Profiling and Code Optimization

Real-Time Operative Systems Course

Paulo Pedreiras, DETI/UA/IT

December 20, 2021

1. Preliminaries

2. Code optimization techniques

3. Profiling

Last lecture

Other Topics Relevant for Real-Time Operative Systems

- Non-preemptive scheduling
- Practical aspects related with the implementation of applications on a RTOS
  - Cost of tick handler
  - Cost of context switching
  - Measuring of WCET
  - Cost of ISR
  - Impact of Release Jitter

Agenda for today

Profiling and Code Optimization

- Code optimization techniques
  - Introduction
  - Processor-independent techniques
  - Techniques dependent on memory architecture
  - Processor-dependent techniques
- Profiling tools
  - Objectives and methodologies
  - Tools

Why optimizing code?

- Many real-time systems are used in applications:
  - Highly cost-sensitive
  - Where energy consumption must be minimized
  - Where physical space is restricted, . . .
- If the execution time or footprint is too large just buying a faster processor IS NOT a solution!
- Code optimization allows to produce:
  - Faster programs:
    - Enables the use of slower processors, with lower cost, lower energy consumption.
  - Shorter programs:
    - Less memory, therefore lower costs and lower energy consumption

Low-level vs high-level languages

Is using assembly the solution?

- Assembly language programming
  - It potentially allows a very high level of efficiency, but:
    - Difficult debugging and maintenance
    - Long development cycle
    - Processor dependent - non-portable code!
    - Requires long learning cycles
- Programming in high-level languages (e.g. "C")
  - Easier to develop and maintain, relatively processor independent, etc. but:
    - Generated code potentially less efficient than code written in assembly

"C" vs Assembly

- Assembly programming, while potentially more efficient, in general turns out not to be a good approach:
  - Focus on implementation details and not on fundamental algorithmic issues, where the best optimization opportunities usually reside
    - E.g .: instead of spending hours building an "ultra-efficient" library for manipulating lists, it will be preferable to use "hash-tables";
    - Good quality compilers produce more efficient code than the one produced by an "average" programmer;
    - And finally:

John Levine, on Comp.Compilers

"Compilers make it a lot easier to use complex data structures, compilers don't get bored halfway through, and generate reliably pretty good code."

Then what is the best approach?

As in almost everything in life, “virtue is in the middle”, or otherwise, in the “war” between “C” and Assembly wins ... whoever chooses both!

- General approach:
  - The programmer writes the application in a high-level language (e.g. "C")
  - The programmer uses tools to detect "hot spots" (points where the application spends more resources)
  - The programmer analyzes the generated code and ...
    - Re-write critical sections in assembly
    - and/or restructures high-level code to generate more suitable assembly code

CPU independent optimization techniques

Elimination of common sub-expressions

- Formally, the occurrence of an expression "E" is called a common sub-expression if "E" was previously calculated and the values of the variables in "E" have not changed since the last calculation of "E".
- The benefit is obvious: less code to run!

Code before (left) and after(right) optimization

```
b:
   t6 = 4 * i // E1
   x  = a[t6]
   t7 = 4 * i // E1
   t8 = 4 * j // E2
   t9 = a[t8]
   a[t7]  = t9
   t10    = 4 * j // E2
   a[t10] = x
   goto b
```

```
b:
    t6 = 4* i
    x  = a[t6]
    t8 = 4 * j
    t9 = a[t8]
    a[t6] = t9
    a[t8] = x
    goto b
```

CPU independent optimization techniques

Elimination of "dead code"

- If a certain set of instructions is not executed under any circumstances, it is called "dead code" and can be removed
- E.g. replacing the "if" with "#ifdef" allows the compiler to remove debug code at the pre-processing stage (no memory and CPU wasted)

```
...
debug = 0;
...
if (debug){
    print .....
}
```

```
...
#define DEBUG
...
#ifdef DEBUG
    print .....
#endif
```

CPU independent optimization techniques

Induction and force reduction variables

- An "X" variable is called an "L" cycle induction variable if each time "X" is changed in cycle "L", it is increased or decreased by a constant value
  - When there are two or more induction variables in a cycle, it may be possible to remove one of them
  - Sometimes it is also possible to reduce its "strength", i.e., its cost of execution
  - **Benefits**: lower and/or less costly computations

```
    j = 0
label_XXX:
    j  = j + 1
    t4 = 11 * j // t4 depends on j
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

