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 динамічного керування багатоядерними комп'ютерними системами»

КвРКІ. 190168.22.02.45 ПЗ

Виконала: студент 2 курсу, група КІ2м-22-2 ____ Антоній СВИСТУН
Підпис Ім'я, прізвище

Керівник канд. екон. наук, доцент ____ Світлана САЧЕНКО
Науковий ступінь, вчене звання Підпис Ім'я, прізвище

До захисту допускаю:
Зав. кафедри КІІС, д.т.н., проф.
Тетяна ГОВОРУЩЕНКО
20 05 2024 р.

Хмельницький, 2024

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 КІІС
Тетяна ГОВОРУЩЕНКО
____
" 01 " 09 2023 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Свистун Антоній Анатолійович
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод динамічного керування багатоядерними комп'ютерними системами

Керівник проекту (роботи) к.е.н., доцент Саченко С.І.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.01.2024р. № 1

2. Строк подання студентом проекту (роботи) на кафедру 03.05.2024 р.

3. Вихідні дані до проекту (роботи) Завдання на кваліфікаційну роботу

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

аналіз відомих методів динамічного керування багатоядерними комп'ютерними системами;

архітектура багатоядерни[комп'ютерни[систем;

метод динамічного керування багатоядерними комп'ютерними системами.

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

6. Консультанти розділів дипломного проекту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Лисенко С.М., професор кафедри КІІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання «_06_»________09__________2023 р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики ДРМ з керівником	01.09.2022	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	05.12.2023	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	25.02.2024	виконано
4	Робота над розділом 2 – розробка архітектури для вирішення поставленої задачі	01.04.2024	виконано
5	Робота над тезами	05.03.2024	виконано
6	Робота над розділом 3 – розробка методу для вирішення поставленої задачі	15.04.2024	виконано
7	Робота над розділом 4 – проектування та розробка засобів для вирішення поставленої задачі, експериментальна частина	25.04.2024	виконано
8	Оформлення пояснювальної записки згідно вимог	30.04.2024	виконано
9	Попередній захист ВКР	01.05.2024	виконано
10	Захист ВКР на засіданні ЕК	До 30.05.2024	

Студент ____________ _Антоній СВИСТУН__
Підпис Ініціали, прізвище

Керівник роботи ____________ _Світлана САЧЕНКО__
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема кваліфікаційної роботи: «Метод динамічного керування багатоядерними комп'ютерними системами»

Автор роботи: Свистун Антоній Анатолійович

Керівник роботи: Саченко С. І.

Пояснювальна записка: 71 с., 2 таблиці, 81 джерело.

ПЕРЕЛІК КЛЮЧОВИХ СЛІВ: операційні системи; ядро; процесор; багатопроцесорні системи.

Об’єктом дослідження є процес забезпечення динамічного керування багатоядерними комп'ютерними системами.

Предметом дослідження є методи динамічного керування багатоядерними комп'ютерними системами.

Метою кваліфікаційої роботи є розробка методу динамічного керування багатоядерними комп'ютерними системами.

Для розв’язання поставлених задач використовувалися методи теорії комп’ютерних мереж, архітектури комп’ютерів.

Наукова новизна отриманих результатів:

- удосконалено метод динамічного керування багатоядерними комп'ютерними системами.

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 динамічного керування багатоядерними комп'ютерними системами.

У другому розділі здійснено дослідження предметної області та визначено стратегію забезпечення динамічного керування багатоядерними комп'ютерними системами.

У третьому розділі розроблено спосіб обробки завдань та використання

декскриптору. Його реалізація базується на використанні апаратного пристрою. Також, розроблено метод динамічного керування багатоядерними комп'ютерними системами.

У четвертому розділі здійснено дослідження ефективності запропонованого рішення з апаратним пристроєм.

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

Практична значимість отриманих результатів полягає у забезпеченні динамічного керування багатоядерними комп'ютерними системами.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 3

ВСТУП 4

1 АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ ДИНАМІЧНОГО КЕРУВАННЯ БАГАТОЯДЕРНИМИ КОМП'ЮТЕРНИМИ СИСТЕМАМИ 6

1.1 Аналіз предметної області 6

1.2 Спеціальні апаратні засоби багатоядерних систем 14

1.3 Висновки до першого розділу 19

1.4 Постановка задачі дослідження 19

2 МОДЕЛЬ ЗАБЕЗПЕЧЕННЯ АПАРАТНОЇ ПІДТРИМКИ КЕРУВАННЯ БАГАТОПРОЦЕСОРНИМИ СИСТЕМАМИ 19

2.1 Дослідження предметної області 21

2.2 Організація зв’язку в багатопроцесорній системі 32

2.3 Висновки до другого розділу 40

3 **МЕТОД ДИНАМІЧНОГО КЕРУВАННЯ БАГАТОЯДЕРНИМИ КОМП'ЮТЕРНИМИ СИСТЕМАМИ** 41

3.1 Спосіб обробки завдань та використання декскриптору 41

3.2 Метод динамічного керування багатопроцесорними системами 50

3.3 Висновки до третього розділу 59

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТ АПАРАТНОГО РІШЕННЯ 60

4.1 Реалізація апаратного пристрою 60

4.2 Підвищення продуктивності за рахунок додавання ядер до виконання завдань 66

4.3 Висновки до четвертого розділу 72

ВИСНОВКИ 74

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 75

ДОДАТОК А Презентація до захисту 85

ДОДАТОК Б Наукова праця здобувача 95

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

АП – апаратний підхід з використанням апаратного пристрою

ПП – програмний підхід з використанням програмної моделі

ОС - операційна система

ПЗ - програмне забезпечення

ВСТУП

Тенденції в архітектурі комп'ютерів зосереджені на багатоядерних плантформах. Метою цих нових платформ є масштабування продуктивності системи за рахунок кількості ядер. Однак продуктивність поточних архітектур обмежена через додаткові витрати на паралелізм на рівні потоків і програмованість. ПП — це модель програмування, заснована на завданнях, яка полегшує програмованість багатоядерних процесорів і намагається розширити функціональний паралелізм у застосунках. Однак продуктивність ПП не масштабується ефективно для дрібнозернистих завдань, оскільки для таких завдань додаткові витрати на керування завданнями стають значними в порівнянні з виконанням завдань. АП — це динамічна система апаратної підтримки, яка має на меті зменшити поточні додаткові витрати ПП, розвантажуючи процес вирішення залежностей і синхронізацію з ядрами на апаратному забезпеченні. Тому, реалізуємо АП, визначаючи та підключаючи нове обладнання в симуляторі архітектури Cell. Масштабованість, продуктивність і пропускна здатність реалізації оцінюються для різних розмірів завдань і кількості ядер, використовуючи кілька шаблонів залежностей. Крім того, оцінимо різні параметри конфігурації, такі як розміри нового обладнання, вставленого в існуючу архітектуру.

Актуальність роботи полягає в необхідності удосконалити метод динамічного керування багатоядерними комп'ютерними системами.

Метою кваліфікаційої роботи є розробка методу динамічного керування багатоядерними комп'ютерними системами.

Поставлена мета досягається розв’язанням таких основних завдань:

- проаналізувати відомі методи динамічного керування багатоядерними комп'ютерними системами;

- удосконалити метод динамічного керування багатоядерними комп'ютерними системами;

- реалізувати розроблений метод динамічного керування багатоядерними комп'ютерними системами;

- здійснити еспериментальні дослідження згідно розроблених рішень.

Об’єктом дослідження є процес забезпечення динамічного керування багатоядерними комп'ютерними системами.

Предметом дослідження є методи динамічного керування багатоядерними комп'ютерними системами.

Для розв’язання поставлених задач використовувалися методи теорії комп’ютерних мереж, архітектури комп’ютерів.

Наукова новизна отриманих результатів:

- удосконалено метод динамічного керування багатоядерними комп'ютерними системами.

Практична значимість отриманих результатів полягає у забезпеченні динамічного керування багатоядерними комп'ютерними системами.

За темою кваліфікаційної роботи опубліковано одну публікацію [81] у Збірнику наукових праць за матеріалами IX Науково-практичної конференції «Інтелектуальні комп’ютерні системи та мережі» (Тернопіль, 2024).

1 АНАЛІЗ МЕТОДІВ І ТЕХНОЛОГІЙ ДИНАМІЧНОГО КЕРУВАННЯ БАГАТОЯДЕРНИМИ КОМП'ЮТЕРНИМИ СИСТЕМАМИ

1.1 Аналіз предметної області

Архітектура комп'ютерів стала переважно багатоядерною [1-4], оскільки попередні методи підвищення продуктивності процесорів показали зниження продуктивності. Ці попередні методи підвищували продуктивність за рахунок використання паралелізму на рівні команд і використання зростаючих частот процесора [4, 5]. Експлуатація рівнів команд, однак, принесла з собою великий обсяг енергоспоживання, і показала слабкі результати [5, 6]. У той же час обмеження потужності обмежили збільшення частот процесора [7]. В результаті індустрія розвернулася в бік багатоядерних платформ [8].

Серед нових багатоядерних платформ найсучаснішим процесором є Cell Broadband Engine (CBE) [9-11]. Це гетерогенне багатоядерне ядро. Архітектура цієї платформи поєднує в собі процесний елемент і вісім процесорних елементів, з'єднаних через шину з'єднання елементів [12-14]. Підвищенню продуктивності, яке може бути досягнуто за допомогою цього типу багатоядерних платформ, загрожує кілька факторів, серед яких додаткові витрати на зв'язок, паралелізм на рівні потоків [13]. Щоб забезпечити масштабованість продуктивності, ці проблемні завдання необхідно вирішити.

Для поліпшення програмованості багатоядерних останнім часом було запропоновано кілька моделей програмування [14]. Значна частина з них заснована на завданнях, тобто вони намагаються використовувати функціональний паралелізм у застосунках. Однією з таких моделей програмування на основі завдань, яка може бути використана є ПП [15]. Модель програмування ПП дозволяє програмісту вказувати функції, які можуть працювати паралельно, шляхом анотування серійного коду. Система середовища виконання ПП динамічно визначає залежності між завданнями,

виходячи з їх вхідних і вихідних операндів. Завдання, залежності яких розв'язані, плануються на доступні ядра для виконання. Простота програмування, яку пропонує ПП, досягається за рахунок додаткових витрат, запроваджених системою часу виконання. Хоча ПП досить добре масштабується для великих завдань і невеликої кількості ядер, його додаткові витрати обмежують масштабованість для дрібнозернистих завдань і великої кількості ядер [16].

Для подолання обмежень ПП використовуємо АП, удосконалення моделей задачного програмування, архітектура якого був запропонований в [17-22]. АП — це апаратна система підтримки, яка спрямована на зменшення додаткових витрат, що виникають у програмних моделях. Він заснований на прискоренні процесу вирішення залежностей завдань і забезпеченні масштабованої синхронізації між ядрами. АП реалізовуватимемо в симуляторі СВЕ. Оцінюватимемо його продуктивність, масштабованість і пропускну здатність за допомогою декількох тест продуктивностіів і різних розмірів завдань. Крім того, оскільки різні параметри конфігурації в АП можуть впливати на його продуктивність і масштабованість, для вивчення їх впливу оцінюється кілька випадків.

Було виконано кілька робіт в області апаратної підтримки керування завданнями [19-23]. Однак більшість цих робіт не пропонують розв'язання залежностей між завданнями, оскільки вони обмежені плануванням доступних завдань для виконання. Тому, ці підходи покладаються на програміста, який належним чином виконує завдання. В роботі [13] пропонується апаратна черга завдань, яка досягає низьких затримок при отриманні завдань. Більш гнучке планування завдань досягається в [22] за рахунок використання міжпотокової синхронізації. В [11] запропонована багатоядерна платформа, що містить блок, який забезпечує централізоване планування завдань. У [18] запропонований апаратний інтерфейс для керування завданнями з метою полегшення роботи ОС. В [14] архітектурна підтримка була зосереджена для моделі програмування. Для розв'язання

залежностей серед завдань з'явилося небагато робіт. В [23] було запропоновано блок керування завданнями з випередженням, який зменшує затримку при отриманні завдань. Така одиниця визначає лише завдання, які готові до виконання, оскільки залежності між завданнями мають бути визначені користувачем, а не генеруватися автоматично. Для прогнозування готових завдань залежності завдань використовуються у зворотному порядку. Однак затримка цієї системи досить велика через те, що ця система програмована. Цей недолік був усунутий в [5], хоча він втратив загальність, оскільки орієнтований на декодування. Для прискорення керування програмними завданнями був розроблений процесор набору інструкцій [7].

Апаратне прискорення для моделей задачного програмування було запропоновано в [24-26], що є найбільш схожою роботою з АП. Тут спостерігається схожість залежностей введення і виведення завдань і інструкцій. Таким чином, пропозиція авторів в роботі [12] є апаратним забезпеченням для розв'язання залежностей, принцип роботи якого схожий на позачерговий планувальник інструкцій [12]. Він заснований на гетерогенній архітектурі з двома типами ядер, кожне з яких оптимізоване для певної ролі, і спільною і когерентною схемою пам'яті. Він був розроблений, щоб запропонувати продуктивність у широкому спектрі застосунків, з особливим акцентом на інтенсивні обчислювальні середовища, такі як мультимедіа та ігри.

Двопотоковий процесор загального призначення з набором інструкцій і з розширеннями векторних носіїв для підтримки організації функціонування, який є ефективним у прискоренні мультимедійних застосунків. Цей процесор має ієрархію кешу двох рівнів. На першому рівні є два різні кеші. Кеш першого рівня складається з двох елементів, один для даних, а інший для інструкцій. І кеш другого рівня, що відповідає за роботу операційної системи [27]. Вони працюють автономно, але не працюють під керуванням операційної системи і залежать від роботи, яку вони повинні виконувати.

Елементи для обробки є незалежними процесорами, оптимізованими для

виконання великих робочих навантажень. Вони реалізують архітектуру набору інструкцій. Їх внутрішня конструкція містить локальне сховище і контролер потоку пам'яті з внутрішнім контролером прямого доступу до пам'яті. Замість того, щоб використовувати архітектуру на основі кешу, він працює в локальному сховищі, яке служить основним сховищем. Саме звідси завантажуються інструкції, завантажуються та зберігаються дані. Він має розмір і є найбільшим елементом. Для доступу до даних він запрограмований на виконання асинхронного прямого доступу до пам'яті. За допомогою прямого доступу до пам'яті можна отримати доступ до основної пам'яті, а також локальних сховищ інших пристроїв. Для програмування цієї одиниці прямого доступу до пам'яті або для доступу до будь-якої іншої одиниці він має в своєму розпорядженні інтерфейс каналу, який служить для зв'язку з іншими суб'єктами. Модуль може видавати послідовність передач прямого доступу до пам'яті, записуючи команду на інтерфейсі каналу. Контролер потоку пам'яті може обробляти одночасно багато передач, кожна з яких має певний розмір даних. Стан можна перевірити, зчитавши його регістри. Цю перевірку можна виконати з будь-якого спеціалізованого пристрою. Доступ до цих регістрів здійснюється з власного пристрою через інтерфейс каналу та з інших модулів, оскільки вони включені в загальний адресний простір [28-32].

Шина з'єднання елементів для зв'язку є спільною для всіх контролерів та елементів обробки, які присутні в архітектурі. Ця з'єднувальна шина підтримує багатопроцесорні операції, що дозволяє будувати кластери процесорів. Він утворений чотирма кільцями, кожне з яких має дані і одночасно передає їх, що дорівнює розміру рядка кешу. Кожне з цих кілець може обробляти передачу даних одночасно, маючи можливість видавати передачі для прямого доступу до пам'яті між основною пам'яттю з максимальною внутрішньою пропускною здатністю за тактовий цикл процесора. Заради швидкості програмісти повинні пам'ятати про те, що затримка залежить від ідентифікатора, пов'язаного з кожним з елементів, підключених до пристроїв, оскільки затримка збільшується

зі збільшенням кількості переходів, що розділяють два краї зв'язку. Для того, щоб мати можливість адресувати кожен модуль в архітектурі, кожен блок, підключений до нього, зіставляється з певною фізичною адресою. Таким чином, різні пристрої або основна пам'ять відображаються в різних фізичних діапазонах. Відповідно, він спрямовує кожну частину даних залежно від цільової фізичної адреси. Найбільш інноваційною та новаторською особливістю архітектури є відсутність ієрархії кешу, а також наявність локального сховища та контролера прямого доступу до пам’яті. Ідея прямого доступу до пам’яті, яка використовується для доступу до основної пам'яті, використовується для подолання проблем з використанням пам'яті. В архітектурі, заснованій на кеші, дані збираються щоразу, коли це необхідно, створюючи великі затримки через велику відстань від ядер до основної пам'яті. У підході, заснованому на технології прямого доступу до пам’яті, потрібно заздалегідь продумати, які дані потрібні. Але в той час, коли відбувається передача даних, деяка обробка може виконуватися одночасно, і таким чином можемо приховати затримки пам'яті. Концепція локальної пам'яті, безпосередньо доступної з процесора і доступної з будь-якого іншого модуля, а також асинхронної передачі прямого доступу до пам’яті підвищує передбачуваність і масштабованість глобального архітектури. У схемі, заснованій на кеші, можливо, що проблеми кешу впливають на продуктивність процесора, роблячи систему непередбачуваною. Ще однією важливою характеристикою платформи є те, що кожне локальне сховище відображається на карті пам'яті всього процесора, тому будь-який модуль обробки доступний з будь-якого іншого модуля обробки, присутнього в архітектурі. Цей факт дозволяє здійснювати передачу даних з локального магазину в локальне сховище за допомогою операцій прямого доступу до пам’яті. Однак цей тип зв'язку між двома пов’язаними процесорами не є узгодженим, оскільки інформація, що міститься в місцевому магазині, могла змінитися без будь-якого попередження в бік цільового локального магазину [33]. Оскільки сучасні тенденції в мікропроцесорах розвиваються у напрямку розміщення багатьох ядер в одному мікропроцесорі, передбачаючи платформи,

які містять від сотень до тисяч ядер, то програмні моделі потрібно розробляти [34]. Ці моделі розвиваються з метою допомоги програмістам у створенні ефективних реалізацій, здатних приносити прибуток від існуючого паралелізму на застосунках. Прикладом таких інструментів є модель програмування ПП [35]. ПП — це модель програмування, призначена для використання функціонального паралелізму. Програмісту не обов'язково знати, на якій платформі запущено програму. Достатньо анотувати код шляхом додавання процесорів. Є також деякі елементи, які використовуються для проблем синхронізації. В задачі позначають функцію, яка може виконуватися як завдання паралельно в залежності від вхідних і вихідних операндів. Це буде динамічно перевірено системою виконання. Потім він будує граф залежностей завдань під час виконання, де завдання, які не містять залежностей між собою, можуть виконуватися паралельно. Анотації початку та закінчення завдання вказують на початок і кінець цієї системи виконання, відповідно [36].

Реалізація ПП виконується на одному ядрі з двома потоками і декількома робочими ядрами, які виконують завдання програми. Перший потік виконує основний програмний код і додає завдання в систему виконання. Другий потік розкладає завдання на ядра воркерів і обробляє зв'язок з ними. Таким чином, існує проста відповідність між головним ядром і робочими ядрами. Два потоки керування будуть виконуватися на ядрі, в якому розподілятимуть завдання між робочими ядрами. Отже, він подаватиме сигнал щоразу, коли з'являться готові завдання, які потрібно виконати. Як тільки вони дізнаються про нове завдання, робочі ядра виконають запит прямого доступу до пам’яті, щоб отримати дані у своє локальне сховище, що обробляється системою виконання, і виконають завдання, після чого вони подадуть сигнал, вказуючи, що вони виконали завдання. Система виконання має деякі параметри конфігурації для оптимізації продуктивності системи, наприклад, мінімальну кількість завдань, які потрібно запланувати процесору, або мінімальну кількість завдань, оброблених перед початком призначення завдань. Як наслідок, вона пропонує можливість збирати завдання на графіках у пакети з метою розкриття локалізації даних. Таким

чином, можна зменшити середні додаткові витрати на одне завдання та покращити масштабованість. Недоліками цієї моделі програмування є великі додаткові витрати, які використовуються для обробки готових завдань, особливо коли вони характеризуються малими розмірами [37].

Основні результати, представлені в [38-40] повторюються, оскільки вони показують оцінку ефективності, конкретно для його реалізації. У [40] було протестовано три штучні тести для різних розмірів завдань. Кожен з тестів був виконаний з різною схемою залежностей над матрицею, розділених на квадратні блоки. Кожен блок представляє собою одну задачу, яка повинна бути виконана на робочих ядрах, маючи загальну однакову кількість завдань.

