виконання кожної фази пов'язане з певною функцією. Таким чином, можна визначити послідовність виконання яка пов'язує кожну фазу з відповідною виконуваною функцією. При цьому норми виробництва/споживання для кожного ребра виходу/входу також визначаються для кожної фази. Таким чином, для кожної вершини можна визначити наступні послідовності.

Послідовність споживання для кожного вхідного ребра. Довжина всіх цих послідовностей дорівнює числу фаз. Фази вершин виконуються циклічно. Тобто під час виклику виконується функція. Аналогічно, для кожного вхідного ребра витрачаються токени, а для кожного вихідного ребра виробляються токени. Сукупна кількість токенів, спожитих викликами з вершини від його вхідного ребра дорівнює кумулятивній кількості токенів, вироблених викликами вершини від його вхідного ребра. Тоді, потрібно здійснити знаходження невизначеного періодичного графу.

Виходячи з цього вектора повторення можна вивести статичний невипереджаючий граф для графу, який може повторюватися вічно за допомогою обмежених буферів. Цією властивістю володіють розклади. Існує, також, альтернативний граф. Для послідовного та вільного від дедлоків графа враховуючи вектор повторення графа, можна визначити концепцію ітерації графа. Це потрібно для отримання відомостей про цикли в багатопроцесорній системі.

Ітерація графа – це виклик кожної вершини. Будемо використовувати поняття суб'єктів системи. Ця концепція є спільною для обох підходів. Формально, будь-яка вершина є без стану, оскільки відношення між спожитими та виробленими токенами під час виклику визначається функцією. Бо функція не має стану. Однак, іноді необхідно моделювати вершини, для яких результат поточного виклику залежить від даних, отриманих у попередніх викликах. Усі ці залежності від попередніх викликів моделюються за допомогою саморебер. На цих саморебрах виклик може записувати токени даних, які представляють «стан» вершини і які можуть бути прочитані послідовними викликами.

Вершина називається порожньою, якщо вона не має власних ребер, які

використовуються для моделювання її стану. Мережа багатогранних процесів використовується головним чином в контексті систем найкращих зусиль. Процеси представляють завдання програми і взаємодіють між собою для використання каналів. Кожен процес має набір вхідних портів і набір вихідних портів, через які процес зчитує і записує дані. Канали, підключені до вхідного і вихідного портів процесу називаються вхідними і вихідними каналами. Процес обробляє маркери зчитування/запису даних з/на канали за допомогою портів введення/виведення. Процеси синхронізуються через канали, тобто процеси, які намагаються прочитати з порожнього каналу з черги, будуть блокуватися. Однак, буфери мають обмежений розмір, тому процеси також змушені блокуватися при спробі запису в повний канал.

Контроль і синхронізація в таких системах повністю розподілені, що дозволяє змінювати відображення процесів під час виконання з незначними зусиллями. Ще одне обмеження полягає в тому, що процеси мають чітку структуру. Кожне виконання процесу відповідає певному значенню ітераторів циклу. Значення цих ітераторів можна представити у вигляді вектору ітерацій. Кожен процес виконується наступним чином. По-перше, процес зчитує дані зі своїх вхідних портів (підмножини). Підмножина вхідних портів, з яких зчитуються дані, залежить від значення вектора ітерацій. Наприклад, процес зчитує дані, якщо виконується умова, в іншому випадку дані не зчитуються. Потім вхідні дані обробляються функцією. Ця функція відображає обчислювальну поведінку процесу. Нарешті, процес записує вироблені дані на свої вихідні порти. Підмножина вихідних портів, на які записуються дані, залежить, знову ж таки, від значення вектору ітерацій.

Зв'язок між вхідними та вихідними даними процесу визначається функцією, яка за визначенням не має стану. Для моделювання процесів, для яких результат поточної ітерації залежить від даних, отриманих на попередніх ітераціях, можна використовувати власні канали. На цих власних каналах ітерація може записувати токени даних, які представляють «стан» процесу і які можуть бути прочитані послідовними ітераціями. Тому, цей стан зберігається поза самим процесом. Однак, набір портів вводу/виводу, який читається /записується процесом виводиться на основі його вектору ітерацій. Таким чином, вектор ітерації фактично є єдиним станом

процесу.

Обмеження, що накладаються, призводять до наступної важливої властивості. Будь-яка послідовна програма, визначена як статична афінна програма вкладеного циклу, може бути автоматично перетворена в еквівалентну паралельну специфікацію. Статична афінна програма вкладеного циклу це програма, в якій кожен оператор програми оточений одним або декількома циклами та оператором розгалуження, і де петлі мають постійний розмір кроку, цикли мають границі, які є афінними виразами ітераторів замикаючого циклу, статичними параметрами програми та константами, оператори розгалуження мають афінні умови в термінах ітераторів циклу, статичних параметрів програми та констант, індексний вираз посилань на масив, тобто афінні вирази ітераторів охоплюючого циклу, статичних параметрів програми та констант, потік даних між операторами в циклі є явним, що забороняє, щоб два оператори, які містять виклики функцій, взаємодіяли через спільні змінні, невидимі на рівні системи.

Для автоматичного перетворення статичних програм афінного вкладеного циклу у паралельні специфікації та визначення розмірів буферів, які гарантують безтупикове виконання використаємо компілятор. Незважаючи на те, що компілятор накладає деякі обмеження на специфікацію вхідного застосунку, великий набір потокових застосунків може бути ефективно визначений. Можна цим підходом також моделювати програми з різних областей, таких як обробка зображень/відео, обробка звуку та більш складніші обчислення. Більшість потокових застосунків можна вказати за допомогою моделі синхронного потоку даних. Модель є більш виразною, тому її можна ефективно використовувати для моделювання більшості потокових програм.

Формально представляємо модель періодичних завдань в реальному часі і результати планування для багатопроцесорних систем. Ця модель завдання та методи аналізу відіграють важливу роль у підходах. У моделі періодичного завдання реального часу завдання визначається кортежем з показниками найгіршого часу виконання завдання, періоду виконання завдання, часу початку виконання завдання, терміну виконання завдання. Періодичне завдання починається з часу і є повторюваним, з постійним часом міжприбуття. Тобто, періодична задача

викликається в моменти часу для всіх процесорів. Кожен виклик називається роботою. Завдання має завершити своє виконання до встановленого часу. Припустимо, що завдання мають неявні терміни виконання для кожного завдання. У цьому випадку абсолютний термін виконання завдання збігається з приходом завдання. Завдання можуть бути випереджені в будь-який момент часу. Потреба в періодичному завданні в режимі реального часу визначається наступним чином. Вимога періодичного виконання завдання в режимі реального часу повинна бути обов’язково. Вимога періодичної задачі реального часу в інтервалі це сумарний час, за який виконуються завдання. Узагальнимо техніку планування, на якій базуються підходи, спрямовані на системи. Ця техніка планування розглядає вхідний застосунок, змодельований у вигляді ациклічного графа з вершинами. Потім ця модель застосування перетворюється на множину періодичних завдань у реальному часі. У загальному випадку завдання не мають однакового часу початку, тобто це асинхронна задача. Модель завдань є узагальненням моделі періодичних завдань. Робочі місця звільнені завданням повинні бути відокремлені в часі мінімальним міжприхідним інтервалом.

Розглянемо однорідні багатопроцесорні системи. Тобто, в розглянутих системах всі процесори ідентичні і швидкість виконання завдань на процесорах однакова. Зокрема, розглянемо систему, що складається з множини однорідних процесорів. Багатопроцесорні алгоритми планування в реальному часі є складними і потребують адаптації під певну архітектуру системи, що досліджується. Концепції та результати аналізу планування в реальному часі є важливими для підходів, що будуть розроблятись. Пріоритетним є розроблення та удосконалення алгоритмів планування, які обробляють періодичні набори завдань у режимі реального часу.

Алгоритми багатопроцесорного планування відіграють ключову роль в організації ефективного розподілу обчислювальних ресурсів у системах з кількома процесорами. Їх основне завдання полягає у вирішенні двох фундаментальних питань, що стосуються розподілу завдань. По-перше, необхідно визначити, які саме завдання мають виконуватися на конкретних процесорах, тобто здійснити розподіл завдань між наявними процесорами. По-друге, потрібно розв'язати проблему

пріоритетності, тобто визначити, коли саме і в якому порядку відносно інших завдань має виконуватися кожне із них. Обидва ці аспекти є критично важливими для досягнення оптимальної продуктивності системи, забезпечення своєчасного виконання завдань та уникнення затримок.

Залежно від того, як алгоритми підходять до розподілу завдань, то їх можна поділити на кілька категорій. Одна з найважливіших класифікацій стосується підтримки міграції завдань між процесорами. Деякі алгоритми повністю виключають можливість міграції, тобто кожне завдання жорстко прив’язане до одного процесора і не може переміщуватися на інші. Такий підхід спрощує управління виконанням завдань, оскільки не потребує складних механізмів переміщення даних та синхронізації між процесорами. Однак, він має свої недоліки, оскільки може призвести до нерівномірного завантаження процесорів і, як наслідок, до неефективного використання ресурсів системи.

Існують також підходи, які допускають міграцію на рівні завдань, коли різні частини одного й того ж завдання можуть виконуватися на різних процесорах. У такому випадку кожна частина завдання виконується лише на одному процесорі, але загалом завдання може мігрувати між процесорами у процесі виконання. Це забезпечує більш рівномірне навантаження системи, але вимагає ретельного управління залежностями між частинами завдання. Ще більш гнучким є підхід із міграцією на рівні робочих місць, коли завдання може переміщуватися між процесорами під час виконання. Однак при цьому не допускається паралельне виконання одного і того ж завдання на кількох процесорах одночасно. Такий підхід забезпечує високу гнучкість у використанні ресурсів, але вимагає складніших механізмів управління станами завдань, щоб уникнути конфліктів і узгодженості даних.

Найгнучкішими є глобальні алгоритми, які дозволяють вільну міграцію як на рівні завдань, так і на рівні робочих місць. Це дозволяє максимально ефективно використовувати ресурси системи, перерозподіляючи завдання відповідно до поточного завантаження процесорів. Однак така гнучкість супроводжується значними складнощами в управлінні, оскільки вимагає складних механізмів

синхронізації та обміну даними між процесорами. На противагу їм, розділені алгоритми повністю обмежують міграцію, тобто кожне завдання або його частина виконується лише на одному процесорі без можливості переміщення. Це спрощує управління розкладом, але може призводити до дисбалансу в завантаженні процесорів, особливо якщо деякі завдання є більш ресурсомісткими, ніж інші.

Крім підходів до міграції, алгоритми багатопроцесорного планування також розрізняються за методами встановлення пріоритетів. Один із підходів передбачає фіксований пріоритет для кожного завдання, який не змінюється протягом усього його виконання. Це забезпечує передбачуваність у розкладі, але може бути неефективним у випадках, коли робоче навантаження є динамічним. Інший підхід передбачає фіксований пріоритет для кожної окремої роботи в межах завдання, але різні роботи можуть мати різні пріоритети. Це підвищує гнучкість планування, але ускладнює аналіз розкладу, оскільки потрібно враховувати залежності між роботами одного завдання. Найбільш динамічним є підхід із змінними пріоритетами, коли пріоритет завдання може змінюватися під час його виконання залежно від залишкового часу до завершення або інших факторів. Це дозволяє більш ефективно використовувати ресурси, але потребує складних обчислень для прийняття рішень.

Завдання вважається здійсненним, якщо існує алгоритм, здатний забезпечити виконання всіх можливих послідовностей робіт цього завдання в межах встановлених термінів. Алгоритм вважається оптимальним, якщо він здатний запланувати всі можливі набори завдань, які відповідають моделі завдань і є здійсненними в системі. Поняття планованості відноситься до можливості виконання завдання в межах найгіршого часу відгуку, який не перевищує встановлений термін. Якщо це виконується для всіх завдань у наборі, то набір вважається планованим.

У теорії планування в реальному часі використовуються аналітичні тести на планованість, які поділяються на достатні, необхідні та точні. Достатні тести гарантують, що всі набори завдань, визнані плановаными, дійсно можуть бути виконані в межах термінів. Необхідні тести стверджують, що всі набори, визнані непланованими, насправді не можуть бути виконані вчасно. Точні тести є одночасно достатніми і необхідними, тобто точно визначають планованість набору завдань. Для

періодичних завдань із жорсткими термінами корисним показником продуктивності є найгірша межа використання процесора, яка показує максимально можливе завантаження, за якого всі завдання можуть бути виконані вчасно.

Таким чином, алгоритми багатопроцесорного планування забезпечують ефективний розподіл обчислювальних ресурсів у складних системах, враховуючи різні підходи до міграції та встановлення пріоритетів. Вибір алгоритму залежить від вимог системи, характеристик завдань та цільових показників продуктивності.

В результаті, класифікація алгоритмів залежить від підходу до міграції завдань між процесорами. У деяких алгоритмах міграція заборонена, і кожне завдання виконується тільки на одному процесорі, що спрощує управління, але може спричинити нерівномірне навантаження. Інші алгоритми допускають міграцію на рівні завдань або робочих місць, забезпечуючи кращий баланс навантаження, але вимагають складного управління залежностями та станами завдань. Найгнучкішими є глобальні алгоритми, які дозволяють вільну міграцію, однак вони потребують складних механізмів синхронізації. Натомість розділені алгоритми обмежують міграцію на всіх рівнях, що спрощує управління, але може призвести до дисбалансу завантаження процесорів. Методи встановлення пріоритетів також впливають на ефективність планування. При фіксованому пріоритеті завдання має постійний пріоритет, що забезпечує передбачуваність, але неефективне в динамічних умовах. Фіксований пріоритет робіт дозволяє різні пріоритети для частин завдання, підвищуючи гнучкість, але ускладнюючи аналіз розкладу. Динамічний пріоритет змінюється під час виконання завдання, що оптимізує використання ресурсів, але потребує складних обчислень. Завдання вважається здійсненним, якщо існує алгоритм, який забезпечує виконання всіх можливих послідовностей робіт у встановлені терміни. Алгоритм вважається оптимальним, якщо він здатний запланувати всі можливі набори завдань, які відповідають моделі і є здійсненними в системі. Необхідні тести підтверджують, що всі визнані непланованими набори завдань насправді не можуть бути виконані в межах термінів. Точні тести одночасно достатні і необхідні, тобто точно визначають плановність набору завдань. Для періодичних завдань із жорсткими термінами важливим показником є найгірша межа

використання процесора, яка вказує на максимально можливе завантаження, за якого всі завдання можуть бути виконані вчасно. Таким чином, ефективність багатопроцесорного планування залежить від обраного підходу до міграції та пріоритетів, а також від аналітичної оцінки планованості завдань.

2.2 Цільова функція оптимізації завдань у багатопроцесорних системах з міграцією

Оптимізація завдань у багатопроцесорних системах з міграцією вимагає розробки цільової функції, яка враховує всі релевантні параметри для досягнення максимальної ефективності використання ресурсів та своєчасного виконання завдань. Основними параметрами, які слід враховувати, є час виконання завдань, затримки при міграції, навантаження на процесори, енергоспоживання, пріоритети завдань та дотримання встановлених термінів. Важливо мінімізувати загальний час завершення всіх завдань, забезпечити баланс навантаження між процесорами та мінімізувати застосункові витрати, пов'язані з міграцією завдань. Крім того, оптимізація повинна враховувати енергоспоживання, оскільки надмірне використання ресурсів призводить до зростання енерговитрат.

Цільову функцію для такої оптимізації задамо наступним чином:

$$F = \min\left[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{k=1}^{K} E_k + \delta \sum_{i=1}^{J} T_j\right], \qquad (2.1)$$

де C_i – час завершення завдання i; L_m – коефіцієнт навантаження на процесор m, визначений як відношення часу зайнятості процесора до загального часу виконання; E_k – енергоспоживання процесора k під час виконання завдань; T_j – загальний час міграцій для завдання j, що враховує затримки на передачу даних та налаштування середовища виконання на новому процесорі; $\alpha, \beta, \gamma, \delta$ – вагові коефіцієнти, які визначають відносну важливість кожного з параметрів у цільовій функції.

Мета оптимізації полягає в мінімізації цільової функції F, яка враховує баланс

між часом завершення завдань, навантаженням на процесори, енергоспоживанням та витратами на міграцію. Вагові коефіцієнти можна налаштовувати відповідно до специфіки системи та цілей оптимізації. Наприклад, у системах реального часу пріоритет може надаватися мінімізації часу завершення завдань та дотриманню термінів, тоді як у системах з обмеженими енергоресурсами – мінімізації енергоспоживання. Оптимізація такої цільової функції може здійснюватися за допомогою методів динамічного програмування, еволюційних алгоритмів або методів машинного навчання для адаптивного налаштування параметрів системи в реальному часі.

У багатопроцесорних системах з міграцією кількість завдань, що впливають на процес оптимізації, є важливим параметром, оскільки вона визначає складність розподілу навантаження, витрати на міграцію та загальний час виконання. Нехай N – загальна кількість завдань, які потрібно запланувати, а M – кількість процесорів у системі. У моделі передбачається, що кожне і-те завдання має власний час виконання C_i, може мігрувати між процесорами, спричиняючи затримки T_i, і споживати енергію E_i під час виконання. Цільова функція повинна мінімізувати загальний час завершення всіх завдань, балансувати навантаження між процесорами та мінімізувати енергоспоживання та витрати на міграцію. Уточнену цільову функцію з урахуванням таких параметрів задамо так:

$$F = \min[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{i=1}^{N} E_i + \delta \sum_{i=1}^{N} T_i], \qquad (2.2)$$

де $C_i = S_i + W_i + M_i$ – час завершення завдання i, який складається з часу очікування S_i, часу виконання W_i та часу міграцій M_i; $L_m = \frac{\sum_{i=1}^{N} W_{im}}{\sum_{i=1}^{N} C_i}$ – навантаження на процесор m, обчислене як відношення часу виконання всіх завдань на процесорі m до загального часу виконання; $E_i = P_i \cdot W_i$ – енергоспоживання завдання i, яке залежить від потужності процесора P_i та часу виконання W_i; $T_i = \sum_{k=1}^{K} t_{ik}$ – сумарний час міграцій для завдання i, де t_{ik} – час переміщення завдання i на процесор k; $\alpha, \beta, \gamma, \delta$

– вагові коефіцієнти, які визначають відносну важливість кожного параметру в цільовій функції оптимізації.

Розглянемо просту систему з $N = 3$ завданнями та $M = 2$ процесорами.

Припустимо, що в системі виконуються завдання.

Завдання 1: $W_1 = 4, M_1 = 1, P_1 = 2$

Завдання 2: $W_2 = 3, M_2 = 2, P_2 = 3$

Завдання 3: $W_3 = 5, M_3 = 0, P_3 = 2$

Обчислимо кожний компонент в цільовій функції за формулою (2.2).

Час завершення:

$$C_1 = W_1 + M_1 = 4 + 1 = 5$$
$$C_2 = W_2 + M_2 = 3 + 2 = 5$$
$$C_3 = W_3 + M_3 = 5 + 0 = 5$$

Навантаження на процесори.

Для процесора 1: $L_1 = \frac{4+3}{5+5+5} = \frac{7}{15}$

Для процесора 2: $L_1 = \frac{5}{15} = \frac{1}{3}$

Енергоспоживання:

$$E_1 = P_1 \cdot W_1 = 2 \cdot 4 = 8$$
$$E_2 = P_2 \cdot W_2 = 3 \cdot 3 = 9$$
$$E_3 = P_3 \cdot W_3 = 2 \cdot 5 = 10$$

Час міграцій:

$$T_1 = M_1 = 1$$
$$T_2 = M_2 = 2$$
$$T_3 = M_3 = 0$$

Припустимо, що вагові коефіцієнти: $\alpha = 1, \beta = 2, \gamma = 1.5, \delta = 1$, тоді цільова

функція обчислюється так:

$$F = 1 \cdot (5 + 5 + 5) + 2 \cdot \left(\frac{7}{15} + \frac{1}{3}\right) + 1.5 \cdot (8 + 9 + 10) + 1 \cdot (1 + 2 + 0)$$

$$F = 15 + 2 \cdot \left(\frac{7}{15} + \frac{5}{15}\right) + 1.5 \cdot 27 + 3$$

$$F = 15 \cdot 2 \cdot \left(\frac{12}{15}\right) + 40.5 + 3$$

$$F = 15 + 2 \cdot 0.8 + 40.5 + 3 = 15 + 1.6 + 40.5 + 3 = 60.1$$

Таким чином, значення цільової функції F=60,1. Це значення можна використовувати для порівняння з іншими розподілами завдань, щоб знайти оптимальне рішення. Такий підхід дозволяє врахувати кількість завдань, їхні характеристики та вплив на процес оптимізації в багатопроцесорній системі з міграцією.

Коли кількість завдань суттєво перевищує кількість процесорів у вбудованій багатопроцесорній системі, оптимізація розподілу завдань стає значно складнішою через кілька ключових факторів. По-перше, необхідно враховувати не тільки балансування навантаження, але й інші обмеження, такі як енергоспоживання, обмеження пам'яті, реальні часові вимоги та міжпроцесорні затримки під час передачі даних.

Зі збільшенням кількості завдань кілька аспектів оптимізації набувають критичної ваги.

Часові витрати на міграцію. При великій кількості завдань часті міграції можуть значно збільшити загальні витрати часу через додаткові витрати на передачу станів між процесорами. Це може призвести до ситуацій, коли вигода від балансування навантаження нівелюється додатковими витратами на міграцію. Тому, виникає потреба у визначенні оптимальних точок міграції та мінімізації частоти міграцій.

Зі зростанням кількості завдань оптимізаційний алгоритм стикається з експоненційним зростанням простору пошуку рішень, тобто зростає складність планування. Простий жадібний алгоритм, який працює ефективно для малої кількості

завдань, втрачає ефективність через неповний огляд можливих конфігурацій. У таких випадках перспективним підходом є використання методів стохастичної оптимізації, таких як генетичні алгоритми, алгоритми рою частинок або машинне навчання для прогнозування оптимального розподілу.

Енергетична ефективність проявляється в тому, що коли кількість завдань зростає, то загальне енергоспоживання системи також збільшується. Для вбудованих систем це критично, оскільки вони часто мають обмежений ресурс живлення (наприклад, батареї). У цьому контексті необхідно додатково оптимізувати енергоспоживання, використовуючи динамічне регулювання напруги і частоти у поєднанні з міграцією завдань.

Із збільшенням кількості завдань виникає більша ймовірність взаємозалежності між ними. Це означає, що міграція одного завдання може спричинити затримки для інших, особливо якщо завдання часто обмінюються даними. У таких випадках оптимальний розподіл має враховувати топологію зв’язків між завданнями, мінімізуючи міжпроцесорні комунікаційні витрати.

У реальних системах завдання можуть мати динамічний характер з’являтися, завершуватись або змінювати свій пріоритет, тобто проявлятиметься адаптивність алгоритму. Це вимагає адаптивних підходів до оптимізації, які можуть реагувати на такі зміни в реальному часі. Перспективним підходом є використання методів підкріплювального навчання, які дозволяють алгоритму навчатися оптимальному розподілу завдань на основі історичних даних.

Велика кількість завдань створює додаткове навантаження на кеш і локальну пам'ять процесорів. Це може призвести до частих промахів кешу та збільшення затримок доступу до пам'яті. В таких випадках ефективною стратегією є врахування локальності даних при міграції завдань, тобто розміщення завдань так, щоб вони працювали з локальними даними, мінімізуючи передачу даних між процесорами.

Хоча підхід реплікації при міграціях завдань є ефективним для швидкої міграції завдань, але він демонструє обмежену ефективність при значному перевищенні кількості завдань над кількістю процесорів. Виникає необхідність у гібридних підходах, які поєднують реплікацію з іншими стратегіями, такими як відтворення

процесів або динамічне масштабування. Наприклад, можна використовувати реплікацію для коротких завдань, а відтворення — для довгих, енергоємних процесів. Іншою перспективою є застосування мультиагентних систем, де кожен процесор вважається агентом, який автономно приймає рішення на основі локальної інформації та комунікації з іншими агентами. Це дозволяє уникнути центрального планувальника, який може стати вузьким місцем у системі з великою кількістю завдань. Також слід враховувати гетерогенність процесорів у сучасних вбудованих системах. Наприклад, у багатьох системах використовуються поєднання високопродуктивних і енергоефективних процесорів. Оптимізація в таких системах повинна враховувати не тільки навантаження, але й енергоефективність кожного типу процесора.

Таким чином, потребують подальшого удосконалення та розроблення адаптивних, енергоефективних та масштабованих алгоритмів, які можуть оптимально розподіляти велику кількість завдань, враховуючи обмеження вбудованих багатопроцесорних систем. Важливим напрямком є інтеграція методів машинного навчання для прогнозування навантаження та адаптації алгоритмів оптимізації в реальному часі.

Вбудовані багатопроцесорні системи суттєво відрізняються від звичайних багатопроцесорних систем як за архітектурою, так і за обмеженнями, що впливають на реалізацію механізмів міграції завдань. Ці особливості зумовлюють значні складнощі під час оптимізації продуктивності та енергоефективності.

Однією з основних відмінностей є обмежені ресурси вбудованих систем. На відміну від звичайних багатопроцесорних систем, наприклад, серверів або десктопів, які мають значні обсяги пам'яті, високопродуктивні процесори та потужну систему охолодження, вбудовані системи зазвичай мають обмежений обсяг пам'яті, менш потужні процесори та обмежений енергетичний бюджет. Це накладає обмеження на складність алгоритмів планування та міграції завдань, адже кожна операція споживає обмежені ресурси.

