Lab 1 Report

Timothy Dee, Brent Barth

October 11, 2015

Abstract

This report contains a description and analysis of the findings during Lab 1 of CprE 458, Real Time Systems. Contained herein is all relevant program code, and the results from having run this code.

1 Introduction

This paper has sections discussing program code, energy consumption data analysis, and analysis of computation time. Each of these sections discuss design decisions and why those decisions were made. TODO

2 Part 1 - Code

2.1 High Performance Mode

Listing 1 describes our implementation of high performance mode. This code was straightforward due to its similarity with low performance mode.

2.2 Dynamic Frequency Scaling Mode I

Listing 2 describes our implementation of dynamic frequency scaling mode I. This mode varies the frequency range according to according to the CPU load. Lower processor frequencies will conserve power while higher processor frequencies will allow for increased performance. Certain CPU loads require that the processor be set to high performance mode or low performance mode. To accomplish this we use the performClick() method attached to the low performance mode and high performance mode buttons. This has the same effect as the user pressing the button which corresponds to these modes.

2.3 Dynamic Frequency Scaling Mode II

Listings 3 and 4 show our implementation of dynamic frequency scaling mode 2. For this mode we though about the phone states which affect the scarcity of power. We determined that the screen state (on or off) and the charging state of the phone were significant enough to be determining factors in how we choose to manage the processor frequency.

In the case of the screen state the following train of logic was applied. If the screen is on, then we are likely to be interacting with an application. If we are interacting with an application than we will certainly benefit from a more responsive device. Therefore we elect to put the phone in a higher power state while the screen is on. We save energy by setting the phone in a lower power state when the screen radio is off. The decision making for the charging state is straightforward. If the phone is charging then we don't need to worry about power consumption, even if the battery is low.

2.4 Dynamic Frequency Scaling Mode Mixed

Listing 5 shows our implementation of dynamic frequency scaling mode mixed. For this mode we choose to use the charging state, battery charge percent, and cpu load as determining factors in our algorithm. All of these factors have a significant impact on how much we care about our energy usage. In many cases we are willing to sacrifice energy in exchange for performance.

As discussed previously, the charging state impacts our decisions in a straightforward way. If the device is being charged than we do not have to worry about power consumption, and we can activate high performance mode. When the device is not charging set a frequency range as a function of charge percent and cpu load. We set load fraction to be an integer ranging between 0 and 12 depending on the cpu load. A load fracting of 0.0 yields 100000 Khz processor frequency while a load fraction of 12 yields 1300000 Khz processor frequency. load fraction increases linearly with cpu load and decreases linearly with battery percent. battery percent causes the program to pretend like the cpu load is smaller than reality.

2.5 Measuring Computation Time

Listing 6: Measuring Computation Time - Intense Task Launcher

```
private long result_factorial;
private void intense_computation(){
```

Listing 1: High Performance Mode

```
PMbutton2.setOnClickListener(new View.OnClickListener() {

     @Override
     public void onClick(View v) {

       OPERATIONmessage("[High Performance Mode]
           ###########################################");
       //TODO Please program for High Performance Mode here (done)

       DATAname = "1300000"; // Setting up the minimum frequency 1300 Mhz
       DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
       ChangeCPUinfor(CPUname, DATAname, DATAaddress);
       DATAname = "1300000"; // Setting up the maximum frequency at 1300 MHz
       DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
       ChangeCPUinfor(CPUname, DATAname, DATAaddress);

       CPUname = "High Power";
       DATAname = "current min frequency";
       DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
       ReadCPUinfor(CPUname, DATAname, DATAaddress);
       DATAname = "current MAX frequency";
       DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
       ReadCPUinfor(CPUname, DATAname, DATAaddress);

       OPERATIONmessage("[High Performance Mode]
           ###########################################");


     }
    });
```

Listing 2: Dynamic Frequency Scaling Mode I

```java
private void setDFS_1(double cpu_load){
    // determine if I should go to a power mode
    if(cpu_load < .2){
      // go to low performance mode
      PMbutton1.performClick();
    }else if(cpu_load > .9){
      //go to high performance mode
      PMbutton2.performClick();
    }else{
      // set the frequency range between 51 Mhz and 1.3 Ghz
      DATAname = "51000"; // Setting up the minimum frequency 51 Mhz
      DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
      ChangeCPUinfor(CPUname, DATAname, DATAaddress);
      DATAname = "1300000"; // Setting up the maximum frequency at 1300 MHz
      DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
      ChangeCPUinfor(CPUname, DATAname, DATAaddress);

      CPUname = "Dynamic Mode";
      DATAname = "current min frequency";
      DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
      ReadCPUinfor(CPUname, DATAname, DATAaddress);
      DATAname = "current MAX frequency";
      DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
      ReadCPUinfor(CPUname, DATAname, DATAaddress);
    }
  }
```

