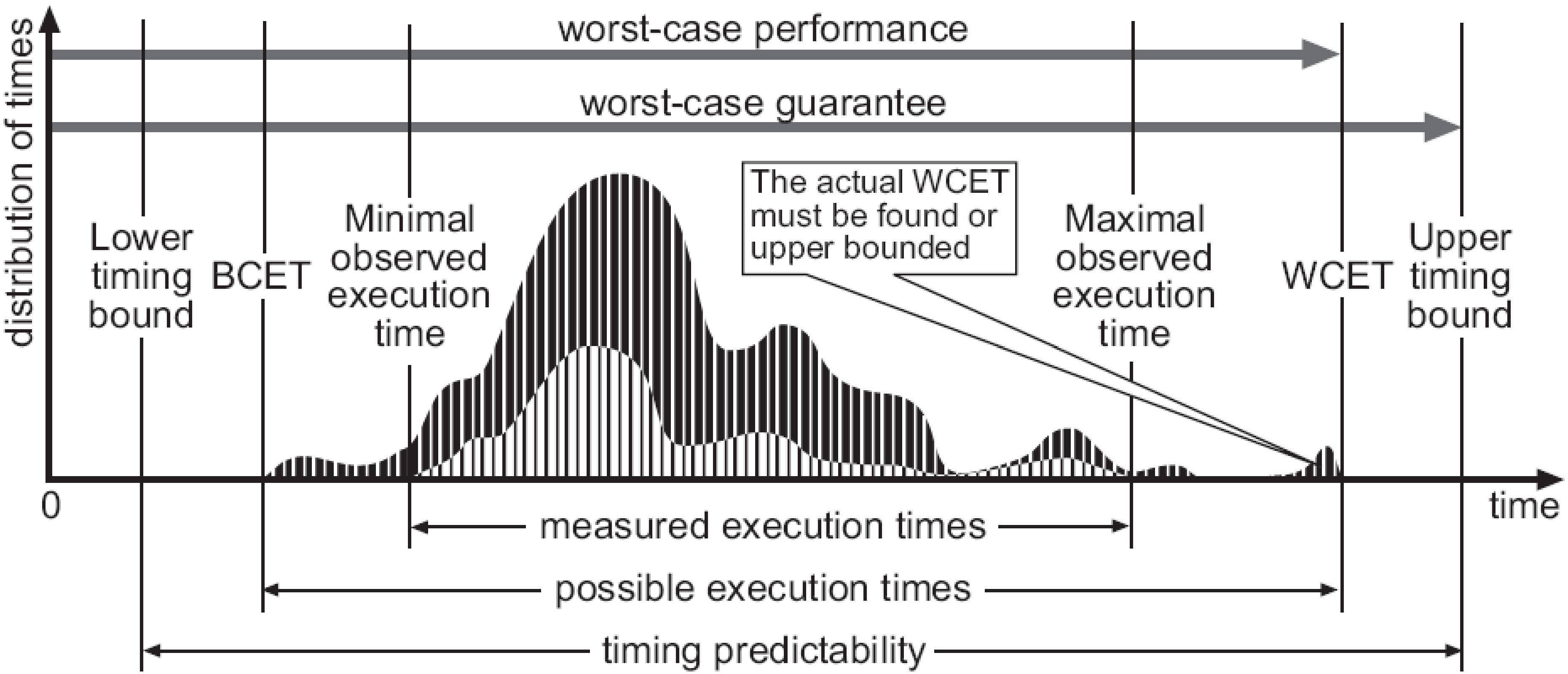
EC535 Introduction to Embedded Systems

- Teaming
- Blocking I/O
- Asynchronous Notification

Team: Pairs for Assignments, HW, Labs

Last Time:

Metrics for Embedded Systems



Architecture-dependent Metrics

- Number of instructions/cycles, memory requirement, etc.

- More accurate, but also more expensive
  - May need to simulate or to use the hardware

- Useful to optimize a software for a target hardware

- Potential problems:
  - Simulation model may introduce some abstraction
    - e.g., a cycle-accurate simulator may not be able to model the processor bus and other peripherals
  - Simulated hardware may be different from the target

Instruction-accurate profiling

- SimIt-ARM instruction-accurate profiling:

```
$ ema -f nwfpe.bin small
The result is 4900
Total user time : 0.070 sec.
Total system time: 0.001 sec.
Simulation speed : 3.088e+07 inst/sec.
Total instructions : 2191774 (2M) including 7211 nullified
Total 4K memory pages allocated : 370
```