Ще однією важливою особливістю є гетерогенність архітектури вбудованих систем. Часто вбудовані багатопроцесорні системи використовують різнорідні

обчислювальні ядра, які мають різну продуктивність і енергоспоживання. Наприклад, у архітектурі можуть поєднуватися високопродуктивні ядра з енергоефективними. Це ускладнює реалізацію міграції, оскільки завдання не можна просто перемістити на інший процесор без урахування характеристик ядра. Потрібно адаптувати стан процесу та налаштування програмного середовища, а це додаткові додаткові витрати.

Жорсткі часові обмеження є характерною особливістю вбудованих систем реального часу. Вони обумовлені вимогами до надійності та передбачуваності виконання завдань, наприклад, у системах автомобільної електроніки чи медичних пристроях. Міграція завдань у таких системах може призвести до непередбачуваних затримок через додаткові витрати на передачу стану процесу та синхронізацію. Тому, звичайні підходи до міграції, які ефективні у звичайних багатопроцесорних системах, тут не підходять. Потрібні детерміновані алгоритми, які гарантують дотримання часових обмежень.

Обмежені можливості комунікації також є критичним аспектом. У вбудованих системах часто використовуються шини з низькою пропускною здатністю або навіть бездротові інтерфейси зв'язку, що мають обмежену швидкість передачі даних і можуть викликати додаткові затримки. Міграція завдань у таких умовах може стати вкрай неефективною, особливо якщо процеси мають великий обсяг стану або часто обмінюються даними. Ще однією проблемою є обмежена підтримка динамічного управління пам'яттю. У звичайних багатопроцесорних системах використовується динамічна віртуальна пам'ять, що полегшує переміщення процесів між ядрами. Натомість у вбудованих системах часто застосовуються статичні адреси пам'яті для зниження додаткових витрат і підвищення передбачуваності часу виконання. Це суттєво ускладнює реалізацію механізмів міграції, оскільки стан процесу не можна просто перенести в інше місце пам'яті. Енергоспоживання є критично важливим обмеженням у вбудованих системах, особливо якщо вони працюють від батареї. Міграція завдань потребує додаткових обчислень і передачі даних, що призводить до збільшення енергоспоживання. Тому, в багатьох випадках потрібно оптимізувати не тільки навантаження, але й енергетичні витрати. У звичайних багатопроцесорних системах такі обмеження не є критичними, оскільки вони зазвичай підключені до

електромережі або мають високі енергетичні запаси.ьБезпека та надійність, також, є більш критичними у вбудованих системах, особливо в промислових або медичних застосуваннях. Будь-яка помилка під час міграції завдань може призвести до відмови системи або навіть до катастрофічних наслідків. У звичайних системах помилки в міграції зазвичай призводять лише до зниження продуктивності, але не до критичних збоїв.

Метод реплікації для міграції завдань у вбудованих системах має певні переваги, такі як швидкість перенесення процесу завдяки готовим копіям коду на кількох ядрах. Проте, він менш ефективний з точки зору пам'яті, яка є обмеженим ресурсом. Більш перспективним є комбінування реплікації з відтворенням процесів, де реплікація застосовується лише до критичних завдань, а відтворення — до менш важливих.

Крім того, слід враховувати гетерогенність архітектури та розробляти адаптивні алгоритми міграції, які можуть враховувати характеристики ядер і динамічно змінювати стратегії розподілу завдань. Перспективними є підходи на основі машинного навчання, які передбачають навантаження та енергоспоживання, а також використовують підкріплювальне навчання для адаптації до змін в реальному часі.

Хоча міграція завдань у вбудованих багатопроцесорних системах стикається з численними обмеженнями, поєднання адаптивних алгоритмів, енергоефективних стратегій і методів машинного навчання відкриває перспективи для подальшої оптимізації продуктивності цих систем.

Таким чином, розроблено цільову функцію для оптимізації завдань в багатопроцесорних вбудованих системах, в якій враховано баланс між часом завершення завдань, навантаженням на процесори, енергоспоживанням та витратами на міграцію.

2.3 Висновки до другого розділу

Встановлено, що ефективність багатопроцесорного планування залежить від

обраного підходу до міграції та пріоритетів, а також від аналітичної оцінки планованості завдань.

Для реалізації аналітичної оцінки було розроблено цільову функцію для оптимізації завдань в багатопроцесорних вбудованих системах, в якій враховано баланс між часом завершення завдань, навантаженням на процесори, енергоспоживанням та витратами на міграцію.

3 МІГРАЦІЇ ПРОЦЕСІВ В БАГАТОЗАДАЧНИХ ВБУДОВАНИХ СИСТЕМАХ

3.1 Алгоритми міграції процесів в багатозадачних вбудованих системах

Алгоритми міграції процесів у багатозадачних вбудованих системах спрямовані на оптимальний розподіл обчислювальних завдань між кількома процесорами або ядрами. Основна мета міграції – збалансувати навантаження, мінімізувати час виконання завдань, зменшити енергоспоживання та забезпечити дотримання реального часу для критичних завдань. Вбудовані системи мають обмежені ресурси, тому ефективне управління міграцією процесів є особливо важливим. Залежно від підходу до розподілу завдань і управління міграцією, алгоритми можна поділити на кілька категорій: глобальні; кластерні; ієрархічні; гібридні. Глобальні алгоритми розглядають усі процесори як спільну множину ресурсів. Будь-який процес може виконуватися на будь-якому процесорі, а міграція відбувається динамічно залежно від навантаження. Основними підходами є збалансування навантаження, розподіл навантаження та динамічний розподіл навантаження. Алгоритми збалансування навантаження переміщують процеси між процесорами для досягнення рівномірного навантаження. Вони використовують поточні метрики, такі як використання процесора або кількість черг процесів. Прикладами таких алгоритмів є "Least Loaded Processor", де завдання переноситься на процесор з найменшим навантаженням, та "Round Robin Migration", який розподіляє завдання по черзі між усіма процесорами. Алгоритми розподілу навантаження розподіляють завдання на вільні процесори в момент їх створення, а не під час виконання, що зменшує застосункові витрати на міграцію, але може призводити до нерівномірного навантаження. Динамічний розподіл навантаження використовує моніторинг стану системи в реальному часі для динамічного перенесення завдань. Прикладом є алгоритм "Work Stealing", де процесори з невеликим навантаженням забирають завдання у завантажених процесорів, забезпечуючи тим самим рівномірний розподіл завдань.

Кластерні алгоритми міграції поділяють систему на групи процесорів (кластери), і міграція відбувається в межах кластеру. Це зменшує затримки під час

передачі даних, оскільки кластери часто використовують спільну кеш-пам'ять або мають швидший обмін даними. Завдання плануються в межах одного кластера, а якщо навантаження не збалансоване, можливе перенесення завдань між кластерами. Також, використовуються алгоритми міграції з урахуванням афінності, які намагаються залишити завдання на тих процесорах, де вони вже виконувалися, щоб мінімізувати кеш-промахи та застосункові витрати на міграцію. Ієрархічні алгоритми об’єднують глобальні та кластерні методи. Система розглядається на кількох рівнях: на верхньому рівні міграція відбувається між кластерами; на нижньому – в межах кластеру. Така структура дозволяє досягти компромісу між гнучкістю глобальних алгоритмів та ефективністю локальних. Прикладом є алгоритм "Multilevel Queue Migration", де завдання групуються за пріоритетами, а міграція здійснюється в межах кожної черги. Гібридні алгоритми поєднують кілька підходів для досягнення балансу між ефективністю та складністю. Вони можуть динамічно перемикатися між різними стратегіями залежно від поточного стану системи.

Прикладами таких алгоритмів є міграція з урахуванням енергоспоживання, де завдання переміщуються на менш завантажені або енергоефективні процесори, і міграція з урахуванням термінів виконання, де завдання з жорсткими обмеженнями часу виконуються на більш продуктивних процесорах, а менш критичні – на енергоефективних ядрах. Одним із популярних алгоритмів є "Push-Pull Migration", де процесори періодично перевіряють навантаження один одного і, якщо різниця перевищує поріг, надлишкові завдання переміщуються до менш завантажених процесорів. Ще одним прикладом є "Hierarchical Load Balancing", який поєднує глобальне та локальне балансування, використовуючи кластери для зменшення застосункових витрат на міграцію. Алгоритм "Energy-Efficient Task Migration" враховує динамічні характеристики завдань та стан процесорів для мінімізації загального енергоспоживання системи.

Основними викликами для алгоритмів міграції у вбудованих системах є обмежені ресурси, вимоги реального часу, низьке енергоспоживання та мінімізація затримок. Ефективна оптимізація включає мінімізацію застосункових витрат на міграцію, оскільки переміщення завдань вимагає витрат часу на передачу даних і

повторну ініціалізацію середовища виконання, врахування кеш-афінності, оскільки переміщення процесів між ядрами може спричиняти кеш-промахи, що негативно впливають на продуктивність, а також адаптацію до динамічних змін в навантаженні та енергоспоживанні, використовуючи методи машинного навчання для прогнозування навантаження та динамічного налаштування параметрів планування.

Розглянемо приклад обчислень для системи з чотирма завданнями та двома процесорами. Завдання мають наступні характеристики: завдання 1 – час виконання 5, енергоспоживання 10; завдання 2 – час виконання 3, енергоспоживання 8; завдання 3 – час виконання 4, енергоспоживання 7; завдання 4 – час виконання 2, енергоспоживання 5. При використанні алгоритму збалансування навантаження завдання розподіляються рівномірно: на процесорі 1 виконуються завдання 1 та 4 із загальним часом виконання 7 та енергоспоживанням 15, а на процесорі 2 – завдання 2 та 3 із загальним часом 7 та енергоспоживанням 15. Загальний Makespan дорівнює 7, а загальне енергоспоживання – 30. Такий підхід забезпечує баланс навантаження та оптимальне енергоспоживання. Вибір алгоритму залежить від цілей оптимізації, таких як, наприклад, мінімізація енергоспоживання.

Розглянемо розробку конкретних алгоритмів міграції в багатопроцесорних вбудованих системах, враховуючи такі умови: фіксована кількість процесорів; певна топологія з’єднань між ними; обмежена кількість завдань із заданим часом виконання; фіксований квант часу для обробки завдань; можливість міграції між процесорами; певна кількість вбудованих систем, у яких здійснюється обчислювальний процес.

Опис умов.

Кількість процесорів P – фіксована, наприклад, $P = 4$.

Топологія – кільцева, де кожен процесор з'єднаний із двома сусідніми. Це забезпечує ефективну передачу завдань із мінімальними застосунковими витратами на комунікацію.

Кількість завдань N – фіксована, наприклад, $N = 8$. Кожне завдання має певний час виконання C_i та пріоритет.

Квант часу Q – фіксований, наприклад, $Q = 5$ одиниць часу.

Потенційні міграції – дозволені тільки між сусідніми процесорами відповідно до топології.

Кількість вбудованих систем S – наприклад, $S = 2$, кожна з яких має свої процесори та набір завдань.

Вибір цільової функції. Метою є мінімізація часу виконання та енергоспоживання, зберігаючи збалансоване навантаження між процесорами. Цільову функцію задамо так:

$$\min\left(\max_{p\in P} T_p\right) + \alpha \sum_{p\in P} E_p, \tag{2.3}$$

де T_p – загальний час виконання завдань на процесорі p, E_p – енергоспоживання процесора p, α – коефіцієнт, що визначає вагу енергоспоживання у цільовій функції.

Розробимо алгоритми оптимізації для різних параметрів вбудованих багатопроцесорних систем.

Алгоритм 1 враховує топологію кільце і виконує міграцію завдань між сусідніми процесорами для досягнення балансу навантаження.

Крок 1.1. Ініціалізація. Розподіл завдань між процесорами у довільному порядку.

Крок 1.2. Оцінка навантаження. Для кожного процесора p обчислюється навантаження $L_p = \sum C_i$ для всіх завдань на ньому.

Крок 1.3. Балансування. Якщо різниця навантаження між сусідніми процесорами перевищує поріг Δ, то виконується міграція завдання з більш завантаженого процесору на менш завантажений. Міграція відбувається по кільцю, зменшуючи затримки передачі.

Крок 1.4. Оновлення навантаження та повторення Кроків 2–3, поки навантаження не буде збалансоване.

Крок 1.5. Виконання завдань у межах кванту часу Q.

Крок 1.6. Перевірка завершення всіх завдань. Якщо всі виконано – завершення,

інакше повернення до Кроку 2.

Алгоритм 2 враховує пріоритети завдань та можливість міграції між сусідніми процесорами, зберігаючи афінність до процесора для зменшення кеш-промахів.

Крок 2.1. Ініціалізація. Призначення завдань процесорам за пріоритетом. Завдання з вищим пріоритетом призначаються спочатку.

Крок 2.2. Виконання завдань протягом кванту часу Q. Якщо завдання не завершено, воно залишається у черзі процесора.

Крок 2.3. Оцінка виконання. Якщо високопріоритетне завдання не може бути виконано вчасно на поточному процесорі, здійснюється міграція на сусідній процесор із меншим навантаженням.

Крок 2.4. Перевірка афінності. Міграція дозволяється лише за умови, що завдання раніше не виконувалося на обраному процесорі, щоб мінімізувати кеш-промахи.

Крок 2.5. Повторення Кроків 2–4, доки не будуть виконані всі завдання.

Алгоритм 3 спрямований на мінімізацію енергоспоживання з урахуванням топології кільця та кванту часу.

Крок 3.1. Ініціалізація. Розподіл завдань між процесорами, враховуючи їх енергоспоживання. Завдання з найменшою енергоємністю призначаються спочатку.

Крок 3.2. Оцінка енергоспоживання. Обчислення поточного енергоспоживання для кожного процесора.

Крок 3.3. Оптимізація. Якщо енергоспоживання перевищує поріг E_{max}, виконується міграція на сусідній процесор з меншим енергоспоживанням.

Крок 3.4. Виконання завдань протягом кванту часу Q.

Крок 3.5. Оновлення енергоспоживання та повторення Кроків 2–4 до завершення всіх завдань.

Приклад обчислень. Припустимо, є 4 процесори та 8 завдань з такими характеристиками:

Завдання 1: $C_1 = 4, E_1 = 3$.

Завдання 2: $C_2 = 5, E_2 = 2$.

Завдання 3: $C_3 = 3, E_3 = 4$.

Завдання 4: $C_4 = 6, E_4 = 5$.

Інші завдання мають подібні параметри.

За допомогою алгоритму 1 завдання розподіляються рівномірно по кільцю. Наприклад, Завдання 1 і 2 призначаються Процесору 1, Завдання 3 і 4 – Процесору 2 і так далі. Якщо навантаження на Процесорі 1 більше, ніж на Процесорі 2, відбувається міграція Завдання 2 на Процесор 2. Після міграції навантаження вирівнюється. Цей підхід мінімізує часове навантаження та забезпечує збалансоване навантаження між процесорами.

Застосування цих алгоритмів дозволяє досягти ефективного балансування навантаження, оптимального енергоспоживання та дотримання обмежень реального часу у багатозадачних вбудованих системах.

Алгоритми міграції в багатопроцесорних вбудованих системах знаходять широке застосування в різних галузях, де потрібно ефективно використовувати обчислювальні ресурси, мінімізувати час виконання завдань і оптимізувати енергоспоживання. Розглянемо конкретні приклади застосування для трьох розроблених алгоритмів.

Алгоритм 1 особливо ефективний у системах з кільцевою топологією, таких як мережеві маршрутизатори та мультипроцесорні обчислювальні кластери. У телекомунікаційних мережах маршрутизатори обробляють пакети даних, причому обсяг трафіку може динамічно змінюватися. В таких системах маршрутизатори часто об’єднані в кільцеву топологію для забезпечення відмовостійкості та зменшення затримок передачі даних. Коли один із маршрутизаторів отримує надмірну кількість пакетів, а сусідні маршрутизатори мають вільні обчислювальні ресурси, відбувається міграція завдань обробки пакетів на менш завантажені маршрутизатори. Це забезпечує збалансоване навантаження, мінімізує затримки в обробці даних і запобігає перевантаженню окремих маршрутизаторів. У результаті підвищується продуктивність мережі та скорочується час затримки передачі даних.

Алгоритм 2 ефективний у реальних системах із критичними часовими обмеженнями, таких як автомобільні вбудовані системи або системи управління БПЛА. У сучасних автомобілях численні електронні блоки керування відповідають

за різні завдання, такі як система гальмування, стабілізація курсу, розпізнавання об’єктів за допомогою камер тощо. Завдання мають різні пріоритети – наприклад, система гальмування має найвищий пріоритет. Якщо процесор, відповідальний за обробку сигналу гальмування, перевантажений менш пріоритетними завданнями (наприклад, розпізнаванням об’єктів), відбувається міграція менш пріоритетних завдань на сусідні процесори з меншим навантаженням. Завдання з високим пріоритетом виконуються негайно, що гарантує дотримання обмежень реального часу (наприклад, мінімальну затримку в активації гальм). Це підвищує безпеку транспортного засобу та забезпечує надійну роботу критично важливих систем.

Алгоритм підходить для вбудованих систем, де енергоспоживання має критичне значення, наприклад, у переносних пристроях або системах Інтернету речей (IoT). Розглянемо розумний годинник із кількома процесорами, які виконують різні завдання, такі як моніторинг серцевого ритму, GPS-навігація, обробка повідомлень тощо. Завдання мають різну енергоємність залежно від складності обчислень. Завдання з високим енергоспоживанням, наприклад, обробка GPS-навігації, можуть виконуватись на процесорах з більш високою ефективністю енергоспоживання. Якщо один процесор перевантажений і його енергоспоживання перевищує допустимий рівень, завдання мігрують на сусідній процесор із меншим енергоспоживанням. Це забезпечує оптимальний розподіл навантаження з мінімізацією загального енергоспоживання, що продовжує час автономної роботи пристрою.

У серверних кластерах для обробки великих обсягів даних алгоритми міграції використовуються для динамічного розподілу завдань між процесорами, щоб мінімізувати час виконання та зменшити енергоспоживання. У системах керування БПЛА алгоритм забезпечує дотримання обмежень реального часу для критичних завдань (наприклад, стабілізація польоту та уникнення перешкод), тоді як менш важливі обчислення (наприклад, обробка відео) можуть мігрувати на інші процесори.

Вбудовані системи у медичних пристроях (наприклад, кардіостимулятори) можуть використовувати алгоритм 3 для забезпечення безперервної роботи в умовах обмеженого енергоспоживання, мігруючи обчислювальні завдання між процесорами для оптимізації енерговитрат.

Для того, щоб досягти адаптивності системи за допомогою міграції процесів також необхідно визначити, як виконати перехід між поточним відображенням і наступним. Тобто потрібно передбачити механізм для виконання міграції процесів. У розроблюваному підході цей механізм реалізується компонентом проміжного програмного забезпечення, а саме міграцією процесів. Розглянемо проблему визначення та впровадження механізму міграції процесів, орієнтованого на процеси, який задовольняє наступним трьом вимогам: якщо процес міграції запущений, то він повинен бути завершений в певні, відомі терміни, тобто це властивість передбачуваності; міграція процесу може бути запущена в системі в будь-який момент, тобто це вимога охопити сценарій, у якому потрібна міграція процесів у відповідь на апаратний збій, момент виникнення якого невідомий. Код, який використовується для міграції процесів, має бути згенерований автоматично, без ручного втручання розробника. Це потрібно, щоб звільнити розробників від трудомісткого та схильного до помилок завдання вставки коду, необхідного для міграції завдань вручну.

Передбачуваний механізм міграції процесів дозволяє перерозподіляти процеси під час виконання між процесорами, що є фундаментальною вимогою для адаптивності системи. Особливість рішення є те, що, використовуючи операційну семантику і структуру процесу, міграція може фактично початися в будь-який момент виконання основного процесу без необхідності переміщення великого стану. Крім того, верхню межу застосункових витрат на міграцію процесу можна знайти на основі топології та розмірів буфера. Нарешті, код, який використовується для міграції процесів, є мінімально інвазивним щодо оригінальної структури коду і може бути згенерований повністю автоматизованим способом.

Управління ресурсами під час виконання є широко вивченою темою в плануванні розподілених систем загального призначення. Зокрема, в цьому контексті потрібно розробити та оцінити механізми міграції процесів, що дозволяють здійснювати динамічний розподіл навантаження, стійкість до відмов, а також покращувати адміністрування системи та локальність доступу до даних. В останні роки все більшої популярності набирає управління часом виконання і знаходить

застосування також і в багатопроцесорних вбудованих системах. Ця сфера накладає жорсткі обмеження, такі як вартість, потужність і передбачуваність, які повинні ретельно враховувати механізми управління часом виконання та міграції процесів.

Цілі масштабованості та адаптивності системи досягаються за рахунок політики розподіленої міграції завдань по мультипроцесору з чисто розподіленою пам'яттю. Їхня платформа програмується за допомогою процесу мережевого процесору. Однак реальна міграція завдання може відбуватися тільки в фіксованих точках, які відповідають повідомленням примітивних викликів. Мігрувати завдання може на будь-якому етапі виконання основного корпусу процесів. Це призводить до швидшого часу реакції на рішення про міграцію, що краще, наприклад, у разі апаратних збоїв. Динамічне перевідображення завдань досягається на рівні користувача або проміжного програмного забезпечення ОС відповідно. В обох цих підходах користувачеві потрібно визначити в коді контрольні точки, де може відбутися міграція. Це може вимагати від розробника послідовних ручних зусиль. Крім того, суттєвою відмінністю є реалізація комунікації між завданнями, яка використовує систему спільної пам'яті. Шаблон апаратної та програмної архітектури дозволяє системі змінювати відображення завдань застосунків під час виконання. Тобто, процес може зчитувати токени даних безпосередньо з пам'яті іншого процесора. Або, навпаки, він використовує повністю розподілені засоби з метою забезпечення кращої масштабованості. Механізм міграції процесів показує, як програмні процедури самотестування, здатні виявляти несправності, можуть бути поєднані з механізмом міграції. У разі несправності завдання міграції обладнання відповідає за вилучення стану процесів із несправної вбудованої системи, щоб зробити її доступною для менеджера ресурсів. Механізм міграфії пропонує альтернативний спосіб вирішення несправностей на рівні застосування. Коли процес переривається через помилку, виконання відновлюється на іншому процесорі шляхом відкату до початку перерваної ітерації процесу. Інший спосіб відновлення несправностей виконання на іншому процесорі здійснюється шляхом перекочування вперед до ітерації процесу, яка слідує за перерваною ітерацією. Це призводить до простішої апаратної реалізації перенесення завдань, хоча вивід програми може бути

тимчасово неправильним.

Запропонований підхід до міграції буде з повністю розподіленою пам'яттю і без прямого віддаленого доступу до пам'яті. Це означає, що обробний елемент завдання може безпосередньо звертатися тільки до вмісту власної локальної пам'яті. Уся комунікація та синхронізація між процесами, відображеними на різних процесорах, може відбуватися лише за допомогою повідомлень. Підхід до реалізації адаптивності системи полягає в розгортанні процесів застосунків, змодельованих на їх перерозподіл часу виконання адаптувати систему до мінливих умов роботи, таких як варіації вимог до якості обслуговування, доступність ресурсів або обмеження бюджету потужності. Зокрема, адаптивність системи підтримується використанням спеціального проміжного програмного забезпечення, виділеного в програмному стеку. У верхній частині програмного стека застосунки задаються як набір процесів, які реалізуються у вигляді окремих потоків. Приклад потоку, що представляє процес. Базова структура процесів буде змінена для полегшення реалізації передбачуваного механізму міграції процесів. У нижній частині стеку програмного забезпечення операційна система відповідає за всі види управління процесами (створення, видалення, встановлення його пріоритету, призупинення або відновлення). Ці функції є важливими для керування часом виконання системи, і, зокрема, для виконання міграції процесів. Більше того, кожен процесор має можливості багатозадачності завдяки ОС. У випадку відображення «багато-до-одного», тобто коли на одному елементі обробки відображено більше одного процесу, то планування керується даними. Це означає, що процес продовжує виконувати послідовні ітерації до тих пір, поки він не заблокується при читанні або записі. Коли процес блокується, то він передає керування процесором наступному процесу в черзі готовності за круговою системою. У проміжку між застосунками та операційною системою буде проміжне програмне забезпечення, яке складається з двох основних компонентів. Перший – це зв'язок, який реалізує зв'язок і синхронізацію між процесами, розташованими в окремих процесорах. Обов’язкова складова – це міграція процесів, яка в основному відповідає за такі дії, що виконуються під час міграції процесів: координує створення та видалення процесів між різними процесорми; гарантує правильну передачу стану

процесу під час міграції процесу.

Таким чином, розроблено алгоритми міграції процесів в багатозадачних вбудованих система, застосування яких дозволяє досягти ефективного балансування навантаження, оптимального енергоспоживання та дотримання обмежень реального часу у багатозадачних вбудованих системах.

