Issue 14, 2023

https://doi.org/10.23939/sisn2023.14.225

УДК 621.396.67

ПОКРАЩЕННЯ СТИСНЕННЯ КОДУ ДЛЯ МІКРОКОНТРОЛЕРІВ ARM CORTEX M ЗА ДОПОМОГОЮ ПОПЕРЕДНЬОЇ ФІЛЬТРАЦІЇ

Микола Щербина

Львівський національний університет імені Івана Франка,
вул. Університетська, 1, Львів, Україна
mykola.shcherbyna@lnu.edu.ua, 0009-0003-9761-9466

© *Щербина М., 2024*

Протягом останніх десятиліть розмір коду уже не є обмеженням, за винятком малих вбудованих систем. ARM Cortex M – типова архітектура мікроконтролерів у таких системах. Запропоновано простий, але ефективний підхід для покращення стиснення коду алгоритмом загального призначення Deflate, оснований на попередній фільтрації двійкового коду Thumb2. Він перетворює перед стисненням інструкції BL (перехід зі збереженням адреси повернення), що вказують на ту саму ефективну адресу, і відновлює оригінальні коди операцій після декомпресії. Тести, виконані на реальному вбудованому програмному забезпеченні, показують, що запропонований алгоритм покращує стиснення коду приблизно на 3 %.

Ключові слова: стиснення коду; вбудовані системи; ARM Cortex M; Thumb2; фільтрація; deflate

Вступ

Наприкінці 80-х років минулого століття оперативна та постійна пам'ять комп'ютерів була обмеженим ресурсом, який вимірювали у кілобайтах. Її ефективне використання потребувало значних зусиль з боку програмістів. Втім, у XXI ст. обсяг пам'яті перестав бути обмежувальним чинником під час розроблення ПЗ. Цьому сприяла низька вартість як ОЗП, так і флеш-пам'яті, та її розмір, який нині вимірюємо у гігабайтах.

Залишилась чи не єдина галузь, де економне використання пам'яті все ще актуальне: вбудовані системи. Мова, звісно, про найменші мікроконтролери, представлені свого часу восьмирозрядною сім'єю PIC компанії Microchip чи AVR компанії Atmel (нині також Microchip).

Останнім часом спостерігається тенденція їх витіснення 32-розрядними мікроконтролерами архітектури ARM Cortex M [1], які, втім, не можуть похвалитися великим запасом вбудованої пам'яті. Наприклад, популярні мікроконтролери сім'ї STM32F0 від компанії STMicroelectronics мають обсяг флеш-пам'яті 16, 32, 64, 128 або 256 кБ (розмір ОЗП – 4, 6, 8, 16 або 32 кБ) [2].

Отже, проблема мінімізації заповнення флеш-пам'яті у вбудованих системах вельми актуальна. Очевидні способи її вирішення такі:

- оптимізація виконавчого коду;
- оптимізація незмінних даних;
- компресія виконавчого коду;
- компресія незмінних даних.

Сучасне вбудоване ПЗ пишуть, переважно, мовою C. Оптимізуючі компілятори для платформи ARM Cortex M (GCC, IAR і Keil) продукують достатньо якісний код. Розробники широко використовують сторонні бібліотеки, тож їх вплив на оптимізацію алгоритмів за розміром є вельми обмеженим. Вважатимемо, що цей напрям вдосконалень вичерпаний.

Незмінні дані, особливо у випадку таблиць, повинні використовувати типи даних мінімально можливого розміру. Втім, ядро Cortex-M0 не підтримує невирівняний (англ. *unaligned*) доступ до даних, тому можливості пакування структур дуже обмежені [3].

Постановка проблеми

Робота зосереджена на компресії виконавчого коду. Передбачено, що стиснений код зберігається у флеш-пам'яті та розтискається в ОЗП перед виконанням. Це може здійснюватися як суто програмно, так і за підтримки апаратури.

