**УДК 681.3**

**В.А. Клименко**
Національний університет “Львівська політехніка”,
кафедра електронних обчислювальних машин

# ОПРАЦЮВАННЯ СТАТИЧНИХ МАСИВІВ В АРХІТЕКТУРІ НА ОСНОВІ ПАМ’ЯТІ З ВПОРЯДКОВАНИМ ДОСТУПОМ



**Наведено алгоритм оптимізації компілятором статичних масивів в архітектурі на основі пам’яті з впорядкованим доступом. Наведено результати виконання оптимізації компілятором.**

**Algorithm of static array optimization by compiler in the architecture with the distributed register file with time access is described in this article. Results of the compiler’s optimization are shown.**

**Вступ.** Сучасні завдання вимагають високопродуктивних вбудованих рішень за низьку вартість [1]. Для досягнення цього необхідно створювати високопродуктивні вбудовані процесори з малою вартістю. Для того, щоб досягнути великих значень продуктивності, необхідно створити спеціалізований процесор під конкретну задачу [2]. Одним зі способів зменшення вартості такого процесора є зменшення вартості розробки за рахунок автоматизації та спрощення процесу проектування [3–4]. Для цього необхідно створити програмне забезпечення, яке б дозволило створювати спеціалізований процесор під конкретний алгоритм. Одним з шляхів з створення такого програмного забезпечення є створення програмного забезпечення з конфігурування базової архітектури процесора відповідно до алгоритму [5–6]. З огляду на гнучкість та можливість до налаштування під конкретний алгоритм найбільш перспективною є базова архітектура на основі пам’яті з впорядкованим доступом. Базовими елементами такої системи проектування є компілятор з мови програмування високого рівня, конфігуратор базової моделі та генератор опису спеціалізованого процесора мовою VHDL.

З огляду на організацію доступу до регістрового файла постає проблема роботи з статичними масивами в мовах програмування високого рівня, адже така організація не передбачає адресацію як таку [3]. Це спричинило завдання з вдосконалення компілятора з мови високого рівня. Такий компілятор має транслювати вхідну мову у проміжний код, у якому немає операцій над адресами. У цій роботі коло задач обмежено статичними масивами та програмами з невеликим циклами та без команд умовного галуження [6].

**Аналіз відомих рішень.** Сьогодні відомі системи проектування системного рівня, які виконують конфігурування базової архітектури. До таких систем можна зарахувати систему PICO Express та системи фірми Mitrion [7–8]. Базова архітектура таких систем передбачає можливість адресації, тому жодних проблем, пов’язаних з нею, у цих системах не виникає. Але базова архітектура в цих системах менш гнучка порівняно з архітектурою з пам’яттю з впорядкованим доступом і орієнтована на визначення кількості паралельних

блоків виконання. В архітектурі з пам'яттю з впорядкованим доступом можливо змінювати всі основні елементи такі, як регістровий файл, пристрій керування та функціональний модуль. Це дозволяє отримати спеціалізований процесор з відповідною продуктивністю та апаратними затратами.

**Постановка задачі.** Під час дослідження було поставлено наступну задачу. Створити компілятор з мови високого рівня ANSI C з обмеженою множиною мовних конструкцій. На вхід компілятору має подаватися програма без команд галуження, таких, як IF ELSE, SWITCH, кількість ітерацій в циклі має бути наперед визначена та невелика. Крім того, програма не має містити команд роботи з адресами та динамічного виділення/звільнення пам'яті. Компілятор має підтримувати цілий тип даних. У результаті роботи компілятора має бути отримано програму, яка складається з так званих трьохадресних асемблерних інструкцій. Таке представлення є, з одного боку, зрозумілим для аналізу людиною, а, з іншого, легко перетворює в машинні коди. Крім того, таке представлення є зручним для аналізу залежностей і виконання розпаралелення програми на рівні інструкцій (в англ. літературі ILP – Instruction Level Parallelism) [6]. Вигляд такої трьохадресної команди такий:

<КОП > <РП> <РД 1> <РД 2> , де

<КОП > – мнемонічний код операції

<РП> регістр приймач

<РД 1> перший регістр джерело

<РД 2> другий регістр джерело

**Програмне рішення.** Оскільки вхідною мовою програмування високого рівня є стандартна мова ANSI C доцільно будувати компілятор на базі існуючого компілятора. Таким компілятором був обраний компілятор GNU GCC, оскільки він є відкритим та добре відтестованим. На компілятор GCC можна покласти задачі лексичного, синтаксичного та семантичного аналізів, та виконання попередньої оптимізації.

