1 – Systems Performance Tuning

List of Slides

1  Systems Performance Tuning
3  My Program Runs Too Slow!
4  What is Performance?
5  Time as a Performance Metric 1 of 2
6  Time as a Performance Metric 2 of 2
7  Rate as a Performance Metric
8  Response Time and Throughput
9  Performance Tuning
10  Observing the Machine
11  But its the software
12  Intro to Profilers
13  Types of Profiling Tools
14  The getrusage system call
<table>
<thead>
<tr>
<th></th>
<th>Performance Counters 1 of 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Performance Counters 2 of 2</td>
</tr>
<tr>
<td>18</td>
<td>Using getrusage</td>
</tr>
<tr>
<td>19</td>
<td>Some Common Unix Profilers</td>
</tr>
<tr>
<td>20</td>
<td>How Unix Statistical Profilers Work</td>
</tr>
<tr>
<td>21</td>
<td>Other Popular Profilers</td>
</tr>
<tr>
<td>22</td>
<td>An example application</td>
</tr>
<tr>
<td>23</td>
<td>Refining binary search 1 of 3</td>
</tr>
<tr>
<td>24</td>
<td>Refining binary search 2 of 3</td>
</tr>
<tr>
<td>25</td>
<td>Refining binary search 3 of 3</td>
</tr>
<tr>
<td>26</td>
<td>General Speedup Tricks</td>
</tr>
<tr>
<td>27</td>
<td>I/O Related Speedup Techniques</td>
</tr>
<tr>
<td>28</td>
<td>Memory Related Speedup Techniques</td>
</tr>
<tr>
<td>29</td>
<td>Profiler Command Syntax</td>
</tr>
<tr>
<td>30</td>
<td>Use of prof and gprof</td>
</tr>
<tr>
<td>31</td>
<td>Using gcov 1 of</td>
</tr>
<tr>
<td>32</td>
<td>Using gcov 1 of</td>
</tr>
<tr>
<td>33</td>
<td>Use of pixie</td>
</tr>
</tbody>
</table>
2 – My Program Runs Too Slow!

You are sitting patiently waiting for a program to finish, yet somehow it seems to take forever. Worse, you know you will wind up running this program many, many, ... many times.

Do you:

1. Give up
2. Spend thousands of dollars on new computers
3. Hire an expensive consultant
4. Try a few tricks to speed things up
3 – What is Performance?

Performance denotes computational speed measurements and is sensitive to scheduling efficiency.

Jain [4] categorizes classic performance measures:

1. Time (e.g. response time)
2. Rate (e.g. throughput)
3. Resource (e.g. utilization)
3.1 – Time as a Performance Metric 1 of 2

There are intervals in user/computer interaction:

1. Response time — Time until either:
   (a) System Begins Response, or
   (b) System Ends Response

2. Reaction time — User Latency

3. Turnaround Time — Process Duration

4. Think Time — User Latency

5. Stretch Factor — Reaction of system to a fixed increased load
3.2 – Time as a Performance Metric 2 of 2

(a) Instantaneous request and response.

(b) Realistic request and response.

Definitions of Response Time
3.3 – Rate as a Performance Metric

Owners of computers want to know the throughput \( \frac{\text{Operations}}{\text{time}} \) of the system:

1. Processing — MIPS, MFLOPS, GFLOPS, TFLOPS, Sustained, Peak
2. I/O — \( \frac{\text{Transactions}}{\text{Second}} \), Bandwidth
3. Network/Data Communications —
   Bandwidth, bps, fps, Mbps, pps
4. Efficiency — Measures \( \frac{\text{Achieved Throughput}}{\text{Available Throughput}} \).

Figure 1: Efficiency of a Multiprocessor System
Figure 2: Response Time vs. Throughput

Note that Throughput $\neq \frac{1}{\text{Response Time}}$ (e.g., consider pipelining).
5 – Performance Tuning

Bentley [2] provides an overview of performance tuning techniques. The basic steps in performance tuning are:

1. Understanding the application.

2. Determining where the source code spends its time.

3. Improving the performance of the slowest parts as per Amdahl’s Law [1, 3].
6 – Observing the Machine

Before trying to fix things, it is a good idea to make sure that it is your software that is slow, and not the machine (or that other user’s job).