Після запуску виконується код початкового завантажувача, який перевіряє прикладне ПЗ та передає керування на нього. Важливо, що практично всі ресурси мікроконтролера (насамперед ОЗП), використані завантажувачем, звільняються після закінчення його роботи. Стиснувши Bootloader, ми можемо збільшити простір для прикладного ПЗ.

Той чи інший розподіл пам'яті задається скриптом компонувальника. В результаті компіляції проєкту вбудованого ПЗ отримують ELF файл, який містить заголовок, перелік сегментів та секцій та іншу службову інформацію (позначену ***). Він також містить код і незмінні дані (рис. 1).

Для стиснення виключно виконавчого коду як вхідні дані беруть секцію **.text**. У загальному випадку беруть всю сукупність секцій типу **PROGBITS**, після компресії яких отримують стиснений блок. Стиснення відбувається на комп'ютері.

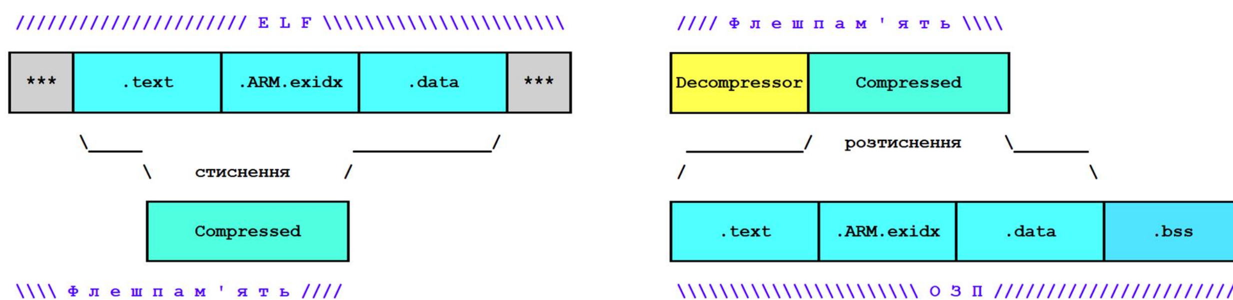

Рис. 1. Стиснення та декомпресія коду з флеш-пам'яті в ОЗП

У флеш-пам'ять записуються стиснений блок та код декомпресора (рис. 1). Отже, розтиснення здійснюється в ОЗП, і вбудоване ПЗ потрібно збирати для розташування там. Під час запуску системи керування передається на декомпресор. Після розтиснення він відновлює початкові значення потрібних регістрів та здійснює перехід на оригінальну точку входу вбудованого ПЗ. Альтернативою є апаратна реалізація декомпресора у системах з XIP та зовнішньою флеш-пам'яттю, яку ми не розглядатимемо.

Для незмінних даних так само можуть використовуватись алгоритми компресії загального призначення. Втім, спеціалізовані алгоритми ефективніші, зокрема для зберігання звуків і зображень. За автоматизації процесу компресії часто важко відрізнити код від даних, тому робота охоплює і це питання.

Аналіз досліджень та публікацій про стиснення коду

Порівняльну характеристику популярних алгоритмів стиснення даних (табл. 1) можна знайти у [4].

Таблиця 1

Порівняння популярних алгоритмів стиснення даних

Алгоритм	Тип	Часова складність	Стиснення
Гаффмана	Статистичний	$N[n + \log(2n - 1)] + Sn$	Середнє
Арифметичне кодування	Статистичний	$N[\log(n) + \mathrm{a}] + Sn$	Середнє
PPM	Статистичний	Залежно від контексту	Добре
CTW	Статистичний	Залежно від контексту	Добре
LZO	Словниковий	$\approx N(d)$	Добре
WKdm	Гібридний	$\approx N(d)$	Добре
WKS	Гібридний	$\approx N(d)$	Добре

Завдання стиснення виконавчого коду становлять теоретичний і практичний інтерес. Вже доволі довго науковці працюють над відповідними алгоритмами, як порівняно універсальними, так і прив'язаними до конкретної архітектури набору команд (ISA). Деякі ідеї алгоритмів компресії розглянуто у [5], хоч і для іншої архітектури MIPS. Втім, порівняльний аналіз показав перевагу алгоритму загального призначення Deflate [6], що використовується у **gzip**.