Спочатку вводиться тест продуктивності, який втілює найскладнішу схему залежностей у досліджуваних системах, де кожен блок залежить від сусіднього блоку. Теоретична максимально досяжна масштабованість для цього тесту продуктивності при використанні ядер обумовлена існуючими залежностями серед завдань. Це особливо важливо на початку і в кінці тесту продуктивності, де мало паралелізму. У цих умовах було використано [41] кілька конфігурацій з метою дослідження масштабованості та додаткових витрат моделі програмування. Для конфігурації за замовчуванням була отримана погана масштабованість навіть для дуже великих розмірів завдань, оскільки система виконання намагається згрупувати кілька завдань для відправки в одне і те ж робоче ядро. Це корисно лише тоді, коли зустрічається достатній паралелізм, але не в кожному випадку. Тому, було встановлено нову конфігурацію, яка розподіляла завдання після того, як вони були готові, досягаючи в цьому випадку максимального значення масштабованості. Ця нова конфігурація передбачає більші додаткові витрати для системи часу виконання, що призводить до гірших відгуків при зменшенні розміру завдання. Незважаючи на те, що була досягнута нова оптимальна конфігурація, результати показали [42], що для невеликих завдань основним і допоміжним потоками створюються перешкоди для їх виконання. Для таких невеликих завдань потоки керування не можуть постійно забезпечувати всіма завданнями для виконання. Взявши,

наприклад, тривалість завдання, що приблизно відповідає довжині завдання при декодуванні, була отримана максимальна масштабованість. Для тесту продуктивності видаляється одна із залежностей з попереднього випадку, і залишається тільки залежність від сусідів. Як наслідок, маємо лише одну залежність, і матриця повністю незалежна від рядків, оскільки зв'язків між рядками більше не існує. Очікування від цього, порівняно з попереднім, полягали в тому, щоб отримати значне покращення ресультів, оскільки створення архітектури тепер є більш ефективним завдяки спрощеній схемі залежностей. Вона демонструє лише незначні поліпшення в порівнянні з тест продуктивностіом, так як масштабованість для завдань становить середнє значення. Проаналізувавши слід [43], встановлено, що основним вузьким місцем системи була основна нитка, що відповідає за додавання завдань. З цього факту випливає, що підтримка і побудова графа залежностей задачі не сильно залежать від шаблону самої залежності. З іншого боку, схоже, що допоміжна нитка значно полегшується можливістю збору блоків, оскільки додаткові витрати, що вводяться зараз, значно нижчі. Цей експеримент вказує, що допоміжний потік є достатньо масштабованим, але це не відповідає дійсності через кругову природу механізму синхронізації [44]. Ця політика відіграє важливу роль, коли він може завершити виконання, оскільки помічник не зверне увагу на свій сигнал, поки не закінчать усі його попередники, що означає найгірший сценарій, коли необхідний час не є мінімальним. Тест продуктивності характеризується відсутністю будь-якої залежності між будь-якими двома завданнями, тому обидва вони незалежні і можуть виконуватися паралельно. Масштабованість досягає значення, що набагато вище, ніж у попередніх випадках. Причина цієї зміни полягає в тому, що, оскільки ефект нарощування не виявлено, і як тільки завдання додаються, то вони готові до виконання. Існування вищого паралелізму виражається в кращій реакції системи при меншій тривалості завдань. При розгляді трасування було помічено, що час, необхідний для побудови та підтримки завдання, дуже схожий на той, що зустрічається в тесті, який дозволяє зробити висновок, що шаблон залежності не

впливає на витрачений час, а є перевіркою залежностей основною причиною затримки.

Таким чином, результати [23-44] показують, що масштабованість досягається для великих розмірів завдань, тоді як для малих значень продуктивність зменшується через додаткові витрати, що вносяться моделлю програмування. Для коротких завдань результати тестів показали три вузькі місця в поведінці системи виконання. Процедура розв'язання проблеми є основним вузьким місцем, яке не може забезпечити виконання завдань досить швидко і, таким чином, призводить до зупинки. Планування та додавання завдань, що виконуються за допомогою середовища виконання, також є вузьким місцем, оскільки вони не здатні відповідати вимогам швидкості. Синхронізація з допоміжним потоком, який перевіряє робочі ядра за допомогою кругової системи, теж є проблемною. Для того, щоб впоратися з наслідками цих вузьких місць, потрібно розробити новий метод динамічної підтримки керування. Перспектвним напрямом для вдосконалення відомих рішень є динамічна підтримка керування базована на перевірці залежностей між завданнями та обробкою синхронізації з робочими ядрами за допомогою регістрів, відображених у пам'яті. Таке централізоване рішення може досягти кращої масштабованості для завдань меншого розміру.

1.2 Спеціальні апаратні засоби багатоядерних систем

Розглядувана система в роботах [23-44] має належну масштабованість для великих завдань, але її продуктивність знижується для дрібнозернистих завдань. Система АП є апаратною підтримкою [45], яка призначена для усунення вузьких місць, які обмежують продуктивністьпершої системи з [23-44]. Ця система намагається підвищити обмежену масштабованість ПП для дрібнозернистих завдань, розміщуючи новий апаратний блок в архітектурі процесора. Першим фактором, що обмежує масштабованість першої системи, була процедура дозволу залежностей, розв'язання залежностей якої здійснюється відповідно до

адрес вхідних і вихідних блоків кожного завдання. Незалежно від моделі залежностей, система не змогла зберегти ефективні характеристики до виконання нових завдань. Більш високе прискорення було б дуже вдячне, оскільки воно дало б можливість ефективно використовувати більше робочих ядер. Пропозиція АП полягає в тому, щоб здійснювати цей процес повністю апаратно. Друге виявлене [45] вузьке місце – планування, підготовка та подача завдань. Це виконується в ПП допоміжним потоком, який намагається зменшити додаткові витрати та експлуатувати локальність, доставляючи групи завдань, а не окремі завдання. Однак ця схема недостатня для невеликих розмірів завдань через великі додаткові витрати, що вносяться системою. Для цього вузького місця потрібна швидкість для ядер [46]. Таким чином, АП пропонує дуже просту реалізацію планування завдань. Також, в частині синхронізації, ПП використовує механізм опитування за круговою системою для синхронізації з робочими ядрами, який погано масштабується. Система виконання ПП використовує поштові скриньки для надсилання завдань та отримання з них готових завдань. Завдяки цьому круговому опитуванню можуть бути завершені потоки, поки йде перевірка інших ядер. Тому, АП пропонує централізовану апаратну чергу, де робочі ядра можуть отримувати завдання та сигналізувати про них після їх завершення. Ця процедура виконується кожним процесором окремо, і потрібно, щоб такий процес синхронізації не займав більше часу для масштабування. Щоб задовольнити ці вимоги, початкова архітектура АП містить два різні блоки: блок завдань; контролер завдань. Контролер завдань буде розміщений в середині, відповідаючи за отримання завдань і сигналізуючи про них назад. Він використовуватиме подвійну буферизацію, щоб приховати затримку прямого доступу до пам’яті, поки ядро виконує раніше отримане завдання. Оскільки передбачаються великі додаткові зусилля з реалізації, оскільки потрібно опрацьовувати розподіл пам'яті всередині робочих ядер, поточна реалізація АП складається включає динамічне керування завданнями [47]. Архітектура процесора з множиною завдань містить засіб АП. Новий блок під’єднується до загального вузла, де до нього можна отримати доступ з будь-

якого іншого блоку системи. Він призначений для роботи як централізована одиниця, яка отримує завдання, вирішує їх в залежності від типів та розподіляє їх по робочих ядрах. Основною проблемою архітектури АП є досягнення високої пропускної здатності, здатної задовольнити вимоги все більшої кількості потоків. З цієї причини вона має високий рівень конвеєра, де один етап виконується одночасно з іншими етапами. Він додає нові завдання, надсилаючи метадані, що описують завдання, які мають форму дескрипторів завдань [48].

Дескриптори завдань - це структури, які містять необхідну інформацію для виконання завдань, а також кількість залежностей і параметри, необхідні для обробки кожної з них. Додатковий пристрій в АП надсилає покажчик дескриптора задачі в основну пам'ять і розмір такого дескриптора завдання. Потім завантажує дескриптор з основної пам'яті в сховище завдань, тобто його пам'ять, відображену в пам'яті. Ця пам'ять діє як локальне сховище множини завдань і зберігає дескриптори кожного завдання до його завершення, запобігання надходженню в основну пам'ять щоразу, коли потрібно отримати доступ до дескриптора завдання, що призвело б до більших затримок. Після отримання дескриптора множина завдань визначає залежності нового завдання з раніше доданими завданнями. Розв'язання залежностей [49] виконується трьома пошуками таблиць, заповнюючи їх необхідною інформацією, що можна зробити дуже швидко. У разі, якщо задача не містить залежностей, то її ідентифікаційний номер записується в готову чергу разом з покажчиком дескриптора завдання на сховище завдань, де його можна прочитати. Готова черга зіставлена з пам'яттю, а отже, до неї можна швидко отримати доступ з будь-якої точки платформи. Коли пристрій вільний для виконання завдання, то він намагається прочитати його з черги готовності. У разі наявності доступного завдання він зчитує дескриптор. В іншому випадку він зупиняється в очікуванні додавання завдання в чергу готовності. Після того, як він отримує дескриптор [49-56], то завантажує дані, необхідні для виконання завдання, з основної пам'яті і виконує завдання відповідно до функції, включеної в дескриптор завдання. Наступним кроком є сигнал про виконання завдання. Це робиться шляхом

запису іншого відображеного в пам'яті регістра — регістра фінішного буфера. Оскільки цей крок включає в себе ще один регістр з відображенням пам'яті, то він завершується за кілька циклів, користуючись перевагами низької затримки. Останній етап полягає в тому, щоб він отримав готове завдання та оновив таблиці знову виконавши пошуки. Крім того, він перевіряє наявність завдань, які не мають залежностей, додаючи їх до черги готовності [56].

Цей підхід усуває вузькі місця, виявлені в моделі програмування ПП [56]. Такий централізований підхід може стати вузьким місцем, оскільки кількість ядер в одному мікропроцесорі збільшується [57]. Це можна вирішити шляхом побудови кластерів з низькою затримкою зв'язку. Навантаження на систему можна збалансувати, дозволивши брати завдання. Блок складається з відображених регістрів пам'яті, сховища завдань, таблиць, що містять інформацію про залежності, і функціональних модулів, які обробляють цю інформацію. Процесор конвеєрний і, таким чином, завантажувач дескрипторів, черга готовності та обробник фінішу працюють паралельно [58]. Обробник дескриптора та кінцевий обробник виконують пошук у таблиці завдань, таблиці виробників і таблиці споживачів, змінюючи свої записи відповідно до поточного стану графа залежностей. Вхідний буфер, черга готовності, буфер завершення та регістр стану можуть бути доступні з будь-якої точки системи за кілька циклів завдяки низькій затримці. Сховище завдань містить дескриптори завдань, що обробляються, а його записи також відображаються в пам'яті [59]. Життєвий цикл задачі передбачає заповнення дескриптору завдання інформацією про його вхідні та вихідні операнди. Ці операнди використовуються пізніше для визначення залежностей задачі з раніше доданими завданнями [60-68]. Процесор надсилає покажчик пам'яті туди, де розміщено дескриптор завдання, і його розмір у буфер. У буфері є кілька записів, які дозволяють додавати завдання, коли завантажувач дескрипторів все ще зайнятий попереднім запитом або зупинився [69-77]. Після того, як покажчик дескриптора нового завдання отримає сигнал, завантажувач дескрипторів будує запит пам'яті на покажчик дескриптора в основній пам'яті. Коли надходить новий дескриптор завдання, він

зберігається у сховищі завдань [78]. Після зберігання дескриптора обробник дескриптора присвоює ідентифікатор завданню та визначає залежності нового завдання двома підстановками в таблиці виробників і таблиці споживачів. Після визначення залежностей обробник дескриптора заповнює інформацію про завдання в таблиці завдань, яка містить ідентифікатор завдання, покажчик дескриптора в сховищі завдань і кількість залежностей. У разі, якщо залежності не існує, в готову чергу додається ідентифікаційний номер завдання і його покажчик дескриптора [79]. Процесор зчитує готові завдання безпосередньо з черги готовності. У разі, якщо завдання не чекає на опрацювання, то вони зупиняються до надходження нових готових завдань. Після отримання нового завдання запитують дескриптор завдань зі сховища завдань, який надішле дескриптор завдання безпосередньо їм. Після цього ядра запросить дані, необхідні для виконання завдання з основної пам'яті, і почнуть виконувати завдання відповідно до інформації про дескриптор завдання. Як тільки це буде зроблено, вони сигналізують про завершення завдання, записуючи ідентифікатор завдання в буфер завершення. Фінішний буфер також має кілька записів для зберігання ідентифікаторів готових завдань, надісланих з кількох ядер [80]. Коли обробник виявляє ідентифікатор завдання в фінішному буфері, то він починає його обробку. Цей обробник, як і обробник дескриптора, перевіряє таблиці споживачів і виробників. Далі обробник оновлює таблиці, видаляючи залежності з готовим завданням, оскільки вони більше не існують. Також, оновлюються записи таблиці завдань, що містять завдання з видаленими залежностями. У разі, якщо будь-яка з них більше не залежить від будь-якого іншого завдання, обробник додає свій ідентифікатор завдання та покажчик дескриптора до готової черги. Останньою операцією обробника є видалення запису таблиці завдань, що містить інформацію про оброблювану задачу, і таким чином дозволяє додати в систему ще одну задачу. Реєстр статусів зберігає цінну інформацію щодо поточного стану. Ця інформація, що міститься в реєстрі станів, вказує на те, запущений або порожній процесор, і чи дозволено додавати новий запис в буфер. Він може зупинитися. Наприклад, коли таблиця завдань

заповнена, обробник дескриптора перестане вставляти нові записи в таблицю. Ця умова сигналізується завантажувачу дескрипторів, який зупиняє обробку покажчиків дескрипторів задачі з буфера. Ядра не можуть продовжувати додавати завдання через повний буфер. Додатковий пристрій виявляє заповнений буфер, зчитуючи регістр стану, і, отже, він не виконує новий запис у буфер.

Таким чином, проведено аналіз предметної області для дослідження. В результаті встановлено, що програмна модель має три основні недоліки. Тому, за основу для напряму досліджень потрібно вибирати апаратний підхід, що може дозволити усунунти проблеми програмної моделі сукупно за трьома базовими недоліками.

1.3 Висновки до першого розділу

В результаті проведеного дослідження предметної області було встановлено недоліки відомих рішень, зокрема для програмної моделі виділено три ключові проблеми. Для їх усунення з метою розробки рішень, проведено аналіз проблем. Встановлено особливості їх функціонування. Важливою напрямом для вирішення проблемних питань в контексті програмної моделі перспективним напрямом визначено розроблення методу з підтримки апаратної моделі.

1.4. Постановка задачі дослідження

Поставлена мета досягається розв’язанням таких основних завдань:

- проаналізувати відомі методи динамічного керування багатоядерними комп'ютерними системами;
- удосконалити метод динамічного керування багатоядерними комп'ютерними системами;
- реалізувати розроблений метод динамічного керування багатоядерними

комп'ютерними системами;

- здійснити есперіментальні дослідження згідно розроблених рішень.

2 МОДЕЛЬ ЗАБЕЗПЕЧЕННЯ АПАРАТНОЇ ПІДТРИМКИ КЕРУВАННЯ БАГАТОПРОЦЕСОРНИМИ СИСТЕМАМИ

2.1 Дослідження предметної області

Побудова та підтримка графа залежностей, який представляє залежності завдань, може бути дуже повільною в програмному забезпеченні. Натомість, процесор обробляє залежності за допомогою простого пошуку в таблицях виробника та споживача, що може працювати дуже швидко, оскільки їм не потрібно перевіряти весь список. Кожен запис цих списків містить виділену там адресу та список початку, що складається з ідентифікаторів завдань, які очікують на звільнення адреси. Крім того, записи списку споживачів також мають поле, яке містить кількість залежностей для цієї адреси. Поля запису таблиці завдань описують ідентифікаційний номер завдання, покажчик на дескриптор, поточний стан задачі в системі та кількість її поточних залежностей. Якщо ідентифікатор завдання більше не зустрічається у жодному зі списків початку, то його кількість залежностей дорівнює нулеві, отже, його можна додати до черги готовності. Таблиця виробників — це механізм, який використовується для уникнення помилок у режимі читання після запису. Кожен елемент у цій таблиці містить адресу, записану певним раніше доданим завданням. Щоразу, коли обробник дескриптора виявляє завдання залежно від одного з елементів цієї таблиці, то він негайно підписує завдання на свій стартовий список. Це завдання не буде запущено, поки його не буде звільнено зі списку. Відповідно до цього, коли завдання завершено, обробник перевіряє свій стартовий список у списку виробників. Якщо він знаходить будь-який елемент у списку, то обробник вилучає їх усі і зменшує кількість залежностей у таблиці завдань. Цього механізму недостатньо, щоб уникнути будь-якої небезпеки, оскільки він не може впоратися з випадком запису двох завдань на одну адресу. Цей випадок відомий як небезпека запису після запису. Якщо завдання записується на ту саму адресу, записану іншим завданням, додавання його до списку, то це не вирішить проблему. У такому випадку, коли завдання з

продукування адреси завершується, то він запускає кожен елемент у списку, незалежно від того, чим вони є. Якщо два завдання або одне були розпочаті в один той же час, то він дасть пошкоджені дані. Щоб впоратися з цією ситуацією, необхідно запровадити додатковий механізм для розрізнення типу завдання, знайденого у стартовому списку. Це робиться шляхом додавання від'ємного ідентифікатора завдання замість реального ідентифікатора завдання для завдань, які записують дані.

Таким чином, коли обробник знайде від'ємний ідентифікатор, то він припинить вилучення елементів зі списку.

Таблиця споживачів використовується для уникнення небезпеки запису після читання. У цій таблиці зберігаються адреси, які зчитуються з раніше доданих завдань. Коли нова задача обробляється обробником дескриптора, то він перевіряє операнди вихідних операндів задачі. У випадку, якщо будь-який з них має запис у таблиці, обробник підписує свій ідентифікатор завдання на початковий список цього запису. Коли завдання завершується виконанням, обробник виконує пошук у таблиці споживачів для кожного з його вхідних операндів. Якщо поле більше одиниці, то воно зменшується на одиницю, що вказує на те, що одне завдання більше не читає його. Якщо лише одне завдання зчитувало цю адресу, то кількість завдань тепер дорівнює нулеві, і обробник завершення має позначити список. У випадку, якщо в цьому списку є завдання, то обробник видаляє їх і зменшує кількість залежностей у записах таблиці завдань. Як і попередній випадок, ця схема не здатна впоратися з усіма можливими випадками. Може статися так, що після додавання до початкового списку, до системи буде додано одного або декількох нових завдань з тієї ж адреси. Оскільки елементи у початкових списках цієї таблиці є номерами завдань, то лише одне з них може бути вилучено, оскільки не можемо мати кілька завдань, які пишуть одночасно в одній позиції. З іншого боку, кілька завдань можуть читати з однієї адреси одночасно. Отже, не можемо додавати завдання як звичайні нові записи до початкового списку, оскільки декілька завдань можуть бути вилучені зі списку одночасно.

Таким чином, додається новий бар'єр, що складається з числа завдань, які очікують адреси, але представлений у вигляді від'ємного значення, щоб відрізнити його від звичайного ідентифікаційний номер завдання. Відповідно, якщо певна кількість завдань чекають на адресу, а між ними немає розподілу, бар'єром буде апаратний пристрій. У такому випадку, коли обробник обробляє завершене завдання, то він перевіряє список. Якщо програма виявить від'ємне значення, вона вилучить елемент зі списку і запише кількість завдань у поле відповідним чином. Для цієї конкретної ситуації немає потреби зменшувати кількість залежностей у таблиці завдань, оскільки вони вже встановлені як залежні у списку виробників. Для простоти передбачається, що виконання першого завдання відбувається повільніше, ніж додавання інших завдань, і що ядра ніколи не зупиняється.

Таким чином, коли перша задача закінчується, інші завдання вже оброблені обробником дескриптора. Псевдокод обробника дескриптора та обробника розглянемо на кожному кроці додавання та вилучення завдань до початкових списків або зі списків. Перше додане завдання — це виробник даних, оскільки він має блок для виходу. Після завантаження в таблицю завдань обробник дескриптора безуспішно перевіряє, чи присутня його позиція в таблиці виробників. Таким чином, він записує новий елемент у властивість таблиця, а кількість її залежностей залишається порожньою. Наступним кроком буде перевірка таблиці споживачів, в якій також немає запису за цією адресою. Цього разу запис у таблиці споживачів не виділяється, оскільки поточне завдання не споживає дані. Через нульову кількість залежностей він додається до черги готовності до виконання. При появі другого завдання обробник дескриптора виявляє його залежність від блоку у списку виробників, і таким чином це завдання додається до списку. Наступним кроком є додавання нового запису в таблицю споживачів з встановленим рівнем, що вказує на те, що один зчитувач блокує адресу. Це поле записів таблиці споживачів містить кількість завдань, які в даний момент звертаються до адреси запису. У цьому випадку кількість залежностей завдання дорівнює одиниці. Далі приходить завдання і, оскільки

воно є основним, то повинно додати запис до списку продюсерів. Однак для цієї адреси вже існує запис. Тому повинні розділити список бар'єром, щоб вказати на наявність другого завдання. Цей бар'єр є запереченням ідентифікаційного номеру завдання. Він також знаходить існуючий запис у таблиці споживачів, тому його ідентифікатор завдання також додається до початкового списку, але тепер у позитивному представленні. Кількість залежностей у цьому завданні встановлено рівним два, оскільки воно залежить від однієї адреси в таблиці споживачів та однієї адреси в таблиці виробників. Коли зчитувач завдання надійде, то його потрібно додати до стартового списку таблиці виробників, як це сталося із завданням два, розмістивши його після бар'єру. Тепер знайдено єдиний випадок для списку споживачів, оскільки це завдання має додати новий елемент до списку споживачів, але виявляє, що елемент вже існує. Таким чином, він записується як бар'єр, який вказує на те, що є один споживач, який чекає на адресу. Незважаючи на те, що нові елементи були записані в обидва початкові списки, цього разу кількість залежностей дорівнює одиниці. Випадок із завданням дуже схожий на раніше додане завдання, оскільки вони обидва намагаються прочитати блок даних. Відповідно, ідентифікаційний номер додається до стартового списку таблиці виробників одразу після ідентифікаційного номеру стартового завдання. Для таблиці споживачів, замість додавання нового бар'єру стає бар'єр, що вказує на два завдання, які очікують на адресу. Кількість залежностей така сама, як і для попереднього завдання, встановлена на одиницю. Останнім завданням є завдання виробника, яке є таким самим, як і завдання споживача, додаючи його заперечуваний ідентифікатор завдання в записі списку виробників, а його позитивне значення в таблиці споживачів після бар'єру. У записі таблиці завдань він має кількість залежностей.

