Volume 3, No. 3, May-June 2012

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Optimizing Performance of Source Code for Real Time System

R A Tiwari
Department of Information Technology
V.Y.W.S. POLYTECHNIC
Badnera,India
softyraja@gmail.com

Miss R R Tuteja
Department of Computer Technology
P.R.M.I.T
Badnera,India
ranu.tuteja@gmail.com

M A Pund
Department of Computer Technology
P.R.M.I.T
Badnera,India
mapund@gmail.com

M R Dhande
Department of Information Technology
V.Y.W.S. POLYTECHNIC
Badnera, India
mrdhande@gmail.com

Abstract: To optimize the performance of a program for real time system does not always mean what we might think. It is not just a matter of outright speed; sometimes it is about tuning the code and data so that it fits into a small memory footprint. It would be hard-pressed to find a programmer that does not want to make programs that run faster, regardless of the platform. Embedded programmers are not exception for that some take an almost fanatical approach to the job of optimizing their code for performance. As hardware becomes faster, cheaper, and more copious, some argue that performance optimization is less critical--particularly people that try to enforce deadlines on software development. Now a days most advanced hardware, combined with the latest in compiler optimization technology can not come even close to the performance benefits that can be attained by fixing some small probes at the beginning of programs or even going with an entirely different and much faster design.

Keywords: performance optimization; real time system; pre compilation; processor time;

I. INTRODUCTION

At the time of programming we have several ideas that can be applied to programs that will make them to perform better. By keeping these ideas in mind while writing the C code, we can expect better and faster programs. When we work to optimize performance, several different things are arise in picture which we need to consider. One thing is the absolute amount of time it takes the software to complete a given task. To understand imagine a web server where web server serves the client requests perfectly well, there can be a delay of a few seconds before the server begins to responding (send the pages) to the client every time. In such a case, the web server is failing to perform adequately in terms of the total time required to complete the task.

Another thing to consider is the amount of processor time required by a program. First we define what is processor time? It is a measure of the time spent by the computer's processors to execute the code. Many programs tend to spend most of their time waiting for something to happen--input to arrive, output to be written to disk, etc. While waiting, the processor will usually be serving other requests and hence the program is not using processor time. However, some programs may be primarily processor bound programs and for such programs, a savings in the amount of processor time required may result in a substantial savings in absolute time. It is important to note here that if your program uses a lot of processor time, it can slow down all the processes on your system. The processor time can further be separated into system and user time. The system time is the amount of processor time used by the kernel on your behalf. This could accrue by calling functions such as open() and fork(). The user time (i.e., amount of processor time used by your program) might be used by string manipulations and arithmetic. A third thing to consider for performance is the time spent doing I/O. Consider an example some programs such as network servers, they spend most of their time on handling I/O. Other programs spend little time with tasks related to I/O. Thus, I/O optimization can be very critical with some projects and completely unimportant with some others.

The performance optimization basically consists of the following steps

a. Define the performance problem.
b. Identify the bottlenecks and carry out a root cause analysis.
c. Remove the bottlenecks by appropriate methodologies.
d. Repeat steps 2 and 3 until we have a satisfactory resolution.

It is important to note here that bottlenecks occur at various points in a program.

Determining the bottlenecks is a step-bystep procedure of narrowing down the root causes. Performance optimization is relatively a complex process that requires correlating many types of information with source code to locate and analyze performance problem bottlenecks. When focusing on performance optimization, a programmer needs certain tools to measure and monitor the situations as well as to identify the bottlenecks. On Linux, various tools are available to do this. gprof and gcov utility is provides a snapshot of the program at the moment it is being viewed. In next three section we

CONFERENCE PAPER
National Conference on Information and Communication Technology for Development
Organized by PRMITR, Amravati (MS) India
http://mitra.ac.in/forthcoming.html

discuss some performance bottlenecks that pin point by GCC utility and solution on same at precompilation time [1].

II. PROBLEM WITH LOOP, DATA TYPE AND BLOCK SIZE

Let us first understand the performance problems caused by loops. Loops magnify the effects of otherwise minor performance problems. This is because the code within the loop will get executed several times. Always make sure to move the code outside the loop that need not be executed each time (dead code). Also look for minimizing dependency on loops too whenever it possible.

Let us consider the following code segment (test.c.gcov)

```
1 : main()
{
int i,k;
int a[50];
52 : for(i=1;
i<=50;
50 : i++)
{
1375: for(k=50;
k>=i;
1275: k--)
{
1275: a[i]=i;
1275: printf("%d",a[i]);
}
50 : printf("\n");
}
1 : }
```

If we carefully observe this code, we can see that several things can be possible to apply around the loop. In given segment we use for loop which is replaced by the combination of do-while and goto. It makes much sense when we replacing for loop with its most suitable counterparts. After replacement of for loop, code segment take 51 less iterations to execute, saves processors time on iterative line executions in program, without affecting output. Results are motivating, because test code is just eleven line long, imagine code of programmers who bounds several hundreds of code line inside loop, take an almost fanatical approach to the job.

When we talk about optimizing the performance, we need to make sure that unless there is an absolute need to use float variable, we should never try to use floating point data types such as "float" and "double." This is because of the fact that they take more space to store and time to calculate than do their integer counterparts like short. Also, if we have a function that is called very frequently, it is better to declare it as "inline". Also, another way to improve the performance is to increase the block size. As we know, many operations are done on blocks of data. By increasing the block size, we will be able to transfer more data at once. This will reduce the frequency with which we call more time consuming. Then someone ask question what is the optimal size of block to hold data?

III. TAKE CARE OF EXPENSIVE CALLS

It is clear that when we are interested in optimizing the code, we always want to get rid of the expensive operations (that take more time) with inexpensive calls.