Listing 3: Dynamic Frequency Scaling Mode II

```java
 private void setDFS_2(){
   // get the battery level
   IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
   Intent batteryStatus = this.registerReceiver(null, ifilter);

   int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
   int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

   float battery_percent = level / (float)scale;

   // determine whether the screen is on
   boolean is_display_on = false;
   DisplayManager dm = (DisplayManager) this.getSystemService(Context.DISPLAY_SERVICE);
   for (Display display : dm.getDisplays()) {
     if (display.getState() != Display.STATE_OFF) {
       is_display_on = true;
     }
   }

   // get the charging state
   int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
   boolean is_charging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
       status == BatteryManager.BATTERY_STATUS_FULL;

   //(continued on next page)
```

Listing 4: Dynamic Frequency Scaling Mode II (continued)

```
// determine if we are charging
   if(is_charging){
     // if we are charging we don't care about power usage.... set to high performance
         mode
     setHighPerformanceMode();
     Log.d("high","performance");
   }else if(battery_percent < .3) {
     // we want to conserve energy because we are almost out of it, set to low
         performance mode
     setLowPerformanceMode();
     Log.d("low","performance");
   }else{
       // if we're not charging and not low battery we have some decisions to make
       // if the wireless radio is on we are likely doing something online.
       // If we are doing something we will like a more responsive device
       // so allow the processor to vary between two fairly high frequency states
       if (is_display_on) {
         // set to fairly high processing state
         // set the frequency range between 700 Mhz and 1.3 Ghz
         DATAname = "700000"; // Setting up the minimum frequency 51 Mhz
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
         ChangeCPUinfor(CPUname, DATAname, DATAaddress);
         DATAname = "1300000"; // Setting up the maximum frequency at 1300 MHz
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
         ChangeCPUinfor(CPUname, DATAname, DATAaddress);

         CPUname = "Dynamic Mode";
         DATAname = "current min frequency";
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
         ReadCPUinfor(CPUname, DATAname, DATAaddress);
         DATAname = "current MAX frequency";
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
         ReadCPUinfor(CPUname, DATAname, DATAaddress);
       } else {
         // set to lower processing state
         // set the frequency range between 51 Mhz and 700 Mhz
         DATAname = "51000"; // Setting up the minimum frequency 51 Mhz
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
         ChangeCPUinfor(CPUname, DATAname, DATAaddress);
         DATAname = "700000"; // Setting up the maximum frequency at 1300 MHz
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
         ChangeCPUinfor(CPUname, DATAname, DATAaddress);

         CPUname = "Dynamic Mode";
         DATAname = "current min frequency";
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
         ReadCPUinfor(CPUname, DATAname, DATAaddress);
         DATAname = "current MAX frequency";
         DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
         ReadCPUinfor(CPUname, DATAname, DATAaddress);
       }
     Log.d("Moderate","Performance");
   }
 }
```

Listing 5: Dynamic Frequency Scaling Mode Mixed

```java
private void setDFS_Mixed(double cpu_load) {
  // get the battery level
  IntentFilter ifilter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
  Intent batteryStatus = this.registerReceiver(null, ifilter);

  int level = batteryStatus.getIntExtra(BatteryManager.EXTRA_LEVEL, -1);
  int scale = batteryStatus.getIntExtra(BatteryManager.EXTRA_SCALE, -1);

  float battery_percent = level / (float)scale;

  // get the charging state
  int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
  boolean is_charging = status == BatteryManager.BATTERY_STATUS_CHARGING ||
     status == BatteryManager.BATTERY_STATUS_FULL;

  // determine if we are charging
  if(is_charging){
   // if we are charging we don't care about power usage.... set to high performance
       mode
   setHighPerformanceMode();
  }else {
   // vary the processor frequency based on the battery level.
   // the higher the battery level, the higher the frequency.
   // anywhere from 100000 khz to 1300000 khz
   double step_size = 1.0/12.0;

   // pretend like the cpu_load is less if the battery percent is lower
   int load_fraction = new Double(Math.ceil( (cpu_load * battery_percent) /
       step_size)).intValue();

   // load_fraction is at greatest 1
   int high_frequency = 100000 * (Integer.valueOf(load_fraction) + 1);
   int low_frequency = high_frequency;

   Log.d("low_frequency",new Integer(low_frequency).toString());
   Log.d("high_frequency", new Integer(high_frequency).toString());

   // set to fairly high processing state
   DATAname = String.valueOf(low_frequency); // Setting up the minimum frequency at low
       frequency
   DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
   ChangeCPUinfor(CPUname, DATAname, DATAaddress);
   DATAname = String.valueOf(high_frequency); // Setting up the maximum frequency at
       high frequency
   DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
   ChangeCPUinfor(CPUname, DATAname, DATAaddress);

   CPUname = "Dynamic Mode";
   DATAname = "current min frequency";
   DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq";
   ReadCPUinfor(CPUname, DATAname, DATAaddress);
   DATAname = "current MAX frequency";
   DATAaddress = "/sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq";
   ReadCPUinfor(CPUname, DATAname, DATAaddress);
  }
}
```

```
    int computation = 3;

    // do factorial of n recursivly,
        measure computation time
    long before_time =
        SystemClock.currentThreadTimeMillis();

    result_factorial=11111l;
    long n = 33;
    for(int i=0;i<50;i++) {
      if(computation == 1) {
        recursive_factorial(n);
      }else if(computation == 2){
        square_big_numbers(n);
      }else {
        fibonacci(n);
      }
    }

    long run_time =
        SystemClock.currentThreadTimeMillis()
        - before_time;

    Log.d("Factorial Result", new
        Long(result_factorial).toString());

    //print the run_time as op message
    OPERATIONmessage2("Run Time" +
        run_time);
  }
```