Фіналізація завдань впливає на таблиці. Згідно з початковим припущенням, коли завдання закінчується, то інші завдання вже додані в систему. Після завершення кожного завдання та обробки залежностей обробником запис таблиці завдань, пов'язаний із завершеним завданням,

видаляється. Обробник обробляє завдання, перевіряючи таблицю виробників, оскільки це завдання є виробником. Він вилучає завдання зі списку стартових завдань, поки не буде встановлено бар'єр включно. У цьому випадку завдання будуть видалені. Крім того, обробник зменшить кількість залежностей записів таблиці завдань для обох завдань. Оскільки у задачі не залишилося залежностей, то вона буде додана до черги готовності та виконана. Після того, як він завершив виконання і оскільки він є споживачем, цього разу обробляється таблиця споживачів. Обробник зменшує кількість завдань, що містяться в полі адреси, прочитаної завданням, яка досягає нуля і, таким чином, він повинен запускати елементи зі списку один за одним. Причина, по якій вони видаляються по одному, полягає в тому, що елементи тут є виробниками і не можуть писати на одну і ту ж адресу одночасно. Видаленим елементом є завдання, яке більше не має позитивної кількості залежностей і тому переміщується до черги готовності. Оскільки наступним елементом є бар'єр з певним значенням, то цей елемент також видаляється і значення записується в поле для цього запису. Після того, як завдання закінчиться, оскільки це виробник, то він повинен перевірити таблицю виробників, вилучивши кожен елемент до бар'єру включно. Це призведе до того, що завдання досягнуть нульової залежності і одночасно будуть переміщені до черги готовності. Оскільки вони не змінюють дані в блоці, то вони можуть працювати одночасно. Але в таблиці споживачів у нього на одну залежність більше, тому він поки що не може запуститися. Припускаючи, що завдання виконано першим, він обробить запис адреси в таблиці споживачів, зменшуючи кількість завдань з цієї адреси. На відміну від попереднього випадку, він не досягає нуля, тому обробник не вилучає жодного елемента зі списку, поки не завершиться завдання. Коли це відбувається, обробник видаляє ідентифікаційний номер завдання зі списку та зменшує кількість його залежностей у таблиці завдань. Оскільки він більше не має залежностей, завдання додається до готової черги та обробляється. У цей час не потрібно утримувати запис в таблиці споживачів і, отже, він видаляється. Після виконання задачі обробник не може знайти будь-який елемент, що залежить від завдання у

таблиці виробників, тому цей запис видаляється.

Схема вирішення залежностей з АП намагається подолати вузькі місця в моделі програмування ПП. Це робиться за допомогою простого пошуку таблиць, які динамічно розв'язують залежності задач і визначають завдання, доступні для виконання. Для цього він використовує кілька паралельних етапів конвеєра в якості обробників, завантажувача дескрипторів або інтерфейсу регістрів, який виконує читання і запис з або в регістри, відображені в пам'яті. Ці регістри, доступ до яких можна отримати з будь-якого блоку системи, використовуються для побудови ефективної процедури синхронізації і повинні виконувати операції читання і запису за кілька циклів. Для вивчення впливу блоку АП на гетерогенну багатоядерну платформу метою даної роботи є моделювання його поведінки та порівняння з програмним забезпеченням. В останні роки з'явилося кілька симуляційних середовищ, які намагаються змоделювати багатоядерні форми пластин з метою дозволити дослідникам отримати знання про їх поведінку. Тим не менш, переважній більшості з них не вистачає загальності, оскільки вони розроблені для конкретних архітектур процесорів. Це є перешкодою для еволюції через велику складність перенесення одного модуля, що використовується в одному інструменті, на інший, що робить порівняння між різними конструкціями мікросхем громіздким. Крім того, більшість доступних імітаційних фреймворків є складними, тому, крім того, що експерименти складні у виконанні, вони також дорогі. Система моделювання була створена для того, щоб подолати всі ці потреби, пропонуючи модульний і безкоштовний симулятор. Це інструмент, який використовується для імітації апаратної підтримки. Система моделювання – це структурний симуляційний фреймворк, який розділяє архітектуру на різні блоки, кожен з яких відповідає одному з елементів архітектури. Його основна характеристика полягає в тому, що він відображає апаратне забезпечення безпосередньо на симуляторі, роблячи імплементацію дуже зрозумілою, а також покращуючи її читабельність і простоту. Крім того, симулятор дозволяє повторно використовувати блоки та взаємодіяти між різними середовищами, використовуючи рішення, для

середовища емуляції. Це досягається шляхом розподілу симуляторів у модулі з розширенням, тоді як зв'язки між цими блоками, що складають архітектуру, встановлюються. Така особливість досягається також визначенням стандартизованого інтерфейсу зв'язку між модулями. Інтерфейс децентралізує логіку керування, встановлюючи натомість розподілене керування між блоками, що полегшує виділення нового модуля в архітектурі або модифікацію існуючого блоку. Крім того, існує широкий набір повносистемних функціональних симуляторів, здатних завантажувати всю операційну систему, хоча в випадку дослідження буде використовуватися лише невелика частина з них. Він також дозволяє включати в модулі для виклику можливостей системи спеціальні функції, де можемо знайти моделювання потужності, вибірку або налагодження.

Існує дві різні методики побудови тренажера, в залежності від призначення конструкції. Коли інтерес в основному зосереджений на функціональній поведінці системи, а не на вимірюванні продуктивності, то методологія є найкращим вибором. Він забезпечує вищу швидкість моделювання, але недоліком є відсутність часу, тому затримки будуть проігноровані. З іншого боку, коли продуктивність системи є головною проблемою, то методологія пов’язана з продуктивністю є найбільш підходящим варіантом, оскільки вона пов'язана з часовими проблемами в моделюванні. Тим не менш, це вимагає більшого часу моделювання, оскільки враховується кожна затримка сигналу. Інтерфейс зв'язку має вигляд простого трьохсигнального рукостискання між двома різними модулями. Для того, щоб підключити модуль, потрібно визначити цей набір сигналів, присутній у вхідному та вихідному портах. Коли першому блоку потрібно відправити щось до будь-якого іншого блоку, то він спочатку починає доставляти дані через шлях даних, що з'єднує їх. Після того, як пункт призначення отримав їх, він може прийняти або відхилити дані, використовуючи сигнал прийняття. Тим не менш, модуль не обробляє нові значення, поки не буде підтверджено сигнал увімкнення, щоб запобігти прийняттю раніше оброблених даних. Цей останній механізм є частиною елемента керування, яка служить синхронізацією, коли джерело доставляє дані

для кількох адресатів. Для обробки цих сигналів він дозволяє визначати чутливі методи. Вони викликаються щоразу, коли змінюється один із цих сигналів.

Модульний симулятор призначений для створення прототипів гетерогенних мультипроцесорів. Завданням системи є тестування працездатності різних архітектурних схем. Це досягається завдяки його високій модульності, обранню методології, запропонованої для дослідження продуктивності. Незважаючи на те, що він може бути налаштований на дослідження різних гетерогенних архітектур процесорів, його першим призначенням є моделювання процесора. Саме тут потрібно додати апаратну підтримку, щоб оцінити її вплив. В рамках реалізації деякі параметри можуть бути скориговані з метою дослідження впливу таких змін на поведінку процесора. Кожен апаратний блок архітектури має своє представлення на симуляторі. Вони мають форму класів і будуть в модулі, відповідно до вимог фреймворку моделювання. Ці компоненти повинні мати дані про вищезгадані сигнали зв'язку, приймати і включати, а також чутливі до них методи. Крім того, відповідно до методології, також вимагається, щоб усі модулі мали спільний тактовий сигнал із пов'язаним чутливим методом, який використовується симулятором для виконання функціональності кожного модуля, пов'язаного з поточним циклом. Взаємозв'язок сигналів між модулями визначається в межах оригінального формату, а синхронізація блоків є відповідальністю фреймворку. Він обробляє виклики методу при зміні сигналів. Незважаючи на те, що він вже був перевірений раніше, щоб відповідати реальному процесору, все-рівно потребує перевірки, оскільки відбулися зміни в управлінні потоками. Незважаючи на те, що він точно відповідає реальному процесору, є кілька відмінностей. Модуль пам'яті використовується для моделювання як основної пам'яті, так і різних локальних підсистем з використанням одного і того ж класу пам'яті. Це досягається завдяки тому, що затримка доступу та кількість портів кожного модуля можуть бути налаштовані так, щоб вони діяли як локальне сховище. Друга відмінність впливає на ієрархію кешу. На відміну від реальної апаратури, яка містить два рівні кешу, в симуляторі реалізований тільки один

рівень кешу. Зв’язки моделюються на функціональному рівні, а не на рівні циклу. Інструкції не виконуються, виходячи з ширини випуску та наявності функціональних блоків. Параметр може бути змінений для визначення статичної швидкості вибірки інструкцій. Неточність моделі може полягати в плануванні потоків. Тому всім їм присвоюється однаковий часовий проміжок процесора, незалежно від того, простоюють ці потоки чи ні. У реальному процесорі, натомість, лише активним потокам надається часовий зріз процесора. На зміну такому підходу до моделювання потрібно ввести підхід з моделюванням міжмережевого з'єднання, що складається з топології, як спроба використовувати більш масштабований підхід. Оригінальне з'єднання дуже ефективне, але не дуже масштабоване. Хоча вони відрізняються, параметри дозволяють зіставити продуктивність.

Для інтерфейсу зв'язку в симуляторі обреремо загальну схему для підключення кожного модуля у вигляді класу. Кожен порт модуля вводу-виводу містить такий клас, як сигнал даних, разом з сигналами, що використовуються в рукостисканні. Усередині цього контейнера даних міститься деяка цінна інформація про суб'єктів, які беруть участь у спілкуванні.

Таким чином, всередині цієї інформації є поля, які визначають, чи є пакет вантажем або сховищем, а також адреси джерела і призначення. У разі, якщо це сховище, передані дані також включаються в пакет. Кожен модуль процесора зіставляється з фізичним діапазоном адрес. Тому він також потребує виконання відображення пам'яті своїх модулів на різні діапазони адрес у глобальному адресному просторі. Відображення пам'яті дозволяє підключати змінну кількість процесорних блоків і пам'ять. Змінна позначає кількість ядер, присутніх у поточній конструкції, яка в цьому випадку дорівнює одиниці, оскільки підключено лише одне ядро. Для корпусів індекси, що додаються до їх назв, вказують на номер елементу. Таким чином, для цих двох випадків елементи з однаковим номером, прикріпленим до назви модуля, належать до одного і того ж ядра. Наприклад, ці елементи знаходяться всередині одного ядра.

Це гнучке відображення призначене для дослідження різних архітектур.

Така гнучкість досягається визначенням діапазонів адрес для кожної групи елементів. Коли модулю потрібно відправити дані в будь-який інший модуль, то він відправляє дані з цільовою адресою, згідно з таблицею, в кеш отримує дані з одного зі своїх вхідних портів. Після того, як він буде отриманий, з'єднання надішле до блоку, що містить цільову адресу, через вихідний порт, призначений для цієї таблиці. Оскільки визначається новий апаратний блок, то новий діапазон фізичної пам'яті повинен бути зарезервований для того, щоб мати можливість адресувати нашу апаратну підтримку, і тому новий елемент був би включений в таблицю. Передачі прямого доступу до пам’яті не збігаються з реальним процесором за затримкою. Оскільки затримка прямого доступу до пам’яті є параметром, який використовується симулятором для моделювання цих передач прямого доступу до пам’яті, на симуляторі потрібно запускати різні затримки прямого доступу до пам’яті, щоб відповідати реальній продуктивності. Результати тестів прямого доступу до пам’яті завжди показували менші затримки, ніж на реальній платформі, навіть незважаючи на те, що тестувалися великі затримки, без будь-якого значного покращення. Результатом аналізу коду стане те, що параметр затримки прямого доступу до пам’яті не буде призначений деяким типам команд прямого доступу до пам’яті. Двома основними командами прямого доступу до пам’яті є ті, які використовуються для доставки даних до будь-якого пункту призначення, і ті, які використовуються для запиту даних з будь-якого джерела. Вихідний код розрізняє, коли застосовувати затримку в залежності від типу команди, використовуючи для цього оператор. Зокрема, затримки прямого доступу до пам’яті заборонені для команд, відмінних від типу наступних. Щоб вирішити цю проблему потрібно отримати більш точну модель часу, витраченого на прямого доступу до пам’яті. Розглянемо прив'язку блоку до існуючої архітектури.

Оскільки стратегія рішення це вплив на продуктивність нового апаратного блоку, який позбавляє систему від вузького місця, виявленого в дослідженні роботи процесорів в багатоядерних системах, то необхідне визначення такого нового обладнання та його підключення до міжмережевої мережі. Оскільки його

реалізація має відбуватися за правилами симулятора, то також необхідно призначити фізичний діапазон сторінок, щоб до нього можна було звертатися з інших модулів. Основне ядро додає завдання до множини завдань, а робочі ядра отримують готові завдання для виконання, сигналізуючи їм про їх повернення, як тільки вони були виконані. Було додано загальний сигнал годинника. Відповідно до конструкції АП є два вхідних і два вихідних порти, один для регістрів і один для зберігання завдань. Вони незалежні один від одного. Після того, як сигнали визначені, потрібно призначити кожен вхідний сигнал методу, який викликається щоразу, коли виявляється зміна сигналу. Основна функціональність компонента буде розподілена між методами. Обидва сигнали даних будуть згруповані в один і той же метод, так як виконували одні і ті ж операції, але тільки розрізняючи об'єкт таких операцій. Аналогічно буде і з сигналами прийому. Функція використовується для ініціалізації блоку на початку симуляції. Така ініціалізація викликається з файлу, де до функції додається новий рядок. Ця функція запускає всі блоки, що містяться в симуляторі, подібно до глобального сигналу скидання, де вона представляє весь симулятор. Кожен елемент цього запису є елементом архітектури. Останнім файлом, який братиме участь у визначенні множини завдань, є файл, який містить конфігурацію блоків за замовчуванням. Після того, як блок визначений як компонента архітектури, наступним кроком є побудова його зв'язку з мережею взаємозв'язку. Це потрібно зробити у файлі, де викликаються та зв'язуються класи, що містять модулі. Тому в цьому файлі внесемо кілька змін, для того, щоб викликати новий блок і зв'язати його з існуючим архітектураом. Першим доданим рядком буде включення файлу, що містить клас, який знаходиться в папці процесора. Оскільки новий блок з двома портами підключений до нього, то кількість портів міжмережевого з'єднання тепер збільшується на два для кожного процесора в архітектураі, тому цей параметр змінюється. Наступні два рядки відповідатимуть створенню екземпляру нового модуля, вказуючи на кількість портів, визначених у параметрах конфігурації за замовчуванням. Вони призначені для модуля налагодження, для підключення

його до спільного глобального годинника та для створення екземпляра класу.

Таким чином, дослідження предметної області щодо доповнення архітектури апаратнтним засобом показало, що такий підхід є перспективним. Він може подолати проблеми програмної моделі. Для доведення його ефективності потрібно використовувати симулятор.

2.2 Організація зв’язку в багатопроцесорній системі

З блоком вже визначено і підключено його до годинника, то потрібно буде ще побудувати дроти з мережею. Це робиться в циклі, де кожен вихідний порт під’єднується до пов'язаного з ним вхідного порту, роблячи те ж саме для вхідних портів з його відповідним вихідним портом. Це встановлює фізичний зв'язок між новим модулем та рештою архітектури. Видалення після закінчення симулятора необхідне, щоб звільнити місце в пам'яті. Однак, крім фізичних зв'язків, необхідно також встановити діапазон пам'яті, щоб мати можливість звертатися до нього. Оскільки новий модуль має бути доступний з інших модулів, то він повинен мати дійсний діапазон у спільному адресному просторі. Для майбутньої роботи, коли може бути присутнім більше одного процесора, вони можуть бути розміщені відразу після попередньої, послідовно. Модуль має два порти, один для інтерфейсу регістра, а інший для зберігання завдань, і, отже, необхідно призначити два різні набори адрес. Блок зберігання завдань набагато більший за блок інтерфейсу регістрів, оскільки він повинен містити дескриптори завдань кожного завдання, що міститься в множинаі. Інтерфейс регістра містить черги елементів, які вже не є цілими числами на кожен елемент черги, буфер та регістр стану. Крім того, цей інтерфейс регістру також має логіку керування, яка бере участь у синхронізації з іншими модулями та управлінні завданнями. Відповідно, встановлення діапазону інтерфейсу регістрів залишає достатньо місця для черг, регістрів і логіки керування. Діапазон зберігання завдань було встановлено, оскільки завантажується вся інформація щодо завдань. Цього діапазону адрес також бути достатньо, оскільки призначено зберігати не більше

кількох тисяч дескрипторів завдань. Будучи вже визначеним діапазоном адрес, він повинен бути доданий в таблиці маршрутизації, які використовуються мережею взаємозв'язку для визначення призначення даних. Таблиця маршрутизації задається як параметр при виклику симулятора. Такий файл виходить за викликом програми, розташованої в папці таблиць маршрутизації. Створений файл містить інформацію про початкову та кінцеву адресу кожного блоку, порт, пов'язаний з кожним із них, та існування кешу.

Новий компонент має фізичний зв'язок з мережею взаємозв'язку та призначений дійсний фізичний діапазон адрес. Однак достукатися з будь-якого модуля поки що неможливо. Причина в тому, що він має фізичну адресу, але не віртуальну, а вони оперують віртуальними адресами, а не фізичними. Відповідно, застосунок, що працює на процесорі, повинен вибудовувати зв'язок між реальною адресою і фізичною адресою. За встановлення зв'язку між віртуальною адресою та фізичною адресою відповідає операційна система, що працює на ядрі. Таким чином, це робиться шляхом виклику функції, яка повертає віртуальну адресу, присвоєну йому. Ця функція виконує два системних виклики. Був відкритий перший системний виклик, який відкриває дескриптор файлу, що вказує на нього, заданий ім'ям модуля як параметром. Значення, яке повертає ця функція, використовується для зв'язку фізичної адреси з віртуальною. Однак це неможливо зробити з початковим набором системних викликів. Файловий модуль, де реалізований функціонал відкритого системного виклику, не включає в себе опцію створення файлового дескриптора для множини завдань, і таким чином він був доданий. Він викликає клас для відкриття файлового дескриптора, використовуючи для цього клас. Цей клас використовується для запиту фізичної адреси, пов'язаної з блоком, а також його розміру. Відповідно до цього також необхідно змінити корту, написавши функцію, що повертає початок фізичного діапазону адрес, і розмір блоку. Після відкриття дескриптора файлу наступним кроком є виконання фізичного відображення пам'яті, виконання системного виклику. Ця функція отримує в якості параметрів покажчик дескриптора файлу і розмір запитуваної віртуальної адреси. Цей системний виклик запитує вільний

діапазон віртуальних адрес із заданим значенням розміру, повертаючи першу віртуальну адресу такого діапазону. В рамках цієї функції віртуальна адреса присвоюється фізичній адресі. Значення, повернуте останнім системним викликом, використовується системою для посилання на ядро і має бути надіслано апаратному пристрою.

Під час проведення симуляції важливо валідувати симулятор, щоб отримати надійні результати. Це дозволяє порівнювати результати з цільовою платформою, оскільки передбачає точний підхід до реального обладнання. Важливою відмінністю є відсутність кешу другого рівня в ядрі симулятора. Цей факт створює невідповідність між затримками для кешу 1-го рівня в реальному процесорі та симуляторі. У той час як в реальному процесорі він намагається отримати дані з кешу другого рівня, то в симуляторі він отримує дані безпосередньо з основної пам'яті. Прямий доступ до основної пам'яті пов'язаний з більшими затримками, ніж до кешу другого рівня, тому цей ефект має бути збалансованим. Для цього затримка пам'яті була зменшена як спроба змоделювати середню затримку, спричинену відмінностями кешу першого рівня. Однак зменшення затримки пам'яті також впливає на затримку пам'яті. Оскільки всі звернення до пам'яті є передачею прямого доступу до пам’яті, то затримка повинна бути збільшена, щоб компенсувати ефект коротшої затримки пам'яті. Параметри використовуються для валідації тренажера. Три значення пов'язані з ядром, оскільки це затримка для кожного доступу до основної пам'яті, затримка кешу першого рівня. Наступним параметром у таблиці є ідентифікаційний номер, який змінює швидкість вибірки інструкцій, але не ширину випуску. Тим не менш, незважаючи на те, що ці параметри змінюють IPC ядер, це не дорівнює реальному значенню. Реальний IPC вимірюється від виконання і залежить від коду, що виконується в ядрі. Така ситуація пов'язана з неточністю симулятора. У реальному процесорі в межах ядер розрізняють два різних ступені: парний і непарний. Однак це не реалізовано в симуляторі, де витягується і виконується ряд інструкцій, незалежно від етапу конвеєра, до якого належить інструкція. Таким чином, встановивши цей параметр на два, буде

виконано більше інструкцій, ніж у реальному процесорі. Таким чином, необхідно знизити цей параметр і тоді виконається менше інструкцій як спроба збігтися з середнім IPC ядра. Потім наступні два параметри є його відповідною затримкою та пропускною здатністю.

Для отримання набору параметрів конфігурації потрібно проводити кілька тестів, виконуючи один і той же код. Ці тести виміряють кілька фаз нормального виконання, намагаючись ізолювати ефект кожної з них, щоб отримати точний підхід. Спочатку потрібне вимірювання для налаштування і затримки кешу 1-го рівня, виконуючи лише одне завдання, розташоване в кеші. Після того, як буде оптимізовано затримку кешу 1-го рівня можна визначити затримку пам'яті. Для цього тесту з пересиланням треба виконати та виміряти лише передачі прямого доступу до пам’яті в обох випадках, надсилання та отримання даних. Цей тест потрібний для вивчення параметра затримки прямого доступу до пам’яті. У випадку на одному ядрі потрібно виконати лише одне завдання для визначення найкращого підходу. Декодування всієї матриці буде виконано за допомогою послідовної обробки.