3.2 Схеми міграції процесів в багатозадачних вбудованих системах

Розглянемо розроблюваний механізм для виконання міграції процесів через багатопроцесорні вбудовані системи. Ця схема базується на проміжному програмному забезпеченні і вона визначатиме, як виконувати перепризначення процесів під час виконання. Це, в свою чергу, дозволяє проектувальникам реалізовувати стратегії адаптивності системи. Механізм міграції залежить від розглянутого підходу до комунікації. Для розробки механізму міграції розглядаємо підхід до комунікації на основі запиту. Такий вибір зроблений тому, що він призводить до значно простішої реалізації механізму міграції, оскільки вимагає меншої кількості точок синхронізації. У той же час він забезпечує продуктивність, порівняну з іншими підходами для застосунків, що домінують в обчисленнях.

Для врахування перерозподілу процесів під час виконання, кожен процесор у багатопроцесорній системі зберігає у своїй локальній пам'яті таблицю проміжного програмного забезпечення. Ця таблиця відіграє ключову роль у перетворенні загальних примітивів зв’язку на специфічні для апаратної платформи примітиви, забезпечуючи ефективну маршрутизацію повідомлень до необхідних процесорів у системі. Такий підхід дозволяє динамічно адаптувати комунікацію між процесами до поточного стану системи, що особливо важливо в умовах нерівномірного навантаження або під час міграції процесів.

Для кожного каналу зв’язку визначаються процеси, які виконують ролі виробника та споживача, а також відслідковується поточне відображення цих процесів у системі. Це дозволяє підтримувати узгодженість даних під час переміщення процесів між процесорами. Окрім того, у таблиці зберігається

допоміжна інформація, наприклад, запити, які очікують на обробку під час ініціації міграції. Такий підхід допомагає забезпечити цілісність обчислень, мінімізуючи ризики втрати або дублювання повідомлень, особливо у випадках, коли міграція відбувається під час активного обміну даними.

Міграція процесів є складним завданням, яке вимагає продуманого підходу до управління станами процесів. Виділяють два основні механізми міграції: реплікація процесу та відтворення процесу. Реплікація процесу передбачає копіювання програмного коду процесу на всі процесори, які потенційно можуть його виконувати. Це дозволяє швидко перемикати виконання процесу між процесорами, оскільки код уже завантажений і готовий до запуску. Коли процес необхідно перенести, він призупиняється на поточному процесорі, а на іншому процесорі запускається його копія з тим самим станом. Для цього стан процесу копіюється з одного процесора на інший, забезпечуючи безперервність обчислень.

Другий механізм – відтворення процесу, який передбачає припинення виконання процесу на одному процесорі та його створення на іншому шляхом передачі коду і стану. Це вимагає підтримки динамічного завантаження процесів операційною системою, що ускладнює реалізацію, але забезпечує оптимальніше використання пам'яті, оскільки в системі в кожний момент часу існує лише один екземпляр коду процесу. Таким чином, відтворення процесу є ефективнішим з точки зору пам'яті, але вимагає більших витрат на передачу коду і стану під час міграції.

Порівнюючи обидва підходи, реплікація процесу менш ефективна у використанні пам'яті, оскільки однаковий код зберігається на кількох процесорах одночасно. Проте вона має перевагу у швидкості міграції, оскільки не потребує додаткових витрат на динамічне завантаження коду. У випадках, коли обмеження пам'яті не є критичним фактором, реплікація виявляється більш привабливим рішенням. Саме тому для запропонованого механізму міграції процесів обирається реплікація, оскільки вона гарантує швидке завершення процедури міграції, що особливо важливо в системах реального часу або в умовах високої динамічності навантаження.

У процедурі міграції беруть участь кілька типів процесорів, кожен з яких

виконує певну роль. Процесор джерела – це процесор, який запускає процес до моменту міграції. Процесор призначення – той, що виконуватиме процес після міграції. Процесори попередника виконують процеси, які передують мігруючому процесу, а процесор наступника відповідає за виконання процесів, які слідують за ним. Така диференціація дозволяє чітко визначити зв’язки між процесами під час міграції, мінімізуючи ризик порушення порядку виконання команд.

Для реалізації ефективної міграції структура процесів була модифікована таким чином, щоб забезпечити можливість перенесення в будь-який момент виконання основних команд. Це означає, що міграція може відбуватися майже у будь-який час під час виконання процесу, що значно зменшує затримку між ініціацією та завершенням міграції. Важливо зазначити, що для досягнення такої гнучкості потрібно забезпечити можливість безперервного збереження та відновлення стану процесу, включно з вмістом регістрів, стеку викликів та вказівників команд.

Крім того, розроблюваний механізм міграції процесів враховує топологію багатопроцесорної системи. Це означає, що під час вибору процесора призначення береться до уваги не лише поточне завантаження процесорів, а й їхнє взаємне розташування, пропускна здатність каналів зв’язку та затримки передачі повідомлень. Такий підхід дозволяє мінімізувати час очікування на обмін даними між процесами, які залишаються на різних процесорах після міграції.

Щоб забезпечити можливість міграції у будь-який момент, процесори були розширені функціоналом для генерації переривань під час надходження повідомлень із певними тегами. Це розширення дозволяє уникнути опитування команд міграції в коді, яке могло б спричинити небажані затримки. Завдяки цьому процедура міграції стає більш динамічною і ефективною, оскільки перемикання між процесорами відбувається практично миттєво після отримання відповідного повідомлення.

Таким чином, запропонований підхід до міграції процесів, заснований на реплікації процесів та динамічній маршрутизації повідомлень із врахуванням топології системи, забезпечує високу ефективність виконання програм. Він дозволяє мінімізувати затримки під час міграції, зберігаючи узгодженість даних між процесами, а також адаптувати обчислювальні ресурси до поточного стану

навантаження системи.

Механізм реплікації є одним із підходів до міграції завдань у багатопроцесорних системах, який забезпечує гнучке і швидке перенесення процесів між процесорами. Суть цього методу полягає в тому, що програмний код процесу заздалегідь копіюється на всі процесори, які потенційно можуть виконувати цей процес. Таким чином, у момент потреби міграції процес не переноситься у традиційному розумінні, а просто призупиняється на одному процесорі і відновлюється на іншому, оскільки необхідний код вже є в локальній пам'яті нового процесора. Це дозволяє значно скоротити час на передачу даних і забезпечити швидке продовження виконання процесу.

Алгоритм реплікації передбачає кілька ключових етапів. Першим етапом є ініціалізація процесу, під час якої код процесу копіюється на всі процесори, які можуть його виконувати. Така копія коду називається реплікою. Важливо зазначити, що на цьому етапі копіюється лише код, а не стан процесу, оскільки стан динамічно змінюється під час виконання. Кожен процесор зберігає у своїй локальній пам'яті копію коду процесу, а також таблицю, яка містить інформацію про стан виконання процесу на кожному з процесорів. Ця таблиця використовується для відслідковування того, на якому процесорі в даний момент активно виконується процес.

Другим етапом є виконання процесу. Процес запускається на одному з процесорів, який визначається на основі поточного навантаження системи та інших параметрів, таких як топологія системи і доступність ресурсів. Під час виконання процесу інформація про його стан постійно оновлюється в локальній таблиці цього процесора. Стан включає значення регістрів, вказівник команд, вміст стеку викликів, а також локальні змінні та буфери обміну повідомленнями. Крім того, стан процесу періодично зберігається у глобальному сховищі станів, яке доступне всім процесорам. Це дозволяє забезпечити узгодженість даних під час міграції процесу.

Третім етапом є ініціалізація міграції, яка може бути викликана кількома причинами, зокрема нерівномірним навантаженням процесорів, змінами в топології системи або зовнішніми подіями, такими як збій обладнання. У момент прийняття рішення про міграцію процес призупиняється на поточному процесорі. При цьому

зберігається повний стан процесу, включаючи вміст регістрів і стеку викликів, у глобальному сховищі станів. Після цього вибирається новий процесор для виконання процесу. Вибір процесора здійснюється на основі аналізу поточного навантаження, топології системи і локальних ресурсів, таких як обсяг вільної пам'яті та доступність обчислювальних ядер.

Четвертий етап полягає у відновленні процесу на новому процесорі. Оскільки копія коду процесу вже є на цьому процесорі, немає потреби передавати код. Процес просто відновлюється зі стану, який був збережений у глобальному сховищі станів. Вказівник команд встановлюється на останню виконану команду, а регістри та стек викликів відновлюються до значень, що були на момент призупинення. Після цього процес продовжує виконання з тієї самої точки, на якій він був призупинений, забезпечуючи цілісність обчислень і узгодженість даних.

П'ятим етапом є оновлення таблиці станів. Після успішного відновлення процесу на новому процесорі локальні таблиці на всіх процесорах оновлюються таким чином, щоб відобразити нове місце виконання процесу. Це забезпечує правильну маршрутизацію повідомлень до процесу після міграції, оскільки система тепер знає, на якому процесорі знаходиться активний екземпляр процесу. Крім того, оновлюється глобальне сховище станів, щоб забезпечити узгодженість даних у всій системі.

Механізм реплікації має кілька ключових переваг. По-перше, він забезпечує швидку міграцію процесів, оскільки немає потреби передавати код процесу між процесорами. По-друге, він мінімізує затримки під час міграції, оскільки передається лише стан процесу, а не весь код. По-третє, цей підхід дозволяє зменшити навантаження на канали зв'язку, оскільки обмін даними обмежується лише оновленням станів. По-четверте, реплікація забезпечує високу гнучкість у виборі процесора для виконання процесу, оскільки код є доступним на кількох процесорах одночасно.

Однак механізм реплікації має і певні недоліки. Основним з них є неефективне використання пам'яті, оскільки однаковий код зберігається на кількох процесорах. Це може стати проблемою в системах з обмеженою пам'яттю або при великій кількості

процесів. Крім того, необхідно забезпечити узгодженість станів між процесорами, що вимагає використання глобального сховища станів і додаткових алгоритмів синхронізації. Також існує ризик виникнення станів гонки або патових ситуацій у разі одночасного доступу до глобального сховища станів кількома процесорами.

Таким чином, механізм реплікації як підхід до міграції завдань у багатопроцесорних системах є ефективним рішенням для систем, де обмеження пам'яті не є критичним фактором, а швидкість міграції і мінімізація затримок є пріоритетом. Він забезпечує гнучке і швидке перенесення процесів між процесорами, дозволяючи динамічно перерозподіляти обчислювальні ресурси відповідно до поточного навантаження системи.

Щоб виконати таке відображення, примітиви зв'язку повинні бути перетворені на набір примітивів зв'язку платформи виконання. Типовими примітивами зв'язку та синхронізації платформи виконання є наступні: перевірка доступності токенів даних у розглянутому буфері; перевірка наявності вільного місця у розглянутому буфері; передача токену з розглянутого буфера в локальний простір процесу; зберігання даних та передача токенів з локального простору процесу до розглянутого буфера; повідомлення про доступність буфера.

Щоб отримати ефективне відображення процесів на платформу виконання, архітектори повинні враховувати структуру самої платформи виконання. Наприклад, важливо знати розташування буферів, які реалізують канали певного процесу, у спільній пам'яті чи в локальній пам'яті, який виконує цей процес.

Кожен процес може звертатися тільки до локальної пам'яті свого процесору. Передача токенів між процесорами виконання обробляється за допомогою підходу проміжного програмного забезпечення, керованого запитами. Таким чином, кожен процес зчитує та записує токени лише у свою локальну пам'ять. Використовуючи наведене перетворення примітивів зв'язку, отримаємо імплементацію процесу на розглянуту платформу виконання. Для того, щоб міграція могла відбутися в будь-який момент виконання основного тіла процесів, потрібна модифікація структури процесу. Для підтримки правильної функціональності програми стан всього процесу має бути однаковим до та після міграції процесу. Поділимо стан системи на дві

складові, таким чином: стан процесів для якого єдиним внутрішнім станом процесу є вектор ітерації, який представляє значення змінних ітератора циклу; стан каналів, який представлений токенами, що в даний момент зберігаються в буферах і реалізують цей канал зв'язку. Щодо другого компонента стану, то відомо, що канал фактично реалізується двома буферами у підході до комунікації, керованої запитами. Один з цих буферів знаходиться на процесорі, на якій відображено виробника, тоді як інший знаходиться на процесорі, в який відображено споживача. Таким чином, під час міграції процесу з його вихідної процесору на цільову процесор потрібно перенести два компоненти стану. Визначивши стан необхідно і передати його під час міграції процесу. Якщо перевірка для стану позитивна, то це означає, що була виконана міграція, тому стан процесу перезавантажується. Обидва компоненти стану передаються з процесору джерела до процесору призначення під час міграції. Якщо прапорець міграції хибний, то це означає, що процес починається з нуля, з порожніми вхідними та вихідними даними. Процеси можуть відрізнятися від базової відображеної структури процесу тому, що ітератори всередині циклів не розпочинаються з нуля в разі міграції. Замість цього вони починаються зі цілочисельних значень, які представляють ітерацію, на якій процес був перерваний міграцією під час роботи на процесорі джерела. Після першого повного виконання внутрішнього циклу значення встановлюється в нуль, щоб наступне виконання внутрішнього циклу починалося коректно. Крім того, порядок виконання примітивів зв'язку та синхронізації відрізняється від того, що використовується в базовій структурі відображеного процесу. Фактично, примітиви платформи виконання, які реалізують примітив, не виконуються в безперервній послідовності. Замість цього вони виконуються в рядках з декількома операціями. Правила переупорядкування зберігають правильність виконання відображеного процесу. Міграційна структура процесу відповідає правилам. Фактичне звільнення ділянок пам'яті виконується операцією, яка споживає токен даних шляхом збільшення покажчика зчитування. Ця операція відбувається тільки за межами основного корпусу. Потім, якщо міграція спрацьовує перед операцією, то може бути коректно відновлений на процесорі призначення, оскільки він знову прочитає той самий вхідний токен, оскільки

покажчик зчитування залишається незмінним. Аналогічно, операція, яка завершує примітив, виконується в кінці відображеного процесу за межами його основного тіла. Завершення операцій в кінці ітерації дозволяє правильно здійснити міграцію процесу в будь-якому місці. Зауважимо, що у випадку декількох вхідних або вихідних каналів, операції всіх каналів виконуються разом відразу після основного тіла процесу, щоб оновити стан буферів в найкоротші терміни.

Міграція процесів не може відбутися в перенесеному процесі в межах рядків, оскільки це призведе до неузгодженості стану. Це пов'язано з тим, що рядки можна розглядати як оновлення стану вихідної і вхідної стратегії планування, що представляє оновлення стану. Якщо, наприклад, міграція відбувається після оновлення стану, але до оновлення набору ітераторів, відбудеться наступний сценарій: стан вхідного та вихідного пристроїв, підключених до центрального процесору, змінюється так, ніби поточна ітерація була успішно завершена; процес перезапускає поточну ітерацію з початку, оскільки вектор ітерації не був оновлений відповідним чином. Такий стан неодмінно викличе глухий кут. Хоча міграція процесів не може відбуватися в межах рядків, але ці рядки коду представляють мінімальну частину виконання процесу, оскільки виконання операцій і оновлення набору ітераторів є питанням кількох простих інструкцій. Тому вимкнення міграції в межах цих розділів суттєво не збільшує затримку міграції.

Принцип, що лежить в основі запропонованої структури міграційного процесу, полягає в тому, що стан процесу має бути послідовним і актуальним на момент виконання міграції. Це дає змогу системі коректно відновити своє виконання, а перенесений процес відображатиметься на процесорі призначення. Використовуючи структуру процесу, такий підхід не вимагає від розробника вказівки контексту, який повинен бути переданий при міграції,. Це навантаження не переноситься на рівень ОС чи проміжного програмного забезпечення. Визначення стану для міграції не потрібне, оскільки стан просто складається з двох компонентів.

Крім того, розроблений підхід не потребує згенерованих розробниками пунктів пропуску/міграції. Менеджер ресурсів може перервати виконання процесу в будь-який момент часу під час виконання основного тіла процесу. Після цього перенесений

процес відновить своє виконання з початку перерваної ітерації. З одного боку, це має на увазі, що якщо міграція спрацьовує в середині виконання функції, то втрачається час, витрачений на обчислення з моменту початку ітерації. З іншого боку, такий підхід призводить до більш ефективної реалізації та передбачуваного часу реагування на міграцію, що вважається більш важливим для таких цілей.

Механізм міграції вимагає дій з усіх процесорів. Менеджер ресурсів відповідає за прийняття рішення про міграцію. Менеджер ресурсів приймає рішення згідно механізму міграції процесу. Коли диспетчер ресурсів приймає рішення про міграцію, то він ініціює міграцію, надсилаючи певне контрольне повідомлення на процесор джерела. Потім процесор джерела пересилає це контрольне повідомлення на процесору призначення, попередника та наступника, щоб повідомити їх про те, що процедуру міграції розпочато. Керуючі повідомлення, які сповіщають задіяні процесору про початок міграції процесу, містять ідентифікатор перенесеного процесу та нове відображення цього процесу. На всіх задіяних процесорх, а також у менеджері ресурсів, таблиці проміжного програмного забезпечення потім оновлюються з урахуванням нового відображення перенесеного процесу.

Поведінка вихідного процесору залежить від функціональності процесорі або його несправності. Якщо вихідний процесор функціонує, то процес міграції зупиняється, а два компоненти стану переміщуються до цільового процесору. Ці компоненти стану передаються за допомогою спеціальних повідомлень. Крім того, проміжне програмне забезпечення оновлюється. Вихідний процесор також дбає про поширення рішення про міграцію на інші процесори, що беруть участь у процедурі міграції. У випадку, якщо вихідний процесор несправний, то дії емулюються спеціальним апаратним IP. Процесор призначення отримує конкретне повідомлення для активації процесу. Процедура міграції здійснюється шляхом створення необхідних програмних каналів та активації репліки перенесеного процесу за допомогою відповідного виклику ОС. Перед запуском репліки процесу прапорець міграції встановлюється на одиницю, щоб стан перенесеного процесу відновився. Звідси випливає, що вхідні та вихідні канали підключені до перенесеного процесу, копіюються, а набір ітераторів встановлюються так, що виконання розпочинається з

того місця, де воно було призупинено на процесорі джерела. На цих процесорх єдиним необхідним кроком є оновлення таблиць проміжного програмного забезпечення відповідно до нового відображення перенесеного процесу. Таким чином, нові токени, призначені для перенесеного процесу будуть відправлені на процесор призначення.

Таким чином, розроблено схему механізму для виконання міграції процесів через багатопроцесорні вбудовані системи, яка базується на проміжному програмному забезпеченні і вона визначатиме, як виконувати перепризначення процесів під час виконання. Це дасть змогу проектувальникам реалізовувати стратегії адаптивності системи.

3.3 Висновки до третього розділу

Розроблено алгоритми міграції процесів в багатозадачних вбудованих система, застосування яких дозволяє досягти ефективного балансування навантаження, оптимального енергоспоживання та дотримання обмежень реального часу у багатозадачних вбудованих системах.

Схема для виконання міграції процесів через багатопроцесорні вбудовані системи базується на проміжному програмному забезпеченні і вона визначатиме, як виконувати перепризначення процесів під час виконання.

4 МЕТОД ОПТИМІЗАЦІЇ ЗАВДАНЬ В БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМАХ

4.1 Метод оптимізації завдань у вбудованих системах з багатьма процесорами

Метод оптимізації завдань базуватимемо на реплікації при міграції завдань у багатопроцесорній системі і оптимізацію будемо здійснювати за допомогою розробленої цільової функції оптимізації, яка враховує основні параметри системи, такі як час виконання завдань, завантаження процесорів, обсяг використаної пам’яті та затримки в обміні повідомленнями. Оптимізація виконання завдань за допомогою методу реплікації передбачає знаходження такого розподілу процесів між процесорами, який мінімізує загальний час виконання завдань при одночасному збереженні балансу навантаження на процесори та мінімізації витрат на комунікацію між ними.

Цільову функція оптимізації задамо так:

$$F = \alpha \sum_{i=1}^{N} T_i + \beta \sum_{j=1}^{M} L_j + \gamma \sum_{k=1}^{P} C_k, \quad (4.1)$$

де T_i — час виконання i-го завдання, L_j — завантаження j-го процесора, C_k — затримка в обміні повідомленнями між процесорами; а α, β та γ — вагові коефіцієнти, які відображають пріоритети оптимізації.

Перший компонент цільової функції спрямований на мінімізацію загального часу виконання всіх завдань. Для цього необхідно забезпечити, щоб завдання виконувалися максимально паралельно і не очікували на ресурси. Метод реплікації забезпечує швидке перемикання виконання завдань між процесорами, оскільки копії коду завдань уже знаходяться в локальній пам'яті всіх потенційних процесорів. Оптимізація цього компонента передбачає динамічне перенесення завдань на найменш завантажені процесори, а також мінімізацію простою завдань.

Другий компонент цільової функції відповідає за баланс навантаження між процесорами. Нерівномірний розподіл завдань може призвести до того, що одні процесори будуть перевантажені, а інші простоюватимуть. Для вирішення цієї

проблеми використовується стратегія динамічного балансування навантаження, яка передбачає регулярну оцінку завантаження процесорів і переміщення завдань з перевантажених процесорів на менш завантажені. Метод реплікації значно спрощує цей процес, оскільки код завдань вже доступний на всіх процесорах, а отже, необхідно лише передати стан завдання. Це зменшує час на перенесення завдань і забезпечує швидке відновлення їх виконання.

Третій компонент цільової функції мінімізує затримки в обміні повідомленнями між процесорами. Це особливо важливо для завдань, які активно взаємодіють між собою. Для цього потрібно розміщувати взаємопов'язані завдання на процесорах, які мають мінімальні затримки зв'язку. Оптимізація цього компонента передбачає аналіз топології системи та розташування процесорів, а також обчислення комунікаційних витрат між процесорами. Для мінімізації затримок використовуються графи залежностей між завданнями, які відображають обсяги та частоту обміну повідомленнями. Завдання, що часто обмінюються даними, розміщуються на сусідніх процесорах з мінімальною затримкою зв’язку.

Метод оптимізації для реалізації цільової функції включає кілька етапів. Першим етапом є початкове розподілення завдань між процесорами. Це може бути виконано за допомогою жадібного алгоритму, який послідовно призначає завдання найменш завантаженому процесору. Другим етапом є динамічне перенесення завдань під час виконання, яке здійснюється на основі аналізу поточного навантаження на процесори та обсягу комунікацій між ними. Якщо виявлено, що процесор перевантажений, частина завдань переноситься на інші процесори з урахуванням мінімізації затримок зв'язку. Третім етапом є періодична оптимізація розподілу завдань шляхом перебору можливих комбінацій розміщення завдань і вибору тієї, яка мінімізує цільову функцію.

Для пошуку оптимального розподілу завдань використовується комбінаторна оптимізація, оскільки кількість можливих розподілів завдань між процесорами зростає експоненціально зі збільшенням кількості процесорів та завдань. Ефективним підходом є використання генетичних алгоритмів або методу імітації відпалу, які забезпечують швидкий пошук оптимального рішення в просторі можливих

розподілів. Генетичний алгоритм моделює процес природного відбору, використовуючи операції кросоверу та мутації для створення нових рішень на основі поточних. Метод імітації відпалу здійснює випадкові зміни в поточному розподілі завдань, поступово зменшуючи ймовірність прийняття гірших рішень, що дозволяє уникнути локальних мінімумів цільової функції.

Таким чином, метод реплікації в поєднанні з цільовою функцією оптимізації забезпечує ефективне виконання завдань у багатопроцесорній системі, мінімізуючи загальний час виконання, забезпечуючи баланс навантаження між процесорами і знижуючи затримки в обміні повідомленнями. Такий підхід є особливо ефективним для систем із високою динамікою навантаження та складною топологією обчислювальних вузлів.

Метод оптимізації виконання завдань у багатопроцесорній системі з використанням реплікації та цільової функції можна подати у вигляді деталізованих кроків. Цей підхід дозволяє динамічно перерозподіляти завдання між процесорами з метою мінімізації загального часу виконання, балансування навантаження та мінімізації затримок в обміні повідомленнями. Кожен крок цього методу спрямований на досягнення оптимального розподілу завдань у системі, зберігаючи гнучкість і швидкість процесу міграції завдяки попередньо створеним реплікам коду завдань.

Крок 1. Ініціалізація системи та створення реплік коду завдань
На першому етапі виконуються такі дії: аналізується загальна кількість завдань та обчислювальних ресурсів системи, включаючи кількість процесорів та їх потужність; для кожного завдання створюються репліки коду на всіх процесорах, які потенційно можуть виконувати це завдання, що забезпечує готовність процесорів до прийняття завдань без додаткових витрат часу на передачу коду; створюється глобальна таблиця станів, яка зберігає інформацію про стан виконання кожного завдання на всіх процесорах, а також локальні таблиці на кожному процесорі для відстеження місця виконання завдань. Крім того, ініціалізується цільова функція оптимізації, значення якої обчислюється за формулою (4.2).

Крок 2. Початковий розподіл завдань. Виконується початковий розподіл

завдань між процесорами на основі жадібного алгоритму. Завдання призначаються тим процесорам, які мають найменше поточне завантаження. Враховується топологія системи для мінімізації затримок зв’язку між процесорами. Завдання, що мають високий рівень взаємодії між собою, розміщуються на сусідніх процесорах. Обчислюється цільова функція для початкового розподілу завдань, яка включає час виконання, завантаження процесорів та затримки зв’язку. Початкове розміщення фіксується в глобальній таблиці станів.