```
    t4 = 0
label_XXX:
    t4 += 11 // J removed
    t5 = a[t4]
    if (t5 > v) goto label_XXX
```

CPU independent optimization techniques

Cycle expansion

- It consists of making multiple iterations of the calculations in each iteration of the cycle
- **Benefits**: reduction of overhead due to the cycle
- **Problems**: increased amount of memory
- Suitable for short cycles

Before:

```
int checksum(int *data, int N){
    int i, sum=0;
    for(i=0;i<N;i++)
    {
        sum += *data++;
    }
    return sum;
}
```

After:

```
int checksum(int *data, int N){
    int i, sum=0;
    for(i=0;i<N;i+=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }
    return sum;
}
```

Cycle expansion example

Before:

```
0x00:    MOV      r3,#0 ; sum =0
0x04:    MOV      r2,#0 ; i= 0
**********************************
0x08:    CMP      r2,r1 ; (i < N) ?
0x0c:    BGE      0x20 ; go to 0x20 if i >= N
0x10:    LDR      r12,[r0],#4 ; r12 <- data++
0x14:    ADD      r3,r12,r3 ; sum = sum + r12
***********************************
0x18:    ADD      r2,r2,#1 ; i=i+1 (N times)
***********************************
0x1c:    B        0x8 ; jmp to 0x08
0x20:    MOV      r0,r3 ; sum = r3
0x24:    MOV      pc,r14 ; return
```

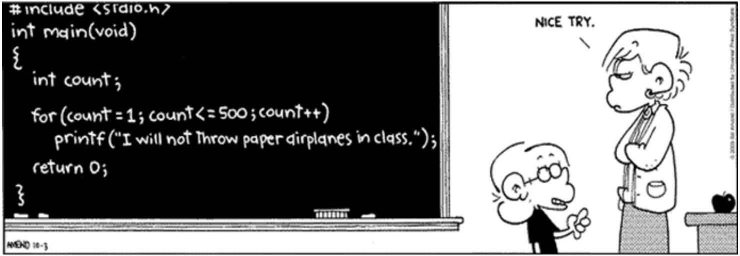
After:

```
0x00:    MOV      r3,#0    ; sum = 0
0x04:    MOV      r2,#0   ; i = 0
0x08:    B        0x30   ; jmp to 0x30
******************************************
0x0c:    LDR      r12,[r0],#4   ; r12 <- data++
0x10:    ADD      r3,r12,r3     ; sum = sum + r12
0x14:    LDR      r12,[r0],#4   ; r12 <- data++
0x18:    ADD      r3,r12,r3     ; sum = sum + r12
0x1c:    LDR      r12,[r0],#4   ; r12 <- data++
0x20:    ADD      r3,r12,r3     ; sum = sum + r12

0x24:    LDR      r12,[r0],#4   ; r12 <- data++
0x28:    ADD      r3,r12,r3     ; sum = sum + r12

******************************************
0x2c:    ADD      r2,r2,#4      ; i = i + 4 (N/4 times)
******************************************
0x30:    CMP      r2,r1   ; (i < N) ?
0x34:    BLT      0xc   ; go to 0x0c if i < N
0x38:    MOV      r0,r3   ; r0 <- sum
0x3c:    MOV      pc,r14   ; return
```

CPU independent optimization techniques

Cycle expansion (cont.)

- As we will see later on, there may be problems associated with this technique ...



CPU independent optimization techniques

Function inlining

- Replace a function call with the function code
  - **Benefits**: reduced overhead associated with calling a function
  - **Problems**: (possible) increased code size
  - Suitable when small functions are called **multiple times** from a small number of locations

Before:

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}
```

After:

```
__inline int max(int a, int b)
{
int x;
x=(a>b ? a:b);
return x;
}
```

Inlining function example (without)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}

int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
max
$a
0x00:    CMP      r0,r1; (x > y) ?
0x04:    BGT      0x0c; return if (x > y)
0x08:    MOV      r0,r1;  else r0 <- y
0x0c:    MOV      pc,r14 return
t