Таким чином, представлене симуляційне середовище та симулятор, який використовується для імплементації АП. Оскільки метою є впровадження АП, то необхідно дотримуватися правил. Потрібно далі визначити і підключити до існуючого тренажера систему з АП. Крім того, реалізувати протокол зв'язку, що містить сигнали та чутливі методи. Для можливості адресувати новий блок з інших блоків архітектури в таблицях маршрутизації симулятора потрібно присвоїти і введений діапазон адрес. Блок зв'язується з іншими модулями архітектури за допомогою вхідних і вихідних портів, а також чутливих методів, які реагують на зміни сигналів, присутніх на цих портах. Функціональність системи розподіляється між методами. Функціональність була розділена на два основних блоки: менеджер залежностей, що містить логіку керування, та таблиці, що беруть участь в розв'язанні залежностей; другу частину, відповідальну за обробку інтерфейсу регістра з відображенням пам'яті. Основне функціональне ядро множини завдань програмується всередині основного

файлу. Разом з функціональними можливостями визначаються черги конструкції, черга в буфері, черга готовності, черга буфера фінішу та деякі внутрішні черги. Серед цих внутрішніх черг одна з черг використовується завантажувачем дескрипторів, щоб вказати обробнику дескриптора на наявність нового дескриптора завдання, який необхідно обробити. Наступна внутрішня черга - це черга, в якій залишаються запити до моменту їх відвідування. Коли вона намагається отримати готове завдання, але в черзі готовності немає жодного елемента, то запит зберігається в цій внутрішній черзі. Останні внутрішні черги – це черги, які відстежують вільні елементи у сховищі завдань та таблиці завдань. Оскільки таблиця завдань і сховище завдань мають фіксований розмір, то не можна виділити більше певної кількості записів у кожному з них. Після досягнення свого ліміту потрібно дочекатися готового завдання, яке випустить запис, щоб розмістити нові записи. Тому необхідно стежити за вільними елементами в списках, які будуть вказувати, чи можна додавати нові предмети чи ні. Це реалізовано у вигляді черги, яка містить кількість вільних елементів у таблиці завдань. У випадку таблиць споживача та виробника замість цього була використана хеш-функція з роздільною здатністю колізій, оскільки ці два списки потребують швидкого доступу до її елементів. Для цих двох останніх таблиць зайнятість кожного запису визначається в самому записі таблиці. Аналогічний випадок таблиці завдань можна застосувати і до сховища завдань, яке також має фіксований розмір. У зв'язку з тим, що дескриптори завдань також мають фіксований розмір, описувачі завдань записуються в індекси сховища завдань. Відповідно, повинні визначити додаткову чергу з індексами цих адрес у сховищі завдань, відображених у пам'яті. Якщо в таблиці завдань сховища завдань немає жодного запису, продовжувати обробку нового завдання не дозволяється. Це означає, що елементи буфера не обробляються і не видаляються, в результаті чого ЗІЗ перестають додавати завдання.

На початку виконання в списках є всі записи присутніх в таблиці завдань і сховищі завдань. Щоразу, коли надходить нове завдання, призначається

елемент як у таблиці завдань, так і в сховищі завдань, утримуючи їх до завершення завдання. Ці ідентифікатори в таблиці завдань і сховищі завдань вивільняються після завершення виконання завдання і обробки. Він також містить код завантажувача дескрипторів, обробника дескрипторів, обробника загального і додаткові функції, необхідні для доповнення їх функціональності. Крім того, в цьому файлі є функція ініціалізації, яка викликається на початку симулятора. Така функція обнуляє внутрішні змінні і вибудовує всі черги конструкції з їх конкретні розміри. Оскільки це ініціалізація, черга в буфері, черга готовності, черга буфера завершення і черга, де ядра залишаються до участі, є порожніми. Навпаки, черги, що містять вільні елементи в таблиці завдань і сховищі завдань, заповнені, оскільки жоден елемент не зайнятий. Регістр стану налаштований таким чином, щоб вказати, що у нього немає ніяких завдань і можна почати їх отримувати. Також є порти модуля та методи, які викликаються, коли будь-який сигнал портів змінюється ззовні. Ці методи охоплюють дані трьох сигналів рукостискання, приймають і дозволяють брати участь у зв'язку з мережею взаємоз'єднання. У разі, якщо дані змінюються в будь-якому з портів, то викликається його конфіденційний метод. Він виконує простий цикл, який перевіряє, чи є щось на портах, стверджуючи сигнал прийняття для портів, куди щось прибуло. Дотримуючись правил, не можна обробляти дані, поки вони не будуть увімкнені з джерела. Якщо сигнал прийняття подається ззовні в будь-якому з портів, призначений метод дозволяє даним, надісланим уздовж цього порту, завершити рукостискання. Коли сигнал прийняття більше не активний, сигнал увімкнення знову встановлюється на нуль. Незважаючи на наявність одного унікального методу включення сигналів, розрізняють два методи, так як кожен порт повинен виконувати різні операції з вхідними даними. В обох випадках інтерфейс регістра викликається відповідно до типу операції, включеної в дані, або навантаження, або сховище. Однак, якщо дані надходять з першого порту, то вони націлені на відображені регістри пам'яті. Навпаки, якщо дані надходять з другого порту, то операцію потрібно виконувати над сховищем завдань. Якщо тип операції є зчитуванням, то не

повинен реагувати негайно, оскільки він несе деяку затримку. Тому після зчитування необхідного регістра адреси зберігання завдань відповідь зберігається у вихідному буфері разом із пов'язаною з нею затримкою. Ця затримка розраховується в залежності від обсягу зчитуваних даних. Методи, чутливі до висхідного і спадаючого краю годинника, також беруть участь в передачі даних.

Таким чином, для випадку висхідного краю, якщо вихідний буфер не порожній, то вихідні дані записуються у вихідний порт після закінчення його відповідної затримки. У випадку падаючого краю, коли дані були передані, цей метод видаляє дані з вихідного порту, якщо вони вже були прийняті. Якщо сигнал прийняття затверджений, то він уже увімкнений одразу після надходження сигналу прийняття. Серед цих двох функцій, чутливих до глобального годинника, називають основний функціонал системи. Отже, завантажувач дескрипторів і обробник дескрипторів викликаються на початку циклу годинника, а обробник викликається в кінці циклу годинника. Після закінчення тактового циклу, якщо відбулися будь-які зміни в регістрі стану, відображений регістр пам'яті стану оновлюється викликом інтерфейсу регістра.

Інтерфейс регістру служить зв'язком блоку з іншими модулями архітектури, і він упакований в файли. Цей інтерфейс розміщується між покажчиком залежностей і мережею взаємозв'язку платформи. Кожен доступ до нього або з нього повинен проходити через цей модуль, який містить різні регістри, що використовуються для синхронізації. Після того, як дані включені в одному з вхідних портів, то це запускає обробку. Ці вхідні дані можуть бути як навантаженням, так і сховищем, включаючи або вимагаючи дані відповідно. Після надходження даних викликається блок керування пам'яттю, щоб перевірити, чи правильна цільова адреса вхідних даних. Залежно від порту, звідки надходять дані, відповідна операція виконується або в відображених регістрах пам'яті, або в сховищі завдань. Схема регістрового інтерфейсу була реалізована аналогічно інтерфейсу регістра для збереження однорідності симулятора. Блок керування пам’яттю реалізований і має функції, необхідні для

обробки фізичних адрес, що викликається, отримує в якості вхідних параметрів своєї ініціалізації фізичні адреси, віднесені до регістрів і сховища завдань, і зберігають їх у внутрішніх змінних. Основний функціонал цього блоку полягає у виконанні трансляції між вхідною фізичною адресою в адреси. При ініціалізації ТПУ цей модуль виконує відображення в пам'яті фізичних адрес в індекси регістрів. Під час виконання симулятора цей модуль викликається для перевірки призначення вхідних даних, розрізняючи сховище завдань і регістри. Крім того, отримавши фізичну адресу, він повертає індекс регістру, пов'язаний з такою адресою. Використовується при зверненні до реєстру банку. Ця функція виконує відображення фізичної адреси в пам'яті в індекси регістрів. Ця функція викликається на початку моделювання, як тільки запускається процесор. Ці функції викликаються, коли він отримує дані зі своїх портів, щоб гарантувати, що призначення таких даних отримано в потрібному порту.

Таким чином, обидва компоненти процесора, менеджер залежностей та інтерфейс регістра, поділяють доступ черги готовності, фінішного буфера та загальної черги. Інтерфейс регістрів дозволяє записувати та читати з елементів з відображених регістрів пам'яті. Регістр буфера та регістр буфера фінішу діють як нижні ребра черг буфера та фінішного буфера. Ці дві черги доступні лише для запису елементів до черги. Натомість, регістр готової черги поводиться як верхній край черги готовності, і його можна прочитати лише ззовні. При зверненні з логіки керування до цих черг режим роботи зворотний: зчитується черга в буфері і черга фінішного буфера; записується черга готовності. Реєстр статусів не включає жодної черги. Він зберігає цінну інформацію, тобто про те, запущений він чи ні, кількість вільних елементів у буфері та кількість елементів, присутніх у черзі готовності. Цей регістр можна зчитувати з будь-якого модуля симулятора, і він оновлюється на падаючому краю тактового циклу, якщо в цих значеннях відбулися будь-які зміни. Це робиться за допомогою керуючої логіки, яка виконує виклик регістра для оновлення регістра при виявленні зміни. Інтерфейс регістра також включає сховище завдань, що використовується як централізована пам'ять, до якої можна швидко отримати доступ. Крім того, що

доступ до нього здійснюється ззовні модуля, він також використовується зсередини. Крім того, керуюча логіка, що міститься, також може виконувати різні виклики інтерфейсу регістра для виконання читання та запису. Ці дві операції можуть бути виконані як в інтерфейсі регістра для оновлення стану, так і для зчитування дескриптора зі сховища завдань.

2.3 Висновки до другого розділу

Здійснено дослідження предметної області та визначено стратегію забезпечення безпеки. Запропоновано використовувати два компоненти процесора менеджер залежностей та інтерфейс регістра. Вони поділяють доступ черги готовності, фінішного буфера та загальної черги.

3 МЕТОД ДИНАМІЧНОГО КЕРУВАННЯ БАГАТОЯДЕРНИМИ КОМП'ЮТЕРНИМИ СИСТЕМАМИ

3.1 Спосіб обробки завдань та використання декскриптору

Після того, як дескриптор готовий, то система перевіряє наявність місця в буфері, зчитуючи регістр стану. Вона продовжує зчитувати цей регістр до тих пір, поки не виявить вільний елемент буфера. Після того, як йому надається вільний елемент, ядро записує основний покажчик пам'яті дескриптора завдання в буфер. Після того, як ця дія буде завершена, система починає готувати новий дескриптор завдання для подання. Вона виявляє нові дані у своїх регістрах, відображених у пам'яті, націлених на інтерфейс регістра. Тому, модуль інтерфейсу регістру називається таким, що має нові дані як параметр. Модуль інтерфейсу регістра розпізнає навантаження в буфер і виконує запис, додаючи новий елемент в нижній частині відповідної черги. Після того, як новий елемент був доданий в чергу в буфері на початку наступного циклу і, якщо завантажувач дескрипторів не зайнятий обробкою будь-якого іншого раніше доданого завдання, то він обробляє нове надходження. Оскільки буфер є чергою, то всі елементи обробляються по порядку. Завантажувач дескрипторів виявляє нове завдання, перевіряючи розмір черги в буфері. Коли ця черга має розмір більший за нуль, то вона відвідується. Перш ніж завантажувати дескриптор, повинна система спочатку переконатися, що в сховищі завдань достатньо місця. Це робиться шляхом перевірки черги, яка містить вільні записи в сховищі завдань. Коли система гарантує це, то він отримує дескриптор з основної пам'яті. Коли ці дані надходять, то модуль інтерфейсу регістра видає дані в сховище завдань для запису цільової адреси. Після того, як дескриптор завантажується в сховище завдань, внутрішня черга використовується для сигналізації дескриптору про наявність нового завдання, що підлягає обробці. Обробник дескриптору перевіряє чергу і виявляє, що є завдання без нагляду. Наступним кроком буде перевірка наявності вільних елементів в таблиці завдань, аналогічно випадку з завантажувачем дескрипторів і сховищем завдань. Також, подібно до черги в

буфері, елементи черги обробляються в порядку надходження. Коли в таблиці завдань немає вільного елемента, то обробник дескриптору не обробляє завдання. Замість цього він перевіряє кожен цикл, чи є вільний запис у таблиці завдань. Оскільки жоден елемент з черги не обробляється, то завантажувач дескрипторів продовжує додавати завдання, поки ця черга не заповниться. Як наслідок, завантажувач дескрипторів досягає стану, коли він не може обробляти завдання з буфера, який також заповнюється, змушуючи його припинити додавання завдань. Якщо інакше, обробнику дескриптору вдається знайти порожній запис, то він продовжує своє виконання, присвоюючи новій задачі вільну позицію таблиці. Ця позиція в таблиці також є ідентифікатором завдання, яке ідентифікує завдання всередині і буде відправлено. Далі обробник дескриптору виконує пошук у таблицях виробників та споживачів. Це визначає поточні залежності нового завдання з раніше доданими завданнями. Після цієї процедури обробник дескриптору заповнює необхідну інформацію в призначеному записі таблиці завдань. У разі, якщо нова задача не містить залежностей, обробник дескриптора додає новий елемент до готової черги. Цього разу не потрібно використовувати інтерфейс регістру, оскільки обробник дескриптору звертається до готової черги звідти всередині. Може статися так, що готова черга вже заповнена завданнями, які чекають на обслуговування. У цьому випадку ідентифікаційний номер завдання зберігається у внутрішній змінній і буде додано до черги готовності наступного такту, де є вільний елемент у черзі готовності. Поки це відбувається, жодне інше завдання не обробляється, поки готове завдання не буде остаточно додано до черги. На цьому етапі, незалежно від додавання завдання до черги готовності, обробник дескриптора завершує роботу в очікуванні видачі нових завдань.

Після того, як всі елементи ініціалізовані, то вони виконують блокуюче читання готової черги в очікуванні виконання завдань. Через свою блокуючу природу вони простоюють до тих пір, поки їм не буде призначено завдання. Система отримує запит на інтерфейс регістру з цільовою адресою, що вказує на готову чергу. Тому, вона викликає модуль інтерфейсу регістра для зчитування

цільового регістра. Якщо жодне завдання не може бути надіслане, то запит ставиться в чергу у внутрішню чергу. Щоразу, коли є доступне завдання, або під час першого запиту, або після того, як його поставили в чергу, готове завдання має бути надіслане до ядра виробника. Ідентифікатор готового завдання та покажчик на сховище завдань дескриптора задачі упаковуються та доставляються всередину класу. Отримавши ідентифікатор завдання та покажчик дескриптора, ядра повинні прочитати дескриптор завдання зі сховища завдань. Таким чином, робиться новий запит до інтерфейсу регістра, але на цей раз вказує на адресу дескриптора завдання. Знову ж таки, влна отримує цей запит і викликає інтерфейс регістру, орієнтований на сховище завдань. Однак це зчитування виконується негайно, оскільки для цієї операції не потрібна перевірка, бо забезпечується доступність дескриптору. Надсилається структура дескриптору завдання, що міститься в тому ж класі, що й запит. Вона отримує дескриптор і завантажує необхідні дані, згідно з дескриптором, з основної пам'яті через команду прямого доступу до пам’яті. Після отримання даних виконує операцію, зазначену в дескрипторі завдання. Після того, як це буде зроблено, він повинен сигналізувати про завершення завдання. Робиться це шляхом запису в фінішний буфер ідентифікаційний номер завдання, отриманого раніше. Аналогічно випадку з першим заповненням буферу, викликається інтерфейс регістра, на цей раз намагається записати готовий ідентифікаційний номер в фінішний буфер. Інтерфейс регістра додає новий елемент до черги завершення, очікуючи на обробку. Після завершення зв'язку вона намагається прочитати інше завдання з черги.

Обидва обробники і обробник дескрипторів мають спільний доступ до таблиці споживачів і таблиць виробників. На реальній платформі умова змагання може відбутися, якщо вони обидва виконують операцію одночасно в одному записі таблиці. Оскільки ця робота проводиться в симуляторі, то ніколи не відбувається жодних умов перегонів, оскільки кожна операція є атомарною.

Таким чином, їх виконання завершується в одному і тому ж тактовому ребрі. Механізм взаємного виключення повинен бути реалізований в реальному

обладнанні, щоб уникнути умов змагання, що передбачає додаткову затримку при кожному виконанні обробників. Одним з можливих рішень є використання атомарної апаратної інструкції для тестування і набору, а також прапорця на кожному записі таблиць споживачів і виробників. Таким чином, кожного разу, коли будь-якому з цих обробників потрібно було отримати доступ до певного запису таблиці, то він повинен був спочатку виконати таку інструкцію на своєму відповідному прапорці. Коли доступ надано, обробнику дозволяється виконувати будь-яку операцію у записі, інакше обробник повинен продовжувати чекати, поки запис більше не буде доступний з іншого обробника. Це не дозволить обом обробникам отримати доступ до одного запису одночасно. Необхідно застосовувати взаємне виключення до кожного запису таблиці окремо. При застосуванні до всієї таблиці продуктивність системи може значно знизитися, оскільки обидва обробники не можуть працювати одночасно, бо вони, як правило, звертаються до таблиць під час кожного виконання.

Інтерфейс регістру і менеджер залежностей поділяють доступ до різних черг множини завдань, Ці черги служать для зв'язку обох сутностей, так як інтерфейс регістра тягне за собою зв'язок з іншими блоками архітектури, а основний функціонал виконує логіка керування.

Завантажувач дескрипторів відповідає за взяття покажчиків дескрипторів задачі з черги в буфері та завантаження дескрипторів задач з основної пам'яті. Після завантаження дескриптора в сховище завдань він повинен сигналізувати обробнику дескриптора про наявність надходжень завдань. Це робиться за допомогою внутрішньої черги. Ця черга містить адреси в сховищі завдань завдання, які нещодавно надійшли і залежності яких ще потрібно обробити. Оскільки цей блок потребує запиту деяких даних з основної пам'яті, то він не може бути впевнений, коли з'явився дескриптор. В результаті, інтерфейс регістру додає покажчик на дескриптор задачі в черзі. Таким чином, цей блок будує запит на пам'ять і вказує у внутрішній черзі, що є одна передача. Така передача не може бути оброблена обробником дескриптору, оскільки його дескриптор завдання ще не знаходиться в сховищі завдання. Однак, з точки зору

дескрипторного завантажувачу, трансфери розглядаються як зайняті елементи в черзі. В іншому випадку не можна було б дізнатися, скільки елементів запитується в основну пам'ять, оскільки можливе перевищення ємності черги. Наприклад, нехай черга повинна бути заповнена дескрипторами завдань вже завантаженими в сховище завдань. Якщо з'являється новий дескриптор завдання, і намагатися додати його до черги, то він перезаписує елемент черги, що спричиняє пошкоджений результат. Після перевірки доступного розміру в черзі та у завданні, завантажувач дескрипторів починає будувати запит дескриптора завдання. Запит є навантаженням доступу до пам’яті націленим на адресу дескриптору. Однак покажчик, що надсилається є не фізичною, а віртуальною адресою. Відповідно, необхідно викликати номер буферу, який підтримує зв'язок між фізичними та віртуальними адресами. Маючи віртуальну адресу як вхідний параметр, він повертає відповідну фізичну адресу. Адреса джерела запиту пам'яті встановлюється як адреса зберігання завдань, раніше взята з черги. Оскільки основна пам'ять реагує на адресу джерела, то відповідь рухається до сховища завдань в інтерфейсі регістра, де вона записується. Коли інтерфейс регістра отримує дескриптор завдання, то він відразу ж записує його в цільову позицію, так як він гарантовано буде вільним. Подібно до випадку, коли зовнішній суб'єкт хотів прочитати сховище завдань, запит пам'яті ставиться в чергу у вихідний буфер порту регістра з пов'язаною з ним затримкою. Після виконання запиту завантажувач дескрипторів збільшує кількість переміщень, які згодом займуть одну позицію в черзі. Після запису сховища завдань адреса дескриптору задачі у сховищі завдань додається до черги. Це сигналізує обробнику дескриптору про те, що йому потрібно прочитати новий дескриптор і обробити його. Завантажувач дескрипторів витрачає деякий час на виконання згаданих операцій. Поки цей час не пройшов, завантажувач зайнятий і тому він не може обробляти нічого іншого. Для імітації такої поведінки можна визначити змінну, яка б містила час протягом якого завантажувач не може запуститися знову. Відповідно, після виконання завантажувачу ця змінна встановлюється на фіксовану затримку, яку можна налаштувати. Припустимо, що його затримка є

стаціонарною, оскільки цей блок виконує ті самі операції, і тому не можна визначити час, який витрачено. В наступні цикли після виконання завантажувача ця змінна зменшується один раз на кожен цикл, поки не досягне нуля. Коли це відбувається, завантажувач дескрипторів знову готовий до обробки нових завдань. Призначення обробника дескрипторів полягає в обробці завдань, надісланих після того, як завантажувач дескрипторів отримав дескриптор цих завдань. Завантажувач дескрипторів сигналізує про наявність нових завдань через внутрішню чергу, призначену для зв'язку з обома сутностями. Обробник дескрипторів отримує покажчики дескрипторів та обробляє залежності і вводить нові записи в таблицю завдань, заповнюючи їх поля після обробки завдань. Крім того, якщо нова задача не має залежностей, обробник дескриптору повинен додати їх до готової черги. Кожного разу, коли обробник дескриптору виявляє новий елемент у черзі, то він намагається його обробити. Однак, як і у випадку з завантажувачем дескрипторів та простором у сховищі завдань, обробник дескриптора повинен перевірити наявність вільного елементу в таблиці завдань. Він використовує той самий механізм, що й завантажувач дескрипторів, тобто чергу, що містить вільні елементи в таблиці завдань. Якщо в таблиці завдань немає жодного доступного запису, то він повинен дочекатися завершення виконання завдання. Якщо це так, то може статися так, що завантажувач дескрипторів продовжить додавати завдання до внутрішньої черги, Але на них не звертають уваги. Як наслідок, ця черга заповнюється, а буфер також пізніше. Це випадок, коли робота зупиняється в очікуванні додавання нових завдань. З іншого боку, якщо є доступний запис, то обробник дескриптору починає визначати залежності нового завдання з раніше доданими.