У дисертації [7] ґрунтовно проаналізовано алгоритми стиснення для різних ISA. Окремий розділ стосується RISC процесорів, зокрема ARM. Автор виокремлює ISA-залежну техніку стиснення, навівши приклад для MIPS із радше низькою щільністю набору інструкцій. Цікавою є ідея заповнення невикористовуваних бітів шаблоном із сусідньої інструкції для кращого стиснення, завдяки мінімізації відстані Геммінга (рис. 2).

Lwcl r3, 653 (r5)	110001	00101	00011	00000	(01010	001101)	\
							зробити однаковими
Add r16, r5, r2	110010	00101	00010	10000	(xxxxx	xxxxxx)	/

Рис. 2. Заповнення невикористовуваних бітів попереднім шаблоном

Загалом, синтез однакових бітових послідовностей дає змогу краще стискати код, що далі буде використано у роботі. Це випливає із самої задачі алгоритмів стиснення: пошук повторень та їх оптимальне кодування. Наприклад, в алгоритмі Deflate [6] кожен блок стискається за допомогою комбінації алгоритму LZ77 і кодування Гаффмана. Дерева Гаффмана для кожного блока є незалежними, алгоритм LZ77 може використовувати посилання на дубльований рядок, що траплявся у попередньому блоці (до 32K попередніх вхідних байтів).

Цікавий підхід запропонували автори [8] та [9], із урахуванням того, що у виконавчому коді часто містяться шаблонні послідовності інструкцій, які відрізняються лише декількома бітами. Наприклад, через різний розподіл регістрів усередині двох схожих функцій те саме значення може міститись у регістрах **r2** та **r3** відповідно. Тоді (для ISA ARM Thumb) у відповідних позиціях код відрізнятиметься лише 1 бітом: **010** проти **011**. Інший типовий шаблон полягає у підготовці виклику підпрограми, де вся відмінність полягає у значенні одного – двох параметрів. Запропоноване кодування стисненого ПЗ наведено на рис. 3.

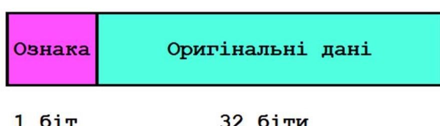

Рис. 3. Кодування бітових шаблонів інструкцій для стиснення

Тип вказує розмір маски (00 – 1 біт, 01 – 2 біти, 10 – 4 біти та 11 – 8 бітів).

Вказаний метод застосували для архітектур TI TMS320C6x, MIPS та SPARC, отримавши коефіцієнт стиснення 55–65 %, що, за оцінкою авторів, на ≈15 % перевершує наявні словникові алгоритми.

Робота [10] безвідносно до алгоритму компресії розглядає зберігання фрагментів вбудованого ПЗ у стисненому вигляді та стратегію їх декомпресії під час виконання. Схожі завдання проаналізовано у [11], де запропоновано двоетапне стиснення за допомогою Dynamic Frequency based Compression (DFC) та Static Frequency based Compression (SFC).

У статтях [12] та [13] розглянуто апаратну реалізацію деяких методів стиснення коду для ARM. Апаратну реалізацію простого словникового методу (де елементами словника є одиничні інструкції) також запропоновано в [14]. У роботі [15] описано метод стиснення CPB-ARM для ISA ARM, де простір невикористаних 32-розрядних кодів залучено для кодування блоків з трьох – чотирьох інструкцій.