0x10:    STMFD    r13!,{r4,r14}; save registers
0x14:    MOV      r2,r0;     r2 <- x
0x18:    MOV      r3,r1;     r3 <- y
0x1c:    MOV      r1,r3;     r1 <- y
0x20:    MOV      r0,r2;     r0 <- x
0x24:    BL       max  ;  r0 <- max(x,y)
0x28:    MOV      r4,r0;      r4 <- a1
0x2c:    MOV      r1,r3;     r1 <- y


0x30:    ADD      r0,r2,#1; r0 <- x+1
0x34:    BL       max  ;         r0 <- max(x+1,y)
0x38:    MOV      r1,r0 ;          r1 <- a2
0x3c:    ADD      r0,r4,#1 ; r0 <- a1+1


0x40:    LDMFD    r13!,{r4,r14} ; restore


0x44:    B        max  ;
```

Inlining function example (with)

```
void t(int x, int y)
{
    int a1=max(x,y);
    int a2=max(x+1,y);

    return max(a1+1,a2);
}

__inline int max(int a, int b)
{
    int x;
    x=(a>b ? a:b);
    return x;
}
```

```
0x00:    CMP    r0,r1 ; (x<= y) ?
0x04:    BLE    0x10  ; jmp to 0x10 if true
0x08:    MOV    r2,r0 ; a1 <- x
0x0c:    B      0x14  ; jmp to 0x14
0x10:    MOV    r2,r1 ; a1 <- y if x <= y
0x14:    ADD    r0,r0,#1; generate r0=x+1
0x18:    CMP    r0,r1   ;   (x+1 > y) ?
0x1c:    BGT    0x24    ;jmp to 0x24 if true
0x20:    MOV    r0,r1   ; r0 <- y
0x24:    ADD    r1,r2,#1 ; r1 <- a1+1
0x28:    CMP    r1,r0 ; (a1+1 <= a2) ?
0x2c:    BLE    0x34  ; jmp to 0x34 if true
0x30:    MOV    r0,r1 ; else r0 <- a1+1
0x34:    MOV    pc,r14
```

Cache impact

The use of techniques such as cycle expansion or inline functions can cause **performance degradation in systems with cache** !

Before cycle expansion:

```
int checksum(int *data, int N)
{
    int i;
    for(i=N;i>=0;i--)
    {
        sum += *data++;
    }
    return sum;
}
```

```
0x0000000c:    LDR    r3,[r2],#4
0x00000010:    ADD    r0,r3,r0
0x00000014:    SUB    r1,r1,#1
0x00000018:    CMP    r1,#0
0x0000001c:    BGE    0xc
```

0x000000c
0x0000010
0x0000014
0x0000018
0x000001c

Instructions cache

Cache impact (cont.)

After cycle expansion:

```
int checksum(int *data, int N)
{
    int i;
    for(i=N;i>=0;i-=4)
    {
        sum += *data++;
        sum += *data++;
        sum += *data++;
        sum += *data++;
    }
    return sum;
}
```

```
0x00000008:   LDR    r3,[r0],#4
0x0000000c:   ADD    r2,r3,r2
0x00000010:   LDR    r3,[r0],#4
0x00000014:   ADD    r2,r3,r2
0x00000018:   LDR    r3,[r0],#4
0x0000001c:   ADD    r2,r3,r2
0x00000020:   LDR    r3,[r0],#4
0x00000024:   ADD    r2,r3,r2
0x00000028:   SUB    r1,r1,#4
0x0000002c:   CMP    r1,#0
0x00000030:   BGE    0x8
```

?????????
?????????
?????????
?????????
?????????

Cache capacity smaller than cycle code

Successive cache misses and updates!

Optimization techniques dependent on memory architecture

Memory access order

- In matrices, the "C" language defines that the rightmost index defines adjacent memory positions
- Significant impact on cache memory data in structures with high dimension

Array p[j][k]

...	...
j=0	k=0 k=1 k=2
j=1	k=0 k=1 k=2
j=2	k=0 k=1 k=2
...	...

Optimization techniques dependent on memory architecture

Better performance when the internal cycle corresponds to the rightmost index

```
//Poor performance with cache
for (k=0; k<=m; k++)
    for (j=0; j<=n; j++) )
        p[j][k] = ...
```