Крок 3. Виконання завдань та моніторинг стану системи. Завдання запускаються на відповідних процесорах, які були обрані на попередньому кроці. В процесі виконання стан кожного завдання регулярно зберігається у глобальному сховищі станів, включаючи значення регістрів, вказівник команд, стек викликів та локальні змінні. Відстежується навантаження на кожен процесор та обсяги обміну повідомленнями між процесорами. Проводиться регулярний моніторинг значень цільової функції для оцінки ефективності поточного розподілу завдань.

Крок 4. Прийняття рішення про міграцію завдань. Якщо цільова функція перевищує встановлені порогові значення, ініціюється процес міграції. Виконується аналіз поточного навантаження на кожен процесор, а також затримок в обміні повідомленнями. Виявляються «вузькі місця», такі як перевантажені процесори або висока затримка зв’язку між завданнями, що активно обмінюються даними. Вибираються завдання для міграції, зокрема ті, які мають низьку локальність даних або генерують значний обсяг повідомлень до віддалених процесорів.

Крок 5. Вибір цільового процесора для міграції. Виконується оцінка всіх потенційних процесорів для виконання завдання, що мігрує, на основі наступних критеріїв: поточне навантаження на процесор; затримки зв’язку з іншими процесорами, які виконують взаємопов’язані завдання; обсяг вільної пам’яті та доступність обчислювальних ядер; обирається процесор, який мінімізує цільову функцію, забезпечуючи баланс між навантаженням та комунікаційними витратами; виконується бронювання ресурсів на обраному процесорі для уникнення конфліктів під час міграції.

Крок 6. Виконання міграції завдання. Завдання призупиняється на поточному

процесорі з повним збереженням його стану в глобальному сховищі станів. Оскільки код завдання вже є на новому процесорі, передається лише стан завдання. На цільовому процесорі завдання відновлюється зі збереженого стану. Оновлюється локальна таблиця станів на обох процесорах, а також глобальна таблиця станів для відображення нового розташування завдання. Завдання продовжує виконання з тієї ж точки, на якій воно було призупинено.

Крок 7. Оновлення цільової функції та оптимізація. Після виконання міграції цільова функція перераховується з урахуванням нового розподілу завдань. Якщо цільова функція не досягла мінімального значення, повторюється процес міграції для інших завдань. Для оптимізації використовується один із методів: генетичний алгоритм виконує кросовер і мутації для створення нових комбінацій розподілу завдань; метод імітації відпалу здійснює випадкові зміни в розподілі завдань з поступовим зниженням ймовірності прийняття гірших рішень. Процес оптимізації продовжується доти, доки цільова функція не досягне заданого мінімуму або не буде досягнуто стійкого розподілу завдань.

Крок 8. Завершення оптимізації та підтримка балансу. Оптимізоване розподілення фіксується в глобальній таблиці станів. Підтримується динамічний баланс навантаження шляхом періодичного оновлення цільової функції та переналаштування розподілу завдань. Забезпечується узгодженість даних та синхронізація станів між процесорами.

Таким чином, деталізований метод оптимізації виконання завдань з використанням реплікації у багатопроцесорній системі дозволяє ефективно мінімізувати загальний час виконання, забезпечити баланс навантаження і мінімізувати затримки зв’язку. Особливістю методу є реалізація міграції завдань згідно реплікації з використанням цільової функції оптимізації.

4.2 Дослідження ефективності методу оптимізації завдань в багатопроцесорних вбудованих системх

Для реалізації методу оптимізації виконання завдань з використанням

реплікації та цільової функції оптимізації у багатопроцесорній вбудованій системі розроблено програму на C++. Програма враховує:

1) кількість процесорів та завдань;

2) схему міграції (на основі цільової функції);

3) стан процесів (виконуються чи очікують);

4) типи завдань (довгі чи короткі).

Програма реалізує:

1) ініціалізацію системи з заданою кількістю процесорів і завдань;

2) початковий розподіл завдань між процесорами;

3) моніторинг стану системи та динамічну міграцію завдань для оптимізації цільової функції.

Програму реалізує метод оптимізації виконання завдань із використанням реплікації та цільової функції оптимізації в багатопроцесорній вбудованій системі. Основні особливості програми:

1) ініціалізація системи з випадковими завданнями (короткі або довгі);

2) початковий розподіл завдань між процесорами на основі їх навантаження;

3) lинамічна оптимізація з міграцією завдань для балансування навантаження;

4) вивід процесу міграції в консолі.

З вокристанням розробленої програми було проведено першу серію експерименту. На рис. 4.1 зображено графіки функцій основних параметрів оптимізації для простого випадку з процесами і пропесорами, тобто коли немає підвищеного рівня завантаженості багатопроцесорної вбудованої системи.

Результати роботи програми:

1) початковий стан навантаження процесорів - [12, 14, 14, 11];

2) кінцевий стан після оптимізації - [12, 14, 14, 11].

Графік показує динаміку навантаження процесорів під час оптимізації. Значного перерозподілу не відбулося, оскільки початковий розподіл був відносно збалансованим.

На рис. 4.2 зображено графіки функцій основних параметрів оптимізації для простого випадку з процесами і процесорами, тобто коли немає підвищеного рівня

завантаженості багатопроцесорної вбудованої системи. Виділено на відміну від першої серії експерименту динаміку навантаження процесорів та динаміку витрат енергії процесорів.

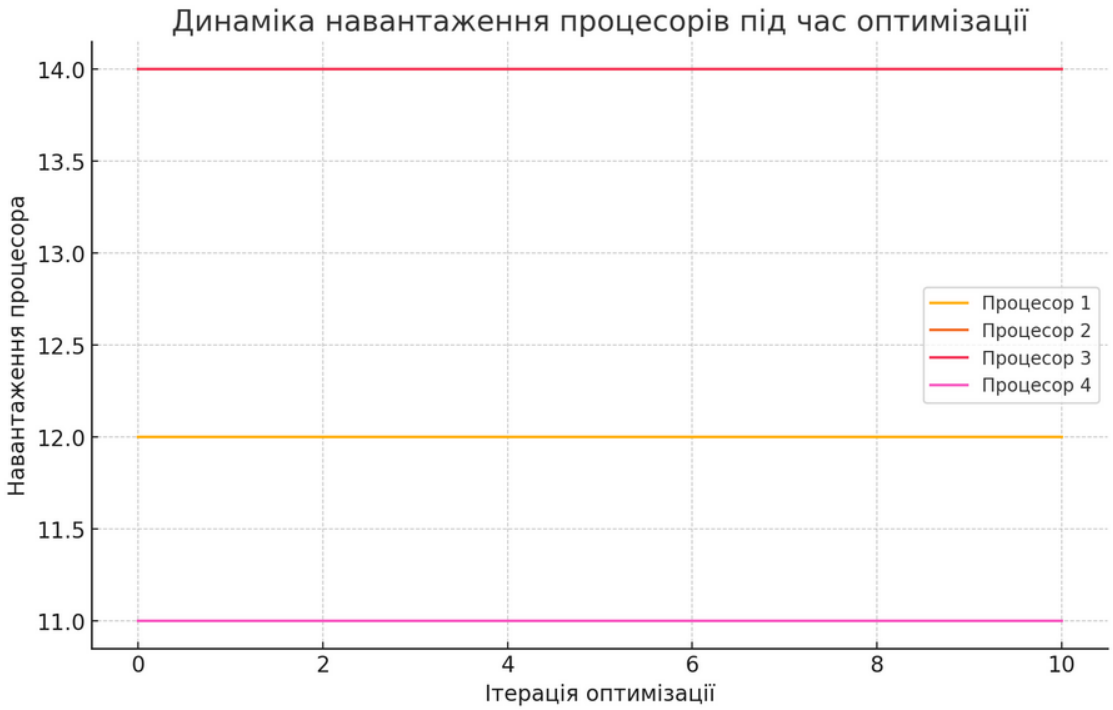


Рисунок 4.1 – Результати першої серії експерименту

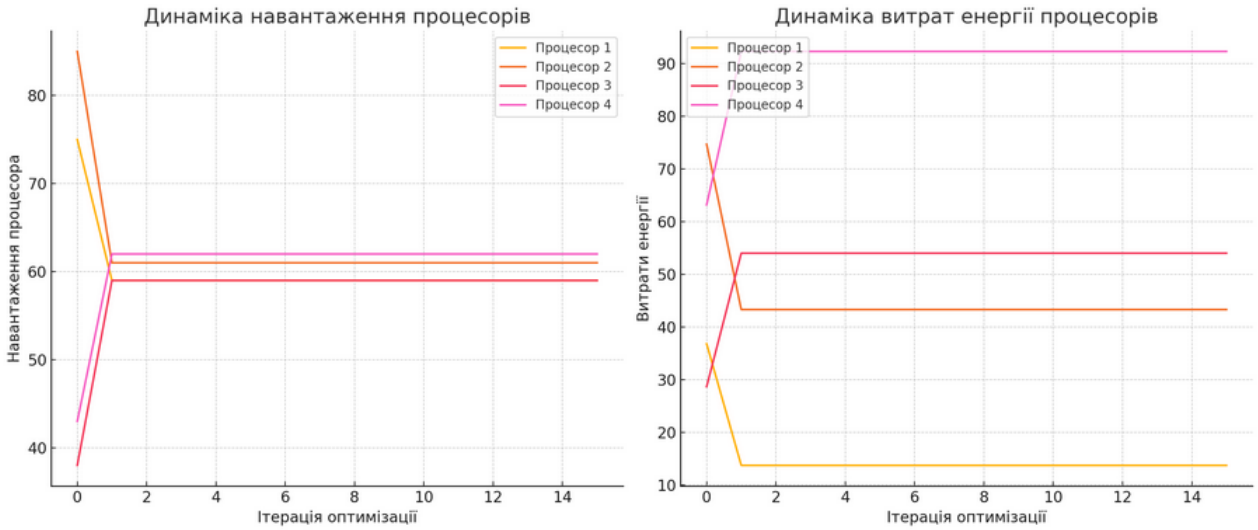


Рисунок 4.2 – Результати другої серії експерименту

Результати експерименту:

1) початкове навантаження процесорів - [75, 85, 38, 43];

2) кінцеве навантаження після оптимізації - [59, 61, 59, 62];

3) початкові витрати енергії: [36.85, 74.71, 28.71, 63.20];

4) кінцеві витрати енергії: [13.78, 43.34, 54.03, 92.31].

Аналіз експерименту: спочатку навантаження було навмисно розбалансованим, щоб перевірити ефективність методу оптимізації; алгоритм виконав міграцію завдань, зрівнявши навантаження на процесори, про що свідчить кінцевий стан, де всі процесори мають приблизно однакове навантаження. Однак витрати енергії стали менш рівномірними, оскільки завдання з більшим споживанням енергії залишилися на певних процесорах, а інші процесори отримали коротші, але енергоємні завдання.

На графіках ліва частина показує, як навантаження процесорів вирівнюється з часом, а права частина ілюструє динаміку витрат енергії, яка не була критерієм оптимізації, тому рівномірності тут не досягнуто.

Цей експеримент демонструє, що обраний метод оптимізації ефективно вирівнює навантаження, але для оптимізації енергоспоживання потрібні додаткові цільові функції.

Здійснимо, також, моделювання з використанням середовища MATLAB.

Обґрунтуємо вибір MATLAB для моделювання багатозадачної вбудованої системи. MATLAB є інструментом для моделювання складних систем завдяки своїм численним перевагам у сфері обчислень, візуалізації та простоти розробки алгоритмів. Вибір MATLAB для моделювання багатозадачної вбудованої системи є обґрунтованим з кількох причин.

Швидкість розробки та зручність програмування. MATLAB має високорівневу мову програмування, що дозволяє швидко розробляти алгоритми без необхідності управління пам'яттю чи оптимізації низькорівневого коду. Це значно спрощує процес моделювання, особливо для досліджень та експериментів, де потрібно часто змінювати параметри системи або саму модель.

Гнучкість у моделюванні та симуляціях. MATLAB забезпечує гнучкість у моделюванні складних систем завдяки таким можливостям. Підтримка багатозадачності та паралельних обчислень та легке налаштування параметрів

системи, таких як кількість процесорів, завдань, швидкість виконання тощо є перевагами. А також можливість швидко реалізовувати складні алгоритми розподілу завдань та оптимізації.

Великий набір математичних функцій та інструментів оптимізації. У MATLAB є вбудовані засоби для оптимізації (Optimization Toolbox), які можна використовувати для мінімізації цільової функції, наприклад, часу виконання завдань або енергоспоживання. Це важливо для розробленої системи, де необхідно знайти оптимальний розподіл завдань між процесорами.

Візуалізація даних та аналіз результатів. Однією з ключових переваг MATLAB є його потужні засоби візуалізації: легке створення графіків для аналізу завантаження процесорів, часу виконання завдань та кількості міграцій; налаштовувані графіки, які допомагають швидко аналізувати динаміку виконання системи та знаходити вузькі місця; можливість створювати анімації, які дозволяють візуалізувати процес міграції завдань у динаміці.

У випадку розробленого методу оптимізації моделюється багатозадачна система з кількома процесорами. MATLAB має вбудовані інструменти для паралельних обчислень (Parallel Computing Toolbox), що дозволяє ефективно моделювати розподіл завдань між процесорами.

MATLAB дозволяє легко створювати модульний код із використанням функцій та скриптів. Це дозволяє: повторно використовувати код для різних конфігурацій системи; легко додавати нові моделі процесорів чи завдань; швидко змінювати алгоритми розподілу та міграції завдань.

MATLAB добре підходить для моделювання складних систем завдяки своїй інтеграції з Simulink. Це дозволяє в майбутньому розширити модель, додавши: деталізовані моделі вбудованих систем; імітацію апаратного забезпечення та зовнішнього середовища; вбудовування енергоспоживання та інших обмежень системи.

MATLAB є стандартом для моделювання складних систем у багатьох галузях: від вбудованих систем до телекомунікацій та робототехніки. Це забезпечує: доступ до великої кількості наукових статей та прикладів коду; легку інтеграцію з іншими

інженерними інструментами та платформами.

Вибір MATLAB для моделювання багатозадачної вбудованої системи обумовлений його гнучкістю, потужними засобами візуалізації, вбудованими інструментами оптимізації та підтримкою паралельних обчислень. Він дозволяє швидко реалізовувати складні алгоритми розподілу завдань, проводити експерименти з різними параметрами та ефективно аналізувати результати. Завдяки всім цим перевагам MATLAB є оптимальним вибором для такого типу завдань.

Розробимо в середовищі MATLAB таку модель: багатозадачна вбудована система; наявні процесори; наявні завдання; реалізовано механізм реплікації при міграції завдань та цільова функція оптимізації. Завданням моделювання є забезпечення моделювання такої системи з використанням середовища MATLAB і написати програму на MATLAB для здійснення моделювання.

Щоб розпочати моделювання багатозадачної вбудованої системи в MATLAB, потрібно визначити кілька ключових компонентів:

1) процесори та їх характеристики, тобто кількість, продуктивність, енергоспоживання тощо;

2) характеристики завдань, такі як час виконання, пріоритети, залежності між завданнями;

3) механізм реплікації та міграції завдань, коли і як завдання можна перенести з одного процесора на інший;

4) цільова функція оптимізації, наприклад, мінімізація часу виконання, енергоспоживання чи балансування навантаження.

Алгоритм дій:

1) моделювання архітектури системи полягає в створенні масиву для опису процесорів і завдань, встановенні початкового розподілу завдань між процесорами;

2) моделювання виконання завдань передбачає реалізацію циклічного обходу завдань і процесорів, визначення правила виконання (наприклад, за пріоритетами);

3) механізм реплікації та міграції вимагає впровадити умови для міграції (наприклад, перевантаження процесора) та описати алгоритм реплікації завдань;

4) оптимізація повинна бути забезпечена обраною цільовою функцією та

реалізованим алгоритмом оптимізації (наприклад, генетичний алгоритм або алгоритм рою часток).

Структура коду в додатку В.

Пояснення до програмного коду такі:

1) ініціалізація параметрів системи полягає в налаштуванні процесорів і завдань;

2) розподіл завдань полягає в початковому розподілі завдань між процесорами;

3) механізм міграції реалізовано як просту міграцію, коли на процесорі більше двох завдань;

4) цільова функція подана як мінімізація максимальної тривалості виконання завдань;

5) візуалізація здійснена як побудова стовпчикової діаграми, що показує завантаження кожного процесора.

Можливі вдосконалення моделювання:

1) додати більш складний механізм міграції та реплікації;

2) икористати оптимізаційні алгоритми (наприклад, генетичний алгоритм);

3) врахувати енергоспоживання або інші обмеження системи.

Результати моделювання з різними параметрами графіками та поясненнями до них зображено на рис. 4.3. Моделювання та дослідження результатів за різних параметрів подано графіками. Основні параметри при моделюванні такі:

1) кількість процесорів змінювали від 2 до 6;

2) кількість завдань варіюєвали від 10 до 30;

3) швидкість процесорів від 1 до 3 умовних одиниць;

4) цільова функція використана для мінімізації максимального часу виконання завдань.

Метою дослідження є аналіз змін завантаження процесорів за різної кількості завдань і процесорів, виявення оптимального співвідношення кількості процесорів та завдань, оцінювання ефективності міграції завдань.

Алгоритм дослідження розроблено для проведення моделювання для різних комбінацій параметрів.

Графіки для аналізу відображають:

1) завантаження процесорів;

2) час виконання завдань;

3) кількість міграцій завдань.

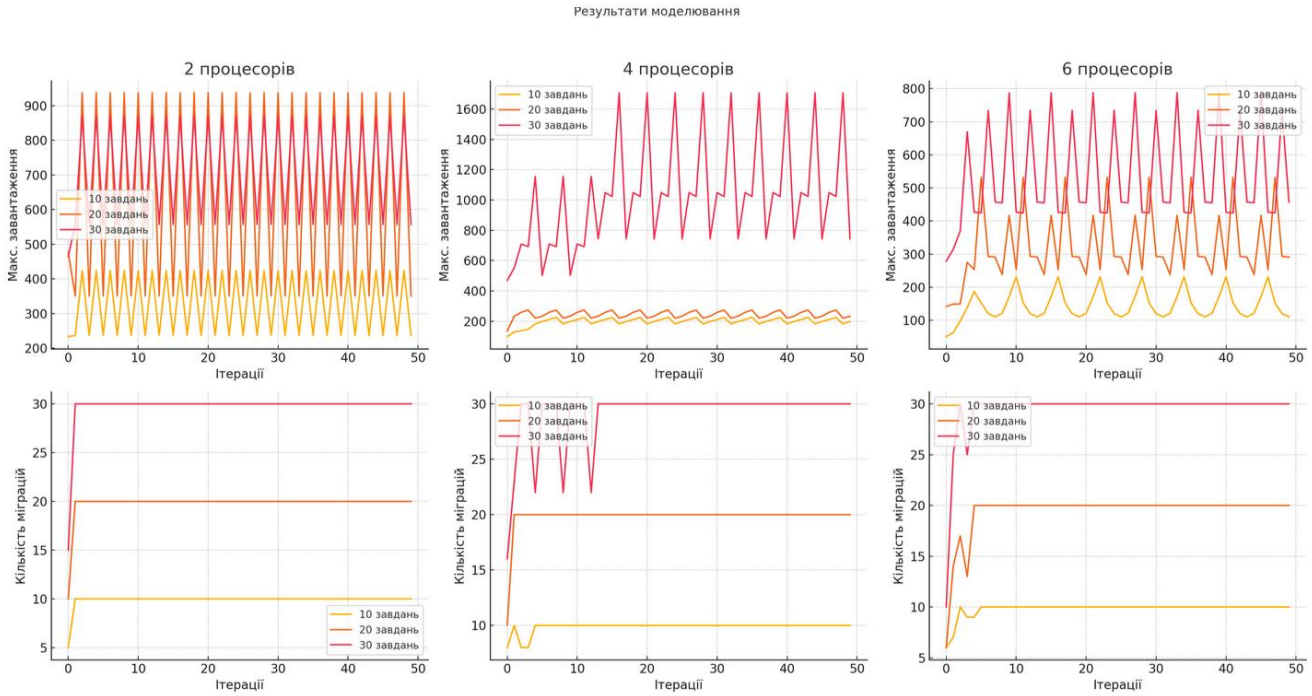


Рисунок 4.3 – Результати моделювання системи

Розглянемо аналіз графіків з рис. 4.3.

Графіки максимального завантаження процесорів (верхній ряд) показують, як змінюється максимальне навантаження на процесори впродовж ітерацій. За більшої кількості процесорів максимальне навантаження менше, оскільки завдання краще розподіляються. За більшої кількості завдань (наприклад, 30) максимальне навантаження зростає, що свідчить про обмеженість ресурсів.

Графіки кількості міграцій (нижній ряд) відображають динаміку міграцій завдань між процесорами. На початкових ітераціях міграцій більше, оскільки система намагається збалансувати навантаження. Згодом кількість міграцій зменшується, коли навантаження стабілізується. За більшої кількості процесорів міграцій менше, оскільки навантаження легше збалансувати.

Таким чином, збільшення кількості процесорів призводить до зменшення

максимального завантаження і кількості міграцій, збільшення кількості завдань підвищує навантаження на систему та кількість міграцій на початкових етапах, механізм міграції ефективно балансує навантаження, особливо на початкових етапах виконання.

4.3 Висновки до четвертого розділу

Розроблений метод оптимізації [84] виконання завдань з використанням реплікації у багатопроцесорній системі дає змогу ефективно мінімізувати загальний час виконання, забезпечити баланс навантаження і мінімізувати затримки зв’язку. Особливістю методу є реалізація міграції завдань згідно реплікації з використанням цільової функції оптимізації.

Проведений експеримент з системою продемонстрував, що обраний метод оптимізації ефективно вирівнює навантаження, але для оптимізації енергоспоживання потрібні додаткові цільові функції.

Результати моделювання показують, що збільшення кількості процесорів призводить до зменшення максимального завантаження і кількості міграцій, збільшення кількості завдань підвищує навантаження на систему та кількість міграцій на початкових етапах, механізм міграції ефективно балансує навантаження, особливо на початкових етапах виконання.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень розроблено новий метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах та отримано такі результати.

1. Проаналізовано відомі методи та засоби оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

2. Розроблено метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

3. Розроблено цільової функції оптимізації завдань у багатопроцесорних вбудованих системах.

