Code Efficiency in ARM CPU Architecture

Bashar Abu-Hweij, Nicole Hessner, Mary Oh
Group 1, CS 533 – Computer Architecture
School of Technology & Computing, City University of Seattle
bhweij @cityuniversity.edu, hessnernicole @cityuniversity.edu,
ohmary1@cityuniversity.edu

Abstract

This paper highlights the importance of code efficiency in the ARM CPU architecture and the impact of the processing power of ARM Microcontrollers. ARM assembly language, a low-level programming language, allows programmers to better understand the architecture toolchains and limitations of their hardware. Code efficiency includes the use of efficient algorithms (e.g. avoiding division where possible), and arranging the code's sequence of statements to speed up program execution without wasting resources. The goal of this paper is to have two assembly code implementations for the Bubble Sort and Bucket Sort (single value buckets) algorithms and compare the efficiency when sorting a defined set of small positive integers using ARM microcontrollers.

Keywords: Arm Microcontrollers, Bubble Sort, Bucket Sort, CPU Architecture, Assembly Code.

1. PROJECT DESCRIPTION

Programmers who are developing heavy workloads often face the challenge of needing to optimize their code for specific processors. This allows the programmer to reduce consumption and completion time as much as possible while minimizing risk to the business or operating environment.

Code efficiency is achieved through several factors, including the use of efficient algorithms, and the rearrangement of the sequence of statements to speed up the program execution without wasting resources. For this project, we will develop one Bubble Sort and one Bucket Sort program to sort the same set of data and measure the difference in efficiency between the two algorithms.

2. USEFULNESS

Software product quality can be assessed by evaluating the efficiency of the code used. Quality is directly linked to algorithmic efficiency, which affects the speed of runtime execution for software, which is a key measure of high performance.

3. PROJECT TOOLS

Keil µVision is a Windows-based software development platform that combines a robust and modern editor with a project management tool (Arm Limited, *n.d.*).

In this project, µVision is used to run and debug the two sorting assembly codes. Logic, Performance, and System Analyzers, from within the Keil µVision software development environment, are also utilized to explore any performance advantages.

4. LITERATURE REVIEW

Moore's Law has transformed society and improved computing systems. "Moore's Law is a techno-economic model that has enabled the information technology industry to double the performance and functionality of digital electronics roughly every 2 years within a fixed cost, power, and area" (Shalf, 2020). However, as hardware-driven gains slow down, it has become much harder than ever to keep up with Moore's Law. Thereby, the challenge of code optimization will become increasingly prevalent in the quest to continuously improve computers.

There are many ways to implement an algorithm and creating code that works is not enough. It is important to write code that minimizes computing resource waste. With recent advancements in embedded systems and the employment of more software applications, code efficiency contributes towards the reduction of power and resource consumption.

Understanding computational complexity as well as the underlying hardware are some of the challenges in writing code efficiently. The scope of this project involves comparing the efficiencies of two assembly code implementations for Bubble Sort and Bucket Sort algorithms. This will then be used to compare efficiency when sorting a defined set of small positive integers using ARM microcontrollers.

Bucket Sort distributes the elements of an array into a designated number of buckets. Then, the buckets are sorted again, using either another Bucket Sort or another sorting algorithm. Finally, the buckets are reassembled into a sorted array. This algorithm assumes a uniform distribution. Its average time complexity is O(n+k), and worst case is O(n^2), and its space complexity is O(n+k) (Sehgal, 2019).

Bubble Sort compares consecutive items in an array, and if they are in the wrong order (determined by sort direction), the items are swapped. The algorithm iterates until all of the numbers are in the correct spot, building from the furthest right element of the array. The average time complexity of Bubble Sort is O(n2) (Upadhyay, 2022). The worst-case time complexity Bubble Sort, like Bucket Sort, is O(n^2)(Michael Sambol, 2016). The space complexity is O(1), because Bubble Sort only requires a fixed amount of extra space for its variables. It is very inefficient for large datasets. (Upadhyay, 2022).

5. Methodology and Assumptions

For this project, we are using Assembly code subroutines that were written in Keil µVision IDE, using the ARM toolset, for the two sorting algorithms, Bubble Sort and Bucket Sort.
The subroutines were tested using the STM32F411VETx by STMicroelectronics.