Першими кроками є присвоєння ідентифікаційного номеру завдання новому завданню та завантаження його дескриптора завдання. Такий ідентифікатор завдання є індексом задачі в таблиці завдань, який служить для розрізнення задачі. Після того, як завдання отримає унікальний ідентифікатор, обробник дескриптору завантажить дескриптор завдання, щоб розпочати обробку його залежностей. Це робиться для виклику функції інтерфейсу

регістру, яка повертає дескриптор завдання, заданий його індексом. Таким індексом є елемент, отриманий з черги, який використовується для зв'язку цього обробника з завантажувачем дескрипторів. Таким чином, він обробляє залежності кожного операнду завдання відповідно до схеми. Оскільки діапазон можливих адрес дуже великий і він не зміг поміститися в таблиці, то не може призначити окремий запис для кожної адреси. Тому потрібен механізм зіставлення кожної адреси із записом у таблицях, для мінімізації кількості відповідностей одному й тому самому елементу. Функція приймає адресу як вхідний параметр і повертає відображений запис у таблиці. Тут розрізняють два випадки: коли обробник хоче додати ідентифікатор задачі до стартового списку існуючого елемента; коли він хоче записати новий запис чи встановити бар'єр. Для першого випадку, якщо шукана адреса знайдена в таблиці, то ця функція повертає адресу, де вона знаходиться. Таким чином, обробник дескриптору додає новий елемент до списку початкового каталогу. Якщо для цієї адреси не знайдено жодного запису, то функція повертає некоректний запис, а обробник продовжує виконання, не додаючи завдання до будь-якого списку. Коли обробник хоче ввести новий елемент або встановити бар'єр, ця функція обов'язково повинна повертати індекс. Після того, як цей індекс буде повернуто, обробник повинен перевірити, запис на заповненість. Якщо він порожній, то вводиться новий елемент, змінюючи статус запису на зайнятий. В іншому випадку це означає, що адреса вже обробляється іншим завданням і тому обробник встановлює бар'єр. Після того, як всі залежності, що стосуються кожного параметра, буде розв'язано, обробник дескриптора заповнює запис у таблиці завдань. Цей запис містить покажчик дескриптора завдання в сховищі завдань, кількість поточних залежностей і стан завдання, який дорівнює одиниці, якщо воно не має залежностей і, отже, воно готове. Якщо задача не має жодної залежності, то це означає, що вона може виконуватися, а отже, обробник дескриптору повинен додати її до готової черги. Якщо черга готовності ще не заповнена, то переміщуємо ідентифікатор нового готового завдання в нижню частину черги разом з його покажчиком дескриптору. Може бути так, що готова

черга вже заповнена завданнями, які чекають на виконання. У такому випадку обробник дескриптора зберігає покажчики ідентифікаційного номеру і дескриптор буде у тимчасовій змінній та встановлює прапорець, який вказує на те, що є один елемент, який потрібно додати. Після цих двох умов обробник дескриптору виходить з виконання. Після відповідної затримки обробник дескриптору знову викликається. Якщо він знаходить елемент готовим, але не доданим до черги, то цей обробник надає пріоритет для цього випадку та намагається додати елемент до черги. Таким чином, жодне нове завдання не обробляється до тих пір, поки цей готовий елемент не буде остаточно введений в чергу. Затримка обробника дескриптору обчислюється аналогічно затримці завантажувача дескрипторів. У зв'язку з такою затримкою, в разі, якщо будь-який блок повинен бути записаний в чергу готовності, його запис затримується і блок зупиняється. Крім того, обробник не виконується знову, поки не закінчиться затримка. На початку будь-якого виконання обробника дескриптору змінна, що містить його затримку, дорівнює нулю. Під час виконання це значення збільшується щоразу, коли потрібно прочитати або записати на будь-яку позицію сховища завдань або в таблицях залежностей. Також береться до уваги хеш-функція та механізм її розв'язання колізій. Хеш-функція та роздільна здатність колізії використовуються щоразу, коли елемент шукається в таблицях споживачів або виробників. При даній процедурі можна прочитати і записати в таблицю кілька позицій. Також враховується затримка, додана циклами, і вона залежить від їх кількості ітерацій. Останнім кроком, що впливає на затримку, є додавання готового елемента в чергу готовності, що також займає деякий час. Вагу кожної дії в загальній затримці, яку несе обробник дескриптора, можна налаштувати. Це дозволяє врахувати вплив затримки обробника на загальну поведінку, встановлюючи вищі або менші затримки.

Коли система завершить виконання завдання, то обробник несе відповідальність за обробку його залежностей. Ядра сигналізують про завершення виконання завдання, записуючи його ідентифікаційний номер у чергу фінішного буфера. Обробник фінішу постійно перевіряє цю чергу, щоб

виявити нові надходження. Як тільки новий ідентифікатор завдання виявлено, то він обробляє свої операнди і намагається додати нові завдання в чергу готовності. Після обробки всіх залежностей в таблицях споживачів і виробників, обробник видаляє запис, який призначений завданню в таблиці завдань і в сховищі завдань. Це дозволяє додавати нові завдання.

Фінішний обробник активний, коли він не зайнятий і він перевіряє чергу буферу завершення, очікуючи на завершені завдання. Якщо в черзі зустрічається новий елемент, то він негайно починає обробляти ідентифікаційний номер. Для цього обробника немає проблем з обмеженням пам'яті, оскільки готове завдання вже виділено в пам'яті, і додаткова пам'ять не потрібна. Першим кроком обробника є читання вказівника дескриптору завдання, пов'язаного з ідентифікаційний номер. Дескриптор записується в запис таблиці завдань, проіндексований цим ідентифікатором завдання. Тому для завантаження структури дескриптора задачі необхідно викликати інтерфейс регістру. Після завантаження дескриптора обробник починає обробку операндів, пов'язаних з готовим завданням. В аналогічному випадку з обробником дескрипторів, обробнику необхідно отримати записи в таблицях виробників і споживачів, пов'язані з операндом поточної адреси за допомогою функції пошуку ідентифікаційного номеру. На відміну від обробника дескрипторів, цьому обробнику потрібно перевірити лише одну таблицю для кожного операнду. Це означає, що для вхідних операндів перевіряється тільки таблиця споживачів. У той час як для вихідних операндів звертаються до таблиці виробників. Для кожного випадку опрацьовується стартовий список відповідного запису. Для кожного елементу, вилученого з цього списку, обробник викликає функцію з видаленим ідентифікатором завдання як вхідним параметром. У межах цієї функції кількість залежностей ідентифікатору, вилученого зі списку, зменшується для запису в таблиці завдань. У випадку, якщо кількість залежностей обробленого завдання дорівнює нулеві, то функція намагається додати їх до готової черги. Якщо це не вдалося, статус запису таблиці завдань змінюється, що означає, що він готовий, але ще не доданий до черги готовності.

Обробник відстежує кількість готових елементів, які не додаються до черги, а отже, це число збільшується. Коли обробник простоює, то він перевіряє стан елементів у таблиці завдань. Це робиться один раз за такт, тобто обробник буде простоювати. Для цього обробник також зберігає індекс, що містить останній позначений елемент, щоб не перевіряти ті самі індекси.

Якщо список є порожнім після вилучення відповідних елементів шляхом зсуву списку, то обробник має вилучити запис. Порожність стартового списку визначається бар'єром, який позначає кінець списку. У цьому випадку обробник викликає функцію пошуку ідентифікаційного номеру. Останнім кроком обробника є звільнення елементів, зайнятих завданням та його дескриптором у таблиці завдань та у сховищі завдань відповідно. Індекси в обох таблицях розміщуються внизу черг, які містять вільні індекси в таблицях. Таким чином, готова задача вважається такою, що більше не присутня в системі, і обробник завершує її виконання.

Робота з дескрипторами є важливою в контексті керування багатопроцесорними системами. Для її вирішення потрібно враховувати копцепцію використовуваного апаратного пристрою і застсоування дескриптору.

3.2 Метод динамічного керування багатопроцесорними системами

Значення обробника визначається врахуванням читань і записів в сховище завдань і таблиць менеджера залежностей, крім доступу до готової черги в разі її необхідності. Також подібно до випадку обробника дескриптора, функція пошуку ідентифікаційного номеру повертає кількість циклів, витрачених на її виконання. Однак в цьому обробнику в чергу готовності може бути додано кілька завдань під час виконання одного і того ж обробника. Оскільки вони обробляються послідовно, два завдання, додані до черги готовності, не повинні ставати доступними одночасно. Це пов'язано з тим, що поки перше завдання готове, інше має бути ще вирішене, а отже, мати додаткову затримку. Щоб

впоратися з цим, визначаємо внутрішню чергу, яка містить елементи, до яких потрібно додати чергу готовності та пов'язані з нею затримки. Після того, як обробник завершив виконання, наступні цикли цієї внутрішньої черги перевіряються в пошуках елементів, які необхідно додати до готової черги. Елементи цієї внутрішньої черги сортуються за часом, коли вони повинні бути додані до черги готовності. Оскільки затримка, пов'язана з кожним завданням, є часом, коли обробник закінчує обробку такого завдання, то максимальна затримка, пов'язана із завданням, є кінцевою затримкою виконання обробника.

Основна функціональність інтерфейсу регістрів досить проста, оскільки він повинен піклуватися про читання та запис, що стосуються відображених регістрів пам'яті та зберігання завдань. Це має бути система з низькою затримкою, оскільки вона повинна забезпечувати дуже швидкий зв'язок з рештою архітектури. Цей інтерфейс повинен враховувати деякі умови, які можуть завдати шкоди системі. Інтерфейс регістра, як менеджер залежностей, має бути з функцію ініціалізації, яка запускає модуль шляхом скидання регістру та встановлення вищезгаданих дозволів. Інша функція для оновлення регістру стану, який повинен викликатися з керуючої логіки щоразу, коли відбувається важлива подія. Ці події стосуються того, чи є завдання і тому він активний, а також кількість елементів у буфері та черзі готовності. Основні функції інтерфейсу регістру викликаються, коли зовнішній суб'єкт хоче прочитати будь-який з двох дозволених до читання регістрів: регістра стану та регістра готової черги. В обох випадках функція повинна повертати клас, який отримано, але тепер включає запитувані дані. Для реєстру статусів це завжди так, оскільки завжди можна прочитати статус. При спробі прочитати перший елемент готової черги регістр повинен подбати про деякі можливі ситуації. В залежності від джерела запиту цей доступ може здійснюватися в одну або дві передачі даних. Якщо елемент отримує ядро, то доступ до нього здійснюється у два кроки, оскільки доступи з ядер можуть отримати лише бітові значення. Готовий елемент черги має два бітових значення: ідентифікаційний номер завдання та покажчик дескриптора. Таким чином, для першого доступу береться перший

елемент черги і відправляється ідентифікаційний номер завдання. Покажчик дескриптору зберігається у внутрішній змінній, яка надсилається вдруге, коли виконує зчитування. Це рішення справедливе лише для випадку, коли в архітектурі присутній лише один процесор з декількома ядрами. У випадку більш ніж одного процесора, що запитують елементи черги готовності, доступи ядер можуть чергуватися, а отже, отримувати неправильні значення. При доступі готовий елемент зчитується відразу, тому що ці ядра виконують зчитування всього регістра, а не двоетапний доступ.

Однак може статися так, що черга готовності буде порожньою і завдання не може бути надіслано. Щоб фінішний буфер не міг бути заповнений, тут враховано ще одну умову. Тому, коли кількість елементів у фінішному буфері перевищує певний поріг, готові елементи також не відправляються. В іншому випадку фінішний буфер може бути заповнений у певний момент, і надходження нового готового завдання перезапише чергу. Таким чином, цей механізм не дозволяє продовжувати додавати завершені завдання, коли раніше завершені ще не були виконані. Поріг також можна змінити у файлі. Коли читання елемента в черзі не допускається, цей випадок обробляється в залежності від елемента, який виконав зчитування. Це дозволяє ядру вирішувати про продовження запитів на завдання. У випадку з робочими ядрами їх унікальною функцією є виконання завдань і, отже, їх запити потрапляють у чергу. Таким чином, запити ядер блокуються. Функція, також включена в цей файл, відповідає за перевірку існування елементів в цій черзі. У разі, якщо черга готовності не порожня, то ця функція відвідує запит і видаляє петицію та готовий елемент з відповідних черг. Така функція викликається з менеджеру залежностей на початку кожного циклу.

Ця функція виконує операції запису в відображений регістр пам'яті. У випадку регістру зчитування дозволяється записувати лише два регістри: регістр у буфері; регістр буферу завершення. Після того, як інтерфейс регістра записує дані, то клас знищується. Хоча початковий архітектура передбачав, що в буфері будуть елементи, що містять покажчик дескриптору та його розмір, поточна реалізація включає лише поле покажчика дескриптору. Таким чином, в даному

випадку немає необхідності розрізняти джерело доступу.

Код цієї функції немає особливих випадків і значення записуються відразу. У буферному регістрі відповідальність за неперевищення розміру черги покладається на ядро, яке перевіряє розмір черги перед додаванням нових значень. У випадку з фінішним буфером після певної кількості елементів у ньому неможливо прочитати елементи з готової черги, що перешкоджає виходу за межі розміру черги. Тому в обох випадках, коли приходить нове значення, а додавання нового елемента перевищує розмір черги, перший елемент черги перезаписується. Запис в будь-який з регістрів змінює кількість завдань. Якщо нове завдання записано в буфер, це означає, що нове завдання має бути оброблене, і кількість завдань збільшується. Запис у готовий буфер зменшує кількість оброблюваних завдань. Після того, як готове завдання видається обробником дескриптору, його запис в таблиці завдань і в сховищі завдань видаляється. Цей лічильник завдань може змінювати стан і тому необхідно викликати функцію стану оновлення. Коли він дорівнює нулю, то завершено обробку завдань і більше не активний пристрій, тоді як додавання нового завдання означає, що він активний, оскільки має принаймні одне елементне завдання, яке обробляється.

Функції, пов'язані зі зберіганням завдань, поділяються на дві групи. Вони використовуються логікою керування для зчитування дескрипторів, пов'язаних із завданням, яке вони обробляють. Ці функції викликається щоразу, коли зовнішня сутність хоче прочитати дескриптор завдання. Оскільки це дозволено в будь-якому випадку, то ця функція лише отримує дескриптор зі сховища завдань і повертає його. У цьому випадку функція виконує операцію запису вхідних даних за адресою. У цьому випадку ця функція повинна сигналізувати обробнику дескриптора про те, що є нові завдання, які потрібно обробити. Індекс дескриптору в сховищі завдань додається в чергу. Наявність вільного місця в цій черзі вже перевірено в завантажувачі дескрипторів. Крім того, оскільки дескриптор завдання вже завантажено, то кількість передач пам'яті зменшується. Враховуючи адресу запитуваного дескриптору завдання, то ця функція повертає

індекс цієї адреси у сховищі завдань. Сховище завдань розглядається як таблиця, що містить дескриптор завдання у вигляді записів. Ця функція викликається з керуючої логіки і повертає запитуваний дескриптор завдання, задавши його індекс як вхідний параметр.

Таблиця виробників і таблиця споживачів мають фіксовану довжину, що обмежує їх кількість записів кількома тисячами. Кожен з цих записів повинен бути придатним для виділення адреси, яка обробляється завданням. Це нестійка ситуація, оскільки вона вимагала б великої кількості даних, зайнятих таблицями. Тому, потрібен механізм, який перетворює з діапазону бітних на кілька тисяч записів. Хеш-таблиці — це структури даних, які розподіляють великий набір вхідних параметрів, відомих як ключі, між набором слотів за допомогою хеш-функції. Вимоги до хеш-функції полягають у тому, що вона повинна мати низьку вартість з точки зору швидкості, а також бути детермінованою та однорідною. Оскільки хеш-функція призначена для пошуку в таблиці, а такі трапляються досить часто, то вона повинна виконуватися швидко, і тому вона не може включати складні операції. Хеш-функції є детермінованими, оскільки для одного і того ж входу вони завжди дають один і той же результат. Це фундаментальна вимога до архітектури, оскільки операнди з однаковою адресою повинні бути зіставлені з однією адресою, щоб розпізнати залежність між ними. Нарешті, хеш-таблиця повинна розподіляти свої входи якомога рівномірніше по своїх виходах, щоб не перевантажувати деякі слоти, поступаючись у багатьох колізіях одному і тому ж запису. Незважаючи на те, що можна отримати будь-який діапазон адрес в межах бітного значення, оброблювані дані, як правило, виділяються в короткому послідовному діапазоні адрес. Таким чином, обрана хеш-функція повинна добре працювати для невеликих послідовних діапазонів. Обрана хеш-функція складається з простих зсувів, доповнень і віднімання, які швидко виконують перетворення. Вона характеризується тим, що є хеш-функцією, де кожен вхідний біт впливає на себе та вищі біти виходу. Звідси випливає, що більш високі вхідні біти не мають ніякого впливу на більш низькі вихідні біти, будучи основним ефектом, створюваним більш низькими вхідними

бітами. Це оптимальний випадок для пошуку в таблицях, де через послідовні адреси пам'яті старші вхідні біти майже не змінюються. Відповідно, на вироблений вихід в основному впливають менші зміни вхідних бітів. Хоча хеш-функція намагається уникнути колізій з одним і тим же записом таблиці, неминуче, що дві різні адреси в кінцевому підсумку зіставляються з одним і тим же записом.

Таким чином, для вирішення цієї проблеми необхідний механізм вирішення колізій. Щоб уникнути ситуації, коли дві різні адреси відображаються в одному записі таблиці, здається, що рішення є двома основними підходами: роздільний ланцюжок; відкрита адресація. Схема роздільного ланцюжка заснована на побудові списку елементів, зіставлених з одним записом. Відповідно, коли після хешування деякого вхідного сигналу він виробляє зайняте місце в таблиці. Окремий ланцюжок виділяє новий елемент і пов'язує його з результуючим записом. Якщо це повторюється, то новий елемент виділяється після останнього елемента списку. Це означало б динамічний розподіл елементів у пам'яті. Однак, оскільки таблиці були розроблені таким чином, щоб мати фіксований розмір, то це неможливо. Замість цього відкрита адресація працює з таблицями з фіксованою довжиною. Для випадку двох входів, зіставлених з одним і тим же елементом, ідея цього методу полягає в тому, щоб виділити новий елемент в наступному елементі, вільному в таблиці. Це означає, що після того, як потрібний предмет зайнятий, відкрита адреса рухається, намагаючись знайти новий предмет. Якщо досягнуто нижньої частини списку, то він продовжується від верхньої частини таблиці вниз, поки не знайде порожню позицію або перший позначений пункт. Якщо він знову досягне хешованого значення, то це означає, що вільного запису в таблиці немає. В окремому ланцюжку цей випадок не відбувся, тому що пам'ять динамічно виділяється.

Обидва механізми мають свої переваги і недоліки. Коли потрібно отримати доступ до одного конкретного елемента таблиці, то роздільний ланцюжок виконує це дуже швидко, оскільки потрібно перевірити лише список,

пов'язаний із хешованим записом. З іншого боку, у випадку відкритої адресації, якщо елемента немає в таблиці, він перевіряє кожен елемент і, таким чином, передбачає велику затримку для найгіршого сценарію. Роздільний ланцюжок тягне за собою швидкий пошук за рахунок динамічно виділеної пам'яті, коли відкрита адресація пропонує більш повільну альтернативу з фіксованим розміром таблиці. Ні динамічна пам'ять, ні великі затримки не підходять для конструкції апаратного пристрою. Замість цього можна обрати комбінацію обох схем.

Ідея об'єднаного хешування полягає в побудові списків елементів, зіставлених з одним і тим же записом, з фіксованим розміром таблиці. Елементами зв'язаних списків є різні слоти таблиці, а не динамічно розподілені структури. Тому, нове поле необхідне в елементах таблиці для того, щоб можна було зв'язати два елементи. Це поле вказує на наступний елемент списку. У разі роботи з останнім елементом списку, це поле містить маркер кінця, що вказує на кінець списку. Крім того, з'явиться поле, що вказує на те, чи є позиція порожньою і, таким чином, туди можна записати новий елемент. Іншим необхідним полем є адреса, яка наразі зіставлена з цією позицією, що дозволяє визначити, хто є власником запису.