- Execution Time = Total Instructions / Simulation Speed

Cycle-accurate profiling

```
$ sima -f nwfpe.bin small
The result is 4900
Total icache reads: 2551776
Total icache read misses: 457
icache hit ratio: 99.982%
Total itlb reads: 2551819
Total itlb read misses: 25
itlb hit ratio: 99.999%
Total dcache writes: 369657
Total dcache write misses: 3654
Total dcache reads: 871393
Total dcache read misses: 207
dcache hit ratio: 99.689%
Total dtlb reads: 1241050
Total dtlb read misses: 26
dtlb hit ratio: 99.998%
Total biu accesses: 4315
biu activity: 3.438%
Total allocated OSMs : 2551819
Total retired OSMs : 2551817
Total cycles : 3131710
Equivalent time on 206.4MHz host: 0.0152 sec.
```

Static memory requirements

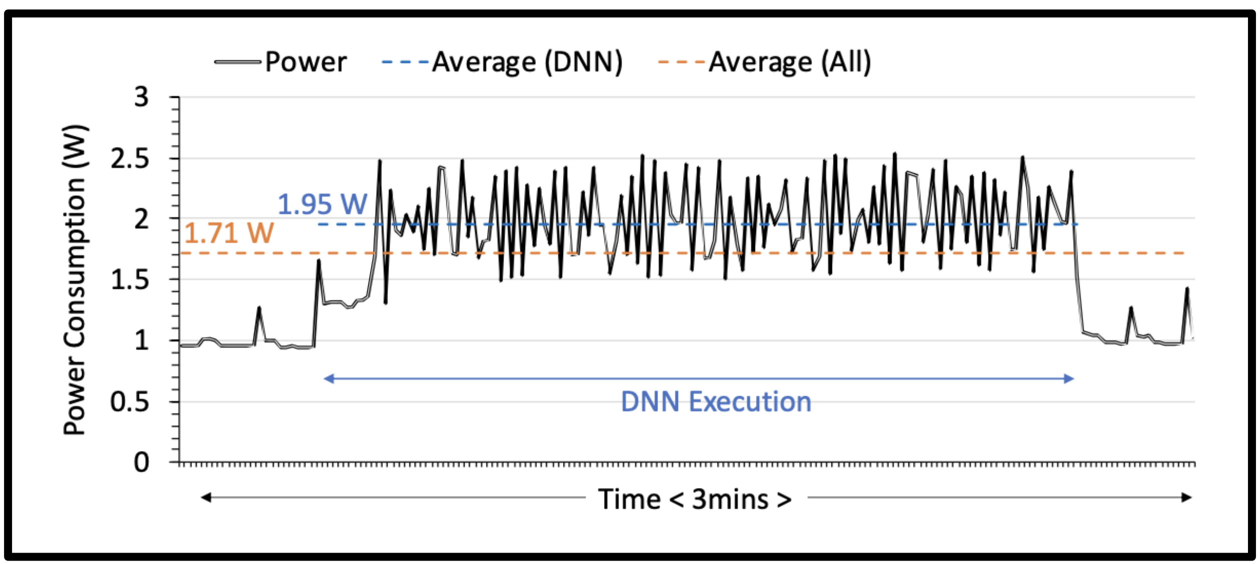
- Code and data space requirements using *size*:

```
$ arm-linux-size small
   text    data     bss     dec     hex filename
 362479    4172    5140  371791   5ac4f small
```

- text: Code size
- data: Initialized data size
- bss: Non-initialized data size
- dec: Total bytes required in decimal
- hex: Total bytes required in hex

Other Metrics?

Cost	$1,000	$15,000	$100,000
Battery	30 min	30 min	90 min



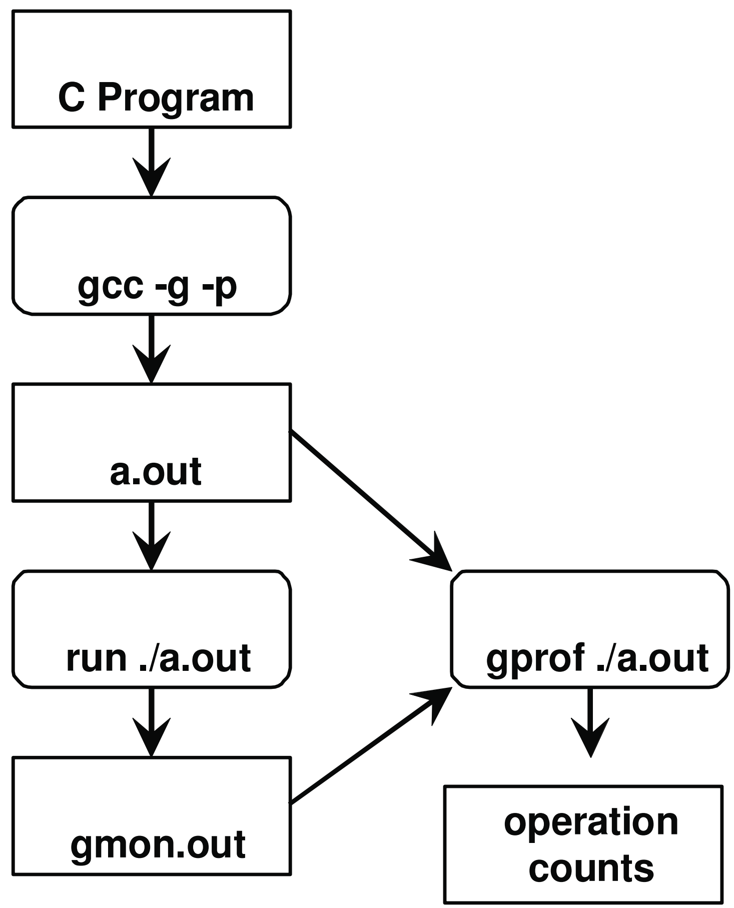
"We want to run as much as we can ***<u>on device</u>***, because it delivers low latency…''