Втім, останні методи використовують низьку щільність інструкцій ARM, що уже не притаманно ISA Thumb2. Нижче запропоновано метод стиснення коду для мікроконтролерів ARM Cortex M, який помітно покращує результати алгоритму загального призначення Deflate [6].

Алгоритм перетворення переходів

Система команд МК ARM Cortex-M

Проєкт ARM започаткувала у 1983 р. британська компанія Acorn Computers з метою створення мікропроцесорів RISC-архітектури. Протягом років архітектура ARM невпинно розвивалася, втім, основні характеристики залишалися незмінними: 32-розрядні мікропроцесори з 16 регістрами (**R13**, **R14** і **R15** виконують функції **SP**, **LR** і **PC** відповідно). 32-бітні інструкції ARM кодувалися уніфіковано. Такий підхід спрощував реалізацію ядра мікропроцесора, проте ціною зменшення щільності коду.

Для збільшення щільності коду в архітектурі ARM7TDMI (1994) були запроваджені 16-бітні інструкції Thumb. Економія досягалась, переважно, переходом на двоадресні команди (замість трьохадресних), зменшення кількості доступних регістрів і умовного виконання лише команд переходу. Інструкції **BL** (та **BLX**, починаючи з архітектури ARMv5) у Thumb є винятком, оскільки кодуються двома 16-бітними півсловами.

Втім, функціональність Thumb виявилась обмеженою, тож в архітектурі ARM1156 (2003) з’явився розширений набір команд Thumb-2, який запровадив різноманітні 32-бітні інструкції на додачу до **BL**/**BLX**. Кодування останніх дещо змінили (зі збереженням зворотної сумісності), збільшивши максимальну віддаль переходів (рис. 4).

```
BL  адреса
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
| 1  1  1  1  0 | S |        imm10        | 1 | 1 |J1| 1 |J2|           imm11              |

I1     = ~(J1 ^ S)
I2     = ~(J2 ^ S)
адреса = PC + 4 + РозширенняЗнаку(S:I1:I2:imm10:imm11:'0', 32)
```

Рис. 4. Кодування інструкції BL у наборі команд Thumb2

У 2004 р. ARM входить на ринок мікроконтролерів, створивши ядро Cortex-M3. Сьогодні на ринку широко пропонують мікроконтролери початкового рівня з ядрами Cortex-M0/M0+/M4. Ця робота стосуватиметься зазначених сімей мікроконтролерів, ISA яких реалізує *підмножину* Thumb та Thumb-2, а 32-бітні інструкції ARM *не підтримуються*.

Загалом, вигляд бітового простору 32-розрядних інструкцій Thumb2 подано на рис. 5.

1-ше півслово (біти 15–0)	2-ге півслово, біт 15	2-ге півслово
1 1 1 1 0	0	Обробка даних (безпосередній операнд) тощо
1 1 1 _ 1 0 1		Обробка даних (не безпосередній операнд)
1 1 1 1 1 0 0		Одиночне завантаження/збереження тощо
1 1 1 0 1 0 0 _ _ 1		Подвійне завантаження/збереження тощо
1 1 1 0 1 0 0 _ _ 0		Множинне завантаження/збереження тощо
1 1 1 1 0	1	Переходи та управління
1 1 1 _ 1 1 1 1		Інструкції копроцесора

Рис. 5. Бітовий простір 32-розрядних інструкцій Thumb2

Тестовий набір програмних кодів

Для перевірки ідеї вирішили використати специфічне вбудоване ПЗ для смарт-годинника PineTime [16], що виконується на мікроконтролері Nordic Semiconductor nRF52832 з ядром Cortex-M4. Розмір файлу **PineTime-mcuboot.bin – 290948 байт**, код починається з адреси **0x8200**. Зазначимо, що він не поміститься в оперативну пам’ять зазначеного мікроконтролера і використовується тільки як дослідний зразок типового вбудованого ПЗ (наявні RTOS, робота з Bluetooth, шифрування AES тощо).

