Data Structure Alignment

Nikeeta R. Patel

Assistant Professor at Department of Computer Science and Applications, Anand Institute of Information Science, India

Abstract *- The objective of this paper is to comprehensively study the Data Structure Alignment in order to maximize storage potential and to provide for fast and efficient memory access. Aligning data elements allows the processor to fetch data from memory in an efficient manner and thereby improves performance. Alignment refers to the arrangement of data in memory and deals with the issue of accessing data in chunks of fixed size from the main memory.*

Keywords - *Data Structure Alignment, Data Alignment, Alignment in C, Data Structure Padding.*

I.INTRODUCTION

Aligning the data allows the processor to fetch data from memory in an efficient way and thus, increasing the performance. Data alignment is a major issue for all the programmers. It avoids the wastage of memory. In this paper we will see how the data is aligned and what the impact of aligning data on memory is.

II. DATA STRUCTURE ALIGNMENT

Alignment relates to the location of data in memory; means the address at which the data should be stored. Data Structure Alignment can be broken down into two issues: data alignment and data structure padding.

III. DATA ALIGNMENT

Most of the computers require data to be aligned. Data Alignment refers to putting the data at the memory location which is multiple of its word size. Aligning the data increases the performance of the system remarkably and thus making it inevitable for the programmers to resolve this issue.

Both Programmer and Processor view the memory in two different aspects.

A. Programmer view of memory:

Programmers view memory as a simple array of bytes. Or simply said, a large block of memory consisting of number of bytes arranged contiguously.

Programmers View of memory

Fig.1

B. Processors View of Memory:

On contrary, processor views the memory as a block of bytes whose size depends on the word size of the processor. Processor always read from / writes in the memory in the form of 2-, 4-, 8-, 16-, or even 32-byte blocks.

Processors View of memory

Fig.1

Processors take this approach because accessing the address on 1-byte is bit painful as compared to accessing the addresses in blocks of 2, 4, 8, 16, or 32-bytes in one stroke.

IV. ALIGNMENT IN C

The way our C compiler lays out basic C data types in memory is constrained in order to make memory access faster. Memory is arranged sequentially and is byte addressable. Memory is a collection of words of 4 byte long. Storage of the data is

dependent on the word size of the compilers / processors.

Storage of the basic C data type does not start at the arbitrary byte addresses. Instead each data type has a certain alignment requirement except the char data type which can start on any byte address. But 2-byte short data type must start at an even address, 4-byte integer or float data type must start at an address which is divisible by 4 and 8-bytes long or double data type must start at an address divisible by 8.

For example, if an integer variable is allocated an address X which is multiple of 4, then the processor will need only one memory cycle to read the entire integer. But if it is allocated at address which is not a multiple of 4, then the integer spans across two words in memory and thus requires two memory cycle to fetch the entire integer. This incurs in significant performance loss in terms of speed.

Some basic data types are self-aligned in the sense they are stored at the addresses which is multiple of the word size of the processor. Self-Alignment makes the access faster as it makes the processing on the data types much faster as compared to unaligned data types.

V. DATA STRUCTURE PADDING

Structure padding is the process of aligning the data members of structure, in accordance with the memory alignment rules specified by the processor. In order to align the data in memory one or more bits are inserted between the memory addresses.

Let's consider a simple structure in a processor having word size of 4.

```
Struct data_pad
{
        char a;
        char b;
        int c;
        char d;
        short e;
};
```

Without taking the concept of data alignment into consider one might draw the conclusion that the above structure will occupy a total of 9 bytes of memory in a sequential manner.

But this is not the case. In actual this structure will occupy a memory size of 12 bytes, in which 5 bytes are padded in order to align the data. Thus the above layout can be visualized as

```
Struct data_pad
{
char a;          (1-byte)
char b;          (1-byte)
char pad1[2];    (2-bytes)
int c;           (4-bytes)
char d;          (1-byte)
char pad2;       (1-byte)
short e;         (2-bytes)
};
```

Variables a and b are char type, thus can be stored at any data address. Variable c is of integer data type and should be stored at the address which is divisible by 4, thus requiring a padding of 2-bytes. d variable again is of char type and can be stored immediately followed by variable c. But variable e is short and can be stored at the address divisible by 2, thus requiring a padding of 1.

This incurs in memory being wastage and in application with huge amount of data this is significant. We minimize this memory wastage by ordering the structure elements such that the largest element comes first, followed by the second largest, and so on.

```
Struct data_pad
{
        int c;
        short e;
        char a;
        char b;
        char d;
};
```

Thus, aligning data properly helps us in increasing the performance significantly. Also this helps us in speeding up the memory access.

CONCLUSION

Software industry is growing rapidly in recent years and a huge amount of data is collected each day. In order to store this data, we require a huge amount of memory, but we have limited amount of memory. So there is a need to utilize the memory efficiently. By aligning the data we speed up the memory access and thereby increasing the performance.

REFERENCES

1. Knuth, D.E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. Addison-Wesley, Reading, Mass., 1968
2. D. L. Rohrbacher, Advanced computer organization study: Volume I—Basic report; Volume II—Appendices, Apr. 1966.
3. Small Data Structure by Charles Weir, James Noble
4. Horowitz, E., Sahni, S., Rajasekaran, S.: Computer Algorithms/C++. Computer Science Press, New York (1998).
5. Goodrich, M.T., Tamassia, R.: Algorithm Design: Foundations, Analysis,and Internet Examples. John Wiley & Sons, Inc., Hoboken (2002).
6.