The STM32F4 family incorporates high-speed embedded memories and an extensive range of enhanced I/Os and peripherals.

The instruction sets used; includes:

- Register Movement: MOV,
- Arithmetic and Logic: ADD, SUB, MUL,
- Load/Store: LDR/STR,
- Comparison: CMP,
- Control Flow: BGT, BLT, BEQ, B, BX

For the purpose of this analysis, the codes for the two sorting algorithms assume that the sorting is to be performed on positive integer values contained in ranges such as 0 to 10 or 10 – 99, etc.

The ARM architecture is a Reduced Instruction Set Computer (RISC) with a relatively simple implementation of a load/store architecture.
Dedicated load and store instructions (LDR/STR) are used to access the main RAM memory.

ARM also supports different addressing modes with all load/store addresses being determined from registers and instructions.

All arithmetic computation (sums, products, logical operations, etc.) is performed on values held in registers.

5.1 Big-O Notation

An algorithm is a step-by-step list of instructions used to perform a required task. A good software engineer will consider both time and space complexities when designing their program. Big-O Notation is a relative representation of the time and space complexities of an algorithm. It describes the execution time and memory space used for a task in relation to the number of steps required to complete it. Big O Notation is one of the most necessary mathematical notations used in computer science to measure an algorithm's efficiency (Kuredjian, 2017).

5.2 Bubble Sort Algorithm

Bubble sort is a simple sorting algorithm that repetitively steps through the values in a list, compares two adjacent elements, and swaps them until they are in the intended order.

The process for sorting the elements in ascending order can be summarized below:

- Starting from the first index, compare the first and the second elements.
- If the first element is greater than the second element, they are swapped.
- Now, compare the second and the third elements. Swap them if they are not in order.
- The above process goes on until the last element.

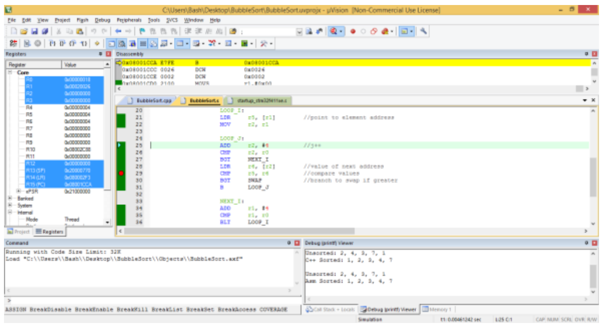
Hence, the average time complexity is n*n = n^2 or O(n^2) as illustrated in the below table:

Cycle	Number of Comparisons
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
…….	……
last	1

Table 1 - Calculating Time Complexity for Bubble Sort Algorithm

The number of comparisons is: (n-1) + (n-2) + (n-3) +…..+ 1 = n(n-1)/2

Placing a breaking point in debugging mode at the comparison operator, inside the inner (J) loop, we can count 10 iterations for the array of 5 elements.


```
0x08001CCA E7FE     B       0x08001CCA
0x08001CCC 0026     DCW     0x0026
0x08001CCE 0002     DCW     0x0002
```

```
LOOP_I:
        LDR     r5, [r1]        //point to element address
        MOV     r2, r1

LOOP_J:
        ADD     r2, #4          //j++
        CMP     r2, r0
        BGT     NEXT_I
        LDR     r6, [r2]        //value of next address
        CMP     r5, r6          //compare values
        BGT     SWAP            //branch to swap if greater
        B       LOOP_J

NEXT_I:
        ADD     r1, #4
        CMP     r1, r0
        BLT     LOOP_I
```

```
Running with Code Size Limit: 32K
Load "C:\\Users\\Bash\\Desktop\\BubbleSort\\Objects\\BubbleSort.axf"
```

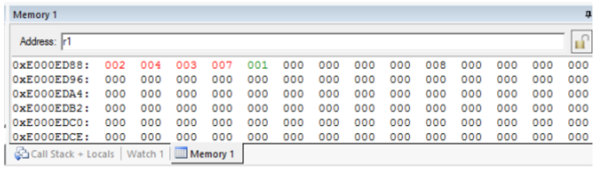
```
Unsorted: 2, 4, 3, 7, 1
C++ Sorted: 1, 2, 3, 4, 7

Unsorted: 2, 4, 3, 7, 1
Asm Sorted: 1, 2, 3, 4, 7
```