Як інтерфейс між компілятором GCC та власною розробкою можна використати RTL код, який генерує компілятор GCC. Цей проміжний код є наближеним до асемблерного коду і тому є найбільш придатним для генерування трьохадресного коду програми.

Оскільки компілятор GCC підтримує багато архітектур та вхідних мов високого рівня, RTL код містить велику кількість інформації, яка не потрібна для нашого компілятора. Було вирішено виконувати компіляцію RTL коду у два етапи. На першому етапі утворюється проміжний трьохадресний код, який прямо відображає інструкції RTL коду. На цьому етапі вибирається лише необхідна для подальшої компіляції інформація: код операції, номери регістрів, константи, мітки тощо.

На другому етапі виконується генерування остаточного трьохадресного коду. На цьому етапі виконується аналіз проміжного коду та виконання необхідної оптимізації: введення окремої команди для завантаження констант, видалення команд пересилань даних (оскільки ми не обмежені у кількості регістрів, кількість яких визначається на етапі конфігурування) тощо. Ці оптимізації дають змогу скоротити кількість інструкцій і відповідно підвищити продуктивність програми.

Оскільки трьохадресний код є компактніший та містить лише необхідну інформацію це дозволяє аналізувати код для виконання додаткової оптимізації.

*Таблиця 1*

**Порівняння RTL коду з проміжним трьохадресним кодом**

| RTL код | Проміжний трьохадресний код |
|---|---|
| (insn 51 48 52 0 (parallel [<br>(set (reg:SI 74)<br>(mult:SI (reg/v:SI 71 [ b ])<br>(reg/v:SI 73 [ d ])))<br>(clobber (reg:CC 17 flags))<br>]) 173 {*mulsi3_1} (nil)<br>(nil)) | mul r74 r71 r73<br><br>add r72 r74 r70<br><br>add r75 r72 r70<br><br>add r76 r75 r73 |
| (insn 52 51 54 0 (parallel [<br>(set (reg/v:SI 72 [ c ])<br>(plus:SI (reg:SI 74)<br>(reg/v:SI 70 [ a ])))<br>(clobber (reg:CC 17 flags))<br>]) 139 {*addsi_1} (nil)<br>(nil)) | |
| (insn 54 52 55 0 (parallel [<br>(set (reg:SI 75)<br>(plus:SI (reg/v:SI 72 [ c ])<br>(reg/v:SI 70 [ a ])))<br>(clobber (reg:CC 17 flags))<br>]) 139 {*addsi_1} (nil)<br>(nil)) | |
| (insn 55 54 56 0 (parallel [<br>(set (reg:SI 76)<br>(plus:SI (reg:SI 75)<br>(reg/v:SI 73 [ d ])))<br>(clobber (reg:CC 17 flags))<br>]) 139 {*addsi_1} (nil)<br>(nil)) | |

Отже, обробка статичних масивів буде виконуватися над проміжним трьохадресним кодом. У результаті обробки ми маємо отримати трьохадресний код, вигляд якого був наведений вище. Масив являє собою виділену область пам'яті. Кожен елемент масиву це відповідна комірка пам'яті в цій області. Доступ до елемента масиву «masiv[i]» відбувається зверненням до комірки пам'яті, адреса якої обраховується додаванням до адреси першої комірки області пам'яті масиву "masiv" індекса «i». У результаті при зверненні до елементів масиву в проміжному трьохадресному коді отримуємо таку послідовність інструкцій (табл. 2).

У разі перетворення з RTL коду звернення до пам'яті відзначається позначкою "addr". Якщо в одній RTL інструкції виконується декілька операцій, результат яких потім записується у відповідний регістр, то на момент перетворення у проміжний трьохадресний код ми не можемо визначити номер регістра. У такому разі регістр позначається позначкою "tmp".

Під час подальшої обробки, оскільки ми маємо необмежений регістровий файл, перетворюємо проміжний трьохадресний код у так звану Static Single Assignment (SSA) форму. SSA форма – це форма представлення програми, в якій кожній змінній присвоюється значення не більше одного разу, проте читатись може скільки завгодно разів. Це дозволяє уникнути використання команд переміщення типу "mov". При виконанні перетворення виконується переприсвоєння номерів регістрів і заміна "tmp" на відповідний вільний регістр.