Для відстеження вільних елементів таблиці використовується список доступних місць. Кожен слот цього списку повинен бути пов'язаний з подвійним посиланням, щоб його можна було швидко видалити з цього списку. Поля служать покажчиками-попередниками і наступниками для цього подвійно зв'язаного списку, відповідно. Перший слот таблиці завжди залишається невикористаним і служить покажчиком на перший елемент у списку доступних пробілів. Відповідно, поле вказує на перший елемент, а поле-покажчик вказує на останній елемент списку. При ініціалізації функції формується список, де першим елементом списку доступного простору є елемент внизу таблиці, а останнім елементом цього списку є верхній елемент таблиці. У початковому списку доступного простору всі елементи правильно зв'язані подвійно. Три основні операції, які потрібно виконати: вставка елемента; пошук елемента;

видалення елемента. Процедура вставки намагається знайти порожній слот, куди записати новий елемент. У цьому випадку функція повертає порожній слот, який буде записано, коли операція виконана успішно, або сигнал переповнення, коли вільних елементів більше немає. Операція вставки починається з виконання хеш-функції на вхідній адресі, яка відповідає значенням, розташованим під списками. У разі, якщо цей слот порожній, то наступний покажчик встановлюється на слот, а поточна адреса записується в його поле адреси, що має місце для адреси. Крім того, поточна позиція видаляється зі списку, що містить вільні елементи, і позначається як зайнята. Щоб вилучити елемент зі списку доступних місць, необхідно змінити поля попереднього вільного елемента та наступного вільного елемента. Таким чином, наступник попереднього вільного предмета змінюється на наступний вільний предмет. Відповідно, попередником наступного вільного предмета буде попередній вільний предмет. Відповідно, адресне поле вказує на елемент, так як є його попередником. Коли перший позначений елемент вже зайнятий, то повинні дійти до кінця його зв'язаного списку. Після того, як він буде досягнутий, то перший елемент списку доступного простору буде взято і вилучено з цього списку. Після цього цей елемент зв'язується зі списком, починаючи з початкової хешованої позиції, а наступне поле встановлюється так, щоб воно вказувало на слот. Видалення у списку доступних місць виконується так само, як і в попередньому випадку, з особливістю попереднього елемента є слот. Операція пошуку досить проста, так як необхідно лише перевірити зв'язані елементи списку, починаючи з індексу, отриманого з хеш-функції. Відповідно, після того, як вхідні дані були перетворені за допомогою хеш-функції, перевіряється отримана хешована позиція. Якщо адреса поля збігається з потрібною адресою, то програма повертає елемент, що означає, що пошук пройшов успішно. Натомість, коли елемент зайнятий іншою адресою, він продовжує перевірку елемента, що міститься в наступному полі. Аналогічно до першого випадку, коли пошук успішний, функція повертає індекс до запису, де зустрічається потрібний елемент. У випадку, якщо перший елемент порожній, або коли останній позначений елемент містить слот, то функція повертає

значення, що вказує на невдалий пошук.

Щоб видалити елемент, першим кроком є пошук цього елемента в таблиці. У разі, якщо його вже немає в списку, то алгоритм завершується. Натомість, коли є збіг, то алгоритм повинен видалити елемент із зв'язаного списку, встановивши наступне поле попереднього елемента на слот. На цьому етапі елемент видалено зі списку, але не додано до списку доступного простору. Процедура видалення елемента зі зв'язаного списку є складнішою, оскільки має на увазі зміну порядку елементів цього списку. Після того, як елемент був видалений, необхідно обробити наступні елементи. В одному списку може бути кілька списків, пов'язаних між собою. Наприклад, коли вставляється новий елемент, може статися так, що його потрібне положення вже зайнято з будь-якого іншого елемента, що утворює список. Тому цей новий елемент пов'язаний з існуючим списком. Отже, після видалення елемента зі списку, наступні елементи повинні бути оброблені, щоб перевірити, чи може список бути розділений.

Таким чином, для кожного з наступних елементів алгоритм перевіряє, чи належать вони до іншого списку з іншим початковим записом. Для наступного елемента обчислюємо хеш-функцію з урахуванням адреси, що міститься в ній. Якщо він повертає індекс вилученого елемента, то він копіюється в ділянку пам’яті, що залишилася після видалення. Ця операція виконується зміною поля адреси останнього вільного елемента на новий вільний елемент, наступного поля нового вільного елемента на останній елемент списку доступного простору, а адресного елемента нового вільного елемента на слот.

Розроблено реалізацію множини завдань АП. Його функціональність розділена на два основні блоки: інтерфейс регістру; менеджер залежностей. Інтерфейс регістру містить кілька функцій для обробки зв'язку з іншими компонентами архітектури, дозволяючи їм записувати та читати відображені в пам'яті регістри. Крім того, інтерфейс регістра також надає корисні функції залежності для доступу до сховища завдань або оновлення регістра стану. З іншого боку, контроль відповідає за процес вирішення залежностей і, отже, є компонентом, який обробляє елементи черги та різні записи таблиць. Для зв'язку

інтерфейсу регістру та менеджера залежностей може бути використано кілька черг. У зв'язку з широким спектром можливих адрес, які повинні бути виділені в таблицях залежностей, хеш-функцію реалізовано з роздільною здатністю колізій. Поточна реалізація механізму розв'язання колізій будує зв'язані списки елементів у таблиці, що дозволяє ефективно шукати таблиці, враховуючи, що є фіксований простір для таблиць. Різні параметри конфігурації можуть бути змінені в реалізації АП, щоб оцінити їх вплив на продуктивність системи, такі як затримки та розміри.

3.3 Висновки до третього розділу

Розроблено спосіб обробки завдань та використання декскриптору. Розроблено реалізацію множини завдань АП.

Розроблено метод динамічного керування багатопроцесорними системами. Різні параметри конфігурації можуть бути змінені в реалізації АП, щоб оцінити їх вплив на продуктивність системи, такі як затримки та розміри.

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТ АПАРАТНОГО РІШЕННЯ

4.1 Реалізація апаратного пристрою

Для більших розмірів завдань система з АП швидше наближається до максимально досяжної масштабованості. Це пов'язано з кращою продуктивністю для дрібнозернистих завдань, що призводить до більш раннього і швидкого зростання в бік теоретичного значення масштабованості. Коли для виконання завдань доступно менше ядер, АП також досягає більш масштабованого підходу. При роботі з ядрами АП забезпечує масштабованість, яка перевищує результат для завдання, де відбувається лише частина передачі прямого доступу до пам'яті. Для меншої кількості ядер всі вони досягають максимальної масштабованості в першому тесті. У разі розміру завдання при використанні ядер масштабованість зростає, залишаючись незмінною для завдань розміром більше тестового. ПП погано працює для невеликих розмірів завдань навіть при використанні лише декількох ядер. Відповідно, АП досягає поліпшення масштабованості по відношенню до ПП при використанні більшої кількості ядер. Більш швидке зростання до максимальної масштабованості зберігається і коли доступно менше ядер. Тому, підвищення масштабованості особливо важливе для невеликих завдань, де додаткові витрати, введені з системи виконання ПП, обмежують продуктивність системи.

Тест продуктивності характеризується спрощеною схемою залежності. Його шаблон залежностей встановлює лише залежності рядків між завданнями, і тому різні рядки можуть виконуватися одночасно. Таким чином, ефекти нарощування шаблону усуваються, і максимальна масштабованість досяжна при використанні всіх ядер. Відповідно, в цьому випадку дозвіл залежностей менш вимогливий і обробники повинні виконувати цю операцію швидше. Додавання завдання було вилучено з трасування виконання, а затримки, пов'язані з завантажувачем дескрипторів і двома обробниками, можуть бути виміряні всередині симулятора.

Для цієї моделі залежностей подача завдання виконується швидше, ніж в

тест продуктивності. Причина такої різниці - в підготовці дескриптору завдання, так як тепер в нього доводиться писати на одну залежність менше. У буфері різниці немає, так як обидва записують покажчик на дескриптор завдання, тому розмір завантажувачу даних залишається незмінним. У завантажувачі дескрипторів немає ніякої різниці, оскільки він виконує одні й ті ж операції для запиту дескриптора завдання, незалежно від інформації, що міститься в ньому. Однак існує різниця в затримках обробника дескриптора та готового обробника. Це, безсумнівно, пов'язано з меншою кількістю залежностей, що зустрічаються в дескрипторі завдання. Оскільки потрібно обробляти меншу кількість операндів, цим обробникам потрібно менше часу для обробки кожного завдання. Як і в попередньому випадку, вони обидва можуть демонструвати дуже схожі затримки; Хоча для цього випадку обробник працює трохи повільніше, ніж обробник дескрипторів. У цьому випадку за рахунок меншої кількості операцій затримки обох обробників менші. Однак фінішному обробнику все одно потрібно виконати видалення готових завдань. Незважаючи на те, що видалень виконується менше, схоже, що це впливає меншою мірою, ніж процес вирішення залежностей. Тим не менш, додавання завдання є вузьким місцем життєвого циклу завдання, навіть незважаючи на те, що його затримка була зменшена.

Масштабованість тесту продуктивності залежнить від обсягу завдання і кількості доступних робочих ядер. Відгук системи дуже схожий на тест продуктивності для невеликих розмірів завдань, що посилює ідею додавання завдання як основного вузького місця. До тих пір, поки завдання не перевищать певний час виконання, коли тільки передача прямого доступу до пам’яті робиться, масштабованість обох тестів продуктивності буде дуже схожа. Для завдань більшого розміру масштабованість цього тесту досягає вищих значень. Максимальна масштабованість для цього випадку досягається при розмірах завдань більше середнього значення. Це свідчить про те, що як тільки додавання завдань перестає бути вузьким місцем, масштабованість, яку пропонує АП, швидко зростає разом із розміром завдання. Це може статися завдяки кращому узгодженню завдань, що виконують завдання ядрами. Для завдань певного

більше середнього розміру виконувані завдання є тільки транферами прямого доступу до пам'яті, і довжина цих передач залежить від кількості залежностей завдання. Це пов'язано з тим, що блоки матриці не можуть залежати від будь-якого іншого блоку. Таким чином, вони запитують лише один блок, замість двох блоків, які потрібні іншим блокам. З іншого боку, для завдань вони виконують лише спрощену версію передачі прямого доступу до пам'яті. Ця спрощена версія однакова для всіх завдань. Як наслідок, кожному ядру потрібен однаковий час для виконання завдання, тоді як для завдань менших є деякі відмінності.

Для цього першого випадку АП масштабується краще, ніж модель програмування ПП. Коли розмір завдання зростає, то відносне покращення АП порівняно з ПП зменшується, і вони обидва наближаються до межі масштабованості.

Тест продуктивності не встановлює залежностей між завданнями, і таким чином роблить застосунок повністю паралельним, оскільки кожен блок може оброблятися паралельно з будь-яким іншим блоком. Таким чином, як і в попередньому випадку, ефекту нарощування не буде виявлено, а теоретична максимальна масштабованість, середні затримки і пропускна здатність різних ступенів будуть розбалансовані. Зменшено час, витрачений на процедуру додавання завдання. Тепер кожен дескриптор задачі має тільки один операнд, так як блоки незалежні один від одного і, таким чином, в дескриптор завантажується тільки адреси оброблюваного блоку. Час витрачання на запис в буфер не має варіацій, так як записується тільки покажчик дескриптору задачі. Затримка завантажувача дескрипторів залишається постійною на завдання. Для цього тест продуктивності може забезпечити затримку обробників менша через відсутність залежностей. Кожне завдання проходить через обробник дескрипторів, не записуючись до жодного списку, але завжди додаючи його до черги готовності. Кінцевий обробник повинен обробляти кожне готове завдання і видаляти його записи з таблиць споживачів і виробників. Оскільки залежностей не буде знайдено, жодних завдань не буде додано, та не вилучено з будь-якого списку початкового етапу, та не додано до черги готовності. Результати тесту

продуктивності подано в табл. 4.1.

Таблиця 4.1 – Тест продуктивності

Етап	Пропускна здатність, тактові цикли	Пропускна здатність, час
Завантаження дескрипторів	45 циклів/завдання	678 завдань/мкс
Оброблення дескрипторів	125 циклів/завдання	698 завдань/мкс
Оброблення загальних завдань	175 циклів/завдання	674 завдання/мкс

Однак, все одно трапляється, що два елементи зіставляються з одним і тим же записом таблиці, і таким чином необхідно виконати процес видалення. У цьому конкретному випадку різниця між затримками обробника дескриптора та обробника більша, ніж у попередньому випадку. Це пов'язано з тим, що на видалення не впливає роздільна здатність залежностей, а обробник дескриптора лише записує інформацію, пов'язану із завданням, і додає її до готової черги. Відсутність залежностей між завданнями впливає на роздільну здатність залежностей як обробника дескриптора, так і обробника. Однак на процес видалення в обробнику не впливають залежності між завданнями, оскільки він залежить лише від результатів від хеш-функції. Це відбувається при використанні ядер і означає дуже близьке наближення до теоретичної межі. Таким чином, на продуктивність системи не сильно впливає збільшення завдання після такого розміру. Аналогічно у випадку з тестом продуктивності для ядер, який показує несподівану поведінку, коли виконується лише частина передач прямого доступу до пам’яті. У цьому конкретному випадку він отримує кращу масштабованість, ніж обробляються всі передачі прямого доступу до пам’яті. Оскільки тепер кожне завдання виконує однакову кількість звернень прямого доступу до пам’яті, то АП масштабується краще, ніж ПП для

дрібнозернистих завдань. При роботі з невеликим розміром завдання апаратна підтримка здатна забезпечити масштабованість. Це означає, що АП працює краще, ніж ПП з точки зору масштабованості для цього розміру завдання. Коли завдання зростають, то різниця між двома системами зменшується, оскільки додаткові витрати ПП мають менший вплив на продуктивність системи. У цьому випадку масштабованість, отримана АП, вже дуже близька до максимально досяжного значення, і тому вона не може сильно зростати. Розрив у продуктивності продовжує зменшуватися, коли завдання більші, оскільки додаткові витрати ПП мають менший вплив. Однак АП може отримати добру масштабованість набагато швидше, ніж модель програмування ПП. Крім того, АП досягає вищої максимальної масштабованості, оскільки його ліміт більший.

Розглянемо масштабованість для змінного розміру завдання. У трьох попередніх тестах оцінювані розміри завдань були встановлені на фіксовані значення. Однак реальний алгоритм декодування показує різні розміри завдань, які залежать від кількох факторів, таких як тип блоку або операції, що виконуються з даними. Варіація розміру завдання декодера була виміряна в реальному процесорі, отримавши розмір завдання. Середнє значення цього розподілу було виміряно, тоді як його дисперсія була виміряна приблизно. Для того, щоб змоделювати цю поведінку в реалізації симулятора, в рамках ядра кожен розмір завдання був обчислений за допомогою функції розподілу, яка повертає затримку своєї змінної. Ця реалізація тесту продуктивностіу була виконана з використанням генератора випадкових чисел. Однак генератор випадкових чисел має рівномірний розподіл. Такий тип розподілу приписує однакову ймовірність кожному можливому числу. Використовуючи цей закон розподілу був складений набір рівномірно розподілених випадкових чисел. Сума їх задається у вигляді обсягу завдання, який визначається при кожному виконанні завдання. У кожному з ядер було використано різне завдання, щоб мати різну картину варіації розміру в кожному з ядер.

Було використано тест продуктивності, оскільки він має найбільш схожу схему залежності з алгоритмом декодування, з якого було отримано еталонне

середнє значення розміру завдання та дисперсії. Зміна розмірів може вплинути на доступну паралельність під час виконання програми. Оскільки це обмеження паралелізму більш помітно, коли кількість ядер більша, для цього експерименту використано максимальну кількість доступних ядер.

Обчислені розміри завдань в табл. 4.1 відповідають лише тим випадкам, коли завдання дійсно виконується. Тестування виконуються лише при передачі прямого доступу до пам’яті та не дадуть жодних значущих результатів, оскільки в цьому часі не може бути жодних змін, і тому результати будуть дуже схожі на ті, що отримані у випадку фіксованого розміру завдання. Результуючі масштабованості змінного розміру задачі та фіксованого розміру завдання показують, що масштабованість, отримана у випадку змінного розміру завдання, трохи нижче, ніж у випадку з фіксованим значенням, а різниця між фіксованим і змінним випадком зростає разом з розміром завдання. Причиною такої поведінки є варіація розміру завдання, що може заважати виконанню завдань. Якщо довжина завдання фіксована, то відбувається ідеальне вирівнювання виконання завдань, тоді як для змінного розміру цього може не відбуватися. Через різницю в розмірі завдання може зайняти більше часу, ніж його середнє значення. Коли це відбувається, завдання, які залежать від цього завдання, не можуть виконуватися, поки поточне завдання не завершить своє виконання.

Таким чином, така ситуація обмежує наявний паралелізм і, отже, масштабованість системи. У випадку, коли середній розмір завдання збільшується, збільшується і його дисперсія, і тому для обробки завдання може знадобитися ще більший час. Відповідно, більші значення ще більше уповільнюють роботу системи, оскільки завдання, які залежать від поточних виконуваних, повинні чекати довше, що призводить до меншої масштабованості.

4.2 Підвищення продуктивності за рахунок додавання ядер до виконання завдань

Передбачалося, що для виконання завдань доступні тільки ядра, а масштабованість визначалася тільки з урахуванням апаратних пристроїв. У той час, як вони виконують завдання, пристрій несе відповідальність за передачу цих завдань. Однак, коли таблиця завдань заповнена, він зупиняється, поки завдання не буде завершено, звільняючи запис у таблиці. За той час, поки пристрій чекає на нове вільне місце в таблиці завдань, він може виконувати готові завдання. Тому, коли він знаходить буфер заповненим, то намагається отримати завдання з черги готовності, щоб виконати його. Це підвищує продуктивність всієї системи, так як економить час при виконанні доступних завдань, і в той же час звільняє запис таблиці завдань. Після того, як він виконає завдання, то продовжує додавати завдання до множини завдань. Більше того, після того, як воно завершено, додавши завдання, він зупиняється, поки всі завдання не будуть оброблені. Якщо ядра також приєднаються до виконання завдань після того, як вони додали всі завдання, то для цього випадку розмір таблиці завдань є важливим параметром, оскільки більший розмір таблиці завдань означає, що ядро починає виконувати завдання раніше.

У цьому випадку можемо виміряти покращення масштабованості, включивши завдання додавання ядра. Перш за все, архітектура ядра відрізняється від архітектури пристрою, оскільки вони оптимізовані для різних цілей. Крім того, це має стати ядром в архітектурі, оскільки він повинен запускати операційну систему та керувати всім. Крім того, ядра не виконують завдання в тих самих умовах. Він виконує завдання лише тоді, коли таблиця завдань заповнена, тоді як ядра виконують завдання, поки є готові завдання. Коли він намагається отримати завдання, то може статися так, що, незважаючи на те, що таблиця завдань заповнена, немає готових завдань для виконання. У цьому випадку він попереджає ядра про цей стан, який зупиняється до тих пір, поки не зможе додати нове завдання. Для вивчення ефекту цієї модифікації був

запущений тест з різними розмірами завдань і змінна кількість ядер. Результати подані в табл. 4.2.

Таблиця 4.2 – Результати тесту продуктивності з модифікаціями

Кількість ядер	Час, мкс
1	3.55145
2	2.74712
4	2.34323
8	2.13633
16	1.04189

Прискорення, одержуване додаванням ядер для виконання завдань, стосовно випадку, коли завдання виконують тільки тест були виміряні лише для випадків з обчисленнями. Якщо обчислення не проводиться, і оскільки пристрій не обробляв будь-яку передачу прямого доступу до пам’яті, то ядра нічого не роблять, бо виконується прямий доступ до пам’яті. Результати показують, що прискорення, отримане при додаванні завдань, збільшується зі збільшенням розміру завдання і зменшується зі збільшенням кількості ядер. Час, витрачений на кожне завдання доповнення, не залежить від кількості ядер. У зв'язку з тим, що ядра виконують завдання, коли вони не можуть обробляти інші завдання.

Таким чином, загальний час, витрачений на додавання завдання, є постійним і не залежить від кількості ядер. Тому, додавання завдань можна розглядати як постійні додаткові витрати. Коли використовується менша кількість ядер або більші розміри завдань, то загальний час виконання системи збільшується, тоді як додаткові витрати залишаються постійними і, отже, менш важливими. Більший загальний час виконання зменшує співвідношення між додатковими витратами та часом виконання, а також дозволяє ядру виконувати більшу кількість завдань. Тому, вплив виконання завдань вищий. Прискорення досягається, коли розмір завдання має найбільше значення і тільки він виконує

завдання. При збільшенні обсягу завдання це прискорення істотно не зросте, так як воно залежить тільки від цього значення. У цьому крайньому випадку додаткові витрати, пов'язані з додаванням завдання, дуже малі в порівнянні з виконанням завдання. Використовуючи їх, продуктивність виглядає так, ніби вони обробляють завдання. Таким чином, це прискорення означає, що ядра як мінімум вдвічі швидше виконують завдання, ніж одне ядро.

Вплив затримки ТПУ на результуючу масштабованість було встановлено в конфігурації блоку. Поки що вони були встановлені відповідно до операції, яку вони представляють, враховуючи, що використовується апаратний блок. Однак очікується, що вплив латентності на поведінку системи не буде вирішальним. Пропускна здатність множини завдань вища, ніж для завдань додавання. У найскладнішій схемі залежностей, тест продуктивності з найнижчою пропускною здатністю в множинаі завдань є обробник дескрипторів. Це найгірший випадок з огляду на цей тест продуктивності, оскільки чим складніша схема залежностей, тим більша кількість операцій, що виконуються в обробниках. З іншого боку, додавання завдань не помічає значущих змін, оскільки різні патерни залежностей впливають лише на кількість параметрів, записаних у дескрипторі завдання. У будь-якому випадку, цей етап конвеєру все одно швидший, ніж який додає завдання. Обробник також має дуже схожу середню затримку з тією, що характеризує обробник дескриптора. Якщо патерн залежностей простіший, як для тесту продуктивності, то можна побачити, що ця різниця ще більша. Отже, еталон був обраний для вивчення впливу затримок на масштабованість системи. Розмір завдання був підібраний таким чином, щоб завдання додавання не обмежували масштабованість, і, таким чином, можемо досліджувати, коли він дійсно стає вузьким місцем системи. Відповідно, розмір завдання для цього експерименту становить так, що масштабованість тесту починає бути складною для початкової затримки.