4. Досліджено розроблений метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах, розроблено програму, проведено експеримент та моделювання схеми оптимізації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Zhou Y., Zhang E., Guo H., Fang Y., Li H. Lifting path planning of mobile cranes based on an improved RRT algorithm. *Adv. Eng. Inform*. 2021, 50, 9. https://doi.org/10.1016/j.aei.2021.101376

2. Zhu A.M., Zhang Z.Q., Pan W. Crane-lift path planning for high-rise modular integrated construction through metaheuristic optimization and virtual prototyping. *Autom. Constr*. 2022, 141, 21. https://doi.org/10.1016/j.autcon.2022.104434

3. Guo H., Zhou Y., Pan Z., Zhang Z., Yu Y., Li Y. Automated Selection and Localization of Mobile Cranes in Construction Planning. *Buildings* 2022, 12, 580. https://doi.org/10.3390/buildings12050580

4. Wang J., Zhang Q., Yang B., Zhang B. Vision-Based Automated Recognition and 3D Localization Framework for Tower Cranes Using Far-Field Cameras. *Sensors* 2023, 23, 851. https://doi.org/10.3390/s23104851

5. Huang L., Pradhan R., Dutta S., Cai Y. BIM4D-based scheduling for assembling and lifting in precast-enabled construction. *Autom. Constr*. 2022, 133, 14. https://doi.org/10.1016/j.autcon.2021.103999

6. Song Y., Xin R., Chen P., Zhang R., Chen J., Zhao Z. Autonomous selection of the fault classification models for diagnosing microservice applications. *Future Generation Computer Systems*, 2024. 153, pp.326-339. https://doi.org/10.1016/j.future.2023.12.005

7. Tao L., Lu X., Zhang S., Luan J., Li Y., Li M., Li Z., Yu Q., Xie H., Xu,R., Hu C. Diagnosing Performance Issues for Large-Scale Microserv ice Systems With Heterogeneous Graph. *IEEE Transactions on Services Computing*. 2024. https://ieeexplore.ieee.org/document/10533869

8. Chen Y., Xu D., Chen N., Wu X. FRL-MFPG: Propagation-aware fault root cause location for microservice intelligent operation and maintenance. *Information and Software Technology*. 2023. 153, p.107083. https://doi.org/10.1016/j.infsof.2022.107083

9. Li X., Wen P., Chen P., Chen J., Wen X., Xia Y. An effective parallel convolutional anomaly multi-classification model for fault diagnosis in microservice system. *Software Quality Journal*. 2024. Pp.1-18. https://doi.org/10.21203/rs.3.rs-5267111/v1

10. Mazraemolla Z.P., Rasoolzadegan A. An effective failure detection method for microservice-based systems using distributed tracing data. *Engineering Applications of Artificial Intelligence.* 2024. 133, p.108558. https://doi.org/10.1016/j.engappai.2024.108558

11. Zhang S., Jin P., Lin Z., Sun Y., Zhang B., Xia S., Li Z., Zhong Z., Ma M., Jin W., Zhang, D. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing*. 2023.16(6), pp.3851-3864. https://doi.org/10.48550/arXiv.2302.10512

12. Chen J., Zhang R., Chen P., Ren J., Wu Z., Wang Y., Li X., Xiong L. MTG_CD: Multi-scale learnable transformation graph for fault classification and diagnosis in microservices. *Journal of Cloud Computing.* 2024. 13(1), p.103. https://doi.org/10.1186/s13677-024-00666-0

13. Zhang B., Wang X., Wang H. Virtual machine placement strategy using cluster-based genetic algorithm. *Neurocomputing.* 2021. 428, Pp. 310–316. https://doi.org/10.1016/j.neucom.2020.06.120

14. Wei P., Zeng Y., Yan B., Zhou J., Nikougoftar E. Vmp-a3c: virtual machines placement in cloud computing based on asynchronous advantage actor-critic algorithm. *J. King Saud Univ. Comput. Inf. Sci.* 2023. 35, 101549. https://doi.org/10.1016/j.jksuci.2023.04.002

15. Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S. Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software*. 2024. 207, p.111857. https://doi.org/10.1016/j.jss.2023.111857

16. Yin J., Li J., Fang Y., Yang A. Service scheduling optimization for multiple tower cranes considering the interval time of the cross-tasks. *Math. Biosci. Eng.* 2023, 20. Pp. 5993–6015. https://www.aimspress.com/article/doi/10.3934/mbe.2023259

17. Daoud A.O., El Hefnawy M., Wefki H. Investigation of critical factors affecting cost overruns and delays in Egyptian mega construction projects. *Alex. Eng.* J. 2023, 83, Pp. 326–334. https://doi.org/10.1016/j.aej.2023.10.052

18. Lin X., Han Y., Guo H., Luo Z., Guo Z. Lift path planning for tower cranes based on environmental point clouds. *Autom. Constr*. 2023, 155, 105046.

https://doi.org/10.1016/j.autcon.2023.105046

19. Pranav T. Mapping and Scheduling on Multi-core Processors using SMT Solvers. 2014. https://www.researchgate.net/publication/281534326_Mapping_and_Scheduling_on_Multi-core_Processors_using_SMT_Solvers

20. Huang C., Wang Z.K., Li B., Wang C., Xu L.S., Jiang K., Liu M., Guo C.X., Zhao X.F., Yang H. Discretized Cell Modeling for Optimal Layout of Multiple Tower Cranes. *J. Constr. Eng. Manag.* 2023, 149, 19. https://doi.org/10.1061/(ASCE)CO.1943-7862.0001206

21. Jiang H., Miao Y.B. Construction Site Layout Planning Using Multiple-Level Simulated Annealing. *In Proceedings of the Computing Conference, Virtual, 15–16 July 2021; Springer International Publishing: Cham, Switzerland*, 2021. https://link.springer.com/chapter/10.1007/978-3-030-80126-7_30

22. Kaveh A., Vazirinia Y. An Upgraded Sine Cosine Algorithm for Tower Crane Selection and Layout Problem. *Period. Polytech.-Civ. Eng*. 2020, 64, 325–343. https://doi.org/10.3311/PPci.15363

23. Khodabandelu A., Park J., Arteaga C. Improving Multitower Crane Layout Planning by Leveraging Operational Flexibility Related to Motion Paths. *J. Manag. Eng.* 2023, 39, 17. https://doi.org/10.1061/JMENEA.MEENG-5402

24. Li R., Chi H.L., Peng Z., Li X., Chan A.P. Automatic tower crane layout planning system for high-rise building construction using generative adversarial network. *Adv. Eng. Inform*. 2023, 58, 19. https://doi.org/10.1016/j.aei.2023.102202

25. Liu C., Zhang F., Han X., Ye H., Shi Z., Zhang J., Wang T., She J., Zhang T. Intelligent Optimization of Tower Crane Location and Layout Based on Firefly Algorithm. *Comput. Intell. Neurosci*. 2022, 2022, 13. https://doi.org/10.1155/2022/6810649

26. Lu Y., Zhu Y.Q. Integrating Hoisting Efficiency into Construction Site Layout Plan Model for Prefabricated Construction. *J. Constr. Eng. Manag*. 2021, 147, 15. https://doi.org/10.1061/(ASCE)CO.1943-7862.0002158

27. Riga K., Jahr K., Thielen C., Borrmann A. Mixed integer programming for dynamic tower crane and storage area optimization on construction sites. *Autom. Constr*.

2020, 120, 15. https://doi.org/10.1016/j.autcon.2020.103259

28. Zhang Z.Q., Pan W., Pan M. Critical considerations on tower crane layout planning for high-rise modular integrated construction. *Eng. Constr. Archit. Manag*. 2022, 29, 2615–2634. https://doi.org/10.1108/ECAM-03-2021-0192

29. Wu K.Y., De Soto B.G. Lifting Sequence Optimization of Luffing Tower Cranes Considering Motion Paths with Dynamic Programming. *J. Constr. Eng. Manag.* 2021, 147, 16. https://doi.org/10.1061/(ASCE)CO.1943-7862.0002129

30. Xu Y., Qiu Z., Gao H., Zhao X., Wang L., Li, R. Heterogeneous data-driven failure diagnosis for microservice-based industrial clouds towards consumer digital ecosystems. *IEEE Transactions on Consumer Electronics.* 2023. https://doi.org/10.1109/TCE.2023.3337351

31. Pan Y., Ma M., Jiang X., Wang, P. DyCause: Crowdsourcing to Diagnose Microservice Kernel Failure. *IEEE Transactions on Dependable and Secure Computing.* 2023. 20(6), pp. 4763-4777. https://github.com/CloudWise-OpenSource/GAIA-DataSet

32. Li Y., Lou J., Tan X., Xu Y., Zhang J., Jing Z. Adaptive kernel learning kalman filtering with application to model-free maneuvering target tracking. *IEEE Access, 2022*. 10, pp.78088-78101. https://ieeexplore.ieee.org/document/9837071

33. Khare S.K., Bajaj V., Acharya U.R. Detection of Parkinson's disease using automated tunable Q wavelet transform technique with EEG signals. *Biocybernetics and Biomedical Engineering*, 2021. 41(2), pp.679-689. https://doi.org/10.1016/j.bbe.2021.04.008

34. Zhang C., Liu Q., Zhang Z. DSGNN: A dynamic and static intentions integrated graph neural network for session-based recommendation. *Neurocomputing*, 2022.468, pp.222-232. https://doi.org/10.1016/j.neucom.2021.10.028

35. Lian J., Hui G. Human evolutionary optimization algorithm. *Expert Systems with Applications*. 2024. 241, p.122638. https://doi.org/10.1016/j.eswa.2023.122638

36. Alboaneen D., Tianfield H., Zhang Y., Pranggono B. A metaheuristic method for joint task scheduling and virtual machine placement in cloud data centers. *Future Gener. Comput. Syst.* 2021. 115, pp. 201–212. https://doi.org/10.1016/j.future.2020.08.036

37. Cavdar M.C., Korpeoglu I., Ulusoy O. A utilization based genetic algorithm for

virtual machine placement in cloud systems. *Comput. Commun.* 2024. 214, pp. 136–148. https://doi.org/10.1016/j.comcom.2023.11.028

38. Burkhardt M., Gienger A., Sawodny O. Optimization-Based Multipoint Trajectory Planning Along Straight Lines for Tower Cranes. *IEEE Trans. Control Syst. Technol*. 2023, 8, 290–297. https://ieeexplore.ieee.org/document/10262144

39. Dutta S., Cai Y., Huang L., Zheng J. Automatic re-planning of lifting paths for robotized tower cranes in dynamic BIM environments. *Autom. Constr*. 2020, 110, 19. https://doi.org/10.1016/j.autcon.2019.102998

40. Li X., Chi H.L., Wu P., Shen G.Q. Smart work packaging-enabled constraint-free path re-planning for tower crane in prefabricated products assembly process. *Adv. Eng. Inform*. 2020, 43, 16. https://doi.org/10.1016/j.aei.2019.101008

41. Lin Z.Y., Petzold F., Hsieh S.H. Automatic Tower Crane Lifting Path Planning Based on 4D Building Information Modeling. *In Proceedings of the Construction Research Congress (CRC) on Construction Research and Innovation to Transform Society, Del E Webb Sch Construct, Arizona State University, Tempe, AZ, USA, 8–10 March 2020; American Society of Civil Engineers: New York, NY, USA*, 2020. https://ascelibrary.org/doi/10.1061/9780784482865.089

42. Ji Y.S., Leite F. Optimized Planning Approach for Multiple Tower Cranes and Material Supply Points Using Mixed-Integer Programming. *J. Constr. Eng. Manag*. 2020, 146, 11. https://ascelibrary.org/doi/10.1061/%28ASCE%29CO.1943-7862.0001781

43. Zhou C., Dai F., Xiao Z., Liu W. Location Optimization of Tower Cranes on High-Rise Modular Housing Projects. *Buildings* 2023, 13, 115. https://doi.org/10.3390/buildings13010115

44. Wu K.Y., De Soto B.G., Zhang F.L. Spatio-temporal planning for tower cranes in construction projects with simulated annealing. *Autom. Constr*. 2020, 111, 17. https://doi.org/10.1016/j.autcon.2019.103060

45. Zhang W., Zhang H., Yu L. Collaborative Planning for Stacking and Installation of Prefabricated Building Components Regarding Crane-Collision Avoidance. *J. Constr. Eng. Manag.* 2023, 149, 04023029. http://dx.doi.org/10.1061/JCEMD4.COENG-12955

46. Khodabandelu A., Park J., Arteaga C. Crane operation planning in overlapping

areas through dynamic supply selection. *Autom. Constr*. 2020, 117, 14. https://doi.org/10.1016/j.autcon.2020.103253

47. Sarathambekai S., Kandaswamy U. Multi-Objective Optimization Techniques for Task Scheduling Problem in Distributed Systems. *The Computer Journal*. 2018. № 61. Pp. 248-263. DOI: 10.1093/comjnl/bxx059. https://doi.org/10.1093/comjnl/bxx059

48. Tariq A., Xiangjie H., Zhicheng C., Jian X. Multi-objective Optimization of Ticket Assignment Problem in Large Data Centers. 2022. DOI: 10.1007/978-981-19-4549-6_4. https://link.springer.com/chapter/10.1007/978-981-19-4549-6_4

49. Yu Q., Nengwen Z., Mingjie L., Zeyan L., Honglin W., Wenchi Z., Kaixin S., Dan P. A survey on intelligent management of alerts and incidents in IT services. *Journal of Network and Computer Applications.* 2024. DOI: 224. 103842. 10.1016/j.jnca.2024.103842. https://doi.org/10.1016/j.jnca.2024.103842

50. Ke X., Shikai G., Hui L., Chenchen L., Rong C., li X., Jiang H. Making Fault Localization in Online Service Systems More Actionable and Interpretable. *ACM Transactions on Software Engineering and Methodology.* 2025. https://doi.org/10.1145/3714466

51. Junjie Z., Xinwen S., Jiaxin L., Jiajia Z., Zihan L. Leveraging Large Language Models for Efficient Alert Aggregation in AIOPs. *Electronics.* 2024. DOI: 13. 4425. 10.3390/electronics13224425. https://doi.org/10.3390/electronics13224425

52. Tarhini H., Maddah B., Hamzeh F. The traveling salesman puts-on a hard hat—Tower crane scheduling in construction projects. *Eur. J. Oper. Res*. 2021, 292, 327–338. https://doi.org/10.1016/j.ejor.2020.10.029

53. Hammad A.W., Rey D., Akbarnezhad A., Haddad A. Integrated mathematical optimisation approach for the tower crane hook routing problem to satisfy material demand requests on-site. *Adv. Eng. Inform*. 2023, 55, 13. https://doi.org/10.1016/j.aei.2023.101885

54. Motyčka V., Gašparík J., Přibyl O., Štěrba M., Hořínková D., Kantová R. Effective Use of Tower Cranes over Time in the Selected Construction Process. *Buildings* 2022, 12, 436. https://doi.org/10.3390/buildings12040436

55. Yin J., Li J., Yang A., Cai S. Optimization of service scheduling problem for overlapping tower cranes with cooperative coevolutionary genetic algorithm. *Eng. Constr.*

Archit. Manag. 2024, 31, 1348–1369. https://doi.org/10.1108/ECAM-08-2022-0767

56. Mousaei A., Taghaddos H., Marzieh Bagheri S., Hermann U. Optimizing Heavy Lift Plans for Industrial Construction Sites Using Dijkstra's Algorithm. *J. Constr. Eng. Manag.* 2021, 147, 16. https://doi.org/10.1061/(ASCE)CO.1943-7862.0002157

57. Wu K., García D.E., Soto B. Spatiotemporal Modeling of Lifting Task Scheduling for Tower Cranes with a Tabu Search and 4-D Simulation. *Front. Built Environ*. 2020, 6, 79. https://doi.org/10.3389/fbuil.2020.00079

58. Gopu A., Thirugnanasambandam K., AlGhamdi A.S., Alshamrani S.S., Maharajan K., Rashid M. Energy-efficient virtual machine placement in distributed cloud using NSGA-III algorithm. *J. Cloud Comput.-Adv. Syst. Appl*. 2023, 12, 20. https://doi.org/10.1186/s13677-023-00501-y

59. Huang C., Li W., Lu W., Xue F., Liu M., Liu Z. Optimization of multiple-crane service schedules in overlapping areas through consideration of transportation efficiency and operational safety. *Autom. Constr.* 2021, 127, 18. https://doi.org/10.1016/j.autcon.2021.103716

60. Zhang Z.Q., Ma S.L., Jiang X.Y. Research on Multi-Objective Multi-Robot Task Allocation by Lin-Kernighan-Helsgaun Guided Evolutionary Algorithms. *Mathematics* 2022, 10, 4714. https://doi.org/10.3390/math10244714

61. Li K., Duan T., Li Z., Xiahou X., Zeng N., Li Q. Development Path of Construction Industry Internet Platform: An AHP—TOPSIS Integrated Approach. *Buildings* 2022, 12, 441. https://doi.org/10.3390/buildings12040441

62. Lysenko S., Bobrovnikova K., Savenko O., Kryshchuk A. BotGRABBER: SVM-Based Self-Adaptive System for the Network Resilience Against the Botnets' Cyberattacks. *Communications in Computer and Information Science*. 2019. Vol. 1039. Pp.127-143, ISSN: 1865-0929. https://doi.org/10.31891/2307-5732-2024-331-2

63. Lysenko S., Savenko O., Bobrovnikova K., Kryshchuk A. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks. *Communications in Computer and Information Science*, 2018. Vol. 860. Pp. 385-401. https://link.springer.com/chapter/10.1007/978-3-319-92459-5_31

64. Savenko O., Sachenko A., Lysenko S., Markowsky G., Vasylkiv N. (2020).

BOTNET DETECTION APPROACH BASED ON THE DISTRIBUTED SYSTEMS. *International Journal of Computing*, *19*(2), 190-198. https://doi.org/10.47839/ijc.19.2.1761

65. Kashtalian A., Lysenko S., Savenko O., Nicheporuk A., Sochor T., & Avsiyevych V. Multi-computer malware detection systems with metamorphic functionality. *Radioelectronic and Computer Systems*. 2024. Vol. 1. Pp. 152-175. doi: https://doi.org/10.32620/reks.2024.1.13 \

66. Savenko B., Kashtalian A., Lysenko S., Savenko O. Malware Detection By Distributed Systems with Partial Centralization. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 2023, pp. 265-270, doi: 10.1109/IDAACS58523.2023.10348773. https://ieeexplore.ieee.org/document/10348773

67. Chong G., Ramiah H., Yin J., Rajendran J., Wong W.R., Mak P.-I., Martin R.P. CMOS cross-coupled differential-drive rectifier in subthreshold operation for ambient RF Energy Harvesting—Model and analysis. *IEEE Trans. Circuits Syst. II Express Briefs* 2019, 66, 1942–1946. https://ieeexplore.ieee.org/document/8630669

68. Kumar S., Gupta U., Singh A.K., Singh A.K. Artificial Intelligence: Revolutionizing Cyber Security in the Digital Era. *J. Comput. Mech. Manag.* 2023, 2, 31–42. https://doi.org/10.57159/gadl.jcmm.2.3.23064

69. Bedratyuk L. and Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. (2018) 79, 407–414. https://doi.org/10.48550/arXiv.1706.00829

70. Oji C., Nwankokwo O., John-Otumu A.M., Adu C. Development of An Enhanced Bandwidth Control Platform for Effective Monitoring and Utilization of Resources in Corporate Networks. *Int. J. Sci. Res. Publ.* 2021, 11, 260–271. http://dx.doi.org/10.29322/IJSRP.11.08.2021.p11636

71. Maksymyuk T., Gazda J. Blockchain-Based Intelligent Network Management for 5G and beyond. *In Proceedings of the International Conference on Advanced Information and Communication Technologies*, Lviv, Ukraine, 2–6 July 2019. https://ieeexplore.ieee.org/document/8847762

72. Oktian Y.E. Witanto E.N. Kumi S. Lee S.-G. ISP Network Bandwidth

Management: Using Blockchain and SDN. *In Proceedings of the International Conference on Information and Communication Technology Convergence*, Jeju, Republic of Korea, 16–18 October 2019. https://ieeexplore.ieee.org/document/8939811

73. Motiwala A., Timbadia P., Upadhyay T., Kunekar P. E-Voting System Using Blockchain. *SAMRIDDHI J. Phys. Sci. Eng. Technol*. 2019, 11, 434–438. https://www.myresearchjournals.com/index.php/SAMRIDDHI/article/view/7565

74. Zhang D., Yu F.R., Yang R. Blockchain-Based Multi-Access Edge Computing for Future Vehicular Networks: A Deep Compressed Neural Network Approach. *IEEE Trans. Intell. Transp. Syst.* 2022, 23, 12161–12175. https://ieeexplore.ieee.org/document/9565824

75. Zhang Y., Lin, S., Tu C. Research on Security Protection of Space-Earth Integrated Network Wireless Link Based on Consortium Blockchain Technology and Application. *In Proceedings of the 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA*), Changchun, China, 24–26 February 2023; pp. 1455–1458. https://ieeexplore.ieee.org/document/10090811

76. Zhao L., Saif M.B., Hawbani A., Min G., Peng S., Lin N. A novel improved artificial bee colony and blockchain-based secure clustering routing scheme for FANET. *China Commun*. 2021, 18, 103–116. https://ieeexplore.ieee.org/document/9495358

77. Guo H., Yu X. A survey on blockchain technology and its security. *Blockchain Res. Appl*. 2022, 3, 100067. https://doi.org/10.1016/j.bcra.2022.100067

78. Lei X., Song Y., Lin J., Feng T., Li P., Wang Y., Yang C. Resilience In-Band Control Path Routing in Blockchain-Based Multi-Domain SDN. *In Proceedings of the 2023 International Wireless Communications and Mobile Computing (IWCMC),* Marrakesh, Morocco, 19–23 June 2023; pp. 1654–1659. https://ieeexplore.ieee.org/document/10182467

79. Subrahmanya S.V., Shetty D.K., Patil V., Hameed B.M., Paul R., Smriti K., Naik N., Somani B.K. The role of Data Science in healthcare advancements: Applications, benefits, and future prospects. *Ir. J. Med. Sci*. (1971-) 2021, 191, 1473–1483. https://link.springer.com/article/10.1007/s11845-021-02730-z

80. Selvi R.T., Lakshmi T.C.S., Karunkuzhali D., Rosaline R.A.A. Improvement of

Graph-Based Assets using Blockchain Quality Control. *In Proceedings of the 2023 International Conference on Innovative Data Communication Technologies and Application (ICIDCA)*, Uttarakhand, India, 14–16 March 2023; pp. 716–720. https://ieeexplore.ieee.org/document/10099935

81. Qiu W., Shi C., Zhou J., Wang F. Joint Dwell TIme and Bandwidth Optimization for Multiple—Target Tracking in Radar Network. *In Proceedings of the International Conference on Control, Automation and Information Sciences: IEEE*, Chengdu, China, 23–26 October 2019. https://doi.org/10.3390/s20051269

82. Ncube T., Dlodlo N., Terzoli A. Private Blockchain Networks: A Solution for Data Privacy. *In Proceedings of the International Multidisciplinary Information Technology and Engineering Conference (IMITEC)*, Kimberley, South Africa, 25–27 November 2020. https://ieeexplore.ieee.org/document/9334132

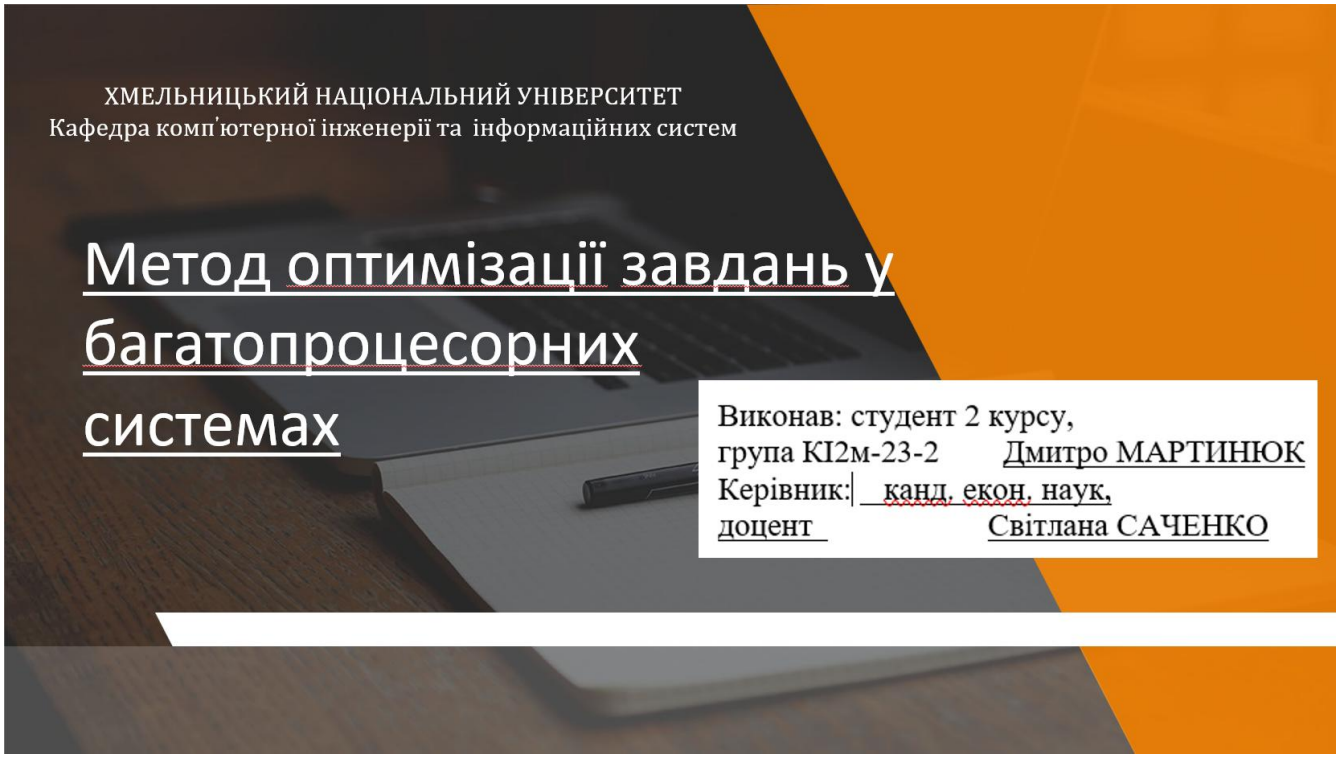
83. Pass R., Seeman L., Shelat A. Analysis of the Blockchain Protocol in Asynchronous Networks. *In Advances in Cryptology—EUROCRYPT 2017; Springer:* Cham, Switzerland, 2017; pp. 643–673. https://link.springer.com/chapter/10.1007/978-3-319-56614-6_22

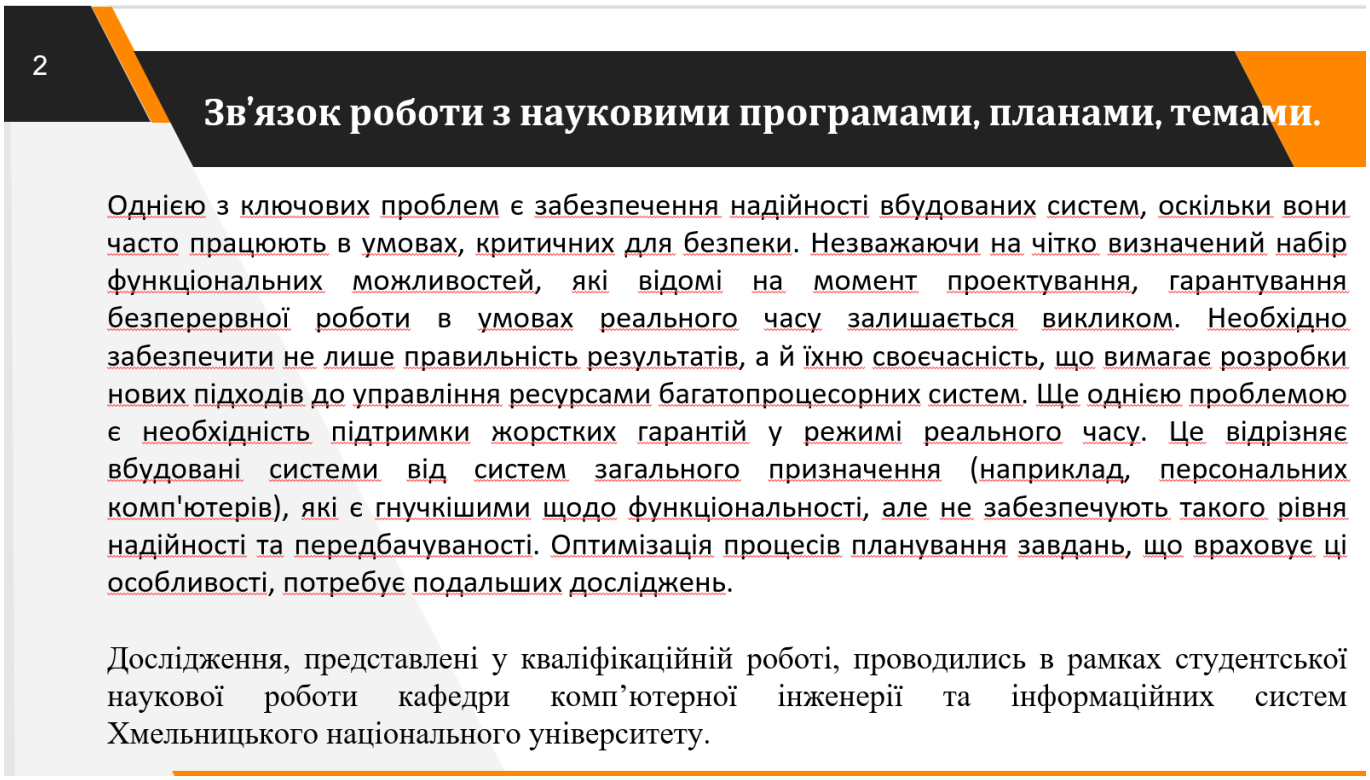
84. Dmytro Martiniuk, Oleksii Lyhun, Andriy Drozd, Oleksii Besedovskyi. TASK OPTIMISATION IN MULTIPROCESSOR EMBEDDED SYSTEMS. *Computer Systems and Information Technologies*, 2025. №1. Pp. 124-134.