*Таблиця 2*

**Приклад звернення до елемента масиву у трьохадресному коді**

| Код на С | Проміжний трьохадресний код |
|---|---|
| int masiv [10];<br>int i=0;<br>for (i=0;i<4;i++)<br>{<br>masiv[i]=in();<br>masiv[i]=masiv[i]+1;<br>out(masiv[i]);<br>} | mov r70 0<br>cmp r17 r70 3<br>in r71<br>mul tmp r70 4<br>add tmp tmp r20<br>add addr tmp -48<br>mov addr r71<br>add r73 r71 1<br>mul tmp r70 4<br>add tmp tmp r20<br>add addr tmp -48<br>mov addr r73<br>out r73<br>add r70 r70 1<br>cmp r17 r70 3<br>in r71<br>mul tmp r70 4<br>add tmp tmp r20<br>add addr tmp -48<br>mov addr r71<br>add r73 r71 1<br>mul tmp r70 4<br>add tmp tmp r20<br>add addr tmp -48<br>mov addr r73<br>out r73 |

У разі роботи з статичними масивами адреси комірок відомі на момент компіляції. Отже, ми можемо під час оптимізації трьохадресного коду, перетворити адреси комірок пам’яті масивів у відповідні регістри. Таке рішення дозволить використовувати статичні масиви у процесорі з архітектурою на основі пам’яті з впорядкованим доступом.

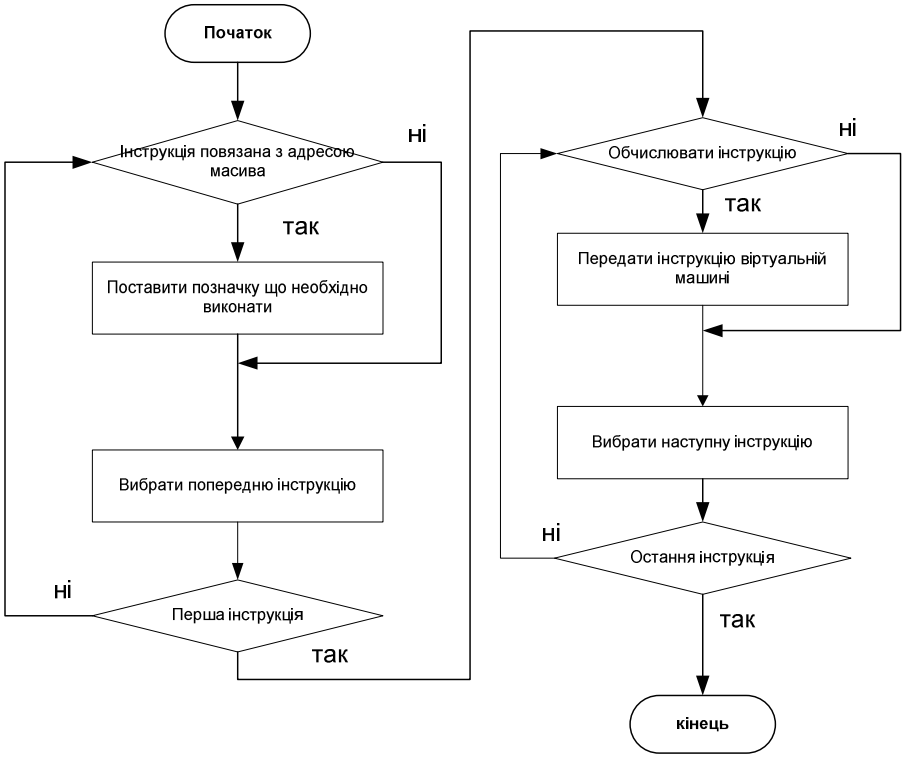
Для цього нам необхідно обрахувати адреси, виконавши на віртуальній машині відповідні трьохадресні інструкції.

Алгоритм виконання оптимізації такий (див. рисунок).

Обробляють у два проходи. При першому проході з кінця програми до початку ми аналізуємо кожну трьохадресну інструкцію. Якщо трьохадресна інструкція містить позначку “addr” або обчислюється регістр, номер якого ми запам’ятали, то запам’ятовуємо операнди, тобто номери регістрів <РД 1>, <РД 2> цієї інструкції, якщо їхні значення невідомі і ставимо позначку, що цю інструкцію треба виконати.

Під час другого проходу ми обробляємо програму з початку до кінця. На цьому етапі знаходимо інструкції, які були відмічені на першому етапі, і надсилаємо їх на віртуальну машину. Віртуальна машина має пам’ять даних, де зберігає результати обчислень, що дає змогу уникнути окремого зберігання проміжних результатів.