```
//Good perfomance with cache
for (j=0; j<=n; j++)
    for (k=0; k<=m; k++)
        p[j][k] = ...
```

- For homogeneous memory access, the performance is identical.
- But in the presence of **cache** the **performance can be very distinct** !
- Thus it depends on the memory architecture

Architecture-dependent optimization techniques

Depending on the processor family used as well as the type of coprocessors available, several optimizations are possible:

- Conversion from floating point to fixed point in the **absence of math co-processor**
  - **Benefits**
    - Lower computational cost,
    - Lower energy consumption,
    - Sufficient signal-to-noise ratio if correctly scaled,
    - Suitable e.g. for mobile applications.
  - **Problems**:
    - Dynamic range reduction,
    - Possible overflows.

Architecture-dependent optimization techniques

Use of assembly specifics

- Example: on ARM architecture it is possible to set flags when doing an arithmetic operation

Before:

```
int checksum_v1(int *data)
{
   unsigned i;
   int sum=0;

   for(i=0;i<64;i++)
    sum += *data++;

   return sum;
}
----------------------------
MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0; i=0
L1  LDR  r3,[r2],#4;   r3=*(data++)
****
    ADD  r1, r1, #1;   i=i+1 (a)
    CMP  r1, 0x40;     cmp r1, 64 (b)
****
    ADD  r0, r3, r0;   sum +=r3
    BCC  L1;      if i < 64, goto L1
    MOV  pc, lr;   return sum
```

After:

```
int checksum_v2(int *data)
{
   unsigned i;
   int sum=0;

   for(i=63;i >= 0;i--)
    sum += *data++;

   return sum;
}
----------------------------
    MOV  r2, r0; r2=data
    MOV  r0, #0; sum=0
    MOV  r1, #0x3f; i=63
L1  LDR  r3,[r2],#4;   r3=*(data++)
    ADD  r0, r3, r0;   sum +=r3
***
    SUBS  r1, r1, #1;  i--, set flags (rep. a and b)
***
    BGE  L1;  if i >= 0, goto L1
    MOV  pc, lr;       return sum
```

Architecture-dependent optimization techniques

There are many other techniques that, due to time limitations, are not covered in this course unit:

- Some classes:
  - Controlling resource use (e.g. variables assigned to registers)
  - Exploring parallelism
  - Multiple memory banks
  - Multimedia instructions
  - ....

1. Preliminaries
2. Code optimization techniques
3. Profiling

Profiling

Task

Given the source code of a program, possibly written by someone else, perform its optimization!

Where to start?

- Analyze the source code and detect inefficient "C" code
- Re-write some sections in assembly
- Use more efficient algorithms

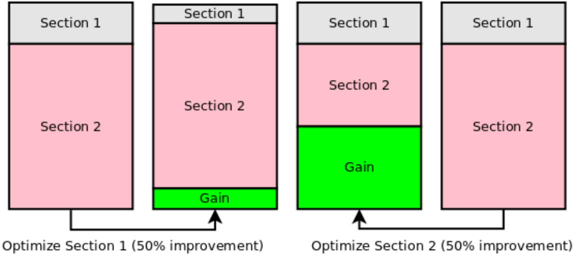
How to determine which sections to optimize?

- A typical application consists of many functions spread over different source files
- **Manual inspection** of the entire application code to determine which sections to optimize is in many cases **unpractical** !

Profiling

Amdahl's law

The performance gain that may be obtained when optimizing a section of code is limited to the fraction of the total time that is spent on that particular section.



- But how to determine the parts of code that consume the more significant share of CPU?

Profiling

Profiling

Collection of statistical data carried out on the execution of an application

- Fundamental to determine the relative weight of each function
- Approaches:
  - **Call graph profiling**: function invocation is instrumented
    - Intrusive, requires access to the source code, computationally heavy (overhead can reach 20
  - **Flat profiling**: the application status is sampled at regular time intervals
    - Accurate as long as functions execution time much bigger than the sampling period

Profiling

Example:

```
Routine        % of Execution Time
function_a     60%
function_b     27%
function_c      4%
...
function_zzz 0.01%
```

"80/20 Law"

In a "typical" application about 80% of the time is spent in about 20% of the code.

Profiling - tools

GNU Gprof

(https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html)

- Profiling requires several steps
  - Compilation and "linking" of the application with debug and profiling active
    - gcc **-pg** -o sample sample.c
  - Run the program to generate statistical data (profiling data)
    - ./sample
  - Run the gprof program to analyze the data
    - gprof ./sample [> text.file]

"-pg" : Generate extra code to write profile information suitable for the analysis program gprof.

Profiling - tools

GNU Gcov

(https://gcc.gnu.org/onlinedocs/gcc/Gcov.html)

- Coverage test, complementary to gprof.
- Indicates the number of times each line is executed
  - Must compile and link with "-fprofile-arcs -ftest-coverage" to generate additional information needed by gcov
    - gcc **-pg -fprofile-arcs -ftest-coverage** -o sample sample.c -lm
  - Run the program to generate statistical data and then run gcov
    - ./sample
    - gcov sample.c
  - File "sample.c.cov" contains the execution data

Profiling - tools

```
 ...  (main) ...
1:                  5:{
-:                  6: int i;
1:                  7: int colcnt = 0;
200000:             8:for (i=2; i <= 200000; i++)
199999:             9: if (prime(i)) {
17984:             10:  colcnt++;
17984:             11:  if (colcnt%9 == 0) {
1998:              12:    printf("%5d\n",i);
1998:           13:    colcnt = 0;
-:                 14:  }
-:                 15: else
15986:             16:   printf("%5d ", i);
-:                 17: }
1:                 18: putchar('\n');
1:                 19: return 0;
-:                 20:}
199999:            21:int prime (int num) {
-:                 22: /* check to see if the number is a prime? */
-:                 23:  int i;


1711598836:        24:  for (i=2; i < num; i++)
1711580852:        25:  if (num \%i == 0)


182015:            26:   return 0;
17984:             27:  return 1;
-:                 28:}
```

Number of executions very high - optimization target

Profiling - tools

- Analyzing the code, an optimization was identified ...

```
....
199999:    22:int prime (int num) {
-:         23: /* check to see if the number is a prime? */
-:         24:  int i;

7167465:   25:  for (i=2; i < (int) sqrt( (float) num); i++)

7149370:   26:  if (num %i == 0)
181904:    27:   return 0;
....
```

Number of executions reduced by a factor of 238!!!

Profiling - tools

- Results with gprof

```
--------------------
Before optimization:
--------------------
Call graph
granularity: each sample hit covers 4 byte(s) for 0.02\% of 40.32 seconds
....
index    % time     self       children     called              name

[1]       100.0      0.01       40.31                          main   [1]
                    40.31        0.00     199999/199999        prime  [2]

-------------------
After optimization:
-------------------
Call graph
granularity: each sample hit covers 4 byte(s) for 2.63\% of 0.38 seconds
...
index    % time     self       children     called              name

[2]       100.0     0.00        0.38                            main   [2]
                    0.38        0.00      199999/199999         prime  [1]
```

Execution time reduced by a factor of 106!!!!

Profiling - tools

There are many other profiling tools. E.g. "perf":

- Performance counters for Linux ("perf" or "perf_events"): Linux tool that shows performance measurements in the command line interface.
- Can be used for finding bottlenecks, analysing applications' execution time, wait latency, CPU cycles, etc.
- Events of interest can be selected by the user ("perf list" allows to see the supported events)
- E.g.:

```
sudo perf stat  ls -al
... (command output ommited)
Performance counter stats for 'ls -al':

          4,97 msec task-clock              #    0,882 CPUs utilized
             2      context-switches        #    0,403 K/sec
             0      cpu-migrations          #    0,000 K/sec
           152      page-faults             #    0,031 M/sec
       7399469      cycles                  #    1,489 GHz
       6022842      instructions            #    0,81  insn per cycle
       1247081      branches                #  251,024 M/sec
         37966      branch-misses           #    3,04% of all branches

   0,005634326 seconds time elapsed

   0,005555000 seconds user
```

Summary (1/2)

- Improving application performance
  - Why optimize
  - Assembly programming vs. "C" programming
  - Architecture-independent optimizations
    - Elimination of common sub-expressions, elimination of dead code, reduction of induction variables, expansion of cycles, inlining
  - Cache impact
  - Memory access

Summary (2/2)

- Architecture-dependent optimizations
  - Conversion of floating to fixed point arithmetic, assembly specifics
- Optimization / profiling
  - General methodology
  - Case study: use of gprof and gconv