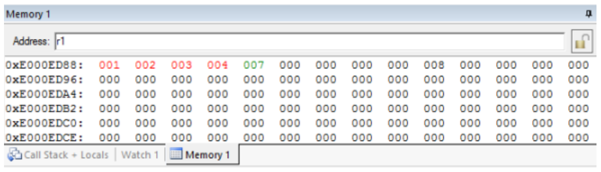
Figure 1 - Screenshot showing break point and final results

The bubble sort algorithm does not require any additional memory and uses the same memory allocated for the original unsorted list. The space complexity is O(n). (Programiz, n.d.)

The below images show the R1 register containing the data before and after running bubble sort:


Address: r1

```
0xE000ED88: 002 004 003 007 001 000 000 000 000 008 000 000 000 000
0xE000ED96: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDA4: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDB2: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDC0: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDCE: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```


Address: r1

```
0xE000ED88: 001 002 003 004 007 000 000 000 000 008 000 000 000 000
0xE000ED96: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDA4: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDB2: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDC0: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
0xE000EDCE: 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

Figure 2 - Screenshots showing in-place memory sorting for Bubble Sort Algorithm

5.3 Bucket Sort Algorithm

Bucket sort is a sorting algorithm that separates the unsorted values in the array into multiple groups, called buckets. If multiple values were grouped in any of the buckets, the bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sorting algorithm.

Bucket sort is mainly useful when input is uniformly distributed over a range. In order to keep the buckets uniformly distributed, and avoid further individual bucket sorting calls, the unsorted data used is limited so that each bucket will contain a single element. In our example code, we have buckets for every single value in the range from 0 to the maximum value of the array elements.

The used algorithm can be summarized in the below steps:

- Loop through the array to determine the maximum value (M)
- Create empty buckets with a range from 0 to M
- Loop through every array element arr[i], inserting (filling) the buckets for the arr[i] value
- Concatenate sorted buckets.

Iterating through the data to find the maximum values and insertion in the corresponding buckets takes O(n), O(m) time, n is the number of elements in the data list and m is the maximum found value.

Unlike bubble sort, bucket sort uses temporary data storage for the grouping buckets, making it a non-in-place algorithm (Geeks For Geeks, 2022)

6. WORKLOAD ASSIGNMENT

6.1 Bashar Abu-Hweij

- Proposal idea
- Abstract
- Project Description
- Project Tools
- Methodology and Assumptions
- Source Codes (Appendix 'A')

6.2 Nicole Hessner

- Literature Review
- Editing

6.3 Mary Oh

- Abstract (Proof editing)
- Usefulness
- Formatting of paper
- Proofreading and editing of the paper
- Literature Review

7. REFERENCES

Arm Limited official website (2022). Arm CPU Architecture. https://www.arm.com/architecture/cpu

Arm Limited official website (2022). µVision User's Guide. https://developer.arm.com/documentation/101407/0537/About-uVision?lang=en

ARM Official Website (2022). Arm CPU Architecture: A Foundation for Computing Everywhere, retrieved from: https://www.arm.com/architecture/cpu

Chen, B., Tarlow, D., Swersky, K., Maas, M., Heiber, P., Naik, A., Hashemi, M., & Ranganathan, P. (n.d.). Learning to Improve Code Efficiency. arxiv.org.

David Patterson and John Hennessy (2018), Computer Organization and Design RISC-V edition

Geeks For Geeks (2022). Bucket Sort, retrieved from: https://www.geeksforgeeks.org/bucket-sort-2/

Kusswurm, D (2020). Modern Arm Assembly Language Programming: Covers Armv8-A 32-bit, 64-bit, and SIMD

Ledin, J. (2020). Modern Computer Architecture and Organization. Packt Publishing

Programiz (n.d). Bubble Sort, retrieved from: https://www.programiz.com/dsa/bubble-sort

Sambol, Michael. (2016, July 26). Bubble sort in 2 minutes [Video]. YouTube. https://www.youtube.com/watch?v=xli_FI7CuzA