Listing 6 depicts the code we used to measure computation time under each of the different modes. Depending on the value of the computation variable, the code will run one of three computationally intense functions. These functions constitute each of the loads we tested for computation time. When running the code we had to vary the number of times the for loop ran in order to make the computation time manageable. We ran recursive_factorial() ten million times, square_big_numbers() ten million times, and fibonacci() 50 times. Each of these tests took about ten seconds to run.

Listing 7: Measuring Computation Time - Sum of Squares

```
  private long square_big_numbers(long n){
    if(n <= 1 ){
      return 1;
    }

    return square_big_numbers(n-1) + n^2;
  }
```

Listing 7 depicts a test which computes the sum on the square of all the numbers from 1 to n. This constitutes a significant computation when it is run ten million times. TODO

Listing 8: Measuring Computation Time - Fibonacci

```
private long fibonacci(long number){
    if ((number == 0) || (number == 1)) //
        base cases
      return number;
    else
      return fibonacci(number - 1) +
          fibonacci(number - 2);
  }
```

Listing 8 depicts a test which computes the nth fibonacci number. This constitutes a significant computation when it is run fifty times. TODO

Listing 9: Measuring Computation Time - Factorial

```
  private long recursive_factorial(long i){
    if(i <= 1){
      return 1;
    }

    return recursive_factorial(i-1)*i;
  }
```

Listing 9 depicts a test which computes the factorial of n. This constitutes a significant computation when it is run ten million times. TODO

2.6 Computing CPU Load

Listing 10: Computing CPU Load

```
protected double ReadCPUload() {

    ProcessBuilder cmd;
    double result=0.0;

    try{

      String[] args = {"/system/bin/cat",
          "/proc/stat"};
      cmd = new ProcessBuilder(args);

      Process process = cmd.start();
      InputStream in =
          process.getInputStream();
      byte[] re = new byte[1024];
      String stat = "";

      while(in.read(re) != -1) {
        stat += new String(re);
        //read the entire input strea
        //result = result + new String(re);
      }

      in.close();

      // parse stat for the load data
      String load;
      Scanner stat_scanner = new Scanner(stat);

      load = stat_scanner.nextLine();
      String[] toks = stat_scanner.nextLine().split(" ");

      //do the cpu load calculation based on the first line of /proc/stat
      int user_time = Integer.valueOf(toks[1]);
      int nice_time = Integer.valueOf(toks[2]);
      int system_time = Integer.valueOf(toks[3]);
      int idle_time = Integer.valueOf(toks[4]);
      int iowait = Integer.valueOf(toks[5]);
      int irq = Integer.valueOf(toks[6]);
      int softirq = Integer.valueOf(toks[7]);
      int steal = Integer.valueOf(toks[8]);
      int guest = Integer.valueOf(toks[9]);
      int guest_nice = Integer.valueOf(toks[10]);

      // Guest time is already accounted in usertime
      int usertime = user_time - guest;
      int nicetime = nice_time - guest_nice;
      // Fields existing on kernels >= 2.6
      // (and RHEL's patched kernel 2.4...)
      int idlealltime = idle_time + iowait;
      int systemalltime = system_time + irq + softirq;
      int virtalltime = guest + guest_nice;
      int totaltime = usertime + nicetime + systemalltime + idlealltime + steal + virtalltime;

      result = ((double)(totaltime - idlealltime))/totaltime;
      OPERATIONmessage2("CPU load = "+ result + "\n");
    } catch(IOException ex){
      ex.printStackTrace();
    } catch(Exception e){
      e.printStackTrace();
    }
    Log.d("cpu_load", Double.toString(result));
    return result;
  }
```

Listing 10 show the process for computing cpu load from the information given in the /proc/stat file.

3 Part 2 - Data Analysis

3.1 Measuring energy consumption

TODO

3.2 Comparison and Analysis

TODO

4 Extra Part - Measuring Computation Time

TODO

5 Conclusion

TODO