Memory Performance Comparison of Heap and Data Segments with Different Compiler Optimizations

Bruno Oliveira Cattelan, Lucas Mello Schnorr
Institute of Informatics
Federal University of Rio Grande do Sul
Caixa Postal 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brazil

Abstract

It is usual for programmers to use at least the O1 optimization level when compiling C programs. However, sometimes this optimization works better with one kind of memory allocation than others. In this paper we show the execution time differences for two programs that rely heavily in memory access. For both programs were made two versions, one that allocated very big arrays in the data segment and the other in the heap. Using the assembly code produced by GCC we then compared the program versions, to try and understand the time execution differences.

1. Introduction

The GCC compiler is widely used in C programming. For this reason, it is imperative that we better understand it's behavior for different memory allocations when using the optimization levels.
This work has been inspired by reading a section in [3], in which was shown that the heap had a bigger cache miss rate than the other kinds of memory. If this was true, execution times would also show this.
To test this differences we created two programs, one that used big arrays and one that used big matrices. For each one we also made a version that allocated the variables in the data segment and one that used the heap. To prevent bias a full factorial approach was used, with the help of a R script. To understand the results, we studied the assembler code generated by GCC.
In methodology we explain how the experiments were executed, along with some details about the machine used and the programs created. In Results we show the resultant behavior of the programs and make some considerations. Finally, in the Conclusion we explain what the results showed us, along with future work.

2. Related Work

Many works have already been done in the study of the GCC optimization levels. A similar work has been done by [1]. Although they used a well known benchmark for their experiments, the work has no distinction between the different memory allocation types that a programmer may use, and their focus was on Integer variables. Our work uses double precision variables, and a closer inspection is made in the resultant program, with a study in the generated assembly code.

3. Methodology

All tests were executed in the following machine:

Parameter	Value
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	32
On-line CPU(s) list:	0-31
Thread(s) per core:	2
Core(s) per socket:	8
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	45
Stepping:	7
CPU MHz:	1202.968
BogoMIPS:	4001.13
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	20480K
NUMA node0 CPU(s):	0-7,16-23
NUMA node1 CPU(s):	8-15,24-31

To prevent bias, we used a full factorial with the size and the kind of memory allocation as factors. This was created using a R script with the DoE package, thoroughly explained in [2]. Also, to be sure that no NUMA effect was interfering with our experiment the following command was used:

```
GOMP_CPU_AFFINITY=0-31 numactl -i all
```

It prevents the OS from moving the threads between the processors and make all allocations following a round-robin policy, rather than by proximity to the processor that is going to use it. In both tests we created two programs, one that used the data segment to store the variables and one that used the heap. They were both compiled using from O0 to O3 optimization levels. Differences in the assembly code were verified using the program "Kompare".

3.1. Array

This very simple program adds the current index divided by the array size to the variable pointed by the index. This is done for the array size times. It is a sequential program, as shown below.

Data Segment code:

```
#include <stdlib.h>
#include "../../lib/hpcelo.h"

double bigArray[100000];

int main(int argc, char *argv[]){
 size_t SIZE = (size_t)atoi(argv[1]);
 HPCELO_DECLARE_TIMER;
 size_t i,j;
 HPCELO_START_TIMER;
 for(j=0;j<SIZE;j++)
  for(i=0;i<SIZE-1;i++)
   bigArray[i] = bigArray[i] + i/SIZE;
 HPCELO_END_TIMER;
 HPCELO_REPORT_TIMER;
}
```

Heap Code:

```
#include <stdlib.h>
#include "../../lib/hpcelo.h"

double *bigArray;

int main(int argc, char *argv[]){
 HPCELO_DECLARE_TIMER;
 size_t SIZE = (size_t)atoi(argv[1]);
 bigArray = (double*)calloc(SIZE, sizeof(double));
 size_t i,j;
 HPCELO_START_TIMER;
 for(j=0;j<SIZE;j++)
  for(i=0;i<SIZE-1;i++)
   bigArray[i] = bigArray[i] + i/SIZE;
 HPCELO_END_TIMER;
 HPCELO_REPORT_TIMER;
}
```

3.2. Matrix Multiplication

To better use the cache, our program multiplied the matrix by first calculating it's transpose. The execution time showed later however is only the multiplication, without the transposing time.

