Int. J. Open Problems Compt. Math., Vol. 3, No. 2, June 2010
ISSN 1998-6262; Copyright © ICSRS Publication, 2010
www.i-csrs.org

An Efficient Manual Optimization for C Codes

Mohammed Fadle Abdulla

Computer Science and Engineering Department,
Faculty of Engineering, University of Aden, Yemen
email: al_badwi@hotmail.com

Abstract

Developing an application with high performance through the code optimization places a greater responsibility on the programmers. While most of the existing compilers attempt to automatically optimize the program code at the lower level, manual techniques remain the predominant method for performing optimization at the source code level. Since deciding where to try to optimize code is difficult, especially for large complex applications, the programmer can use his experiences in writing the code, and then he can use a software profiler in order to collect and analyze the performance data of the code. In this work, we have gathered the most experiences which can be applied to improve the style of writing programs in C language as well as we present an implementation of the manual optimization of the codes using the Intel VTune profiler. The paper includes two case studies to illustrate our optimization on the Heap Sort and Factorial functions.

Keywords: *Optimization, Software, VTune profiler, Compiler, Performance.*

1 Introduction

Software applications are designed to work. Hence, the program functionality is the mean objective of the programmer during the phase of development of a new application. The performance consideration may be added to the design later on.
Several different techniques exist by which a given software program can be made to run faster. However, speeding up a program can also cause an increase in

the program size. In the world of mobile systems, the graphics library should be small enough to run on the mobile device without compromising graphics quality. Devices like mobiles, PDAs etc, have strict memory restrictions [1]. So, the mean objective of the performance improving is to write the code in such a way that memory and speed both be optimized.
There are many ways for measuring an application performances. The homegrown timing functions which are inserted into the code are a more effective way to gather performance data. Other most efficient and accurate ways to gather timing data is to use a good performance profilers, which show the time spent in each function of the program and will also provide an analyses based on this data [2,3].
Once performance data has been collected, it needs to be analyzed to find the routines that are taking the majority of the application time, and the loop that appears to be taking more time than it should. These areas are known as "Hotspots". There are a lot of profilers available for detecting the hotspots either in the source file or in the codes of the standard library [4,5,6], and the optimization should be done on these hotspots. The next step is to test the modified code, if still run correctly with the achieved performance improvement. Otherwise, the optimization cycle is repeated for another iteration.
In this work, we implement manual techniques which can be applied to make a C code optimized for speed as well as memory. We illustrate two ways to perform this. First, by applying the most programming practices in writing C code to make it run faster. Secondly, the VTune profiler is used for father locating the hotspots within the program, which will help us to improve them. The paper includes two case studies to illustrate our optimization on the Heap Sort and Factorial functions.

2 Basic Performance Events

3 Manual Performance Improving

4 Evaluation

4.1 Case Study I :Optimizing the Heap Sort Function

4.2 Case Study II : Optimizing the Factorial Function

5 Results

The process of the performance improvement is taken on a sample application that performs the heap sorting which sorts 100,000 random integers. The time duration is taken as 20 seconds with the Interval of 1 msec time. The calculated value of the Sample_After_Value is 2000000.
The optimization is running in three iterations with three versions of the original heap sort program (HeapS), namely, (i) the version Heap_Optimized1 where the FOR loops are optimized, (ii) the version Heap_Optimized2 where the Heap_Optimized1 is further improved with the optimization of the local and general registers allocations, and (iii) the version Heap_Optimized3 where the Heap_Optimized2 is further optimized with branch removal and Inline technique.

Table 1: The modified program Heap_Optimized2

		Module (Process Heap_Optimized2)					
		Adjust	Hsort	Swap	Gen-array	Rand	memSet
1	CPU_CLK samples	31	4	2	1	0	1
2	INST_Retired samples	27	0	4	0	1	0
3	CPU_CLK %	79.49%	10.26	5.13	2.56	0	2.56
4	INST_Retired.ANY %	84.38%	00	12.5	0	3.3	0
5	CPU_CLK events	62000K	8000K	4000K	2000K	2000K	2000K
6	INST_Retired events	54000K	0	8000K	0	0	0
7	% of the Process	86.11%	6.94	5.56	0	1.39	

In the iteration of the modified program Heap_Optimized2, the analysis is for the module Adjust() is as follows:

- *CPU_CLK samples:*
  Out of a total of 39 samples, 31 samples was collected for Adjust()
- *CPU_CLK % :*
  79.49% out of 100% of Timer samples was collected for Adjust().
- *CPU_CLK events:*

The Sample After value for the Timer event is 2000000. Using the formula, Number of samples X Sample After value, the number of occurrences of the Timer event for Adjust() is 62000000.

- *Process % :*

  86.11% out of 100% of the CPU processes consumed in executing the module Adjust(), followed by the module Swap() (which consumed 7.32%), the next positions in the list (which are consumed 2.44%) are shared by Gen_array and Rand() functions.