Sehgal, K. (2019, December 1). An Introduction to Bucket Sort - Karuna Sehgal. Medium. https://medium.com/karuna-sehgal/an-introduction-to-bucket-sort-62aa5325d124

Shalf, J. (2020). The Future of Computing Beyond Moore's Law. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *378*(2166), 20190061. https://doi.org/10.1098/rsta.2019.0061

Stuart Kuredjian (2017). Algorithm Time Complexity and Big O Notation, retrieved from: https://medium.com/@StueyGK/algorithm-time-complexity-and-big-o-notation-51502e612b4d

Upadhyay, S. (2022, November 15). Bubble Sort Algorithm: Overview, Time Complexity, Pseudocode and Applications. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm

Appendix 'A': Source Code

Assembly Code (Bubble Sort):

```
.text

// Subroutine:
// extern "C" void BubbleSort_(int size, int* array)
.global BubbleSort_                //global export
   BubbleSort_:

               MOV     r4, #4          //constant = 4 bits
               SUB     r0, #1
               MUL     r0, r4          //array size in bits
               ADD     r0, r1          //last address

               LOOP_I:
               LDR     r5, [r1]        //point to element address
               MOV     r2, r1

               LOOP_J:
               ADD     r2, #4          //j++
               CMP     r2, r0
               BGT     NEXT_I
               LDR     r6, [r2]        //value of next address
               CMP     r5, r6          //compare values
               BGT     SWAP            //branch to swap if greater
               B       LOOP_J

               NEXT_I:
               ADD     r1, #4
               CMP     r1, r0
               BLT     LOOP_I

               BX      lr            //branch and exchange instruction set to return value


               SWAP:
               STR     r6, [r1]
               STR     r5, [r2]
               MOV     r5, r6
               B       LOOP_J
```

C++ Code (Bubble Sort):

```
//Library imports
#include <stdio.h>

//Imported external assembly subroutines from asm.s
extern "C" void BubbleSort_(int size, int* integers); // Bubble Sort Subroutine

//Helper functions
//Function to print
void printArray(int, int*);
void printArray(int size, int *array)
{
    for (int i = 0; i < size -1; i++)
    {
        printf("%d, ", array[i]);
    }

    //print last element without separator
    if (size > 1) {
        printf("%d\n", array[size -1]);
    }
}

//Function to bubble sort in c++
void BubbleSort(int, int*);
void BubbleSort(int size, int* array)
{
    for(int i = 0; i < size; i++)
    {
        for(int j = i + 1; j < size; j++)
        {
            if(array[i] > array[j])
            {
                //swap
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
      }
    }
  }
}

//Function to copy arrays
void Copy(int size, const int from[], int to[]);
void Copy(int size, const int from[], int to[])
{
    for(int i = 0; i < size; i++)
    {
        to[i] = from[i];
    }
}

int main()
{
  printf("This program is written by Bashar Abu Hweij\n\n");

  const int integers[] = {2, 4, 3, 7, 1};
```

```
    const int size = sizeof(integers) / sizeof(int);

    //print unsorted
    int testIntegers[size];
    Copy(size, integers, testIntegers);
    printf("Unsorted: ");
    printArray(size, testIntegers);

    //sort using C++
    BubbleSort(size, testIntegers);
    printf("C++ Sorted: ");
    printArray(size, testIntegers);
    printf("\n");

    //reset integers to unsorted
    Copy(size, integers, testIntegers);
    printf("Unsorted: ");
    printArray(size, testIntegers);

    //sort using asm
    BubbleSort_(size, testIntegers);

    //print sorted
    printf("Asm Sorted: ");
    printArray(size, testIntegers);
  return 0;
}
```