ДОДАТОК А

(обов'язковий)

Презентація роботи





3

Перелік публікацій

За темою кваліфікаційної роботи опубліковано одну наукову статтю в науковому журналі категорії Б:

Dmytro Martiniuk, Oleksii Lyhun, Andriy Drozd, Oleksii Besedovskyi. TASK OPTIMISATION IN MULTIPROCESSOR EMBEDDED SYSTEMS. *Computer Systems and Information Technologies*, 2025. №1.

- *Метою кваліфікаційної роботи магістра є покращення ефективності оптимізації завдань у багатопроцесорних системах.*
- *Поставлена мета досягається розв'язанням таких основних завдань:*
- *- проаналізувати відомі методи забезпечення функціонування систем з IoT та захищені протоколи для зв'язку між пристроями IoT;*
- *- розробити удосконалення методу забезпечення функціонування систем з IoT із захищеним протоколом;*
- *- здійснити реалізацію протоколу згідно розробленого методу забезпечення функціонування систем з IoT із захищеним протоколом;*
- *- здійснити дослідження захищеного протоколу.*

- *Об'єктом дослідження є процес оптимізації завдань у багатопроцесорних системах.*
- *Предметом дослідження є методи та засоби забезпечення оптимізації завдань у багатопроцесорних системах.*

- *Наукова новизна отриманих результатів:*
- *- розроблено новий метод оптимізації завдань у багатопроцесорних вбудованих системах, особливістю якого є використання реплікації та цільової функції при міграції завдань.*
- *На основі проведених досліджень розроблено метод оптимізації завдань у багатопроцесорних вбудованих системах.*
- *Практична значимість отриманих результатів полягає у розроблені схеми оптимізації завдань у багатопроцесорних вбудованих системах.*
- *Для розв'язання поставлених задач використовувалися методи оптимізації, теорія комп'ютерних систем.*

Оптимізація завдань у багатопроцесорних системах, враховуючи застосування комп'ютерних систем в різних сферах, набула на сьогодні особливого значення в контексті розроблення вбудованих пристроїв. Більшість вбудованих систем мають такі характеристики: вони призначені для реалізації чітко визначеного набору функціональних можливостей, які відомі на момент проектування; вони повинні бути надійними, оскільки часто працюють в умовах, критичних для безпеки; вони повинні надавати жорсткі гарантії в режимі реального часу, тобто їх вихід повинен бути правильним, а також виробленим в певні терміни. Ці характеристики відрізняють вбудовані системи від систем загального призначення, таких як персональні комп'ютери, які демонструють набагато більшу гнучкість з точки зору функціональності та набагато менше фокусування на гарантії та надійності у режимі реального часу.

Оптимізація завдань у багатопроцесорних системах з міграцією вимагає розробки цільової функції, яка враховує всі релевантні параметри для досягнення максимальної ефективності використання ресурсів та своєчасного виконання завдань. Основними параметрами, які слід враховувати, є час виконання завдань, затримки при міграції, навантаження на процесори, енергоспоживання, пріоритети завдань та дотримання встановлених термінів. Важливо мінімізувати загальний час завершення всіх завдань, забезпечити баланс навантаження між процесорами та мінімізувати застосункові витрати, пов'язані з міграцією завдань. Крім того, оптимізація повинна враховувати енергоспоживання, оскільки надмірне використання ресурсів призводить до зростання енерговитрат.

Цільову функцію для такої оптимізації задамо наступним чином:

$$F = \min\left[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{k=1}^{K} E_k + \delta \sum_{i=1}^{J} T_j\right], \quad (2.1)$$

де C_i – час завершення завдання i; L_m – коефіцієнт навантаження на процесор m, визначений як відношення часу зайнятості процесора до загального часу виконання; E_k – енергоспоживання процесора k під час виконання завдань; T_j – загальний час міграцій для завдання j, що враховує затримки на передачу даних та налаштування середовища виконання на новому процесорі; $\alpha, \beta, \gamma, \delta$ – вагові коефіцієнти, які визначають відносну важливість кожного з параметрів у цільовій функції.

Метод оптимізації завдань базуватимемо на реплікації при міграції завдань у багатопроцесорній системі і оптимізацію будемо здійснювати за допомогою розробленої цільової функції оптимізації, яка враховує основні параметри системи, такі як час виконання завдань, завантаження процесорів, обсяг використаної пам'яті та затримки в обміні повідомленнями. Оптимізація виконання завдань за допомогою методу реплікації передбачає знаходження такого розподілу процесів між процесорами, який мінімізує загальний час виконання завдань при одночасному збереженні балансу навантаження на процесори та мінімізації витрат на комунікацію між ними.

Цільову функція оптимізації задамо так:

$$F = \alpha \sum_{i=1}^{N} T_i + \beta \sum_{j=1}^{M} L_j + \gamma \sum_{k=1}^{P} C_k, \quad (4.1)$$

де T_i — час виконання i-го завдання, L_j — завантаження j-го процесора, C_k — затримка в обміні повідомленнями між процесорами; а α, β та γ — вагові коефіцієнти, які відображають пріоритети оптимізації.

Перший компонент цільової функції спрямований на мінімізацію загального часу виконання всіх завдань. Для цього необхідно забезпечити, щоб завдання виконувалися максимально паралельно і не очікували на ресурси. Метод реплікації забезпечує швидке перемикання виконання завдань між процесорами, оскільки копії коду завдань уже знаходяться в локальній пам'яті всіх потенційних процесорів. Оптимізація цього компонента передбачає динамічне перенесення завдань на найменш завантажені процесори, а також мінімізацію простою завдань.

Другий компонент цільової функції відповідає за баланс навантаження між процесорами. Нерівномірний розподіл завдань може призвести до того, що одні процесори будуть перевантажені, а інші простоюватимуть. Для вирішення цієї проблеми використовується стратегія динамічного балансування навантаження, яка передбачає регулярну оцінку завантаження процесорів і переміщення завдань з перевантажених процесорів на менш завантажені. Метод реплікації значно спрощує цей процес, оскільки код завдань вже доступний на всіх процесорах, а отже, необхідно лише передати стан завдання. Це зменшує час на перенесення завдань і забезпечує швидке відновлення їх виконання.

Третій компонент цільової функції мінімізує затримки в обміні повідомленнями між процесорами. Це особливо важливо для завдань, які активно взаємодіють між собою. Для цього потрібно розміщувати взаємопов'язані завдання на процесорах, які мають мінімальні затримки зв'язку. Оптимізація цього компонента передбачає аналіз топології системи та розташування процесорів, а також обчислення комунікаційних витрат між процесорами. Для мінімізації затримок використовуються графи залежностей між завданнями, які відображають обсяги та частоту обміну повідомленнями. Завдання, що ч[illegible] обмінюються даними, розміщуються на сусідніх процесорах з мінімальною затримкою зв'язку.

Метод оптимізації виконання завдань у багатопроцесорній системі з використанням реплікації та цільової функції можна подати у вигляді деталізованих кроків.

Крок 1. Ініціалізація системи та створення реплік коду завдань.

Крок 2. Початковий розподіл завдань.

Крок 3. Виконання завдань та моніторинг стану системи.

Крок 4. Прийняття рішення про міграцію завдань.

Крок 5. Вибір цільового процесора для міграції.

Крок 6. Виконання міграції завдання.

Крок 7. Оновлення цільової функції та оптимізація.

Крок 8. Завершення оптимізації та підтримка балансу.

Для реалізації методу оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорній вбудованій системі розроблено програму на C++. Програма враховує:

1) кількість процесорів та завдань;

2) схему міграції (на основі цільової функції);

3) стан процесів (виконуються чи очікують);

4) типи завдань (довгі чи короткі).

Програма реалізує:

1) ініціалізацію системи з заданою кількістю процесорів і завдань;

2) початковий розподіл завдань між процесорами;

3) моніторинг стану системи та динамічну міграцію завдань для оптимізації цільової функції.

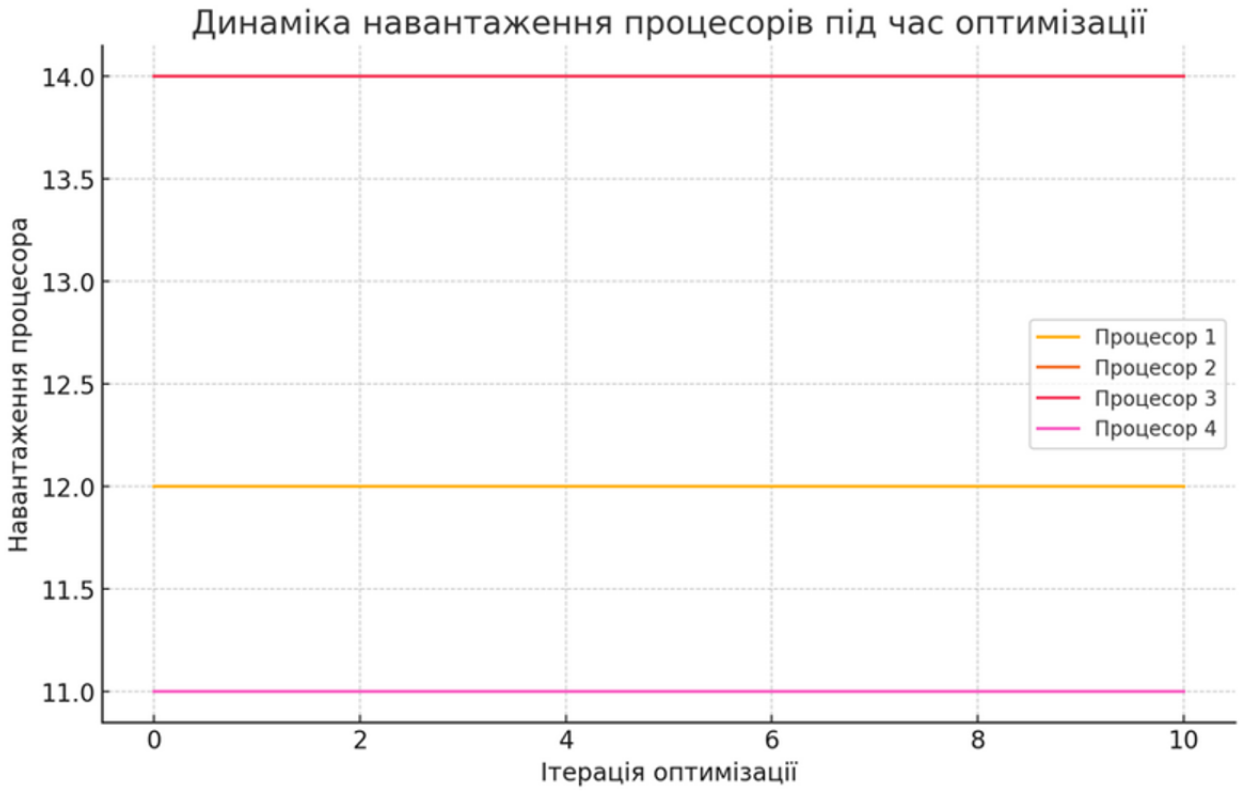


Рисунок 4.1 – Результати першої серії експерименту

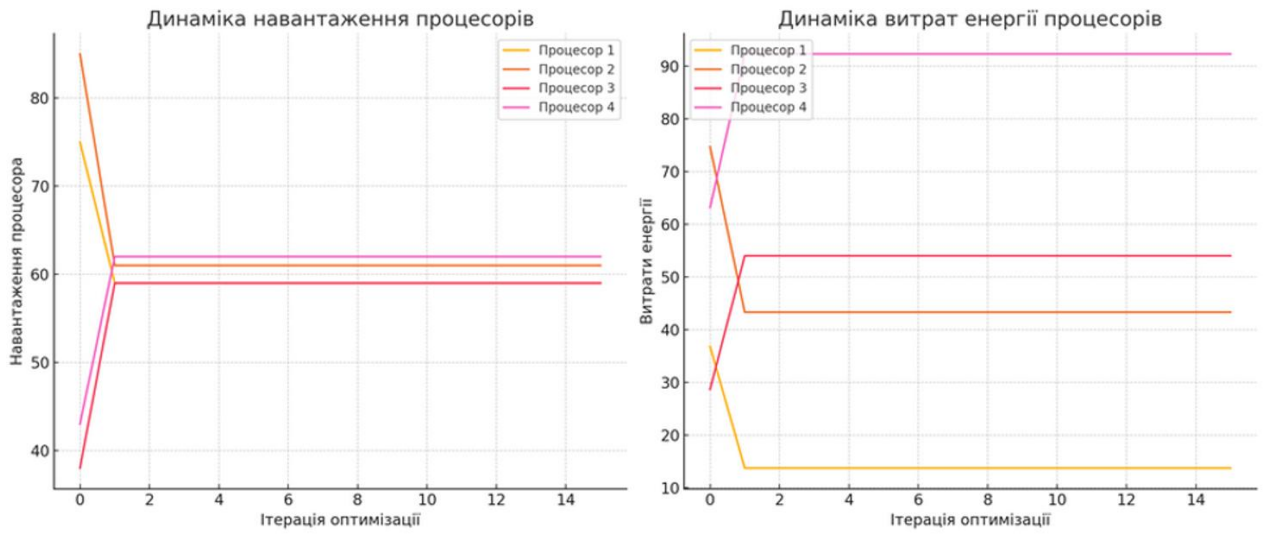


Рисунок 4.2 – Результати другої серії експерименту

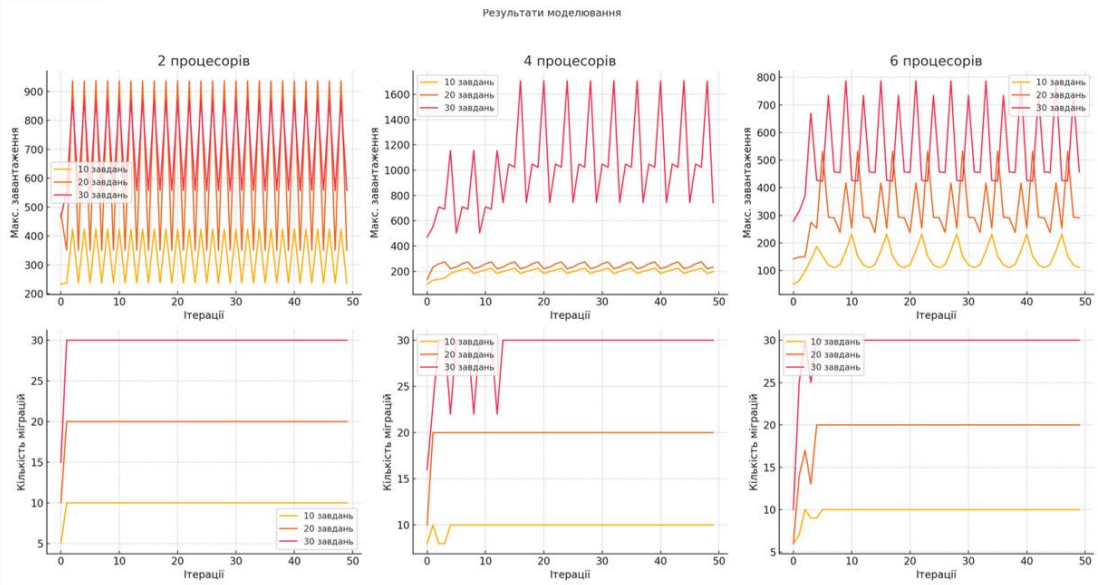


Рисунок 4.3 – Результати моделювання системи

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень розроблено новий метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах та отримано такі результати.

1. Проаналізовано відомі методи та засоби оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

2. Розроблено метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

3. Розроблено цільової функції оптимізації завдань у багатопроцесорних вбудованих системах.

4. Досліджено розроблений метод оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах, розроблено програму, проведено експеримент та моделювання схеми оптимізації.

ДОДАТОК Б

(обов'язковий)

Наукова праця здобувача

INTERNATIONAL SCIENTIFIC JOURNAL ISSN 2710-0766
«COMPUTER SYSTEMS AND INFORMATION TECHNOLOGIES»

https://doi.org/10.31891/csit-2025-1-14
UDC 004.7

Dmytro MARTINIUK, Oleksii LYHUN, Andriy DROZD
Khmelnytskyi National University
Oleksii BESEDOVSKYI
Simon Kuznets Kharkiv National University of Economics

TASK OPTIMISATION IN MULTIPROCESSOR EMBEDDED SYSTEMS

The relevance of this work lies in the fact that the existing task distribution in multiprocessor embedded systems plays a key role in the development of devices used in various industries. Despite the progress made, there are still many research challenges that require in-depth analysis and implementation of effective solutions. One of the main challenges is to ensure the reliability of embedded systems, especially in environments where safety is critical. Although the functionality of such systems is usually defined at the design stage, ensuring their stable operation in real time remains a challenge. It is necessary not only to guarantee the correctness of calculations, but also to adhere to time constraints, which requires new approaches to managing the resources of multiprocessor systems. Another important problem is the need to meet stringent real-time requirements. This is a characteristic feature of embedded systems, which differ from general-purpose systems that have more flexibility in functionality but do not guarantee such predictability and reliability. Therefore, optimization of task scheduling that takes into account the specifics of embedded systems requires further research. It is also important to take into account the variety of embedded systems, which are divided into control systems and streaming systems that have different data processing requirements. Control systems must respond quickly to environmental events while minimizing delays, while streaming systems process continuous data streams, requiring high throughput and efficiency. The development of universal solutions that can optimize the performance of both types of systems is an urgent task for scientists and engineers. Therefore, task optimization in multiprocessor embedded systems has significant potential for development and is relevant for reliability, real-time guarantees, and efficient resource management, which will contribute to the creation of more secure and productive systems.

In this paper, we develop a method for optimizing task execution using replication in a multiprocessor system, which allows to effectively minimize the total execution time, ensure load balance, and minimize communication delays. The peculiarity of the method is the implementation of task migration according to replication using the optimization objective function. An experiment with the system demonstrated that the chosen optimization method effectively balances the load, but additional objective functions are needed to optimize energy consumption. The simulation results show that an increase in the number of processors leads to a decrease in the maximum load and the number of migrations, an increase in the number of tasks increases the system load and the number of migrations at the initial stages, and the migration mechanism effectively balances the load, especially at the initial stages of execution.

The areas of further research are the detailing of embedded devices and their classification. For each class of embedded devices, it will be necessary to adapt the algorithms and method of task optimization, as well as to develop the target optimization function.

Keywords: Multiprocessor systems, embedded systems, computer systems, optimization

Дмитро МАРТИНЮК, Олексій ЛИГУН, Андрій ДРОЗД
Хмельницький національний університет
Олексій БЕСЄДОВСЬКИЙ
Харківський національний економічний університет імені Семена Кузнеця

ОПТИМІЗАЦІЯ ЗАВДАНЬ У БАГАТОПРОЦЕСОРНИХ ВБУДОВАНИХ СИСТЕМАХ

Актуальність даної роботи полягає в тому, що існуючі розподіл завдань у багатопроцесорних вбудованих системах відіграє ключову роль у процесі розробки пристроїв, що застосовуються в різноманітних галузях. Незважаючи на досягнутий прогрес, залишається чимало дослідницьких викликів, які вимагають глибокого аналізу та впровадження ефективних рішень. Одним із головних викликів є забезпечення надійності вбудованих систем, особливо в умовах, де критично важлива безпека. Хоча функціональні можливості таких систем зазвичай визначені ще на етапі проєктування, забезпечення їхньої стабільної роботи в режимі реального часу залишається складним завданням. Потрібно не тільки гарантувати правильність обчислень, але й дотримуватися часових обмежень, що вимагає нових підходів до управління ресурсами багатопроцесорних систем. Ще однією важливою проблемою є необхідність дотримання жорстких вимог реального часу. Це є характерною рисою вбудованих систем, які відрізняються від систем загального призначення, що мають більшу гнучкість у функціональності, але не гарантують такої передбачуваності та надійності. Тому оптимізація планування завдань, яка враховує специфіку вбудованих систем, потребує подальших досліджень. Важливо також враховувати різноманітність вбудованих систем, які поділяються на системи управління та потокові системи, що мають різні вимоги до обробки даних. Системи управління повинні оперативно реагувати на події зовнішнього середовища, мінімізуючи затримки, тоді як потокові системи обробляють безперервні потоки даних, вимагаючи високої пропускної здатності та ефективності. Розробка універсальних рішень, здатних оптимізувати продуктивність обох типів систем, є актуальним завданням для науковців та інженерів.Тому, оптимізація завдань у багатопроцесорних вбудованих системах має значний потенціал для розвитку і є актуальною щодо надійності, гарантій реального часу та ефективного управління ресурсами, що сприятиме створенню більш безпечних та продуктивних систем.

У даній роботі розроблено метод оптимізації виконання завдань з використанням реплікації у багатопроцесорній системі, який дає змогу ефективно мінімізувати загальний час виконання, забезпечити баланс навантаження і мінімізувати затримки зв'язку. Особливістю методу є реалізація міграції завдань згідно реплікації з використанням цільової функції оптимізації. Проведений експеримент з системою продемонстрував, що обраний метод оптимізації ефективно вирівнює навантаження, але для оптимізації енергоспоживання потрібні додаткові цільові функції. Результати моделювання

INTERNATIONAL SCIENTIFIC JOURNAL ISSN 2710-0766
«COMPUTER SYSTEMS AND INFORMATION TECHNOLOGIES»

показують, що збільшення кількості процесорів призводить до зменшення максимального завантаження і кількості міграцій, збільшення кількості завдань підвищує навантаження на систему та кількість міграцій на початкових етапах, механізм міграції ефективно балансує навантаження, особливо на початкових етапах виконання.

Напрямами наступних досліджень є деталізація вбудованих пристроїв і їх класифікація. Для кожного класу вбудованих пристроїв необхідно буде адаптувати алгоритми та метод оптимізації завдань, а також розробляти цільову функцію оптимізації.

Ключові слова: багатопроцесорні системи, вбудовані системи, комп'ютерні системи, оптимізація

Introduction

Task optimization in multiprocessor embedded systems is gaining increasing significance in the development of embedded devices across various industries. Despite significant progress, several research challenges still require in-depth analysis and effective solutions.

One of the key challenges is ensuring the reliability of embedded systems, as they often operate in safety-critical environments. Although their functional capabilities are well-defined at the design stage, guaranteeing continuous real-time operation remains a major challenge. It is essential to ensure not only the correctness of results but also their timely execution, necessitating the development of new approaches to resource management in multiprocessor systems.

Another challenge is the need to maintain strict real-time guarantees. Unlike general-purpose systems (e.g., personal computers), which offer more flexible functionality but lack high reliability and predictability, embedded systems must adhere to stringent real-time constraints. Optimizing task scheduling processes to account for these constraints remains an area of active research.

Furthermore, embedded systems can be categorized into control systems and streaming systems, each with distinct data processing requirements. Control systems respond to external events, requiring rapid reaction times and minimal latency. In contrast, streaming systems handle continuous data flows, demanding high throughput and efficient processing of large data volumes. Developing universal approaches that optimize performance for both types of systems remains a critical challenge for researchers and engineers.

In conclusion, research in task optimization for multiprocessor systems holds significant potential for advancement, particularly in the context of embedded devices. Future efforts should focus on addressing reliability issues, real-time guarantees, and efficient resource management, ultimately leading to the development of safer and more efficient systems.