Profiling C Code

- Profiling:
  - Invasive: modify the program, i.e., code instrumentation
  - Non-invasive: statistic sampling of the program
- Profiles:
  - Flat profile
  - Call graph
  - Annotated sources
- Tools:
  - gprof
  - gcov
  - valgrind
  - oprofile

gprof for profiling C code

gcc inserts calls to a function mcount into prologue of each function



Profiling C code

- small.c

```
#include "stdio.h"
int two(int limit) {
  int a, i;
  a = 0;
  for (i=0; i<limit; i++)
    a += i;
}

int one(int limit) {
  int i, a[50];

  for (i=0; i<limit; i++)
    a[i % 50] = i + two(i);
  return a[49];
}

int main() {
  int j, a;
  a = 0;
  for (j=0; j<1000; j++)
    a = a + one(j);
  printf("The result is %d\n", a);
  return 0;
}
```

gprof for profiling C code

- Enable profiling during compilation

```
> gcc -g -p small.c
```

- When run, the binary will create a file “gmon.out”, which can be analyzed by gprof

```
> ./a.out
> gprof a.out
```

- Some of the output looks like:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 98.61      0.35     0.35   499500     0.71     0.71  two
  1.39      0.36     0.01     1000     5.00   360.00  one
```

gprof for profiling C code

- two() is much more significant
- However, two() is called by one() unnecessarily 98% of the time

In-class exercise:

- Run gprof with small.c
- Optimize something outside the main function to speed up the program
- Run gprof again, observe the change in the flat profile

- Submit a zip file on GradeScope (inclass exercise 3) including:
  - New code in a file named small_new.c
    - Write names/usernames of people in the team as comments
  - The old and the new flat profile

```
#include "stdio.h"
int two(int limit) {
  int a, i;
  a = 0;
  for (i=0; i<limit; i++)
    a += i;
}

int one(int limit) {
  int i, a[50];

  for (i=0; i<limit; i++)
    a[i % 50] = i + two(i);
  return a[49];
}

int main() {
  int j, a;
  a = 0;
  for (j=0; j<1000; j++)
    a = a + one(j);
  printf("The result is %d\n", a);
  return 0;
}
```

gprof for profiling C code

```c
int two(int limit)
{
    int a, i;
    a = 0;
    for (i=0; i<limit; i++)
        a += i;
}

int one(int limit)
{
    int i, a[50];

    for (i=49; i<limit; i+=50) {
            a[i % 50] = i + two(i);
        }
    return a[49];
}

int main()
{
    int j, a;
    a = 0;
    for (j=0; j<1000; j++)
        a = a + one(j);
    printf("The result is %d\n", a);
    return 0;
}
```

```c
#include "stdio.h"
int two(int limit) {
  int a, i;
  a = 0;
  for (i=0; i<limit; i++)
    a += i;
}

int one(int limit) {
  int i, a[50];