Some good tools are (under Linux):

- **top** - command line tool to show system load
- **KNetload** - A KDE network traffic monitor (shows total inbound/outbound traffic).
- **xosview** - based on SGI’s grosvew, shows processor, memror, swap, pager, disk and interrupt activity graphically.
- **xload** - tends to show the systems load over time, more historical than xosview.
- **ktop** - KDE task Manager, has a nice performance meter and top display.
7 – But its the software

Suppose that you find that other users are not to blame, then run your software while observing it, using the tools already mentioned. One good tool is the Unix is the shell’s time command.

The following csh example has the programs output snipped

```
slowpoke:~/talks/profsrc: time bs1gp
60.720u 0.050s 1:01.21 99.2% 0+0k 0+0io 111pf+0w
```

Under bash, it looks like:

```
bash-2.03: time bs1gp
total_result = 5.23776e+09 Average position = 511.5
Command exited with non-zero status 52
60.04user 0.11system 1:01.16elapsed 98%CPU
(0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (112major+19minor)pagefaults 0swaps
```
8 – Intro to Profilers

Profilers are tools for measuring code performance.

Software based methods are *intrusive* in that they increase the very resource usage that we want to measure.

The usual method for this is:

1. Adding instructions to the software to obtain the measurements (often with the compiler’s help).

2. Choosing a set of “representative inputs” (hard to do!).

3. Running the software over its representative inputs, recording the measured values.

4. Analyzing the measured values (also hard!, see Jain [4]).
9 – Types of Profiling Tools

Profilers are tools for measuring code performance, and fall into the following categories:

1. Source Level Profilers: Often compiler supported
   (a) Statistical Measurements — Sample the clock periodically
   (b) Invocation Counts — Count how many times each line is executed

2. Hardware Performance Counters — Measure page faults, cache misses, integer/fp instructions executed, etc.

3. Resource usage measurements — Similar to performance counters, requiring less hardware support.
10 – The getrusage system call

The `getrusage` system call is how the shell’s time facility gets its information.

Its syntax is:

```c
int getrusage (int who, struct rusage *usage);
```

The `who` parameter can be:

- `RUSAGE_CHILDREN` — measures this process and all its children
- `RUSAGE_SELF` — measures only this process (ignores children)
The usage parameter points to a user allocated:

```c
struct rusage{
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss; /* maximum resident set size */
    long ru_ixrss; /* integral shared memory size */
    long ru_idrss; /* integral unshared data size */
    long ru_isrss; /* integral unshared stack size */
    long ru_minflt; /* page reclams */
    long ru_majflt; /* page faults */
    long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
    long ru_msgsnd; /* messages sent */
    long ru_msgrcv; /* messages received */
    long ru_nsignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};
```
Most modern processors have support for counting such events as:

- TLB misses
- Cache misses (L1, L2, etc.)
- Page Faults
- Integer instructions executed
- Floating point instructions executed
- Number of processor cycles used
- Branch Prediction Misses

And other such stuff is kept in architecture dependent counters on the processor.

The way to access these counters is via a system call and/or a higher level utility.
12 – Performance Counters 2 of 2

The following are performance counter based tools:

- SGI IRIX (MIPS R10000) — perfex
- FreeBSD (Intel 586 and later models) — perfmon
- Linux — Not yet standardized
- PAPI — Portable Application Performance counter Interface, at UTK (Jack Dongarra’s group), see:
  http://icl.cs.utk.edu/projects/papi/
- Rabbit — Don Heller at NASA Ames, see:
  http://www.scl.ameslab.gov/Projects/Rabbit/

I think Rabbit and PAPI are Linux Kernel patches (there is no clear winner yet).
13 – Using getrusage

Typically the way to use getrusage is something like:

```c
struct rusage before, after;
int status;

status = getrusage(RUSAGE_SELF, &before);
// status checking omitted for clarity

// The performance measured code
status = getrusage(RUSAGE_SELF, &after);

print (after - before);
```
14 – Some Common Unix Profilers

Most versions of Unix have the following profilers:

- **prof** — statistical sampling
- **gprof** — statistical, keeps a call-graph using stack frame information
- **gcov** — modeled on SUN’s tcov, test coverage analyzer (line counts based).