Таким чином, перша виміряна точка представляє початкову затримку, налаштовану для попередніх тестів, а наступні точки представляють число, на яке було помножено затримку. Для перших протестованих затримок на

масштабованість АП не впливають різні затримки, що споживаються в обробниках. Однак, коли затримка досягає 50-кратного початкового значення, продуктивність системи погіршується. Для більшої затримки масштабованість, яку пропонує АП, продовжує зменшуватись. Причина такої поведінки полягає в тому, що обробники не стають вузьким місцем до тих пір, поки їх пропускна здатність не зможе задовольнити ядер, що виконують завдання. Враховуючи розмір завдання, як обробник дескриптора, так і кінцевий обробник можуть обробляти завдання з більшою швидкістю, ніж вимагається від них. Насправді, коли затримка перевищує початкове значення, це перший випадок, коли обробник дескриптора є вузьким місцем замість завдань, що додають ядра. Однак цей обробник може обробляти, що є достатнім з огляду на поточний розмір завдання. Таким чином, під час виконання одного завдання обробник дескриптора міг видати багато завдань, виконуючи вимоги. У наступному випадку, коли затримки більші, саме тут можемо помітити невелике погіршення масштабованості. Для цієї більшої затримки обробник дескриптора може видавати багато завдань під час виконання, що вже дуже близько до необхідної швидкості при використанні ядер. Для більших затримок пропускної здатності, запропонованої обробником дескрипторів, недостатньо, і, таким чином, продуктивність системи падає. Цей стан продовжує погіршуватися до тих пір, поки затримки всередині не будуть більшими, поки масштабованість одиниці не стане максимально досяжним значенням. Однак, обрані затримки були встановлені враховуючи, що пристрій отримує доступ лише до своєї внутрішньої пам'яті. Тому, хоча в оцінці затримки могла бути допущена деяка помилка, це не вплине на поведінку системи.

Крім того, завдяки цим результатам було продемонстровано, що збільшення розміру пристрою не матиме сильного впливу на очікувану масштабованість. Якщо блок більший, то може знадобитися більше часу, щоб отримати елементи з внутрішнього елемента, а також витратити додатковий час на пошук по пов'язаному списку, оскільки вони можуть містити більше пов'язаних елементів.

Розглянемо вплив розміру таблиці завдань на результуючу масштабованість. Наразі було розглянуто вплив різних розмірів завдань і затримок на масштабованість, яку пропонує АП. Ще однією важливою особливістю реалізації є розмір таблиць, які використовуються для обробки залежностей. Кожен запис цих таблиць містить цінну інформацію, яка використовується обробником дескриптора та обробником. Тим не менш, зберігання всієї цієї інформації пов'язане з деякими недоліками, оскільки зберігання інформації вимагає пам'яті, а отже, займає простір. З іншого боку, необхідно враховувати, що мається на увазі апаратний блок, розміщений всередині багатоядерної платформи, і, таким чином, він не повинен споживати великі обсяги внутрішньої пам'яті. Тому, одна з цілей проектування полягає в тому, щоб мати можливість побудувати цей блок, використовуючи найменшу можливу площу.

Розмір таблиці завдань є важливим параметром реалізації. Розмір таблиці виробників і таблиці споживачів залежать від розміру таблиці завдань. Крім того, сховище завдань можна розглядати як пам'ять, де зберігаються дескриптори, що належать до таблиці завдань. Отже, розмір сховища завдань також залежить від кількості елементів, які можуть поміститися в таблицю завдань. Однак кількість елементів, які можуть одночасно зберігатися в таблиці завдань, також впливають на масштабованість. Розглянемо випадок тесту, де кожен блок залежить від блоку. Для того щоб мати максимальну масштабованість з використанням однакової кількості ядер, необхідно мати всі завдання в таблиці завдань. Таким чином, можемо визначити номінальний розмір як випадок, коли максимально доступний паралелізм завжди може бути використаний. У випадку, якщо зберігаються лише перші рядки, то це обмежує максимальну масштабованість, оскільки він не був би оброблений наступним завданням без залежностей на цьому етапі. Таким чином, існує компроміс між споживаним простором та максимально досяжною масштабованістю. Змодельовані різні розміри таблиці завдань всередині пристрою для вивчення їх продуктивності. Ці розміри варіюються від розміру таблиці завдань. Матриця,

яка використовується в тесті, є розділеною на блоки. Таким чином, в першому випадку гарантується, що перші рядки блоку можуть поміститися в пристрої. Цей випадок використовується для порівняння з іншими тестами, оскільки неможливо отримати кращу продуктивність при іншому розмірі таблиці завдань. У разі, якщо в пристрої виділено лише елементи, то це означає, що на початку моделювання додається лише половина першого рядка, і, таким чином, одночасно може бути виконано лише одне завдання. Продуктивність кожного розміру таблиці завдань вимірюється в порівнянні з продуктивністю, отриманою при використанні таблиці завдань з записами. Таким чином, представлені результати означають відсоток максимально досяжної масштабованості. Продуктивність системи знижується при менших розмірах завдань. При використанні записів в таблиці завдань отримана продуктивність точно така ж, як і у першомк випадку записами. Для цього конкретного випадку існує ідеальний збіг на початку виконання, оскільки він може містити повні рядки матриці, а кількість робочих ядер також дорівнює їх кількості. Менша кількість записів у таблиці завдань означає, що деякі завдання, які могли б виконуватися паралельно, не виконуються, оскільки вони ще не оброблені. Тим не менш, ці результати показують, що продуктивність не знижується лінійно зі зменшенням розміру таблиці завдань. Наприклад, коли ядра виконують завдання, то розмір таблиці завдань не дає відносної продуктивності у випадку записів. Оскільки в таблиці завдань можуть зберігатися тільки завдань, і таким чином бути обробленими, а це еквівалентно повним рядкам матриці. Однак, коли завдання закінчує своє виконання, то воно видаляється з таблиці завдань, а його дескриптор завдання зі сховища завдань. Відповідно, з огляду на характер залежностей тесту продуктивності, при виконанні завдання в рядку розміщені вже виконані, а отже, видалені з таблиці. В тест продуктивності розміщений блок зліва від оброблюваного блоку і він повинен закінчити своє виконання. Таким чином, всі блоки зліва вже оброблені, залишаючи трохи місця для нових завдань. Ці нові завдання виділяються в записах, залишених готовими завданнями. Наприклад, розглядаючи початок виконання матриці, при певному розмірі

таблиці завдань, перший елемент рядка додається в таблицю завдань, коли перший блок матриці оброблений.

Завдяки ефекту нарощування, що зустрічається в тесті продуктивності, максимальна масштабованість потоків зростає з додаванням завдання до множини завдань. Він починає додавати перше завдання, і таким чином може бути виконане лише це завдання. При додаванні другого ряду виконується перший елемент другого ряду і так далі. Таким чином, якщо жодне завдання не було виконано, поки таблиця завдань не була заповнена, то таблиця завдань містила б перші рядки. Тим не менш, це не так, оскільки завдяки конвеєрному процесору завдання обробляються та виконуються одночасно з додаванням завдань. Таким чином, на цьому етапі показано, що крім перших рядків матриці, у зв'язку зі зняттям готового завдання, процесор починає включати також елементи з наступних рядків.

Таким чином, можна вважати, що таблиця завдань із записами страждає від модифікованого ефекту нарощування. Такий ефект нарощування триває довше за рахунок згаданого нижчого темпу зростання для більшої кількості завдань, що виконуються паралельно в даному випадку. Тим не менш, наявність меншої таблиці завдань змінює перший випадок, коли зустрічається нове обмеження масштабованості. На цьому етапі ефект нарощування більше не є проблемою, але розмір таблиці завдань набуває значення. Цей перший випадок залежить від розміру таблиці завдань і кількості робочих ядер, доступних для виконання завдань. Коли розмір таблиці завдань занадто малий для кількості використовуваних ядер, трапляється що він дорівнює масштабованості, Кількість ядер ніколи не змінюється.

4.3 Висновки до четвертого розділу

Результати, отримані для трьох штучних тестів і різних розмірів завдань, показали, що АП отримує цілеспрямоване покращення масштабованості. Продуктивність АП для дрібнозернистих завдань показала більш високу

масштабованість завдяки прискоренню роздільної здатності залежностей і синхронізації з робочими ядрами. Таким чином, три вузькі місця, виявлені в ПП, були усунені. Завдання з додавання апаратного пристрою стали поточним вузьким місцем системи. Зменшуючи кількість завдань у процесорі, обмежуємо максимальну кількість завдань, які можуть виконуватися одночасно, і, отже, обмежуємо максимальну доступну масштабованість. Однак, працюючи з таблицями, зменшеними до половини від їх номінального розміру, масштабованість все одно залишається майже максимальною по відношенню до її максимального значення.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень удосконалено метод динамічного керування багатоядерними комп'ютерними системами та отримано такі результати.

1. Проаналізовано відомі методи керування багатоядерними комп'ютерними системами.

2. Розроблений удосконалений метод динамічного керування багатопроцесорними систмами базується на апаратній підтримці. Для того, щоб додати нове обладнання, новий модуль побудований відповідно до вимог.

3. Функціональність АП була розділена на інтерфейс регістра та менеджер витрат, який містить логіку керування та таблиці, що беруть участь у вирішенні проблем. Інтерфейс регістра визначено як шлюз модуля. На нього покладено відповідальність за синхронізацію множини завдань з ядрами архітектури. Окремо реалізований менеджер залежностей, що містить керуючу логіку, яка обробляє вхідні завдання і готові завдання. Для таблиць конструкції були встановлені фіксовані розміри, а хеш-функція з алгоритмом дозволу колізій була запрограмована на видачу відображення широкого діапазону вхідних даних на обмежену кількість записів у таблицях.

4. Здійснено есперементальні дослідження згідно розроблених рішень. Результати показали, що АП зменшив вузькі місця, виявлені в моделі програмування ПП. Йому вдалося задовольнити вимоги до апаратного забезпечення, прискоривши вирішення залежностей і синхронізацію з ядрами, присутніми в платформі. Результати проілюстрували підвищення масштабованості системи в порівнянні з тими, що були отримані за допомогою моделі програмування ПП.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Sjalander M., Terechko A., Duranton M. A Look-Ahead Task Management Unit for Embedded Multi-Core Architectures, *Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, 2008.

2. Vachharajani M., Vachharajani N., Penry D. A., Blome J. A., August D. I., Microarchitectural Exploration with Liberty. *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*. 2002.

3. Vitter J. S. Implementations for Coalesced Hashing. *Communications of the ACM 25*. 1982. V. 12. P. 911–926.

4. Etsion R. M., Ramirez A., Labarta J. Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline, *Advanced Computer Archi- tecture and Compilation for Embedded Systems, 2009.*

5. Ghiath A.-K., Terechko A. S. A Hardware Task Scheduler for Embedded Video Processing, *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, 2009.*

6. Binkert N. L., Dreslinski R. G., Hsu L. R., Lim K. T., Saidi A. G., Reinhardt S.K. The M5 Simulator: Modeling Networked Systems, *Micro, IEEE 26.* 2006. No. 4. P. 52–60.

7. Castrillon J., Zhang D., Kempf T., Vanthournout B., Leupers R., Ascheid G. Task Management in MPSoCs: an ASIP Approach, *Proceedings of the 2009 International Conference on Computer-Aided Design, 2009.*

8. Rodenas D., Martorell X., Ramirez A., Cabarcas F., Rico A., Ayguad´e E. CellSim: A Validated Modular Heterogeneous Multiprocessor Simulator, *XVIII Jornadas de Paralelismo de Zaragoza, 2007.*

9. Flachs B., Asano S., Dhong S. H., Hofstee H. P., Gervais G., Kim Rю, Le T., Liu Pю, Leenstra J., Liberty J., Michael B., Oh H.-J., Mueller S. M., Takahashi O., Hatakeyama A., Watanabe Y., Yano N., Brokenshire D. A., Peyravian M., To V., Iwata E. The Microarchitecture of the Synergistic Processor for a Cell Processor, Solid-State

Circuits, *IEEE Journal of 41*. 2006. No. 1. P. 63 – 70.

10. Gonnet G. H. Expected Length of the Longest Probe Sequence in Hash Code Searching, *Journal of the ACM 28*. 1981. No. 2. P. 289–304.

11. Hoogerbrugge J., Terechko A. A Multithreaded Multicore System for Embedded Media Processing, Transactions on High-Performance. *Embedded Architectures and Compilers*. 2008.

12. Kahle J. A., Day M. N., Hofstee H. P., Johns C. R., Maeurer T. R., Shippy D. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development 49*. 2005. No. 4.5. P. 589–604.

13. Kumar S., Hughes C. J., Nguyen A. Carbon: Architec- tural Support for Fine-Grained Parallelism On Chip Multiprocessors. *Proceedings of the 34th Annual International Symposium on Computer Architecture*. 2007.

14. Long G., Fan D., Zhang J. Architectural Support for Cilk Computations on Many-core Architectures. *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2009.

15. Magnusson P. S., Christensson M., Eskilson J., Forsgren D., Hallberg G., Hogberg J., Larsson F., Moestedt A., Werner B. Simics: A Full System Simulation Platform. *Computer 35*. 2002. No. 2. P. 50–58.

16. Meenderinck C. H. Improving the Scalability of Multicore Systems. *With a Focus on Video Decoding, Ph.D. thesis*. July 2010.

17. Meenderinck C. H., Juurlink B. H. A Case for Hardware Task Management Support for the StarSS Programming Model. *Proceedings Conference on Digital System Design Architectures, Methods and Tools*. 2010.

18. Ou S.-H., Lin T.-J., Deng X. S., Zhuo Z. H., Liu C. W. Multithreaded Coprocessor Interface for Multi-core Multimedia SoC. *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*. 2008.

19. Papoulis A. Probability, Random Variables and Stochastic Processes. *3rd ed., McGraw-Hill Companies.* February 1991.

20. Perez D. G., Mouchard G., Temam O. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. *Proceedings of the 37th mannual*

IEEE/ACM International Symposium on Microarchitecture. 2004.

21. Planas J., Badia R. M., Ayguad´e E., Labarta J. Hierarchical Task-Based Programming With StarSs. *International Journal of High Performance Computing Applications 23*. 2009. No. 3. P. 284–299.

22. Sanchez D., Yoo R. M., Kozyrakis C. Flexible Architectural Support for Fine-Grain Scheduling. *Proceedings of the Fifteenth Edition of ASP- LOS on Architectural Support for Programming Languages and Operating Systems*. 2010.

23. Choppakatla N., Sivalenka M., Boda R. Task ordering in multiprocessor embedded system using a novel hybrid optimization model. *Multimedia Tools and Applications*. 2024. P. 1-25. 10.1007/s11042-024-19083-1.

24. Müller L., Schumacher N., Steffen L., Haubelt C. Generative Design of the Architecture Platform in Multiprocessor System Design. *Electronics. 13*. 2024. P. 1404. 10.3390/electronics13071404.

25. Liang J., Xiao M., Lin Z. Li C. t/k-Diagnosability of Regular Networks under the Comparison Model. *Sensors. 24*. 2024. P. 2303. 10.3390/s24072303.

26. Nicheporuk A., Savenko O., Nicheporuk A., Nicheporuk Y. An android malware detection method based on CNN mixed-data model. *CEUR Workshop Proceedings Kharkiv, Ukraine*. 2020. V. 2732. P. 198–213.

27. Savenko B., Lysenko S., Bobrovnikova K., Savenko O., Markowsky G. Detection DNS Tunneling Botnets. *In 2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. 2021. P. 64-69.

28. Vavouliotis G, Alvarez L, Karakostas V, Nikas K, Koziris N, Jiménez D and Casas M. Exploiting page table locality for agile TLB prefetching. *Proceedings of the 48th Annual International Symposium on Computer Architecture.* 2021. P. 85-98. https://doi.org/10.1109/ISCA52012.2021.00016

29. Guruswami V., Smith A. Optimal Rate Code Constructions for Computationally Simple Channels. *Journal of the ACM. 63:4*. 2016. P. 1-37. Online publication date: 8-Nov-2016. https://doi.org/10.1145/2936015

30. Miller C., Shi Y. Robust Protocols for Securely Expanding Randomness and

Distributing Keys Using Untrusted Quantum Devices. *Journal of the ACM. 63:4*. 2016. P. 1-63. Online publication date: 8-Nov-2016. https://doi.org/10.1145/2885493

31. Chan S., Lee J., Raghavendra P., Steurer D. Approximate Constraint Satisfaction Requires Large LP Relaxations. *Journal of the ACM. 63:4*. 2016. P. 1-22. Online publication date: 8-Nov-2016. https://doi.org/10.1145/2811255

32. El H. I., Merritt A., Zellweger G., Milojicic D., Achermann R., Faraboschi P., Hwu W., Roscoe T., Schwan K. SpaceJMP. *ACM SIGARCH Computer Architecture News. 44:2*. 2016. P. 353-368. Online publication date: 29-Jul-2016. https://doi.org/10.1145/2980024.2872366

33. Bondhugula U., Acharya A., Cohen A. The Pluto+ Algorithm. *ACM Transactions on Programming Languages and Systems. 38:3*. 2016. P. 1-32. Online publication date: 2-May-2016. https://doi.org/10.1145/2896389

34. Norris B., Demsky B. A Practical Approach for Model Checking C/C++11 Code. *ACM Transactions on Programming Languages and Systems. 38:3.* 2016. P. 1-51. Online publication date: 2-May-2016.https://doi.org/10.1145/2806886

35. Agarwal M., Jailia M. Effect of TLB on system performance. *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*. P. 1-4. https://doi.org/10.1145/2905055.2905280

36. Agarwal M., Jailia M. Inconsistency in translation lookaside buffer. *2016 International Conference on ICT in Business Industry & Government (ICTBIG)*. 2016. P. 1-5. 10.1109/ICTBIG.2016.7892705.978-1-5090-5515-9. http://ieeexplore.ieee.org/document/7892705/

37. Karakostas V., Unsal O., Nemirovsky M., Cristal A., Swift M. Performance analysis of the memory management unit under scale-out workloads 2014 IEEE International Symposium on Workload Characterization (IISWC). 2014. P. 1-12. 10.1109/IISWC.2014.6983034.978-1-4799-6454-3. http://ieeexplore.ieee.org/document/6983034/

38. Fang Z., Zhao L., Jiang X., Lu S., Iyer R., Li T., Lee S. Reducing cache and TLB power by exploiting memory region and privilege level semantics. *Journal of*

Systems Architecture. 10.1016/j.sysarc.2013.04.002. 59:6. 2013. P. 279-295. Online publication date: 1-Jun-2013. https://linkinghub.elsevier.com/retrieve/pii/S1383762113000519

39. Dhawan U., Kwon A., Kadric E., Hritcu C., Pierce B., Smith J., DeHon A., Malecha G., Morrisett G., Knight T., Sutherland A., Hawkins T., Zyxnfryx A., Wittenberg D., Trei P., Ray S., Sullivan G. Hardware Support for Safety Interlocks and Introspection. *Proceedings of the 2012 IEEE Sixth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. 2012. P. 1-8. https://doi.org/10.1109/SASOW.2012.11

40. Morari A., Gioiosa R., Wisniewski R., Rosenburg B., Inglett T., Valero M. Evaluating the Impact of TLB Misses on Future HPC Systems. *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium.* 2012. P. 1010-1021. https://doi.org/10.1109/IPDPS.2012.94

41. Nellans D., Sudan K., Brunvand E. Balasubramonian R. Improving server performance on multi-cores via selective off-loading of OS functionality. *Proceedings of the 2010 international conference on Computer Architecture*. 2010. P. 275-292. https://doi.org/10.1007/978-3-642-24322-6_23

42. Bhattacharjee A., Martonosi M. Inter-core cooperative TLB for chip multiprocessors. *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. P. 359-370. https://doi.org/10.1145/1736020.1736060

43. Bhattacharjee A., Martonosi M. Inter-core cooperative TLB for chip multiprocessors. *ACM SIGPLAN Notices. 45:3.* 2010. P. 359-370. Online publication date: 5-Mar-2010. https://doi.org/10.1145/1735971.1736060

44. Bhattacharjee A., Martonosi M. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 2009. P. 29-40. https://doi.org/10.1109/PACT.2009.26

45. Dooley I., Chao M., Kale L. NOISEMINER: An algorithm for scalable automatic computational noise and software interference detection. *Distributed*

Processing Symposium (IPDPS). 2008. P. 1-8. 10.1109/IPDPS.2008.4536186.978-1-4244-1693-6. http://ieeexplore.ieee.org/document/4536186/

46. Wells P., Sohi G. Serializing instructions in system-intensive workloads: Amdahl’s Law strikes again. *2008 IEEE 14th International Symposium on High Performance Computer Architecture (HPCA).* 2008. P. 264-275. 10.1109/HPCA.2008.4658645. 978-1-4244-2070-4. http://ieeexplore.ieee.org/document/4658645/

47. Samanta R., Surprise J., Mahapatr R. Dynamic Aggregation of Virtual Addresses in TLB Using TCAM Cells. *Proceedings of the 21st International Conference on VLSI Design.* P. 243-248. https://doi.org/10.1109/VLSI.2008.57

48. Sharifi M., Soryani M., Rezvani M. A Simulation-Based Study on Memory Design Issues for Embedded Systems. *Trends in Intelligent Systems and Computer Engineering.* 2008. P. 453-465. 10.1007/978-0-387-74935-8_32. http://link.springer.com/10.1007/978-0-387-74935-8_32

49. Hunt G., Larus J. Singularity. *ACM SIGOPS Operating Systems Review. 41:2.* 2007. P. 37-49. Online publication date: 1-Apr-2007. https://doi.org/10.1145/1243418.1243424

50. Chakraborty K., Wells P., Sohi G. Computation spreading. *ACM SIGPLAN Notices. 41:11.* 2006. P. 283-292. Online publication date: 1-Nov-2006. https://doi.org/10.1145/1168918.1168893

51. Bhandarkar D., Clark D. Performance from architecture: comparing a RISC and a CISC with similar hardware organization. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. P. 310-319. https://doi.org/10.1145/106972.107003