Порівнюватимемо результати компресії з максимальним стисненням файлу **PineTime-mcuboot.bin** за допомогою **gzip**:

```
gzip -9n PineTime-mcuboot.bin
```

Отриманий файл **PineTime-mcuboot.bin.gz** має розмір **193459 байт**. Коефіцієнт стиснення 100 % × 193459 / 290948 ≈ **66 %** (без урахування розміру декомпресора).

Спробуємо оцінити необхідний розмір декомпресора (можлива похибка у десяток інструкцій тут прийнятна, оскільки вище ми оперували сотнями кілобайт). Класична реалізація Inflate у бібліотеці **zlib** не оптимальна за розміром, використаємо реалізацію з [17]. Для розтиснення достатньо викликати функцію:

```
int TINFCC tinf_zlib_uncompress(void *dest, unsigned int *destLen,
                    const void *source, unsigned int sourceLen);
```

Після компіляції виявилось, що розмір декомпресора становить **1419 байт**. Отже, стиснений код разом з декомпресором займає 193459 + 1419 = **194878 байт**. Справжній коефіцієнт стиснення 100 % × 194878 / 290948 ≈ **67 %**.

Фільтрація

Фільтрація даних – додатковий крок у алгоритмах стиснення (рис. 6) та полягає у певному перетворенні вхідних даних для оптимальнішого їх стиснення наявним алгоритмом. Термінологію запозичено зі специфікації PNG.

Перед етапом стиснення здійснюємо кодування вхідних даних. Розмір не змінюється (може навіть трохи збільшуватись за рахунок службової інформації, наприклад, у PNG додається байт алгоритму фільтрації перед кожним рядком зображення).

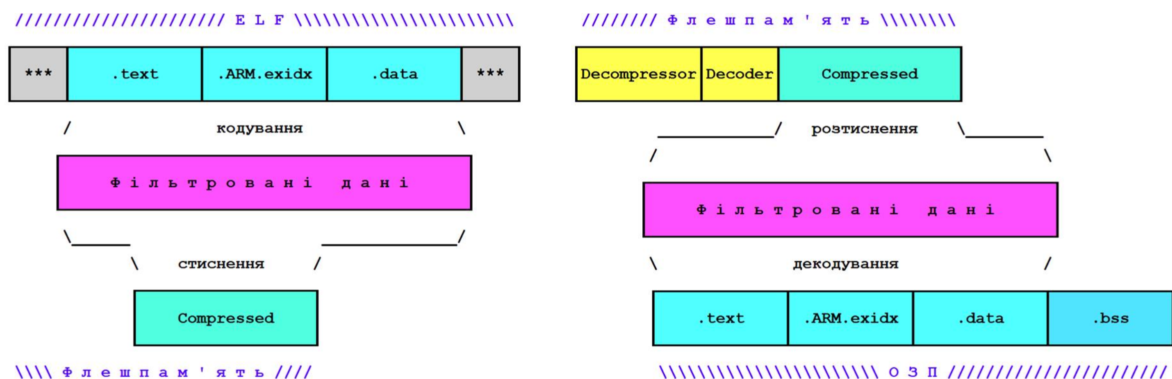

Рис. 6. Стиснення та декомпресія коду в ОЗП із застосуванням фільтрації

Ідея фільтра

Сформулюємо гіпотезу щодо типових властивостей вбудованого ПЗ. Програмний код містить доволі багато викликів підпрограм (інструкції **BL**) – використання *inline*-функцій збільшує код, отже, має застосовуватися лише у виняткових випадках. Деякі підпрограми викликаються багаторазово, зокрема функції стандартної бібліотеки. Не є поодинокими виклики певної функції з однаковим набором параметрів. В теорії алгоритм стиснення повинен "бачити" подібні шаблони та ефективно зменшувати розмір коду. Втім, на практиці це неможливо через *відносну* адресацію переходів у команді **BL**:

адреса **= PC + 4 + РозширенняЗнаку(*S*:*I1*:*I2*:*imm10*:*imm11*:'0', 32)**

За абсолютної адресації алгоритм стиснення "побачить" виклики однакових функцій. Втім, це обмежить адресний простір. Запропоновано натомість мати таблицю адрес функцій, яку заповнюють під час (де)кодування, та замінювати відносну адресу підпрограми на її індекс у таблиці. Залишиться впевнитися, що типова кількість підпрограм у вбудованому ПЗ не ставить занадто великих вимог до розміру згаданої таблиці.