System calls in general are expensive operations. Let us have a look at some of the expensive system calls:

- ***a.*** ***Fork:*** A fork system call is very useful. It isn't slow, but if we use it frequently, it can add up. Consider a scenario where a web server might fork for each new request. This is not a good practice, and select() can be used for multiplexing.
- ***b.*** ***Exec:*** This is one used immediately after a fork. This call can be very expensive as the new program will have to a lot of initialization such as loading libraries, etc.
- ***c.*** ***System:*** This invokes a shell to run the specified command and invoking a shell can be quite expensive. Therefore, frequently using a system is definitely a bad idea.

If we come across a code piece such as system ("/tiwary's/proc/memstat"); we can see how expensive this is. The program first has to fork and execute the shell. The shell needs to do initialization and then it forks and executes /proc, definitely not a piece of code to desire.

The first step in getting the system tweaked for both speed and reliability is to chase down the latest versions of required device drivers. Another useful key is to understand what the bottlenecks are and how they can be taken care of. We can come to know about the various bottlenecks by running various system monitoring utilities, such as the gprof and gcov command.

IV. OPTIMIZING DISK ACCESS

It is always worth giving attention to disk access. There are various techniques that can produce significant improvements in disk performance.First, read up on the hdparm command and you will notice that it sets various flags and modes on the IDE disk driver subsystem. There are two options we need to look at the -c option can set 32 bit I/O support and the -d option enables or disables the using_dma flag for the drive. In most cases, this flag is set to 1, but if yours hasn't, then you are going to suffer from performance issues. Try changing it by placing a command like this

hdparm –d 1 /dev/hda

at the end of the /etc/rc.d/rc.local file.

Similarly,

hdparm –c 1 /dev/hda

at the end of /etc/rc.d/rc.local file will set the support for 32 bit I/O.

hdparm –B

option –B use to set power management to low (aggressive) or high (better performance)[1,2].

V. GNU PROFILER (GPROF)

After we have taken enough measures in optimizing our code, the compiler can be helpful with optimization as well. Two tools that we can use to analyze program's execution is

CONFERENCE PAPER
National Conference on Information and Communication Technology for Development
Organized by PRMITR, Amravati (MS) India
http://mitra.ac.in/forthcoming.html

the GNU profiler (gprof) and coverage (gcov). With these, we can come to know where the program is spending most of its processors time. With profile information we can determine which pieces of program are slower than expected. These sections are definitely good candidates for to be rewritten so that program can execute faster. The profiler collects data during the execution of a program. Profiling can be considered as another way to learn the source code.

The following are the requirements to profile a program using gprof

a. Profiling must be enabled when compiling and linking the program.
b. A profiling data file is generated when the program is executed.
c. Profiling data needs to be analyzed.

For you to use this gprof utility, the package must be installed on your system. In order to analyze the program with gprof, we need to compile the program with a special option. Assuming that we have a program test.c, the following can be used to compile it

$ gcc –Wall -c –pg test.c
$ gcc –Wall –pg test.o
To link, linker with input file.
$ gprof -b a.out

About command use to display flat profile. Note here that –pg option enables the basic profiling support in gcc. The program will run somewhat slower when profiling is enabled. This is because of the fact that it needs to spend time in collecting data as well. The profiling support in the program creates a file named gmon.out in the current directory. This file is later used by gprof to analyze the code.

We can run the following command to get the output (which we have redirected to a file):$ gprof a.out gprof is useful not only to determine how much time is spent in various routines, but it also tells you which routines invoke other routines. By using gprof, we will be able to know which sections of our code are causing the largest delays. Analyzing the source code with gprof is considered as an efficient way determining which function is using a large percentage of the overall time spent in executing the program.

Another one, gcov is a test coverage program. Use it in concert with GCC to analyse your programs to help create more efficient, faster running code and to discover untested parts of your program.You can use gcov as a profiling tool to help discover where your optimization efforts will best affect your code. Profiling tools help you analyse your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

a. How often each line of code executes
b. What lines of code are actually executed
c. How much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. gcov helps you determine where to work on optimization. It use your source code and create graph (*.gcno) and data (*.gcda) files for analysis. Use $gcov test.c then $cat test.c.gcov One can use gcov along with, gprof, to assess which parts of your code use the greatest amount of computing time [1,3,4].

VI. CONCLUSION

Efficient analysis and optimization methods are needed and can be developed for the implementation of embedded programming. Program optimization for embedded system demands uses more effective optimization techniques. Off course Compilers will increase the efficiency of these applications and will solve major bottlenecks regarding memory reference and speed in real time systems. But hot spots in program identified and properly handle by programmer at code development stage than its impact seen as dramatic change in the performance of application. In this paper we introduce a straight forward approach which helps to programmer to highlight major culprits of best performance in program. This can be easily done by utilizing existing GNU compiler collection on Linux. Until now, we are applied different profiling options to spot bottle necks in C program. The next step is to implement front end using GCC that provide optimal optimization levels to program.

Typical compiler present a variety of optimization level to program where each level represent some fixed sequence of optimization's that compiler applies to the program. We tried several sequences in different phases that prevent in reduced sequence of proposed front end.

VII. REFERENCES

[1]. Philip G. Ezolt, “Optimizing Linux Performance: A Hands-On Guide to Linux performance tools”, Prentice Hall PTR GNU Manual

[2]. http://www.gnu.org/software/binutils/ manual/ Linux Tools http://www.yolinux.com/TUTORIALS/LinuxTutorialOptimization.html

[3]. GNU Compiler Collection

http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

CONFERENCE PAPER
National Conference on Information and Communication Technology for Development
Organized by PRMITR, Amravati (MS) India
http://mitra.ac.in/forthcoming.html