Both of these use compiler embedded information from monitor and profil calls. On some machines (e.g. SGI IRIX) prof decodes input generated by other profilers (e.g. pixie).
15 – How Unix Statistical Profilers Work

The Unix profilers call profil:

```c
int profil(char *samples, int size,
           int offset, int scale)
```

Where the parameters are:

- **samples** — a pointer to a list of 16 bit bins in the histogram
- **size** — the size the buffer pointed to samples, in bytes
- **offset** — the lowest program counter (pc) address to sample.
- **scale** — each bin measures the time spent in a region $2 \times \frac{64KB}{\text{scale}}$ bytes of data, where scale is a power of 2 between 1 and 64KB typically.
16 – Other Popular Profilers

Other profilers tend to be architecture/compiler tool specific.

- tcov — On SunOS, can do line counts by basic block
- pixie — On SGI IRIX, part of the SpeedShop package generates exacting basic block line counts and operation counts.
- speedshop — A newer SGI performance tuning kit which includes pixie as a tool
- gcov — the GNU gcc/g++ test case coverage analyzer

Using the -a flag on gcc/g++ will produce a basic block counts profile in bb.out.
17 – An example application

The application in bs1.c gives basically does a lot of binary searches in a random order on a list of 1024 numbers.

This application is meant to use the approaches described in Bentley’s article.

Our profile, of the application bs1.c indicates that it is spending more than 80% of its time in the binary search routine my_bsearch, which corresponds to:

```
L := 1; U := N
loop
  /* Invariant: if T is in X, it is in X[L..U] */
  if L > U then
    P := 0; break
  M := (L+U) div 2
  case
    X[M] < T:  L := M+1
    X[M] = T:  P := M; break
    X[M] > T:  U := M-1
```

Figure 3: Traditional Binary Search
18 – Refining binary search 1 of 3

Instead of taking any occurrence of value \( T \) in the array \( X \), we modify the program to locate the first instance. This code now uses only one comparison per iteration in the loop body. The source is in bs2.c.

\[
L := 0; U := N+1 \\
\text{while } L+1 \neq U \text{ do} \\
\quad /* Invariant: } X[L] < T \text{ and } X[U] \geq T \text{ and } L < U */ \\
\quad M := (L+U) \text{ div } 2 \\
\quad \text{if } X[M] < T \text{ then} \\
\quad \quad L := M \\
\quad \text{else} \\
\quad \quad U := M \\
\quad /* assert } L+1 = U \text{ and } X[L] < T \text{ and } X[U] \geq T */ \\
\quad P := U \\
\quad \text{if } P > N \text{ or } X[P] \neq T \text{ then } P := 0
\]

Figure 4: Binary Search Refinement 1
19 – Refining binary search 2 of 3

By constraining the interval to search iteratively to be an exact power of 2, the search remove some operations from the loop body (a branch and an addition, and halves an exact power of 2 which is fast).

```plaintext
I := 512; L := 0
if X[512] < T then L := 1000+1-512
while I ≠ 1 do
    I := I div 2
    if X[L+I] < T then
        L := L+I
    /* assert I = 1 and X[L] < T and X[L+I] >= T */
    P := L+1
    if P > 1000 or X[P] ≠ T then P := 0
```

Figure 5: Binary Search Refinement 2
20 – Refining binary search 3 of 3

Unrolling the loop in the previous example gives yet more speedup, as seen in bs5.c.

```c
L := 0
if X[512] < T then L := 1000+1-512  
if X[L+256] < T then L := L+256  
    /* assert X[L] < T and X[L+256] >= T */
if X[L+128] < T then L := L+128
if X[L+64]  < T then L := L+64
if X[L+32]  < T then L := L+32
if X[L+16]  < T then L := L+16
if X[L+8]   < T then L := L+8
if X[L+4]   < T then L := L+4
if X[L+2]   < T then L := L+2  
if X[L+1] < T then L := L+1  
P := L+1
if P > 1000 or X[P] != T then P := 0
```

Figure 6: Binary Search Refinement 2
21 – General Speedup Tricks

Several well known techniques to get (real or apparent) speed up are:

1. Pick an efficient algorithm for your problem size. Not only does the big $O$ notation matter, but constants count too.