The performance figures for the different modules within the modifies program Heap_Optimized2 is shown in Table 1. Table 2 illustrates the result for the next modification of the program, namely, Heap_Optimized3. The complete performances view of the three versions of the original HeapSort algorithm is shown in Table 3. The actual improvement in the code performances within each technique for the HeapSort and Factorial algorithms are shown in Table 4.

Table 2: The modified program Heap_Optimized3

		Module (Process Heap_Optimized3)					
		Adjust()	Hsort()	Swapt ()	Gen-rray	Rand.c	memSet
1	CPU_CLK_Core samples	35	2	2	1	1	1
2	INST_Retired.ANY samples	33	0	2	1	0	0
3	CPU_CLK_Unhalted_Core %	83.3%	4.76	4.76	2.38	2.38	2.38
4	INST_Retired.ANY %	91.67%	0	5.56	2.78	0	0
5	CPU_CLK_Core events	70000K	4000K	4000K	2000K	2000K	2000K
6	INST_Retired.ANY events	66000K	0	4000K	2000K	0	0
7	% of the Process	86.37%	2.44	7.32	2.44	2.44%	

Table 3: Performances for the different versions of HeapSort Algorithm.

		Process			
		HeapSort	Heap_ Optimized1	Heap_ Optimized2	Heap_ Optimized3
1	Clockticks samples	58	56	55	11
2	Instructions Retired samples	28	27	29	6
3	Cycles per Instructions (CPI)	1.242	1.22	1.19	0.72
4	Clockticks %	4.26%	4.11	4.04	0.81%
5	Instructions Retired %	4.83%	4.66	5	1.03
6	Clockticks Events	139200K	134400K	132000K	26400K
7	Instructions retired events	67200000	64800000	69600000	14400000

Function	No. of Reduction in Samples						Total Reduction (%)
	Function InLining	Register Allocation	Branch Removal	AND/OR Conv	Loop Termin	Global Variables	
Heap_sort Alg	22	40	44	3	22	22	12%
Factorial Alg	20	38	20	4	15	12	16%

Table 4 : Improvement with different techniques.

6 Conclusions and Open Problems

Most of the compilers automatically optimized the code at a much lower level and performing optimizations specific to a targeted processor. However, the manual optimization performed by the programmer at the source code level remains the predominant method for generating an efficient software code. In this paper, we have gathered the most experiences which can be applied in writing the C code to be optimized for speed as well as memory, and the further using of a profiler for locating the hotspots within the code which will help us to improve them.

We have seen how various small changes in the hotspots of the program can dramatically affect its performance: In particular, instructions inside a loop should be very carefully written so as to minimize loop length. A tool like VTune can help a lot in this process.

However, unix systems impose restrictions for such profiling because of security considerations. Profilers like *gprof* can help in this case with statistics for time spent and the relative frequency of the instructions.

We have shown that by using a good compiler and some manual optimization techniques tested in this paper, programmers can much more easily create high performance applications.

References

[1] Malik P., and Richard S., "Optimization and analysis of performance In Simulation", the 30th conference on winter simulation, (1998), pp 1689-1692.

[2] Creating High Performance Embedded Applications Through Compiler Optimizations, White Paper, (2005).

[3] A. Myers, "Practical mostly-static information flow Control " , In proceeding of ACM SIGPLAN_SIGACT symposium on Principles of Programming languages, San Antonio, (1999).

[4] Amjad and Neelakanth N., A performance optimization for C/C++ systems that Employ Time-stamping, (2004).

[5] L. Liu, S. Rus, "A Context Sensitive Performance Adviser for C++ Programs", In Proceeding of the 2009 International Symposium on Code Generation and Optimization", (2009), pp 265-274.

[6] Jing Xu," Rule_Based Automatic Software Performance Diagnosis and Improvement," In Proceedingsof the 7th International Workshop on Software and Performance, Princeton, USA, (2008), pp 23-26.
[7] T. Moseley, D.Grunwald , R. Ramanujan, and R. Peri, “LoopProf: Dynamic Techniques For Loop Detection and Profiling", Technical Report.
[8] S. Gurumani, A. Milenkovic,"Execution Characteristics of SPEC CPU2000 Benchmarks: Intel C++ Vs Microsoft VC++", In proceeding of ACM, (SE'04), Huntsville, USA, (2004).
[9] Rudy Chukran, “ Accelerating AIX: Performance Tuning for Programmers and System Administrators”, Addison Wesley, (1998).
[10] Budge, K.G., "C++ optimization and excluding middle-level code", Conference on object-oriented numeric, OONSKI '94, US (1994).
[11] M. Anderson, “Continuous Profiling Where Have All the Cycles Gone”, SRC Technical Note, (19 97).