Related works

Optimization of tasks in multiprocessor systems [1, 2], considering the application of computer systems in various fields, has become particularly significant in the context of embedded device development. Most embedded systems have the following characteristics: they are designed to perform a clearly defined set of functional capabilities known at the design stage; they must be reliable, as they often operate in safety-critical environments; and they must provide strict real-time guarantees, meaning their output must not only be correct but also produced within specific time constraints. These characteristics [3, 4] distinguish embedded systems from general-purpose systems, such as personal computers, which offer much greater flexibility in terms of functionality but place significantly less emphasis on guarantees and reliability in real-time operation.

Embedded systems can be categorized into two groups [5, 6] based on the type of functionality they provide: control systems wait for incoming events (or signals) from the environment and respond accordingly, making them suitable for applications such as industrial automation; streaming systems process a continuous, potentially infinite flow of data from the environment, and are commonly used in applications such as audio and video processing.

Given the complexity [7] of task optimization in multiprocessor systems, we will focus on embedded streaming systems as the object of our research. Examples of streaming applications include audio/video encoding and decoding, signal processing, computer vision, medical imaging, navigation systems, security camera systems, and many others.

Complex embedded systems [8, 9], such as those controlling autonomous vehicle driving, actually consist of multiple subsystems that interact with each other. In the case of autonomous driving, some of these subsystems fall into the category of streaming systems, while others belong to control systems.

For example, self-driving cars collect vast amounts of data in the form of continuous streams from onboard cameras and LiDAR sensors. These data streams must be constantly updated and processed to perform motion planning, which involves determining the optimal path and speed for the vehicle, and collision avoidance, which entails detecting and evading unexpected obstacles. These decisions are made by streaming subsystems [10] and then transmitted to control subsystems, which execute actions such as steering, braking, or accelerating the vehicle.

Autonomous vehicle control systems implement motion planning and collision avoidance algorithms [11], which require extremely high levels of interaction. Additionally, these algorithms must produce results within a short and predictable timeframe to ensure the vehicle can quickly respond to external events. For instance, if a pedestrian suddenly crosses the street in front of the vehicle, it must stop as quickly as possible. The high complexity of these algorithms, combined with the need for rapid execution, presents a challenge for system

designers to achieve high performance. This requirement is common across many modern embedded systems.

In fact, embedded systems have consistently demonstrated a growing demand for increased performance over the years. Until the mid-2000s, most computing systems were built using single-processor architectures [12], and this increasing performance demand was met by enhancing the computational power of a single processor. However, performance gains between successive generations of single processors began to slow significantly in the early 2000s, primarily due to [13, 14]: diminishing returns from new processor architecture improvements, a very slow increase in clock frequencies due to power leakage issues, and the growing gap between processor and memory speeds.

As a result, to further improve system performance, semiconductor manufacturers shifted their research and development efforts toward multiprocessor architectures starting in the mid-2000s. This technological trend [15, 16], which has influenced both general-purpose and embedded systems, remains promising. Indeed, an increasing number of architectures, proposed by both research institutions and industry, feature a growing number of processing elements. Today, embedded system designers frequently integrate multiple processors, memory units, interconnects, and other hardware peripherals into a single chip, forming what is known as a multiprocessor system-on-chip (MPSoC).

The design of embedded MPSoC [17] is a process that involves multiple stages. It begins with defining the required system functionality using an application model along with the specification of the execution platform on which the application will run. After a series of refinement stages, the design process is completed when a detailed description of the system's hardware and the software running on each processor is obtained.

Design trends, such as the widespread adoption of multiprocessor architectures and scalable interconnections, introduce new design methodologies aimed at achieving high system performance on a multiprocessor architecture. However, ensuring high system performance is not the sole objective of embedded system developers [20].

The term "system adaptability" [18, 19] refers to a system's ability to adjust to changing environmental conditions. These conditions are represented by parameters that can be classified into two categories [21, 22]. The first category includes application-related parameters that affect how the program is executed [23, 24]. For instance, the resolution of a video decoding application is typically defined by two parameters specifying the frame height and width. The second category consists of parameters describing the state of the execution platform. For example, a parameter may indicate the number of active processors in the system. Achieving system adaptability in response to changes in the second set of parameters those describing the execution platform's state is a promising direction for development.

The system is powered by a battery, and as the battery charge depletes, the user may choose to disable a certain number of components to reduce the system's power consumption. This may lead to a decrease in the application's quality of service, such as a reduced video encoding speed. Additionally, this scenario requires the system to redistribute tasks between those that will be disabled and those that will remain active, meaning that task mapping must be adjusted during execution.

System adaptability can be implemented in a computing system in various ways. In the case of embedded multiprocessors, applications are typically scheduled by the operating system, which acts as an intermediate layer. The OS serves as an interface between the application layer and the hardware layer, which resides at the bottom of the stack [25, 26].

Depending on the nature of the tasks and performance requirements, different approaches to migration management can be chosen. Flexible systems with adaptive control are well-suited for scenarios where resilience to changes and unexpected failures is crucial, while deterministic schemes are beneficial for highly regulated environments, such as flight control systems or mission-critical industrial applications. In any case, the choice of method depends on efficiency requirements, predictability, and the overall architecture of the hardware platform.

The objective function for task optimization in multiprocessor systems with migration.

Task optimization in multiprocessor systems with migration requires the development of an objective function that considers all relevant parameters to maximize resource utilization efficiency and ensure timely task execution. The key parameters to take into account include task execution time, migration delays, processor load, energy consumption, task priorities, and adherence to predefined deadlines. It is essential to minimize the overall completion time of all tasks, maintain load balancing across processors, and reduce application overhead associated with task migration. Additionally, optimization should consider energy consumption, as excessive resource utilization leads to increased power costs.

The objective function for such optimization can be defined as follows:

$$F = \min\left[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{k=1}^{K} E_k + \delta \sum_{i=1}^{J} T_j\right], \quad (1)$$

where C_i – task completion time i; L_m – the load factor of processor m, defined as the ratio of processor busy time to the total execution time; E_k – the energy consumption of processor k during task execution; T_j – the total migration time for task j, which accounts for data transfer delays and the setup time of the execution

environment on the new processor; $\alpha, \beta, \gamma, \delta$ – weight coefficients that determine the relative importance of each parameter in the objective function.

The goal of optimization is to minimize the objective function F, which balances task completion time, processor load, energy consumption, and migration costs. The weight coefficients can be adjusted according to the system's characteristics and optimization objectives. For example, in real-time systems, priority may be given to minimizing task completion time and meeting deadlines, whereas in systems with limited energy resources, minimizing energy consumption becomes the primary focus. Optimization of such an objective function can be performed using dynamic programming methods, evolutionary algorithms, or machine learning techniques for adaptive parameter tuning in real time.

In multiprocessor systems with migration, the number of tasks affecting the optimization process is a crucial parameter, as it determines the complexity of load balancing, migration costs, and overall execution time. Let N be the total number of tasks to be scheduled, and M be the number of processors in the system. The model assumes that each task i has its own execution time C_i, it can migrate between processors, causing delays T_i, and consume energy E_i during execution.

The objective function should minimize the total completion time of all tasks, balance the load across processors, and minimize energy consumption and migration costs. The refined objective function, considering these parameters, is defined as follows:

$$F = \min[\alpha \sum_{i=1}^{N} C_i + \beta \sum_{m=1}^{M} L_m + \gamma \sum_{i=1}^{N} E_i + \delta \sum_{i=1}^{N} T_i], \quad (2)$$

де $C_i = S_i + W_i + M_i$ – task completion time i, which consists of waiting time S_i, execution time W_i and migration time

M_i; $L_m = \frac{\sum_{i=1}^{N} W_{im}}{\sum_{i=1}^{N} c_i}$ – CPU load m, calculated as the ratio of the execution time of all tasks on the processor m to the total execution time; $E_i = P_i \cdot W_i$ – task energy consumption i, which depends on the processor's power P_i and execution time

W_i; $T_i = \sum_{k=1}^{K} t_{ik}$ – total migration time for the task i, where t_{ik} – task migration time i to the processor k; $\alpha, \beta, \gamma, \delta$ – weight coefficients that determine the relative importance of each parameter in the optimization objective function.

When the number of tasks significantly exceeds the number of processors in an embedded multiprocessor system, optimizing task distribution becomes much more complex due to several key factors. First, it is necessary to consider not only load balancing but also other constraints, such as energy consumption, memory limitations, real-time requirements, and interprocessor delays during data transfer.

As the number of tasks increases, several aspects of optimization become critically important. Migration time costs: With a large number of tasks, frequent migrations can significantly increase overall time costs due to the additional overhead of transferring states between processors. This may lead to situations where the benefits of load balancing are offset by the additional migration costs. Therefore, there is a need to determine the optimal migration points and minimize the migration frequency.

As the number of tasks increases, the optimization algorithm faces exponential growth in the solution space, i.e., the complexity of planning increases. A simple greedy algorithm, which works effectively for a small number of tasks, loses efficiency due to an incomplete exploration of possible configurations. In such cases, a promising approach is to use stochastic optimization methods, such as genetic algorithms, particle swarm algorithms, or machine learning, to predict the optimal distribution.

Energy efficiency becomes crucial as the number of tasks grows, causing the overall energy consumption of the system to increase as well. For embedded systems, this is critical since they often have limited power resources (e.g., batteries). In this context, energy consumption must also be optimized through dynamic voltage and frequency scaling, in combination with task migration.

With the increase in the number of tasks, the likelihood of interdependencies between them grows. This means that migrating one task may introduce delays for others, especially if tasks frequently exchange data. In such cases, the optimal distribution must take into account the topology of the dependencies between tasks, minimizing interprocessor communication costs.

In real systems, tasks can have a dynamic nature, appearing, completing, or changing their priority, meaning the algorithm must exhibit adaptability. This requires adaptive optimization approaches that can respond to such changes in real-time. A promising approach is to use reinforcement learning methods, which allow the algorithm to learn the optimal task distribution based on historical data.

A large number of tasks creates additional load on the cache and local memory of the processors. This can lead to frequent cache misses and increased memory access latency. In such cases, an effective strategy is to consider data locality when migrating tasks, i.e., placing tasks in such a way that they work with local data, minimizing data transfer between processors.

Although the replication approach for task migrations is effective for fast task migration, it shows limited efficiency when the number of tasks significantly exceeds the number of processors. This creates a need for hybrid

approaches that combine replication with other strategies, such as process spawning or dynamic scaling. For example, replication can be used for short tasks, while process spawning is more suitable for long, energy-intensive processes.

Another promising direction is the use of multi-agent systems, where each processor is considered an agent that autonomously makes decisions based on local information and communication with other agents. This approach helps avoid a central scheduler, which can become a bottleneck in a system with a large number of tasks.

It is also important to consider the heterogeneity of processors in modern embedded systems. For example, many systems use a combination of high-performance and energy-efficient processors. Optimization in such systems should take into account not only the load but also the energy efficiency of each type of processor.

Thus, there is a need for further improvement and development of adaptive, energy-efficient, and scalable algorithms that can optimally distribute a large number of tasks while considering the constraints of embedded multiprocessor systems. An important direction is the integration of machine learning methods for load prediction and real-time adaptation of optimization algorithms.

Process migration algorithms in multitasking embedded systems

Process migration algorithms in multitasking embedded systems are aimed at optimally distributing computational tasks across multiple processors or cores. The main goal of migration is to balance the load, minimize task execution time, reduce energy consumption, and ensure real-time compliance for critical tasks. Embedded systems have limited resources, so efficient process migration management is particularly important. Depending on the task distribution approach and migration management, algorithms can be divided into several categories: global, cluster-based, hierarchical, and hybrid.

Global algorithms treat all processors as a common set of resources. Any process can run on any processor, and migration occurs dynamically based on the load. Key approaches include load balancing, load distribution, and dynamic load distribution. Load balancing algorithms move processes between processors to achieve uniform load. They use current metrics such as processor utilization or the number of process queues. Examples of such algorithms include the "Least Loaded Processor" algorithm, where tasks are moved to the processor with the least load, and "Round Robin Migration," which distributes tasks in turn among all processors.

Load distribution algorithms assign tasks to free processors at the moment of their creation, rather than during execution, which reduces application overhead for migration but may lead to uneven load distribution. Dynamic load distribution uses real-time system state monitoring to dynamically migrate tasks. An example is the "Work Stealing" algorithm, where processors with low loads steal tasks from overloaded processors, ensuring a more even distribution of tasks.

Cluster migration algorithms divide the system into groups of processors (clusters), and migration occurs within the cluster. This reduces data transfer delays, as clusters often share common cache memory or have faster data exchange. Tasks are scheduled within a single cluster, and if the load is not balanced, task migration between clusters is possible. Additionally, affinity-based migration algorithms are used, which aim to keep tasks on the processors where they were previously executed to minimize cache misses and application migration overhead.

Hierarchical algorithms combine global and cluster-based methods. The system is considered at multiple levels: at the top level, migration occurs between clusters; at the lower level, migration happens within the cluster. This structure allows for a compromise between the flexibility of global algorithms and the efficiency of local ones. An example is the "Multilevel Queue Migration" algorithm, where tasks are grouped by priorities, and migration occurs within each queue.

Hybrid algorithms combine multiple approaches to achieve a balance between efficiency and complexity. They can dynamically switch between different strategies depending on the current system state.

The main challenges for migration algorithms in embedded systems are limited resources, real-time requirements, low energy consumption, and minimizing delays. Effective optimization involves minimizing application overhead for migration, as task movement requires time for data transfer and reinitialization of the execution environment, considering cache affinity since process migration between cores can cause cache misses, which negatively affect performance. It also involves adapting to dynamic changes in load and energy consumption, using machine learning methods to predict load and dynamically adjust scheduling parameters.

The goal is to minimize execution time and energy consumption while maintaining balanced load across processors. We define the objective function as follows:

$$\min\left(\max_{p\in P} T_p\right) + \alpha \sum_{p\in P} E_p, \tag{3}$$

where T_p – the total execution time of tasks on a processor p, E_p – the energy consumption of the processor p, α – a coefficient that determines the weight of energy consumption in the objective function.

We will develop optimization algorithms for various parameters in embedded multiprocessor systems.

Algorithm 1 takes into account a ring topology and performs task migration between neighboring processors to achieve load balancing.

Step 1.1. Initialization. Tasks are distributed among processors in a random order.

Step 1.2. Load evaluation. For each processor p, the load is calculated $L_p = \sum C_i$ for all tasks on it.

Step 1.3. Balancing. If the load difference between neighboring processors exceeds a threshold, Δ, migration of a task is performed from the more loaded processor to the less loaded one. The migration occurs in a ring, reducing transfer delays.

Step 1.4. Update the load and repeat Steps 2–3 until the load is balanced.

Step 1.5. Execute tasks within the time quantum Q.

Step 1.6. Check if all tasks are completed. If all tasks are done, finish the process; otherwise, return to Step 2.

Algorithm 2 takes into account task priorities and the possibility of migration between neighboring processors, maintaining processor affinity to reduce cache misses.

Step 2.1. Initialization. Assign tasks to processors based on priority. Tasks with higher priority are assigned first.

Step 2.2. Execute tasks during the time quantum Q. If the task is not completed, it remains in the processor's queue.

Step 2.3. Performance evaluation. If a high-priority task cannot be completed on time on the current processor, migration is performed to a neighboring processor with a lower load.

Step 2.4. Affinity check. Migration is allowed only if the task has not been executed previously on the selected processor to minimize cache misses.

Step 2.5. Repeat Steps 2–4 until all tasks are completed.

Algorithm 3 is aimed at minimizing energy consumption while considering the ring topology and time quantum.

Step 3.1. Initialization. Distribute tasks among processors, taking into account their energy consumption. Tasks with the lowest energy consumption are assigned first.

Step 3.2. Energy consumption evaluation. Calculate the current energy consumption for each processor.

Step 3.3. Optimization. If the energy consumption exceeds the threshold E_{max}, migration is performed to a neighboring processor with lower energy consumption.

Крок 3.4. Execution of tasks during the time quantum Q.

Step 3.5. Update energy consumption and repeat Steps 2–4 until all tasks are completed.

The application of these algorithms enables efficient load balancing, optimal energy consumption, and adherence to real-time constraints in multitasking embedded systems.

Migration algorithms in multiprocessor embedded systems are widely used in various industries where there is a need to efficiently utilize computational resources, minimize task execution time, and optimize energy consumption. Let's consider specific application examples for the three developed algorithms.

Algorithm 1 is particularly effective in systems with a ring topology, such as network routers and multiprocessor computing clusters. In telecommunications networks, routers process data packets, and the traffic volume can dynamically change. In such systems, routers are often organized in a ring topology to ensure fault tolerance and reduce data transmission delays. When one router receives an excessive number of packets, and neighboring routers have available computational resources, task migration for packet processing occurs to less-loaded routers. This ensures load balancing, minimizes data processing delays, and prevents overloading individual routers. As a result, network performance improves, and data transmission latency is reduced.

Algorithm 2 is effective in real-world systems with critical time constraints, such as automotive embedded systems or UAV (unmanned aerial vehicle) control systems. In modern vehicles, numerous electronic control units are responsible for various tasks, such as braking systems, course stabilization, object recognition using cameras, etc. These tasks have different priorities for instance, the braking system has the highest priority. If the processor responsible for processing the braking signal is overloaded with lower-priority tasks (such as object recognition), lower-priority tasks are migrated to neighboring processors with lighter loads. High-priority tasks are executed immediately, ensuring that real-time constraints are met (for example, minimizing the delay in activating the brakes). This improves vehicle safety and ensures the reliable operation of critical systems.

The algorithm is suitable for embedded systems where energy consumption is critical, such as in portable devices or Internet of Things (IoT) systems. Consider a smart watch with multiple processors performing various tasks such as heart rate monitoring, GPS navigation, message processing, and more. Tasks have different energy requirements depending on the complexity of the computations. High-energy tasks, such as GPS navigation processing, can be executed on processors with higher energy efficiency. If one processor becomes overloaded and its energy consumption exceeds the acceptable level, tasks are migrated to a neighboring processor with lower energy consumption. This ensures optimal load distribution with minimal overall energy consumption, extending the device's battery life.

In server clusters for processing large volumes of data, migration algorithms are used for dynamic task distribution between processors to minimize execution time and reduce energy consumption. In Unmanned Aerial Vehicle (UAV) control systems, the algorithm ensures the adherence to real-time constraints for critical tasks (such as flight stabilization and obstacle avoidance), while less critical computations (such as video processing) can migrate to other processors.

Embedded systems in medical devices (e.g., pacemakers) can use Algorithm 3 to ensure continuous operation under limited energy consumption, migrating computational tasks between processors to optimize energy usage.

To achieve system adaptability through process migration, it is also necessary to determine how to transition between the current mapping and the next one. In other words, a mechanism for performing process migration must be anticipated. In the proposed approach, this mechanism is implemented by middleware, specifically through process migration. Let's consider the problem of defining and implementing a process-oriented migration mechanism that meets three requirements: if the migration process is initiated, it must be completed within certain, known deadlines, ensuring predictability; process migration can be triggered at any time in the system, meaning the mechanism must account for scenarios where process migration is required in response to a hardware failure, the occurrence of which is unknown; the code used for process migration must be automatically generated, without manual developer intervention, to relieve developers from the labor-intensive and error-prone task of manually inserting the necessary migration code.

The predictable process migration mechanism allows processes to be redistributed during execution across processors, which is a fundamental requirement for system adaptability. The uniqueness of the solution lies in the fact that by using the operational semantics and structure of the process, migration can actually begin at any point during the execution of the main process without needing to move large states. Furthermore, the upper bound for application costs of process migration can be determined based on the topology and buffer size. Finally, the code used for process migration is minimally invasive to the original code structure and can be fully automated in generation.

Resource management during execution is a well-studied topic in the scheduling of general-purpose distributed systems. Specifically, within this context, mechanisms for process migration must be developed and assessed, enabling dynamic load balancing, fault tolerance, and improving system administration and data access locality. In recent years, execution-time management has become increasingly popular and is also being applied in multiprocessor embedded systems. This domain imposes strict constraints such as cost, power, and predictability, all of which need to be carefully considered by execution-time and process migration management mechanisms.

The proposed migration approach will use fully distributed memory with no direct remote memory access. This means that the task processing element can only directly access its own local memory. All communication and synchronization between processes mapped to different processors can occur only through messaging. The approach for implementing system adaptability involves deploying application processes modeled for their execution time redistribution to adapt the system to changing operating conditions, such as variations in quality of service requirements, resource availability, or power budget constraints. Specifically, system adaptability is supported by using specialized middleware within the software stack. At the top of the software stack, applications are specified as a set of processes implemented as separate threads. An example of a thread representing a process. The basic process structure will be modified to facilitate the implementation of the predictable process migration mechanism. At the bottom of the software stack, the operating system is responsible for all types of process management (creation, deletion, priority setting, suspension, or resumption). These functions are important for managing the system's execution time, particularly for process migration execution. Furthermore, each processor has multitasking capabilities provided by the OS. In the case of a "many-to-one" mapping, where more than one process is mapped to a single processing element, scheduling is governed by data. This means that the process continues its successive iterations until it blocks on a read or write. When the process blocks, it passes control to the next process in the ready queue using a round-robin system. Between the applications and the operating system, there will be middleware consisting of two main components. The first is the communication mechanism, which implements communication and synchronization between processes located on separate processors. A mandatory component is process migration, which is mainly responsible for actions performed during process migration: coordinating the creation and deletion of processes across different processors; ensuring the correct transfer of the process state during migration.

Thus, algorithms for process migration in multitasking embedded systems have been developed, and their application enables efficient load balancing, optimal energy consumption, and adherence to real-time constraints in multitasking embedded systems.

The method of task optimization in embedded systems with multiple processors

The task optimization method will be based on replication during task migration in a multiprocessor system, and the optimization will be carried out using a developed optimization objective function that takes into account the main system parameters, such as task execution time, processor load, memory usage, and communication delays. Optimizing task execution using the replication method involves finding the distribution of processes among processors that minimizes the total task execution time while maintaining load balancing on the processors and minimizing communication costs between them. The optimization objective function is defined as follows:

$$F = \alpha \sum_{i=1}^{N} T_i + \beta \sum_{j=1}^{M} L_j + \gamma \sum_{k=1}^{P} C_k, \quad (4)$$

where T_i — execution time of the i-th task, L_j — load j-th processor, C_k — message passing delay between processors; α, β and γ — weight coefficients that reflect the optimization priorities.

The first component of the objective function aims to minimize the total execution time of all tasks. To achieve this, it is necessary to ensure that tasks are executed as parallel as possible and do not have to wait for resources. The replication method enables quick task switching between processors, as task code copies are already stored in the local memory of all potential processors. Optimizing this component involves dynamically moving tasks to the least loaded processors, as well as minimizing task idle time.

The second component of the objective function is responsible for load balancing between processors. An uneven distribution of tasks can lead to some processors being overloaded while others remain idle. To address this issue, a dynamic load balancing strategy is used, which involves regularly assessing processor load and migrating tasks from overloaded processors to less loaded ones. The replication method significantly simplifies this process since task code is already available on all processors, meaning only the task state needs to be transferred. This reduces the task migration time and ensures quick task resumption.

The third component of the objective function minimizes communication delays between processors. This is especially important for tasks that actively interact with each other. To achieve this, interrelated tasks need to be placed on processors that have minimal communication latency. Optimizing this component involves analyzing the system's topology and processor placement, as well as calculating communication costs between processors. To minimize delays, dependency graphs between tasks are used to reflect the volume and frequency of message exchanges. Tasks that frequently exchange data are placed on neighboring processors with minimal communication latency.

The optimization method for implementing the objective function consists of several stages. The first stage is the initial distribution of tasks across processors. This can be done using a greedy algorithm, which sequentially assigns tasks to the least loaded processor. The second stage is the dynamic migration of tasks during execution, based on the analysis of the current load on processors and the volume of communication between them. If a processor is found to be overloaded, some tasks are migrated to other processors, taking into account the minimization of communication delays. The third stage involves periodic optimization of task distribution by evaluating possible task placement combinations and selecting the one that minimizes the objective function.

To find the optimal task distribution, combinatorial optimization is used, as the number of possible task distributions between processors increases exponentially with the number of processors and tasks. An effective approach is the use of genetic algorithms or simulated annealing, which provide a quick search for the optimal solution within the space of possible distributions. The genetic algorithm simulates the process of natural selection, using crossover and mutation operations to generate new solutions based on the current ones. Simulated annealing makes random changes to the current task distribution, gradually reducing the probability of accepting worse solutions, which helps avoid local minima of the objective function.

Thus, the replication method combined with the optimization objective function ensures the efficient execution of tasks in a multiprocessor system, minimizing overall execution time, ensuring load balancing between processors, and reducing message exchange delays. This approach is particularly effective for systems with high load dynamics and complex computational node topologies.

The optimization method for task execution in a multiprocessor system using replication and an objective function can be presented in detailed steps. This approach allows dynamic task redistribution between processors to minimize total execution time, balance the load, and minimize message exchange delays. Each step of this method aims to achieve the optimal task distribution within the system, maintaining flexibility and speed in the migration process due to pre-created task code replicas.

Step 1. System initialization and task code replication at the first stage, the following actions are performed: the total number of tasks and the computational resources of the system, including the number of processors and their capacities, are analyzed; for each task, code replicas are created on all processors that are potentially capable of executing the task, ensuring that processors are ready to accept tasks without additional time costs for code transfer; a global state table is created, which stores information about the execution status of each task on all processors, as well as local tables on each processor to track the task execution locations. Additionally, the optimization objective function is initialized, with its value computed according to formula (5).