52. Aiken M., Fähndrich M., Hawblitzel C., Hunt G., Larus J. Deconstructing process isolation. *Proceedings of the 2006 workshop on Memory system performance and correctness*. P. 1-10. https://doi.org/10.1145/1178597.1178599

53. Mogul J., Borg A. The effect of context switches on cache performance. *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. P. 75-84.

https://doi.org/10.1145/106972.106982

54. Lazowska E. Operating system support for high-performance architectures. *Operating Systems of the 90s and Beyond*. 10.1007/BFb0024520. (40-43). http://www.springerlink.com/index/10.1007/BFb0024520

55. Lindblad C., Tennenhouse D. The VuSystem. *IEEE Journal on Selected Areas in Communications. 14:7*. 2006. P. 1298-1313. Online publication date: 1-Sep-2006. https://doi.org/10.1109/49.536481

56. Jaleel A., Jacob B. In-Line Interrupt Handling and Lock-Up Free Translation Lookaside Buffers (TLBs). *IEEE Transactions on Computers. 55:5*. 2006. P. 559-574. Online publication date: 1-May-2006. https://doi.org/10.1109/TC.2006.77

57. Millstein T. Practical predicate dispatch. *ACM SIGPLAN Notices. 39:10*. 2004. P. 345-364. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1029006

58. Bracha G., Ungar D. Mirrors. *ACM SIGPLAN Notices. 39:10*. 2004. P. 331-344. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1029004

59. Streckenbach M., Snelting G. Refactoring class hierarchies with KABA. *ACM SIGPLAN Notices. 39:10*. 2004. P. 315-330. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1029003

60. Guéhéneuc Y., Albin-Amiot H. Recovering binary class relationships. *ACM SIGPLAN Notices. 39:10*. 2004. P. 301-314. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1029002

61. Georges A., Buytaert D., Eeckhout L., De Bosschere K. Method-level phase behavior in java workloads. *ACM SIGPLAN Notices. 39:10*. 2004. P. 270-287. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1028999

62. Hauswirth M., Sweeney P., Diwan A., Hind M. Vertical profiling. *ACM SIGPLAN Notices. 39:10*. 2004. P. 251-269. Online publication date: 1-Oct-2004. https://doi.org/10.1145/1035292.1028998

63. Zhang L., Krintz C. Adaptive code unloading for resource-constrained JVMs. *ACM SIGPLAN Notices. 39:7*. 2004. P. 155-164. Online publication date: 11-Jul-2004. https://doi.org/10.1145/998300.997186

64. Stärner J., Asplund L. Measuring the cache interference cost in preemptive real-time systems. *ACM SIGPLAN Notices. 39:7*. 2004. P. 146-154. Online publication date: 11-Jul-2004. https://doi.org/10.1145/998300.997184

65. Seznec A. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. *IEEE Transactions on Computers. 53:7*. 2004. P. 924-927. Online publication date: 1-Jul-2004. https://doi.org/10.1109/TC.2004.21

66. Black A., Walpole J. Objects to the rescue! *Proceedings of the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs*. P. 100-104. https://doi.org/10.1145/504390.504418

67. Engler D., Kaashoek M., O'Toole J. The operating system kernel as a secure programmable machine. *Proceedings of the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs*. P. 62-67. https://doi.org/10.1145/504390.504407

68. Nagle D., Uhlig R., Mudge T., Sechrest S. Optimal allocation of on-chip memory for multiple-API operating systems. *Proceedings of the 21st annual international symposium on Computer architecture*. P. 358-369. https://doi.org/10.1145/191995.192070

69. Wahbe R., Lucco S., Anderson T., Graham S. Efficient software-based fault isolation. *Proceedings of the fourteenth ACM symposium on Operating systems principles*. P. 203-216. https://doi.org/10.1145/168619.168635

70. Chen J., Bershad B. The impact of operating system structure on memory system performance. *Proceedings of the fourteenth ACM symposium on Operating systems principles*. P. 120-133. https://doi.org/10.1145/168619.168629

71. Stodolsky D., Chen J., Bershad B. Fast interrupt priority management in operating system kernels. *USENIX Symposium on USENIX Microkernels and Other Kernel Architectures Symposium*. V. 4. P 9. /doi/10.5555/1295468.1295477

72. Markowsky G., Savenko O., Sachenko A. Distributed Malware Detection System Based on Decentralized Architecture in Local Area Networks. *In: Shakhovska, N., Medykovskyy, M. (eds) Advances in Intelligent Systems and Computing III. CSIT 2018. Advances in Intelligent Systems and Computing. Springer, Cham*. 2019. V. 871.

https://doi.org/10.1007/978-3-030-01069-0_42

73. Setia S., Squillante M., Tripathi S. Processor scheduling on multiprogrammed, distributed memory parallel computers. *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems.* P. 158-170. https://doi.org/10.1145/166955.167002

74. Hidaka Y., Koike H., Tanaka H. Multiple threads in cyclic register windows. *Proceedings of the 20th annual international symposium on computer architecture*. P. 131-142. https://doi.org/10.1145/165123.165149

75. Nagle D., Uhlig R., Stanley T., Sechrest S., Mudge T., Brown R. Design tradeoffs for software-managed TLBs. *Proceedings of the 20th annual international symposium on computer architecture*. P. 27-38. https://doi.org/10.1145/165123.165127

76. Hidaka Y., Koike H., Tanaka H. Multiple Threads In Cyclic Register *Windows 20th Annual International Symposium on Computer Architecture.* 10.1109/ISCA.1993.698552. 0-8186-3810-9. P. 131-142. http://ieeexplore.ieee.org/document/698552/

77. Vaswani R., Zahorjan J. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Proceedings of the thirteenth ACM symposium on Operating systems principles*. P. 26-40. https://doi.org/10.1145/121132.121140

78. Steenkiste P. Analyzing communication latency using the Nectar communication processor. *Conference proceedings on Communications architectures & protocols*. P. 199-209. https://doi.org/10.1145/144179.144278

79. Bershad B., Redell D., Ellis J. Fast mutual exclusion for uniprocessors. *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. P. 223-233. https://doi.org/10.1145/143365.143523

80. Torrellas J., Gupta A., Hennessy J. Characterizing the caching and synchronization performance of a multiprocessor operating system. *Proceedings of the fifth international conference on Architectural support for programming languages*

and operating systems. P. 162-174. https://doi.org/10.1145/143365.143506

81. Свистун А. А. Динамічне керування багатоядерними комп'ютерними системами / Збірник наукових праць за матеріалами IX Науково-практичної конференції «Інтелектуальні комп’ютерні системи та мережі». Тернопіль, 2024.

ДОДАТОК А Презентація роботи

Метод динамічного керування багатоядерними комп'ютерними системами

Антоній СВИСТУН

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

Хмельницький
2024

Актуальність роботи.

Тенденції в архітектурі комп'ютерів зосереджені на багатоядерних плантформах. Метою цих нових платформ є масштабування продуктивності системи за рахунок кількості ядер. Однак продуктивність поточних архітектур обмежена через додаткові витрати на паралелізм на рівні потоків і програмованість. ПП — це модель програмування, заснована на завданнях, яка полегшує програмованість багатоядерних процесорів і намагається розширити функціональний паралелізм у застосунках. Однак продуктивність ПП не масштабується ефективно для дрібнозернистих завдань, оскільки для таких завдань додаткові витрати на керування завданнями стають значними в порівнянні з виконанням завдань. АП — це динамічна система апаратної підтримки, яка має на меті зменшити поточні додаткові витрати ПП, розвантажуючи процес вирішення залежностей і синхронізацію з ядрами на апаратному забезпеченні. Тому, реалізуємо АП, визначаючи та підключаючи нове обладнання в симуляторі архітектури Cell. Масштабованість, продуктивність і пропускна здатність реалізації оцінюються для різних розмірів завдань і кількості ядер, використовуючи кілька шаблонів залежностей. Крім того, оцінимо різні параметри конфігурації, такі як розміри нового обладнання, вставленого в існуючу архітектуру.

2

Актуальність роботи полягає в необхідності удосконалити метод динамічного керування багатоядерними комп'ютерними системами.

Метою кваліфікаційої роботи є розробка методу динамічного керування багатоядерними комп'ютерними системами.

Поставлена мета досягається розв'язанням таких основних завдань:

- проаналізувати відомі методи динамічного керування багатоядерними комп'ютерними системами;
- удосконалити метод динамічного керування багатоядерними комп'ютерними системами;
- реалізувати розроблений метод динамічного керування багатоядерними комп'ютерними системами;
- здійснити еспериментальні дослідження згідно розроблених рішень.

Об'єктом дослідження є процес забезпечення динамічного керування багатоядерними комп'ютерними системами.

Предметом дослідження є методи динамічного керування багатоядерними комп'ютерними системами.

3

Для розв'язання поставлених задач використовувалися методи теорії комп'ютерних мереж, архітектури комп'ютерів.

Наукова новизна отриманих результатів:

- удосконалено метод динамічного керування багатоядерними комп'ютерними системами.

Практична значимість отриманих результатів полягає у забезпеченні динамічного керування багатоядерними комп'ютерними системами.

4

Архітектура комп'ютерів стала переважно багатоядерною, оскільки попередні методи підвищення продуктивності процесорів показали зниження продуктивності. Ці попередні методи підвищували продуктивність за рахунок використання паралелізму на рівні команд і використання зростаючих частот процесора. Експлуатація рівнів команд, однак, принесла з собою великий обсяг енергоспоживання, і показала слабкі результати. У той же час обмеження потужності обмежили збільшення частот процесора. В результаті індустрія розвернулася в бік багатоядерних платформ.

Результати дослідження показують, що масштабованість досягається для великих розмірів завдань, тоді як для малих значень продуктивність зменшується через додаткові витрати, що вносяться моделлю програмування. Для коротких завдань результати тестів показали три вузькі місця в поведінці системи виконання.

5

Процедура розв'язання проблеми є основним вузьким місцем, яке не може забезпечити виконання завдань досить швидко і, таким чином, призводить до зупинки. Планування та додавання завдань, що виконуються за допомогою середовища виконання, також є вузьким місцем, оскільки вони не здатні відповідати вимогам швидкості. Синхронізація з допоміжним потоком, який перевіряє робочі ядра за допомогою кругової системи, теж є проблемною. Для того, щоб впоратися з наслідками цих вузьких місць, потрібно розробити новий метод динамічної підтримки керування. Перспектвним напрямом для вдосконалення відомих рішень є динамічна підтримка керування базована на перевірці залежностей між завданнями та обробкою синхронізації з робочими ядрами за допомогою регістрів, відображених у пам'яті. Таке централізоване рішення може досягти кращої масштабованості для завдань меншого розміру.

Теоретичні та практичні результати роботи впроваджено при виконанні науково-дослідних робіт, які виконувались в Хмельницькому національному університеті.

6

Суть удосконалення методу:

Таким чином, обидва компоненти процесора, менеджер залежностей та інтерфейс регістра, поділяють доступ черги готовності, фінішного буфера та загальної черги. Інтерфейс регістрів дозволяє записувати та читати з елементів з відображених регістрів пам'яті. Регістр буфера та регістр буфера фінішу діють як нижні ребра черг буфера та фінішного буфера. Ці дві черги доступні лише для запису елементів до черги. Натомість, регістр готової черги поводиться як верхній край черги готовності, і його можна прочитати лише ззовні. При зверненні з логіки керування до цих черг режим роботи зворотний: зчитується черга в буфері і черга фінішного буфера; записується черга готовності. Реєстр статусів не включає жодної черги. Він зберігає цінну інформацію, тобто про те, запущений він чи ні, кількість вільних елементів у буфері та кількість елементів, присутніх у черзі готовності. Цей регістр можна зчитувати з будь-якого модуля симулятора, і він оновлюється на падаючому краю тактового циклу, якщо в цих значеннях відбулися будь-які зміни. Це робиться за допомогою керуючої логіки, яка виконує виклик регістра для оновлення регістра при виявленні зміни. Інтерфейс регістра також включає сховище завдань, що використовується як централізована пам'ять, до якої можна швидко отримати доступ. Крім того, що доступ до нього здійснюється ззовні модуля, він також використовується зсередини. Крім того, керуюча логіка, що міститься, також може виконувати різні виклики інтерфейсу регістра для виконання читання та запису. Ці дві операції можуть бути виконані як в інтерфейсі регістра для оновлення стану, так і для зчитування дескриптора зі сховища завдань.

- Кроки методу:
- 1) встановити граф залежностей для поточного стану багатоядерної системи;
- 2) визначити завантаження ядер;
- 3) якщо ядра завантажені помірно (близько 50 відсотків), то залучити менеджер залежностей;
- 4) розподілити процеси за чергами готовності;
- 5) здійснити аналіз буферу для процесів;
- 6) встановити розмір загальної черги та кількість процесів в ній;
- 7) якщо ядра завантажені надмірно, то залучити інтерфейс регістра.
- Таким чином, для динамічного керування багатоядерними системами пропонується комбінувати менеджер залежностей та інтерфейс регістра. Вони поділяють доступ черги готовності, фінішного буфера та загальної черги.

8

Таблиця 4.1 – Тест продуктивності

Етап	Пропускна здатність, тактові цикли	Пропускна здатність, час
Завантаження дескрипторів	45 циклів/завдання	678 завдань/мкс
Оброблення дескрипторів	125 циклів/завдання	698 завдань/мкс
Оброблення загальних завдань	175 циклів/завдання	674 завдання/мкс

Таблиця 4.2 – Результати тесту продуктивності з модифікаціями

Кількість ядер	Час, мкс
1	3.55145
2	2.74712
4	2.34323
8	2.13633
16	1.04189

10

ВИСНОВКИ

- У роботі за результатами виконаних теоретичних та практичних досліджень удосконалено метод динамічного керування багатоядерними комп'ютерними системами та отримано такі результати.
- 1. Проаналізовано відомі методи керування багатоядерними комп'ютерними системами.
- 2. Розроблений удосконалений метод динамічного керування багатопроцесорними систмами базується на апаратній підтримці. Для того, щоб додати нове обладнання, новий модуль побудований відповідно до вимог.

ВИСНОВКИ

- 3. Функціональність АП була розділена на інтерфейс регістра та менеджер витрат, який містить логіку керування та таблиці, що беруть участь у вирішенні проблем. Інтерфейс регістра визначено як шлюз модуля. На нього покладено відповідальність за синхронізацію множини завдань з ядрами архітектури. Окремо реалізований менеджер залежностей, що містить керуючу логіку, яка обробляє вхідні завдання і готові завдання. Для таблиць конструкції були встановлені фіксовані розміри, а хеш-функція з алгоритмом дозволу колізій була запрограмована на видачу відображення широкого діапазону вхідних даних на обмежену кількість записів у таблицях.
- 4. Здійснено есперриментальні дослідження згідно розроблених рішень. Результати показали, що АП зменшив вузькі місця, виявлені в моделі програмування ПП. Йому вдалося задовольнити вимоги до апаратного забезпечення, прискоривши вирішення залежностей і синхронізацію з ядрами, присутніми в платформі. Результати проілюстрували підвищення масштабованості системи в порівнянні з тими, що були отримані за допомогою моделі програмування ПП.

12

Свистун А. А.
студент 2 курсу ФІТ ХНУ
Науковий керівник к.т.н. Нічепорук А. О., кафедра КІІС ХНУ

ДИНАМІЧНЕ КЕРУВАННЯ БАГАТОЯДЕРНИМИ КОМП'ЮТЕРНИМИ СИСТЕМАМИ

Вступ. Тенденції в архітектурі комп'ютерів зосереджені на багатоядерних плантформах [1, 2]. Метою цих нових платформ є масштабування продуктивності системи за рахунок кількості ядер. Однак продуктивність поточних архітектур обмежена через додаткові витрати на паралелізм на рівні потоків і програмованість. Перспективною є модель програмування, заснована на завданнях, яка полегшує програмованість багатоядерних процесорів і намагається розширити функціональний паралелізм у застосунках. Однак її продуктивність не масштабується ефективно для дрібнозернистих завдань, оскільки для таких завдань додаткові витрати на керування завданнями стають значними в порівнянні з виконанням завдань [3]. Тому, дослідники активно розробляють апаратний підхід.

Постановка задачі. Об’єкт дослідження –процес забезпечення динамічного керування багатоядерними комп'ютерними системами. Мета дослідження – розробка методу динамічного керування багатоядерними комп'ютерними системами.

Основний матеріал. Побудова та підтримка графа залежностей, який представляє залежності завдань, може бути дуже повільною в програмному забезпеченні. Натомість, процесор обробляє залежності за допомогою простого пошуку в таблицях виробника та споживача, що може працювати дуже швидко, оскільки їм не потрібно перевіряти весь список. Кожен запис цих списків містить виділену там адресу та список початку, що складається з ідентифікаторів завдань, які очікують на звільнення адреси. Крім того, записи списку споживачів також мають поле, яке містить кількість залежностей для цієї адреси. Поля запису таблиці завдань описують ідентифікаційний номер завдання, покажчик на дескриптор, поточний стан задачі в системі та кількість її поточних залежностей.

Таблиця виробників використовується для уникнення помилок у режимі читання після запису. Кожен елемент у цій таблиці містить адресу, записану певним раніше доданим завданням. Щоразу, коли обробник дескриптора виявляє завдання залежно від одного з елементів цієї таблиці, то він негайно підписує завдання на свій стартовий список. Таким чином, коли обробник знайде від'ємний ідентифікатор, то він припинить вилучення елементів зі списку. Таблиця споживачів використовується для уникнення небезпеки запису після читання.

Висновки. Таким чином, додається новий бар'єр, що складається з числа завдань, які очікують адреси, але представлений у вигляді від'ємного значення, щоб відрізнити його від звичайного ідентифікаційний номер завдання. Відповідно, якщо певна кількість завдань чекають на адресу, а між ними немає розподілу, бар'єром буде апаратний пристрій. У такому випадку, коли обробник обробляє завершене завдання, то він перевіряє список. Якщо програма виявить від'ємне значення, вона вилучить елемент зі списку і запише кількість завдань у поле відповідним чином. Практичне значення роботи полягає у введенні нового елементу для покращення керування бгатопроцесорними системами.

Список літератури

1. Choppakatla, N., Sivalenka, M. and Boda, R. Task ordering in multiprocessor embedded system using a novel hybrid optimization model. *Multimedia Tools and Applications.* 2024. P. 1-25. 10.1007/s11042-024-19083-1.
2. Müller, L., Schumacher, N., Steffen, L., and Haubelt, C. Generative Design of the Architecture Platform in Multiprocessor System Design. *Electronics.* 2024. H. 13. 1404. 10.3390/electronics13071404.
3. Liang, J., Xiao, M., Lin, Z. and Li, C. t/k-Diagnosability of Regular Networks under the Comparison Model. *Sensors.* 2024. P. 24. 2303. 10.3390/s24072303.

Ім'я користувача:
Кафедра КІ

ID перевірки:
1016250159

Дата перевірки:
14.05.2024 11:25:00 EEST

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
14.05.2024 11:45:13 EEST

ID користувача:
100005591

Назва документа: **Свистун_Метод динамічного керування багатоядерними комп'ютерними системами**

Кількість сторінок: **84** Кількість слів: **22235** Кількість символів: **165226** Розмір файлу: **250.37 KB** ID файлу: **1016036023**

0.96% Схожість

Найбільша схожість: **0.82%** з джерелом з Бібліотеки (ID файлу: **1015988391**)

0.44% Джерела з Інтернету | 13 .... Сторінка 86

0.96% Джерела з Бібліотеки | 46 .... Сторінка 86

0% Цитат

Цитати | 1 .... Сторінка 87

Посилання | 1 .... Сторінка 87

0% Вилучень

Немає вилучених джерел

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 1.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 7%**

<table>
<tr><td rowspan="3">ID: 126168
Назва: МКР Метод динамічного керування багатоядерними комп'ютерними системами
Додано в БД: 2024-05-14
Автора: Антоній СВИСТУН
Керівники: Світлана САЧЕНКО
Консультанти:
Опоненти:</td><td colspan="2">Документ</td><td colspan="2">Сумарний збіг по Базі Даних</td></tr>
<tr><td>Символи</td><td>Лексеми</td><td>Символи</td><td>Лексеми</td></tr>
<tr><td>152723</td><td>1351</td><td>1358 (1%)</td><td>16 (1%)</td></tr>
</table>

Джерело плагіату

<table>
<tr><td rowspan="2">ID</td><td rowspan="2">Опис</td><td colspan="2">Наявність плагіату в документі</td></tr>
<tr><td>Символи</td><td>Лексеми</td></tr>
</table>

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник: __Антоній СВИСТУН ________________________

Тема: __Метод динамічного керування багатоядерними комп'ютерними системами ____

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень _-__; кількість сторінок записки ___

1. Короткий зміст роботи та прийнятих рішень _У роботі розроблено удосконалений метод динамічного керування багатоядерними комп'ютерними системами____

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 динамічного керування багатоядерними комп'ютерними системами.

У другому розділі здійснено дослідження предметної області та визначено стратегію забезпечення динамічного керування багатоядерними комп'ютерними системами.

У третьому розділі розроблено спосіб обробки завдань та використання декскриптору. Його реалізація базується на використанні апаратного пристрою. Також, розроблено метод динамічного керування багатоядерними комп'ютерними системами.

У четвертому розділі здійснено дослідження ефективності запропонованого рішення з апаратним пристроєм.

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 «добре» 4,00 (С)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) Мартинюк Валерій Володимирович, д.т.н., професор, завідувач кафедри АКІТР ХНУ

"14" травня 2024р.

Завідувачу кафедри КІІС
д-р.техн.наук, проф. Тетяні ГОВОРУЩЕНКО

Антоній СВИСТУН
ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-22-2

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.

30 березня 2024 року

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 динамічного керування багатоядерними комп'ютерними системами

Автор: Антоній СВИСТУН

Спеціальність: 123 – Компютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Саченко Світлана Іванівна, 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 1% 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	____________	Тетяна ГОВОРУЩЕНКО