Система команд мікроконтролерів сім'ї Cortex-M не містить інструкції **BLX** (рис. 7), оскільки режим ARM у них відсутній.

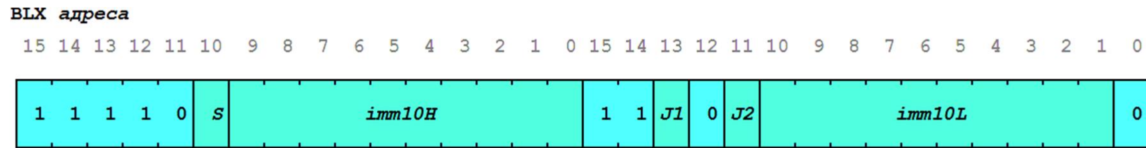

Рис. 7. Кодування інструкції BLX у наборі команд Thumb2

Будемо замінювати інструкції **BL** на **BLX**, а індекс записувати у поля:

index* = *imm10H*:*imm10L

(для простоти реалізації поля ***S***, ***J1*** та ***J2*** не чіпатимемо).

Роботу завершимо, коли дійдемо до кінця вхідних даних, або коли натрапимо на код непідтримуваної інструкції **BLX** (це неможливо у секції коду, проте можливо у секції незмінних даних).

Код фільтра

У наведеному нижче фрагменті на мові C 16-бітові інструкції зберігаються у масиві *t* довжини *n*, а адреси підпрограм – у масиві *fn*:

```
/* Encoding */
for (i = 0U; i + 1U < n; i++, pc += sizeof(uint16_t)) {
  if ((t[i] & 0xF800U) == 0xF000U) {
   if ((t[i + 1U] & 0xD000U) == 0xD000U) { /* BL */
     bool     S, J1, J2;
     uint16_t imm10, imm11;
     int32_t  imm32;
     uint32_t addr;
     size_t   j;
     S     = !!(t[i]      & 0x0400U);
     imm10 =    t[i]      & 0x03FFU;
     J1    = !!(t[i + 1U] & 0x2000U);
     J2    = !!(t[i + 1U] & 0x0800U);
     imm11 =    t[i + 1U] & 0x07FFU;
     imm32 = (S ? 0xFF000000 : 0)
         | (int32_t)!(J1 ^ S) << 23U
         | (int32_t)!(J2 ^ S) << 22U
         | (int32_t)imm10     << 12U
         | (int32_t)imm11     <<  1U;
     addr  = pc + 4U + imm32;
     (void)printf("BL\t0x%08" PRIX32 "\n", addr);
     /* Look for previous occurences */
     for (j = 0U; j < k; j++){
       if (fn[j] == addr) {
         /* Convert into "BLX" with index */
         t[i]      &= 0xFC00U;
         t[i]      |= (uint16_t)(j >> 10U);
         t[i + 1U] &= 0xE800U;
         t[i + 1U] |= (uint16_t)(j << 1U & 0x07FEU);
         break;
       }
     }
     /* New one */
```

```
          if (!!(t[i + 1U] & 0x1000U)) {
             assert(k < sizeof(fn) / sizeof(fn[0]));
             fn[k++] = addr;
          }
       }
       else if ((t[i + 1U] & 0xD001U) == 0xC000U) { /* BLX */
          break;  /* cannot appear in Cortex-M code, stop */
       }
    }
    else if ((t[i] & 0xEC00U) != 0xE800U /* Skip 16-bit */
         && (t[i] & 0xEF00U) != 0xEF00U) {  /* opcodes */
       continue;
    }
    i++, pc += sizeof(uint16_t);  /* for 32-bit opcodes */
 }
```

Перевірка гіпотези

Наведений фрагмент також видає перелік дизасембльованих інструкцій **BL**. Для файлу **PineTime-mcuboot.bin** отримаємо їх список (табл. 2) та запишемо у файл **encoder.out**.