Можливо застосувати алгоритм з обчисленням у один етап. До того ж використати рекурсивний виклик функції для обчислення операндів інструкції. Це є недоцільним, оскільки вираз для визначення індексу елемента масиву може бути складним, для визначення адреси може бути необхідно обчислити велику кількість трьохадресних інструкцій. Тому у разі використання рекурсивного виклику функції з обчислення можливе переповнення стеку.




*Алгоритм обробки масиву у трьохадресному коді*

У результаті виконаних оптимізацій коду було досягнуто зменшення обсягів програми, що дозволяє пришвидшити виконання алгоритму. Ці результати було досягнуто за рахунок видалення всіх команд, результати виконання яких не використовувалися далі по коду. Використання SSA форми дало змогу видалити всі команди пересилання даних. Такі команди є зайвими для процесора, побудованого на основі пам'яті з впорядкованим доступом через специфіку організації регістрового файла. Крім того, оскільки кожній комірці пам'яті масиву було поставлено у відповідність окремий регістр, відпала необхідність у виконанні обчислень адрес відповідної комірки пам'яті. Експериментально було досліджено, що кожне обчислення адреси вимагає приблизно 3–4 команди, отже, застосовуючи цей метод, можемо зменшити обсяг коду програми на 3–4 команди для кожного звернення до пам'яті. Оскільки виконувалася розгортка циклу, відбулося значне зростання команд звернення до елементів масиву. Як можна пересвідчитися з результатів порівняння неоптимізованого та оптимізованого коду, було отримано значне зменшення (в середньому в 7–8 разів) обсягів коду програми (табл. 3), а отже, є потенціал для підвищення продуктивності роботи процесора.

*Таблиця 3*

**Результат оптимізації трьохадресного коду**

| № | Алгоритм | Кількість інструкцій неоптимізованого коду | Кількість інструкцій оптимізованого коду |
|---|---|---|---|
| 1 | Множення масиву на константу | 54 | 13 |
| 2 | Множення вектора 4 на матрицю 4x4 | 1814 | 165 |
| 3 | Множення матриць 4x4 | 1609 | 144 |

**Висновки.** В результаті виконаних досліджень отримано такі результати:

1) розроблено алгоритм оптимізації коду програми, яка містить статичні масиви. Отриманий в результаті оптимізації код є придатний для виконання на процесорі з архітектурою на основі пам'яті з впорядкованим доступом;

2) розроблений алгоритм оптимізації дозволив значно зменшити (приблизно у 7–8 разів) обсяг програми з статичними масивами. Отже, це дасть змогу значно збільшити продуктивність роботи процесора.

*1. Anatoly Melnyk, Andriy Salo. Instruction set architecture of the determined memory access processor// Досвід розробки та застосування САПР в мікроелектроніці – CADSM'2003. – с. 198–199. 2. Мельник А.О., Сало А.М. Методика проектування паралельного процесора на основі пам'яті з детермінованою вибіркою // Вісник Нац. ун-ту "Львівська політехніка". – Львів, 2005. – №546. – С. 96–101. 3. Мельник А.О., Сало А.М. Методика проектування паралельного процесора на основі пам'яті з детермінованою вибіркою.// Вісник Нац. ун-ту "Львівська політехніка". – Львів, 2005. №546. – С. 96–101. 4. Мельник А.О., Сало А.М. Організація регістрових файлів програмованих процесорів // Вісник Нац. ун-ту "Львівська політехніка", №573. – Львів, 2006. – С. 138–147. 5. Мельник А.О., Сало А.М. Проектування спеціалізованих процесорів на основі їх конфігурованих моделей // Моделювання та інформаційні технології. Зб. наук. пр. ІПМЕ НАН України. – Вип. 39. –К., 2007. – С. 136–149. 6. Мельник А.О., Сало А.М., Клименко В.А. Апаратна реалізація циклів програмованих конфігурованих процесорів // Вісник Нац. ун-ту "Львівська політехніка" Компютерні системи та мережі: Вид-во Нац. ун-ту "Львівська політехніка". – Львів, 2007. №603. – С. 94–102. 7. Synfora PICO Express Datasheet http://www.synfora.com/products/files/PicoExpressDatasheet_V18.pdf. 8. Goran Sandberg. The Mitrion-C Development Environment for FPGAs // www.mitrionics.com.*