It is a parallel program that uses the openmp library. Since it's code is much more complicated than the Array, we decided to omit it.

4. Results

4.1. Array

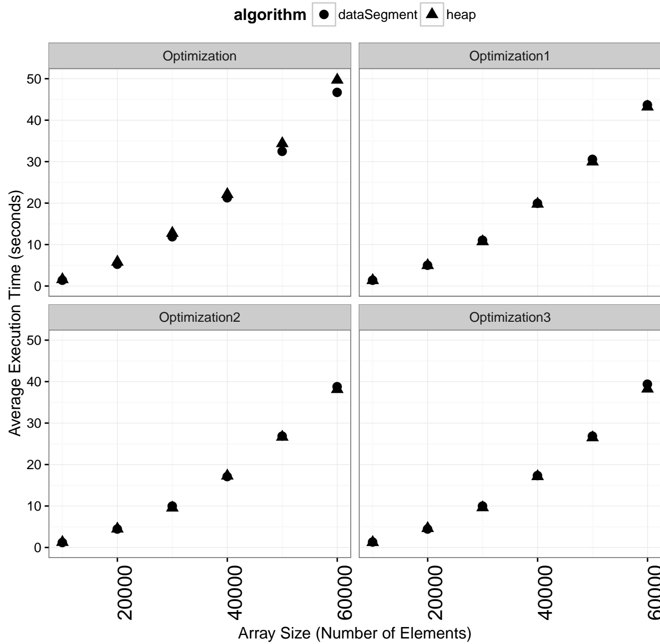

Figure 1. Array Execution Time

The data segment program for O0 shows a better execution time, as shown in 1. However, for O3 the heap is actually the fastest. For simplicity, we compared the assembler code only for O0 and O1. The differences are shown below.

Data Segment:

```
.L6:
  movq   -40(%rbp), %rax
  movsd  bigArray(,%rax,8), %xmm1
```

Heap:

```
.L6:
  movq   bigArray(%rip), %rax
  movq   -40(%rbp), %rdx
  salq   $3, %rdx
  leaq   (%rax,%rdx), %rcx
  movq   bigArray(%rip), %rax
  movq   -40(%rbp), %rdx
  salq   $3, %rdx
  addq   %rdx, %rax
  movsd  (%rax), %xmm1
```

Above we show the main differences in the main loop of the data segment program and the heap program using the O0 optimization level. The later clearly has not only more instructions, but also more memory accesses, which are the parentheses in the code.

Data Segment:

```
.L8:
  movq   %rcx, %rax
  movl   $0, %edx
  divq   %rbx
  testq  %rax, %rax
  js     .L4
  cvtsi2sdq  %rax, %xmm0
  jmp    .L5
```

Heap:

```
.L8:
  movq   bigArray(%rip), %rax
  leaq   (%rax,%rcx,8), %rsi
  movq   %rcx, %rax
  movl   $0, %edx
  divq   %rbx
  testq  %rax, %rax
  js     .L4
  cvtsi2sdq  %rax, %xmm0
  jmp    .L5
```

Above we show the main differences in the main loop of the data segment program and the heap program using the O1 optimization level. Although the heap program still has more instructions than the data segment, in 1 it showed a lower execution time. This might be explained by other differences in the rest of the code:

Data Segment:

```
.L5:
  addsd  bigArray(,%rcx,8), %xmm0
  movsd  %xmm0, bigArray(,%rcx,8)
  addq   $1, %rcx
  cmpq   %rsi, %rcx
  jne    .L8
.L7:
  addq   $1, %rdi
  cmpq   %rdi, %rbx
  jbe    .L2
.L3:
  testq  %rsi, %rsi
  je     .L7
  movl   $0, %ecx
  jmp    .L8
```