Таблиця 2

Фрагмент переліку інструкцій BL з PineTime-mcuboot.bin

BL	0x00008918
BL	0x00008928
BL	0x00008E14
…	…
BL	0xFF8A285A
BL	0x0023356C
BL	0xFFF93180

Також отримаємо кількість опрацьованих команд (останнє значення *i* = **121854**). Вона знадобиться для декодування.

У кінці можна побачити дивні “адреси переходу”. Це не проблема – *false positives* можливі, коли кодування незмінних даних збігається з інструкцією **BL**.

Дослідимо, наскільки часто трапляються однакові виклики функцій:

```
uniq -c encoder.out | sort -r >bl_stats.txt
```

Верхня десятка з **bl_stats.txt** виглядає доволі обнадійливо (табл. 3).

Таблиця 3

Перша десятка підпрограм з найбільшою кількістю викликів

19	BL	0x0001330E
18	BL	0x00020464
12	BL	0x0000DA06
11	BL	0x0002049A
9	BL	0x00025668
9	BL	0x00019C02
8	BL	0x00020DBA
8	BL	0x00019C74
8	BL	0x00013270
8	BL	0x00011374
…	…	…

Тепер спробуємо стиснути перетворений файл **PineTime-mcuboot.flt**:

gzip -9n PineTime-mcuboot.flt

Отримаємо файл **PineTime-mcuboot.flt.gz** розміром **185329 байт**. Це краще, ніж 193459 байт після компресії оригіналу. Втім, крім декомпресора нам тепер потрібен ще й декодер. Написаний також на C, після компіляції він займає **352 байти**. Стиснений код разом з декомпресором та декодером займатиме приблизно 1419 + 352 + 185329 = **187100** байт, що помітно менше, ніж **194878** без застосування фільтра. Коефіцієнт стиснення становить 100 % × 187100 / 290948 ≈ **64 %**, тобто на **3 %** краще, ніж звичайний Deflate.

Висновки

У результаті дослідження вдалося розробити метод стиснення коду для мікроконтролерів ARM Cortex M, оснований на попередній фільтрації даних, який покращує результати відомого алгоритму загального призначення Deflate. Здійснено успішні тести на реальних зразках вбудованого ПЗ, де новий метод покращив коефіцієнт стиснення на 3 %.

Використання подібних методів дає змогу економити кошти завдяки використанню мікроконтролерів чи мікросхем із меншим об'ємом флеш-пам'яті. Водночас це доцільно лише на великих (≥100 тис. шт.) партіях виробів за умови, що компресія дасть можливість перейти на модель з нижчим обсягом пам'яті (переважно вдвічі), або ж за критичної потреби додавання коду й/або незмінних даних у вже наявній вбудованій системі.