Step 2. Initial Task Distribution. The initial task distribution is performed between processors based on a greedy algorithm. Tasks are assigned to processors with the least current load. The system's topology is considered to minimize communication delays between processors. Tasks with high interaction levels are placed on neighboring processors. A target function is calculated for the initial task distribution, which includes execution time, processor load, and communication delays. The initial placement is fixed in a global state table.

Step 3. Task Execution and System State Monitoring. Tasks are launched on the corresponding processors selected in the previous step. During execution, the state of each task is regularly saved in the global state repository, including the values of registers, instruction pointer, call stack, and local variables. Processor load and message exchange volumes between processors are monitored. Regular monitoring of the target function values is carried out to assess the efficiency of the current task distribution.

INTERNATIONAL SCIENTIFIC JOURNAL ISSN 2710-0766
«COMPUTER SYSTEMS AND INFORMATION TECHNOLOGIES»

Step 4. Decision on Task Migration. If the target function exceeds predefined threshold values, the migration process is initiated. The current load on each processor and communication delays are analyzed. Bottlenecks such as overloaded processors or high communication delays between tasks that are actively exchanging data are identified. Tasks for migration are selected, particularly those with low data locality or those generating significant volumes of messages to remote processors.

Step 5. Selection of the Target Processor for Migration. An evaluation of all potential processors for executing the migrating task is performed based on the following criteria: current processor load; communication delays with other processors executing related tasks; available free memory and computational core availability. The processor that minimizes the target function, balancing load and communication overhead, is chosen. Resources are reserved on the selected processor to avoid conflicts during migration.

Step 6. Task Migration Execution. The task is paused on the current processor, and its state is fully saved in the global state repository. Since the task's code is already on the new processor, only the task's state is transferred. On the target processor, the task is restored from the saved state. The local state tables on both processors are updated, as well as the global state table to reflect the new task location. The task continues execution from the point where it was paused.

Step 7. Target Function Update and Optimization. After the migration is executed, the target function is recalculated considering the new task distribution. If the target function has not reached its minimum value, the migration process is repeated for other tasks. Optimization is performed using one of the following methods: the genetic algorithm executes crossover and mutations to create new task distribution combinations; the simulated annealing method makes random changes to the task distribution, gradually reducing the probability of accepting worse decisions. The optimization process continues until the target function reaches the specified minimum or a stable task distribution is achieved.

Step 8. Optimization Completion and Load Balancing Maintenance. The optimized distribution is fixed in the global state table. Dynamic load balancing is maintained by periodically updating the target function and reconfiguring the task distribution. Data consistency and state synchronization between processors are ensured.

Thus, the detailed method of task execution optimization using replication in a multiprocessor system effectively minimizes total execution time, ensures load balancing, and minimizes communication delays. A key feature of the method is the implementation of task migration according to replication using the optimization target function.

Research on the Effectiveness of the Task Optimization Method in Multiprocessor Embedded Systems

A program in C++ has been developed to implement the task optimization method using replication and a target optimization function in a multiprocessor embedded system. The program takes into account:

1) the number of processors and tasks;
2) the migration scheme (based on the target function);
3) the state of processes (whether they are running or waiting);
4) the types of tasks (long or short).

The program implements:

1) initialization of the system with a specified number of processors and tasks;
2) initial task distribution among processors;
3) system state monitoring and dynamic task migration to optimize the target function.

The program implements the task optimization method using replication and a target optimization function in a multiprocessor embedded system. The main features of the program include:

1) initialization of the system with random tasks (short or long);
2) initial task distribution among processors based on their load;
3) dynamic optimization with task migration for load balancing;
4) output of the migration process in the console.

Using the developed program, the first series of experiments was conducted. Figure 4.1 shows the graphs of the optimization parameters for a simple case with processes and processors, meaning there is no increased load level in the multiprocessor embedded system.

Program results:

1) initial processor load state - [12, 14, 14, 11];
2) final state after optimization - [12, 14, 14, 11].

The graph shows the dynamics of processor load during optimization. There was no significant redistribution since the initial distribution was relatively balanced.

Figure 4.2 illustrates the graphs of the optimization parameters for a simple case with processes and processors, meaning there is no increased load level in the multiprocessor embedded system. Unlike the first series of experiments, the dynamics of processor load and energy consumption of processors are highlighted.

INTERNATIONAL SCIENTIFIC JOURNAL ISSN 2710-0766
«COMPUTER SYSTEMS AND INFORMATION TECHNOLOGIES»

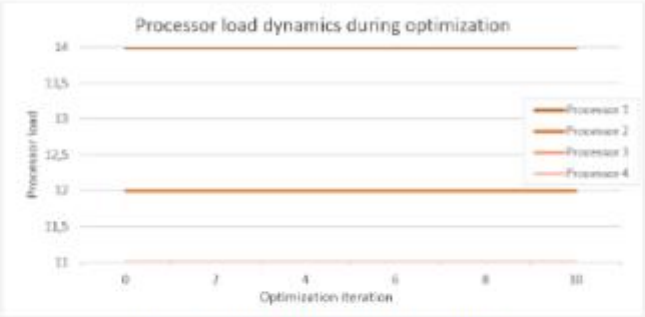


Fig. 1. Results of the first series of experiments

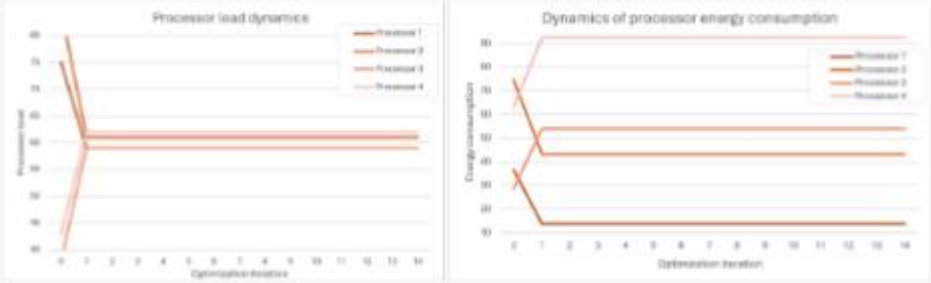


Fig. 2. Results of the second series of experiments

Experiment results:

1) initial processor load - [75, 85, 38, 43];
2) final load after optimization - [59, 61, 59, 62];
3) initial energy consumption: [36.85, 74.71, 28.71, 63.20];
4) final energy consumption: [13.78, 43.34, 54.03, 92.31].

Experiment analysis: Initially, the load was intentionally unbalanced to test the effectiveness of the optimization method. The algorithm performed task migration, balancing the load across processors, as evidenced by the final state where all processors have approximately the same load. However, energy consumption became less uniform, as tasks with higher energy consumption remained on certain processors, while other processors received shorter but more energy-intensive tasks.

In the graphs, the left part shows how the processor load levels out over time, while the right part illustrates the dynamics of energy consumption, which was not a criterion for optimization, so uniformity was not achieved here.

This experiment demonstrates that the chosen optimization method effectively balances the load, but additional target functions are needed for optimizing energy consumption.

Conclusions

The developed task execution optimization method using replication in a multiprocessor system effectively minimizes the overall execution time, ensures load balancing, and reduces communication delays. A key feature of the method is the implementation of task migration according to replication using an optimization target function.

The conducted experiment with the system demonstrated that the chosen optimization method effectively balances the load, but additional target functions are needed for optimizing energy consumption.

Simulation results show that increasing the number of processors leads to a reduction in maximum load and the number of migrations. Increasing the number of tasks increases the system load and the number of migrations during the initial stages, but the migration mechanism effectively balances the load, especially in the early stages of execution. Future research directions include further detailing embedded devices and their classification. For each class of embedded devices, algorithms and task optimization methods will need to be adapted, and an optimization target function will need to be developed.

References

1. Zhou Y., Zhang E., Guo H., Fang Y., Li H. Lifting path planning of mobile cranes based on an improved RRT algorithm. *Adv. Eng. Inform.* 2021, 50, 9. https://doi.org/10.1016/j.aei.2021.101376

2. Zhu A.M., Zhang Z.Q., Pan W. Crane-lift path planning for high-rise modular integrated construction through metaheuristic optimization and virtual prototyping. *Autom. Constr.* 2022, 141, 21. https://doi.org/10.1016/j.autcon.2022.104434

3. Guo H., Zhou Y., Pan Z., Zhang Z., Yu Y., Li Y. Automated Selection and Localization of Mobile Cranes in Construction Planning. *Buildings* 2022, 12, 580. https://doi.org/10.3390/buildings12050580

4. Wang J., Zhang Q., Yang B., Zhang B. Vision-Based Automated Recognition and 3D Localization Framework for Tower Cranes Using Far-Field Cameras. *Sensors* 2023, 23, 851. https://doi.org/10.3390/s23104851

5. Huang L., Pradhan R., Dutta S., Cai Y. BIM4D-based scheduling for assembling and lifting in precast-enabled construction. *Autom. Constr.* 2022, 133, 14. https://doi.org/10.1016/j.autcon.2021.103999

6. Song Y., Xin R., Chen P., Zhang R., Chen J., Zhao Z. Autonomous selection of the fault classification models for diagnosing microservice applications. *Future Generation Computer Systems*, 2024. 153, pp.326-339. https://doi.org/10.1016/j.future.2023.12.005

7. Tao L., Lu X., Zhang S., Luan J., Li Y., Li M., Li Z., Yu Q., Xie H., Xu,R., Hu C. Diagnosing Performance Issues for Large-Scale Microserv ice Systems With Heterogeneous Graph. *IEEE Transactions on Services Computing*. 2024. https://ieeexplore.ieee.org/document/10533869

8. Chen Y., Xu D., Chen N., Wu X. FRL-MFPG: Propagation-aware fault root cause location for microservice intelligent operation and maintenance. *Information and Software Technology*. 2023. 153, p.107083. https://doi.org/10.1016/j.infsof.2022.107083

9. Li X., Wen P., Chen P., Chen J., Wen X., Xia Y. An effective parallel convolutional anomaly multi-classification model for fault diagnosis in microservice system. *Software Quality Journal*. 2024. Pp.1-18. https://doi.org/10.21203/rs.3.rs-5267111/v1

10. Mazreemolla Z.P., Rasoolzadegan A. An effective failure detection method for microservice-based systems using distributed tracing data. *Engineering Applications of Artificial Intelligence*. 2024. 133, p.108558. https://doi.org/10.1016/j.engappai.2024.108558

11. Zhang S., Jin P., Lin Z., Sun Y., Zhang B., Xia S., Li Z., Zhong Z., Ma M., Jin W., Zhang, D. Robust failure diagnosis of microservice system through multimodal data. *IEEE Transactions on Services Computing*. 2023.16(6), pp.3851-3864. https://doi.org/10.48550/arXiv.2302.10512

12. Bedratyuk L. and Savenko O., The star sequence and the general first Zagreb index, *MATCH Communications in Mathematical and in Computer Chemistry*. (2018) 79, 407–414. https://doi.org/10.48550/arXiv.1706.00829

13. Zhang B., Wang X., Wang H. Virtual machine placement strategy using cluster-based genetic algorithm. *Neurocomputing*. 2021. 428, Pp. 310–316. https://doi.org/10.1016/j.neucom.2020.06.120

14. Wei P., Zeng Y., Yan B., Zhou J., Nikougoftar E. Vmp-a3c: virtual machines placement in cloud computing based on asynchronous advantage actor-critic algorithm. *J. King Saud Univ. Comput. Inf. Sci.* 2023. 35, 101549. https://doi.org/10.1016/j.jksuci.2023.04.002

15. Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S. Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software*. 2024. 207, p.111857. https://doi.org/10.1016/j.jss.2023.111857

16. Yin J., Li J., Fang Y., Yang A. Service scheduling optimization for multiple tower cranes considering the interval time of the cross-tasks. *Math. Biosci. Eng.* 2023, 20. Pp. 5993–6015. https://www.aimspress.com/article/doi/10.3934/mbe.2023259

17. Zhang Z.Q., Ma S.L., Jiang X.Y. Research on Multi-Objective Multi-Robot Task Allocation by Lin-Kernighan-Helsgaun Guided Evolutionary Algorithms. *Mathematics* 2022, 10, 4714. https://doi.org/10.3390/math10244714

18. Li K., Duan T., Li Z., Xiahou X., Zeng N., Li Q. Development Path of Construction Industry Internet Platform: An AHP—TOPSIS Integrated Approach. *Buildings* 2022, 12, 441. https://doi.org/10.3390/buildings12040441

19. Lysenko S., Bobrovnikova K., Savenko O., Kryshchuk A. BotGRABBER: SVM-Based Self-Adaptive System for the Network Resilience Against the Botnets' Cyberattacks. *Communications in Computer and Information Science*. 2019. Vol. 1039. Pp.127-143, ISSN: 1865-0929. https://doi.org/10.31891/2307-5732-2024-331-2

20. Lysenko S., Savenko O., Bobrovnikova K., Kryshchuk A. Self-adaptive system for the corporate area network resilience in the presence of botnet cyberattacks. *Communications in Computer and Information Science*, 2018.- 860, - Pp. 385-401. https://link.springer.com/chapter/10.1007/978-3-319-92459-5_31

21. Savenko O., Sachenko A., Lysenko S., Markowsky G., Vasylkiv N. (2020). BOTNET DETECTION APPROACH BASED ON THE DISTRIBUTED SYSTEMS. *International Journal of Computing*, *19*(2), 190-198. https://doi.org/10.47839/ijc.19.2.1761

22. Kashtalian A., Lysenko S., Savenko O., Nicheporuk A., Sochor T., & Avsiyevych V. Multi-computer malware detection systems with metamorphic functionality. *Radioelectronic and Computer Systems*. 2024. Vol. 1. Pp. 152-175. doi: https://doi.org/10.32620/reks.2024.1.13

23. Savenko B., Kashtalian A., Lysenko S., Savenko O. Malware Detection By Distributed Systems with Partial Centralization. *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Dortmund, Germany, 2023, pp. 265-270, doi: 10.1109/IDAACS58523.2023.10348773. https://ieeexplore.ieee.org/document/10348773

24. Chong G., Ramiah H., Yin J., Rajendran J., Wong W.R., Mak P.-I., Martin R.P. CMOS cross-coupled differential-drive rectifier in subthreshold operation for ambient RF Energy Harvesting—Model and analysis. IEEE Trans. Circuits Syst. II Express Briefs 2019, 66, 1942–1946. https://ieeexplore.ieee.org/document/8630669

25. Kumar S., Gupta U., Singh A.K., Singh A.K. Artificial Intelligence: Revolutionizing Cyber Security in the Digital Era. *J. Comput. Mech. Manag.* 2023, 2, 31–42. https://doi.org/10.57159/gadl.jcmm.2.3.23064

26. Chen J., Zhang R., Chen P., Ren J., Wu Z., Wang Y., Li X., Xiong L. MTG_CD: Multi-scale learnable transformation graph for fault classification and diagnosis in microservices. *Journal of Cloud Computing*. 2024. 13(1), p.103. https://doi.org/10.1186/s13677-024-00666-0

Dmytro Martiniuk **Дмитро Мартинюк**	master's degree student, Khmelnytskyi National University, Khmelnytskyi, Ukraine, e-mail: martiniyuk.dim14@gmail.com https://orcid.org/0009-0002-3524-872X	Магістрант, Хмельницький національний університет, м. Хмельницький, Україна
Oleksii Lyhun **Олексій Лигун**	PhD student, Khmelnytskyi National University, Khmelnytskyi, Ukraine e-mail: oleksii.lyhun@gmail.com https://orcid.org/0009-0004-5727-5096	Аспірант, Хмельницький національний університет, м. Хмельницький, Україна
Andriy Drozd **Андрій Дрозд**	PhD student, Khmelnytskyi National University, Khmelnytskyi, Ukraine, e-mail: andriydrozdit@gmail.com https://orcid.org/0009-0008-1049-1911	Аспірант, Хмельницький національний університет, м. Хмельницький, Україна
Oleksii Besedovskyi **Олексій Бесєдовський**	Candidate of Sciences in Economics, Associate Professor, Associate Professor of the Information Systems Department, Simon Kuznets Kharkiv National University of Economics, Kharkiv, e-mail: oleksii.besedovskyi@hneu.net https://orcid.org/0000-0002-9161-4061	Кандидат економічних наук, доцент, доцент кафедри інформаційних систем Харківського національного економічного університету імені Семена Кузнеця, м. Харків

ДОДАТОК В

Результати перевірки на плагіат

StrikePlagiarism.com

Дата звіту 4/15/2025
Дата редагування 4/15/2025

Документ прийнятий

Звіт подібності

метадані

Назва організації
Khmelnytskyi National University
Заголовок
МАРТИНЮК_Метод оптимізації завдань у багатопроцесорних системах
Автор
Дмитро МАРТИНЮК Науковий керівник / Експерт
підрозділ
Кафедра комп'ютерної інженерії та інформаційних систем

Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

Заміна букв	ß	1
Інтервали	A→	0
Мікропробіли	·	168
Білі знаки	ß	1
Парафрази (SmartMarks)	a	66

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.

5.78% 5.78% КП 1

0.18% 0.18% КЦ

25
Довжина фрази для коефіцієнта подібності 2

20229
Кількість слів

161147
Кількість символів

Anti-Plagiarism v-15.260 Educational

Максимальне співпадіння з одним документом 15.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

	Документ		Сумарний збіг по Базі Даних	
ID: 229435 Назва: МКР Метод оптимізації завдань у багатопроцесорних системах Додано в БД: 2025-04-15 Автора: Дмитро МАРТИНЮК Керівники: Світлана САЧЕНКО Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	152951	1211	24723 (16%)	186 (15%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
193124	Назва: Звіт з ПДП Метод оптимізації завдань у багатопроцесорних системах Додано в БД: 2025-03-21 Автора: Мартинюк Д.В. Керівники: Павлова О.О. Консультанти: Опоненти:	23417 (15.0%)	171 (14.0%)

ДОДАТОК Г

Програмний код

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

struct Task {
    int id;
    int duration; // Час виконання завдання
    bool isLong;  // Довге чи коротке завдання
    bool isCompleted;
    int currentProcessor;
};

struct Processor {
    int id;
    vector<Task> tasks;
    int load;
};

int numProcessors = 4;
int numTasks = 10;
vector<Processor> processors(numProcessors);
vector<Task> tasks;

void initializeSystem() {
    srand(time(0));
    for (int i = 0; i < numTasks; i++) {
        Task task;
        task.id = i;
        task.duration = rand() % 10 + 1; // Від 1 до 10 одиниць часу
```

```
        task.isLong = task.duration > 5;
        task.isCompleted = false;
        task.currentProcessor = -1;
        tasks.push_back(task);
    }
    for (int i = 0; i < numProcessors; i++) {
        processors[i].id = i;
        processors[i].load = 0;
    }
}

void initialAssignment() {
    for (auto &task : tasks) {
        int minLoad = INT_MAX;
        int targetProcessor = 0;
        for (auto &processor : processors) {
            if (processor.load < minLoad) {
                minLoad = processor.load;
                targetProcessor = processor.id;
            }
        }
        processors[targetProcessor].tasks.push_back(task);
        processors[targetProcessor].load += task.duration;
        task.currentProcessor = targetProcessor;
    }
}

void simulateExecution() {
    for (auto &processor : processors) {
        for (auto &task : processor.tasks) {
            if (!task.isCompleted) {
                task.duration--;
                processor.load--;
                if (task.duration <= 0) {
                    task.isCompleted = true;
```

```
            }
        }
    }
}

void optimizeTaskMigration() {
    for (auto &task : tasks) {
        if (!task.isCompleted) {
            int currentProcessor = task.currentProcessor;
            int minLoad = processors[currentProcessor].load;
            int bestProcessor = currentProcessor;
            for (auto &processor : processors) {
                if (processor.id != currentProcessor && processor.load < minLoad) {
                    minLoad = processor.load;
                    bestProcessor = processor.id;
                }
            }
            if (bestProcessor != currentProcessor) {
                processors[currentProcessor].load -= task.duration;
                processors[bestProcessor].load += task.duration;
                task.currentProcessor = bestProcessor;
                cout << "Міграція завдання " << task.id << " на процесор " << bestProcessor << endl;
            }
        }
    }
}

void runSimulation() {
    initializeSystem();
    initialAssignment();
    while (true) {
        simulateExecution();
        optimizeTaskMigration();
        bool allCompleted = true;
```

```
        for (auto &task : tasks) {
            if (!task.isCompleted) allCompleted = false;
        }
        if (allCompleted) break;
    }
    cout << "Всі завдання виконані." << endl;
}

int main() {
    runSimulation();
    return 0;
}
```

Моделююча програма

```
% Кількість процесорів і завдань
numProcessors = 4;
numTasks = 10;

% Характеристики процесорів (наприклад, швидкість виконання)
processors = rand(1, numProcessors) * 2 + 1;  % Від 1 до 3 умовних одиниць

% Характеристики завдань (час виконання, пріоритети)
tasks = struct();
for i = 1:numTasks
    tasks(i).executionTime = randi([10, 100]);
    tasks(i).priority = randi([1, 5]);
    tasks(i).assignedProcessor = mod(i, numProcessors) + 1;
end

% Цільова функція оптимізації (наприклад, мінімізація часу виконання)
objectiveFunction = @(x) max(arrayfun(@(p) sum([tasks([tasks.assignedProcessor] == p).executionTime]), 1:numProcessors));
```

```
% Алгоритм розподілу та міграції
for iter = 1:100
    % Оновлення стану системи
    for p = 1:numProcessors
        % Список завдань на поточному процесорі
        taskList = find([tasks.assignedProcessor] == p);

        % Міграція завдань
        if length(taskList) > 2  % Умова міграції
            tasks(taskList(end)).assignedProcessor = mod(p, numProcessors) + 1;
        end
    end

    % Обчислення цільової функції
    currentObjective = objectiveFunction(tasks);

    % Візуалізація розподілу завдань
    clf;
    hold on;
    for p = 1:numProcessors
        bar(p, sum([tasks([tasks.assignedProcessor] == p).executionTime]));
    end
    title(['Ітерація ', num2str(iter)]);
    xlabel('Процесори');
    ylabel('Завантаження');
    pause(0.5);
end
```

Завідувачу кафедри КІІС,
доктору філософії, доц. Ользі ПАВЛОВІЙ

Дмитро МАРТИНЮК

ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-23-2

ЗАЯВА

З правилами чинного Положення «Про систему забезпечення академічної доброчесності у Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений(а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

25 квітня 2025 року

Протокол аналізу звіту подібності експертом

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Дмитро МАРТИНЮК

Співавтор:

Назва: МАРТИНЮК_Метод оптимізації завдань у багатопроцесорних системах

Експерт:

Підрозділ: Кафедра комп'ютерної інженерії та інформаційних систем

Коефіцієнт подібності 1:5.8%

Коефіцієнт подібності 2:2.2%

Мікропробіли: 168

Заміна букв: 1

Інтервали: 0

Білі знаки: 1

Дата створення звіту: 2025-04-15 12:55:23.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2025-04-15 Доцент Андрій Нічепорук

Дата експерт

РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: Метод оптимізації завдань у багатопроцесорних системах

Автор: Дмитро МАРТИНЮК

Спеціальність: 123 – Компютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Світлана САЧЕНКО, к.е.н, доцент

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з 10-40 джерелами на один фрагмент речення;
2) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості, складає менше 6% і адресується до джерел з інтернету та бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру завдання і свідчить на користь кваліфікаційної роботи.

Керівник роботи _____ Світлана САЧЕНКО

Гарант ОП _____ Олег САВЕНКО

Завідувач кафедри КІІС _____ Ольга ПАВЛОВА

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник: Дмитро МАРТИНЮК

Тема: Метод оптимізації завдань у багатопроцесорних системах

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень -; кількість сторінок записки 108

1. Короткий зміст роботи та прийнятих рішень У роботі розроблено метод та засоби оптимізації завдань у багатопроцесорних системах

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота відповідає виданому завданню

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі подано об'єкт та предмет дослідження, мету, наукову новизну та практичну цінність роботи, а також характеристику структури роботи.

У першому розділі проведено аналіз відомих рішень щодо оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах.

У другому розділі здійснено розроблення цільовихї функцій оптимізації завдань у багатопроцесорних вбудованих системах.

У третьому розділі розроблено розроблення алгоритми оптимізації завдань за різними параметрами у багатопроцесорних вбудованих системах.

У четвертому розділі здійснено розроблення методу оптимізації виконання завдань з використанням реплікації та цільової функції оптимізації у багатопроцесорних вбудованих системах, розроблено програму, проведено експеримент та моделювання схеми оптимізації.У висновках підведено підсумки досягнення результатів з розв'язання завдань дослідження.

4. Позитивні сторони роботи: ___

5. Негативні сторони роботи: немає.

6. Оцінка графічного оформлення та пояснювальної записки роботи: –

7. Відгук про роботу в цілому: __Робота виконана на належному рівні.

8. Інші зауваження: ___–

9. Оцінка дипломної роботи:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи вважаю, що робота заслуговує оцінки «відмінно» 5,00 (А)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) __Корецька Людмила Олександрівна, к.т.н., доцент кафедри АКІТР ХНУ

"_2_" __травня__2025р. _____