  for (i=0; i<limit; i++)
    a[i % 50] = i + two(i);
  return a[49];
}

int main() {
  int j, a;
  a = 0;
  for (j=0; j<1000; j++)
    a = a + one(j);
  printf("The result is %d\n", a);
  return 0;
}
```

I/O-oriented programming in Linux

- I/O service is not always available
  - Bandwidth limitation
    - Data not yet ready when one wants to read
    - Old data not yet transmitted when one wants to write again
  - Asynchronous nature of I/O
    - Data arrival time is unpredictable

Simplistic Solution

- Nonblocking file operations on device file
  - If no data or device busy, read/write returns –EAGAIN
    - User process receives 0 as return value
  - If data available, read/write returns actual byte count transferred
    - User process receives the count
- Simplistic solution
  - When requesting data
    - User process keeps reading until read returns >0.
  - When sending data
    - User process keeps writing until write returns <0.

Better Solution

- Blocking file operations
  - If no data or device busy, read/write sleeps
    - Use the “wait queue” data structure
      - wait_event_interruptible
    - User process sleeps as a result
  - When I/O available, read/write wakes up
    - wake_up_interruptible
    - User process wakes up and resumes

Rules about sleeping

- Never sleep while running in atomic context.
  - What can go wrong?
    - Process holds a spinlock, other locks
    - Process disabled interrupts

Rules about sleeping

- Never sleep while running in atomic context.
  - What can go wrong?
    - Process holds a spinlock, other locks
    - Process disabled interrupts

- When process wakes up, it will not know what happened on the CPU while it was sleeping.
  - Make no assumptions based on earlier CPU states, check again.
- Make sure to verify who is going the wake the process up under what condition before implementing sleep.

Simple Sleep

- Wait queue
  - A linked list of sleeping processes
    - Wait events: interruptable, timeout, etc.
    - Use interruptable: can be interrupted by signals
  - Initialize the queue
  - Implement the calls to enter into wait queue and to finish waiting
    - Use wake_up_interruptible: restricts itself to processes that are on interruptable sleep (otherwise you may wake up all processes)
    - Implement a condition for wake_up_interruptible, e.g., a flag

Simple Sleep

- Wait queue
  - A linked list of sleeping processes
    - Wait events: interruptable, timeout, etc.
    - Use interruptable: can be interrupted by signals
  - Initialize the queue
  - Implement the calls to enter into wait queue and to finish waiting
    - Use wake_up_interruptible: restricts itself to processes that are on interruptable sleep (otherwise you may wake up all processes)
    - Implement a condition for wake_up_interruptible, e.g., a flag

Multiple Processes Sleeping?

Simple Sleep

- Wait queue
  - A linked list of sleeping processes
    - Wait events: interruptable, timeout, etc.
    - Use interruptable: can be interrupted by signals
  - Initialize the queue
  - Implement the calls to enter into wait queue and to finish waiting
    - Use wake_up_interruptible: restricts itself to processes that are on interruptable sleep (otherwise you may wake up all processes)
    - Implement a condition for wake_up_interruptible, e.g., a flag

Multiple Processes Sleeping?

Blocking vs. Non-blocking

- Potential problems with blocking operations
  - What if the program does not want to sleep?
    - e.g., need to handle other tasks.
    - Solution: can use multiple threads

Blocking vs. Non-blocking

- Potential problems with blocking operations
  - What if the program does not want to sleep?
    - e.g., need to handle other tasks.
    - Solution: can use multiple threads
- Separation of read/write buffers

Blocking vs. Non-blocking

- Potential problems with blocking operations
  - What if the program does not want to sleep?
    - e.g., need to handle other tasks.
    - Solution: can use multiple threads

- Separation of read/write buffers

- NONBLOCK flag: affects read, write, open operations
  - -EAGAIN: “try it again”
  - Useful for short operations that either succeed or fail quickly

Advanced sleep

- Set process state (running, interruptable/uninterruptable)
  - Interruptbile/uninterruptible: two types of sleep
  - This step does not yield the processor yet
  - Discouraged → prone to bugs
- Check sleep condition and call the scheduler
  - schedule() will yield resources

Scull_pipe: a blocking I/O example
($EC535/examples/ldd3_book)

Another Solution

- Asynchronous Notification
  - User process registers a signal handler function
    - Continues to do other tasks
  - When I/O available, kernel sends signal to user process
    - Signal handler is invoked

Setting I/O Modes

- Blocking/Async mode controlled by *f_flags* of *filp*
  - Changing bits in the flag changes device file behavior
- Fcntl system call can be used to read/write f_flags
  - `oldflag = fcntl(fd, F_GETFL);`
  - `fcntl(fd, F_SETFL, oldflag | FASYNC);`

Team - Shared Reading

- LDD3 Book
  - P. 147, Blocking I/O (Person 1)
  - P. 169, Asynchronous Notification (Person 2)

Team - Shared Reading

- LDD3 Book
  - P. 147, Blocking I/O (Person 1)
  - P. 169, Asynchronous Notification (Person 2)

Questions:

How do you wake up a sleeping process?

When a process receives a SIGIO, if two input files are available, how does the process know which input file has new input to offer?

What could go wrong if a process performing an atomic operation sleeps.

Advanced sleep

- Set process state (running, interruptable/uninterruptable)
  - Interruptbile/uninterruptible: two types of sleep
  - This step does not yield the processor yet
  - Discouraged → prone to bugs
- Check sleep condition and call the scheduler
  - schedule() will yield resources

Scull_pipe: a blocking I/O example
($EC535/examples/ldd3_book)