Heap:

```
.L5:
  addsd  (%rsi), %xmm0
  movsd  %xmm0, (%rsi)
  addq   $1, %rcx
  cmpq   %rdi, %rcx
  jb     .L8
.L7:
  addq   $1, %r8
  cmpq   %r8, %rbx
  jbe    .L2
.L3:
  testq  %rdi, %rdi
  je     .L7
  movl   $0, %ecx
  jmp    .L8
```

Even though both codes have the same number of instructions, and that except for the ”jne” and ”jb” ones they are all the same, the data segment program has some much more complex addresses to calculate than the heap. This may explain the slightly better execution time of the heap. Also, this shows that the GCC compiler can optimize the heap in a more efficient way than the data segment.

4.2. Matrix Multiplication

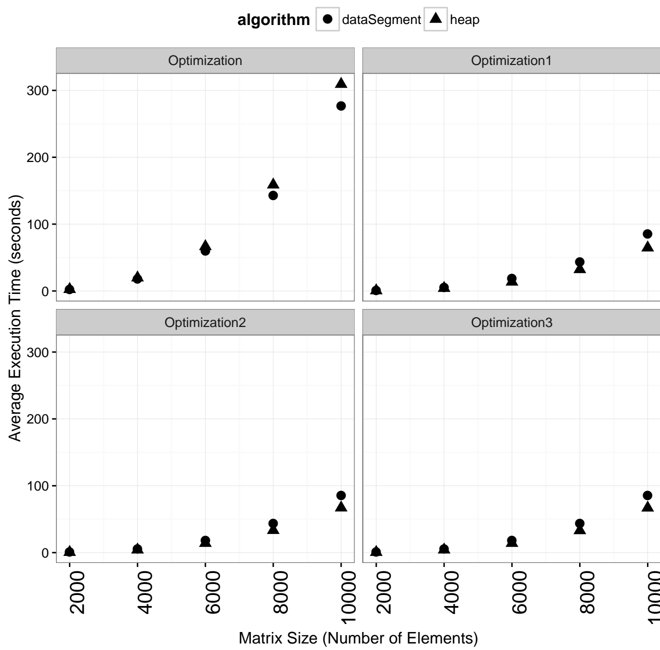

Figure 2. Matrix Execution Time

The program that allocates the matrix in the data segment clearly has a lower execution time than the one that allocates in the heap for O0, as shown in 2. However, as the optimization level is increased the difference becomes the opposite, with the heap being faster than the data segment.
Although this experiment showed the same behavior as the Array, it was made using a much more complex program. Even more, we used such large matrices that they did not even fit inside the L3 cache. With this we intended to rule out any possibility that the cache was responsible for the difference in the execution time of the programs.

5. Conclusions

For higher optimization levels, there seems to be better to use heap allocation. Not only it gives a better flexibility to the program, but also it gives smaller executables and higher efficiency. However, in cases that optimization may not be used, data segment allocation can have a positive impact in the program execution time.
For future work we would like to do a more thorough inspection in both programs, and also in new ones, that not only access the memory in different ways but also study the stack optimization efficiency. To better understand the effects, it would also be interesting to use trace tools like score-p or the intel pcm.

6. Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law n 8248, 1991, and its subsequent updates.

References

[1] R. D. Escobar, A. R. Angula, and M. Corsi. Evaluation of gcc optimization parameters. *Ing. USBMed*, July 2012.
[2] U. Grmping. R package doe.base for factorial experiments. *Reports in Mathematics, Physics and Chemistry*, February 2016.
[3] H. Hadimioglu, D. Kaeli, J. Kuskin, A. Nanda, and J. Torrellas”. *High Performance Memory Systems*. Springer, 2004.