Assembly Code (Bucket Sort):

```
// ------------------------------------------------
// TP - CS533-Computer Architecture Fall 2022
// By: Bashar Abu Hweij
// References: ARM Compiler armasm Reference Guide Version 6.01
// https://developer.arm.com/documentation/dui0802/b/A32-and-T32-Instructions
// ------------------------------------------------

.text


// Subroutine:
// extern "C" void BucketSort_(int size, int* array)
.global BucketSort_                     //global export
BucketSort_:
                //constant registers
                MOV    r4, #4         //Bits
                MOV    r10, #0        //False
                MOV    r11, #1        //True

                SUB    r3, r0, #1     //number of elements -1
                MUL    r3, r4        //array size in bits = length * 4
                ADD    r3, r1        //last address

                //Get max value to determine size of allocated memory
                MOV    r9, #0         //Max placeholder
                B      START_MAX

                LOOP_MAX:
                SUB    r3, #4
                CMP    r3, r1
                BLT    FILL_BUCKETS

                START_MAX:
                LDR    r5, [r3]      //current value
                CMP    r9, r5
                BGE    LOOP_MAX
                MOV    r9, r5
                B      LOOP_MAX


                //Allocate memory of size max

                FILL_BUCKETS:
                ADD    r9, #1         //add bucket for zero
                MOV    r3, #-1

                LOOP_EMPTY:
                ADD    r3, #1
                CMP    r3, r9
                BGT    FILL
                MUL    r5, r3, r4

                STR    r10, [r2, r5] //set at False
                B      LOOP_EMPTY

                FILL:
                MOV    r3, #-1
```

```
LOOP_FILL:
ADD     r3, #1
CMP     r3, r0
BGE     SORT
MUL     r5, r3, r4
LDR     r6, [r1, r5]
MUL     r6, r4

STR     r11, [r2, r6] //Fill with True
B       LOOP_FILL

SORT:
MOV     r3, #-1
MOV     r5, #0
LOOP_SORT:
ADD     r3, #1
CMP     r3, r9
BGE     EXIT
MUL     r6, r3, r4
LDR     r7, [r2, r6]
CMP     r7, r11
BEQ     OUTPUT
B       LOOP_SORT

EXIT:
BX      lr

OUTPUT:
STR     r3, [r1, r5]
ADD     r5, #4
B       LOOP_SORT
```

C++ Code (Bucket Sort):

```
/*
 TP - CS533-Computer Architecture Fall 2022
 By: Bashar Abu Hweij

   Due to evaluation version limitation, <iostream> library cannot be imported
   printif is used in place std::cout and
   Reference: https://www2.keil.com/limits
**/


//Library imports
#include <stdio.h>

//Imported external assembly subroutines from asm.s
extern "C" void BucketSort_(int size, int *integers, int *buckets); // bucket Sort Subroutine

//Helper functions
//Function to print
void printArray(int, int*);
void printArray(int size, int *array)
{
   for (int i = 0; i < size -1; i++)
   {
      printf("%d, ", array[i]);
   }

   //print last element without separator
   if (size > 1) {
      printf("%d\n", array[size -1]);
   }
}

//Function to bubble sort in c++
void BucketSort(int, int*);
void BucketSort(int size, int* array)
{
      //determine size of buckets by finding max value in array
      int max = 0;
      for(int i =0; i < size; i++)
      {
         if (array[i] > max) {
            max = array[i];
         }
      }

      bool *buckets = new bool[max]();
      //Creat and fill buckets
   for (int i=0; i < size; i++) {
         *(buckets + array[i]) = true;
   }

   //Sort from filled buckets
   int j =0;
   for (int i=0; i <= max; i++) {
         printf("%d", buckets[i]);
         if(buckets[i] == true) {
            array[j] = i;
      j++;
    }
  }
}

//Function to copy arrays
void Copy(int size, const int from[], int to[]);
void Copy(int size, const int from[], int to[])
{
  for(int i = 0; i < size; i++)
  {
    to[i] = from[i];
  }
}

int main()
{
 printf("This program is written by Bashar Abu Hweij\n\n");

 const int integers[] = {2, 4, 3, 7, 1};
  const int size = sizeof(integers) / sizeof(int);

  //print unsorted
  int testIntegers[size];
  Copy(size, integers, testIntegers);
  printf("Unsorted: ");
  printArray(size, testIntegers);


  //sort using C++
  BucketSort(size, testIntegers);
  printf("C++ Sorted: ");
  printArray(size, testIntegers);
  printf("\n");

  //reset integers to unsorted
  Copy(size, integers, testIntegers);
  printf("Unsorted: ");
  printArray(size, testIntegers);


  //sort using asm
  int *buckets = new int();
  BucketSort_(size, testIntegers, buckets);

  //print sorted
  printf("Asm Sorted: ");
  printArray(size, testIntegers);
 return 0;
}
```