Список літератури

1. Arm Limited. (2022). Arm Cortex-M Processor Comparison Table. https://documentation-service.arm.com/static/6267de1c7e121f01fd22d677?token=
2. STMicroelectronics. (2023). Microcontrollers & Microprocessors. STM32 32-bit Arm Cortex MCUs. STM32 Mainstream MCUs. STM32F0 Series – Products. https://www.st.com/en/microcontrollers-microprocessors/stm32f0-series/products.html
3. Arm Limited. (2018). ARM®v6-M Architecture Reference Manual. https://documentation-service.arm.com/static/5f8ff05ef86e16515cdbf826?token=
4. Simpson, M. (2003). Analysis of Compression Algorithms for Program Data. Division of Information Technology, University of Maryland. https://terpconnect.umd.edu/~barua/matt-compress-tr.pdf
5. Lekatsas, H., & Wolf, W. (1998). Code Compression for Embedded Systems. Proceedings of the 35th annual Design Automation Conference (pp. 516–521).
6. Deutsch, P. (1996). DEFLATE Compressed Data Format Specification version 1.3. Network Working Group. Request for Comments: 1951. https://www.ietf.org/rfc/rfc1951.txt
7. Talal, B. (2009). Huffman-based Code Compression Techniques for Embedded Systems. Fakultät für Informatik, Universität Fridericiana zu Karlsruhe. https://publikationen.bibliothek.kit.edu/1000017922/1319910
8. Seong, S., & Mishra, P. (2008). Bitmask-Based Code Compression for Embedded Systems. IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems, 27(4), 673–685. https://doi.org/10.1109/TCAD.2008.917563
9. Seong, S., & Mishra, P. (2006). A Bitmask-based Code Compression Technique for Embedded Systems. Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (pp. 251–254). https://doi.org/10.1145/1233501.1233551
10. Ozturk, O., Kandemir, M., & Chen, G. (2008). Access Pattern-Based Code Compression for Memory-Constrained Systems. ACM Transactions on Design Automation of Electronic Systems, 13(4), 1–30. https://doi.org/10.1145/1391962.1391968
11. Shrivastava, K., & Mishra, P. (2011). Dual Code Compression for Embedded Systems. Proceedings of the 24th Annual Conference on VLSI Design, 177–182. https://doi.org/10.1109/VLSID.2011.13
12. Dias, W. R. A., Moreno, E. D., & Barreto, R. da Silva. (2011). An Approach for Code Compression in Run Time for Embedded Systems – A Preliminary Results. Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing, 349–359. https://doi.org/10.1007/978-3-642-24650-0_30

13. Garofalo, V., Napoli, E., Petra, N., & Strollo, A. G. M. (2007). Code compression for ARM7 embedded systems. Proceedings of the 18th European Conference on Circuit Theory and Design, 687–690. http://doi.org/10.1109/ECCTD.2007.4529689

14. Do, Q., & Le, T.C. (2012). Low Power Embedded System Design Using Code Compression. Solid State Systems Symposium, 1–4. https://www.researchgate.net/publication/282150217_Low_Power_Embedded_System_Design_Using_Code_Compression

15. Dias, W. R. A., & Moreno, E. D. (2012). CPB-ARM - A New Code Compression Method for Embedded Systems. 13th Symposium on Computing Systems, 25–32. https://doi.org/10.1109/WSCAD-SSC.2012.20

16. Firmware Release. (2020). https://github.com/lupyuen/pinetime-rust-riot/releases/tag/v1.0.2

17. Joergen Ibsen. Tiny inflate library. (2019). https://github.com/jibsen/tinf

IMPROVING CODE COMPRESSION FOR ARM CORTEX M MICROCONTROLLERS USING PRE-FILTERING

Mykola Shcherbyna

Ivan Franko National University of Lviv, 1, Universytetska str., Lviv, Ukraine
mykola.shcherbyna@lnu.edu.ua, 0009-0003-9761-9466

© Shcherbyna M., 2023

For last decades code size is no longer a concern except small embedded systems. ARM Cortex M is a typical microcontroller architecture of such systems. A simple yet effective approach based on pre-filtering Thumb2 binary code is proposed to improve code compression by the general purpose Deflate algorithm. It transforms BL (branch and link) instructions pointing to the same effective address before compression, and restores original opcodes after decompression. Tests performed on real-life embedded software show that the proposed algorithm improves code compression by approximately 3 %.

Key words: code compression; embedded systems; ARM Cortex M; Thumb2; filtering; deflate