2. For divide and conquer approaches, using an algorithm with poor big $O$ complexity and low constants for small parts can improve run time.

3. For numerical methods, try to use a method with fast convergence (and make sure your problem is well suited for it).

4. Identify the slow part to optimize.

5. Know your application (e.g. don’t optimize the idle loop!).
Several well known techniques to get (real or apparent) speed up are:

1. Prompt the user immediately and process while pending user input.
2. Use asynchronous I/O or data communication when possible.
3. Remove/Tune I/O or Data Communication - they tend to be several orders of magnitude slower than regular processor operations.
4. Run on a local disk, not a network mounted volume.
5. Don’t fill the disk all the way, the OS can place data better if there is free space.
6. On some versions of Unix, the /tmp directory is in virtual memory, not on disk. Filling up /tmp can use up all your memory too.
23 – Memory Related Speedup Techniques

1. Ensure Locality of Reference - This uses the cache and reduces thrashing to disk.
2. Don’t waste memory - Initializing and allocating unused memory is expensive!
3. Use the minimum necessary precision to represent data.
4. Replace recursion with iteration, it is cheaper.
5. Pass small unmodified structures by value.
6. Pass large structures by address/reference.
7. Sometimes (but not always!) it is faster save a value than recompute it.
24 – Profiler Command Syntax

In general using a profiler involves the following steps:

1. Generate executable code with profiling instructions.

2. Run the code and generate profiling information.

3. Generate an execution profile using the measurements and the source code.
24.1 – Use of prof and gprof

Recall that the Unix statistical profilers are prof and gprof.

1. Insert profiling calls into the executable — the compiler typically does this:
   (a) For prof Unix C compilers have a command line options:
       \[ cc \ -g\ -p\ \text{prog.c} \]
   (b) and for gprof, the \(-pg\) is profiling, \(-ax\) is for line numbering:
       \[ gcc\ -g\ -pg\ -ax\ \text{prog.c} \]

2. Gather statistics — running \texttt{a.out} will produce a data file named \texttt{mon.out} (sometimes gprof creates \texttt{gmon.out}).

3. Create an execution profile — run prof or gprof. (syntax varies by Unix Version). On FreeBSD and Linux prof is unsupported and the syntax is:
   \[ \texttt{gprof\ a.out\ gmon.out} \]
24.2 – Using gcov 1 of

Gcov gives (in my opinion) more accurate counts of per line execution than gprof, and gives branch frequency statistics.

- Insert profiling calls into the executable — the compiler does this:
  
gcc -g -ftest-coverage -fprofile-arcs -o prog prog.cpp
- Gather statistics — running ./prog will produce a data files named
  - prog.bb — Raw Basic Block Information.
  - prog.bbg — Basic Block Graph Information.
  - prog.da — Branch Statistics.
24.3 – Using gcov 1 of

- Create an execution profile — run gcov, I like to get branching statistics, so I use the -b option.

```
gcov -b prog.cpp
```

which will create a human readable file `prog.cpp.gcov`. 
On SGI IRIX the powerful pixie profiler can produce extensive profiling statistics.

To use pixie:

1. Insert profiling calls into the executable —
   (a) Be sure to compile symbolic debugging information into your executable (use the -g flag for compiling and linking).
   ```
   cc -g prog.c
   ```
   If you want optimizations level 3 and up can confuse pixie, so it is better to use optimization level 2 or lower.
   ```
   cc -g -02 prog.c
   ```
   (b) Run pixie on the executable:
   ```
   pixie a.out
   ```
   which will generate a pixie version of each shared library (e.g. libc.so.1.pix32) used and a.out.pixie.

2. Gather Statistics — run a.out.pixie,
creating a file a.out.Counts.

3. Create an execution profile — use the prof program to decode the counts file. The example in class shows the lines with the highest number of counts:

prof -h a.out.pixie a.out.Counts

To get the source (plus disassembly)

prof -S a.out.pixie a.out.Counts
References


