Computer Labs: Profiling and Optimization

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

December 18, 2011

Contents

Optimization

- Speed matters, and it depends mostly on the right choice of
  - Data structures
  - Algorithms
- If the program's performance in the end is not satisfactory, you should revisit our design decisions
- However, rather than revisit them all, you should "optimize" starting on those that matter most
  - I.e. those where the program spends most time
- To find out that you can perform **program profiling**

What is (Program) Profiling?

- It is a dynamic program analysis technique, i.e. is an analysis technique:
  - of the behavior of a program
  - that uses information collected while the program runs
- Typically it allows to find out for each function:
  - the number of times it is called
  - the amount of time the program spends in that function
  - the functions it calls

Profiling with `gprof`

- `gprof` is GNU's code profiler
- The program to be analysed must be:
  - compiled, and
  - linked

  with `gcc`'s **`-pg`** option

  ```
  gcc -pg -c foo.c
  gcc -pg foo.o -o foo
  ```

  This way the program is instrumented with additional instructions
- When the program runs, these instructions collect data and write it out to a file with name `gmon.out`
- `gprof` processes the data gathered in `gmon.out` and outputs the "program's profile"

  ```
  gprof foo
  ```

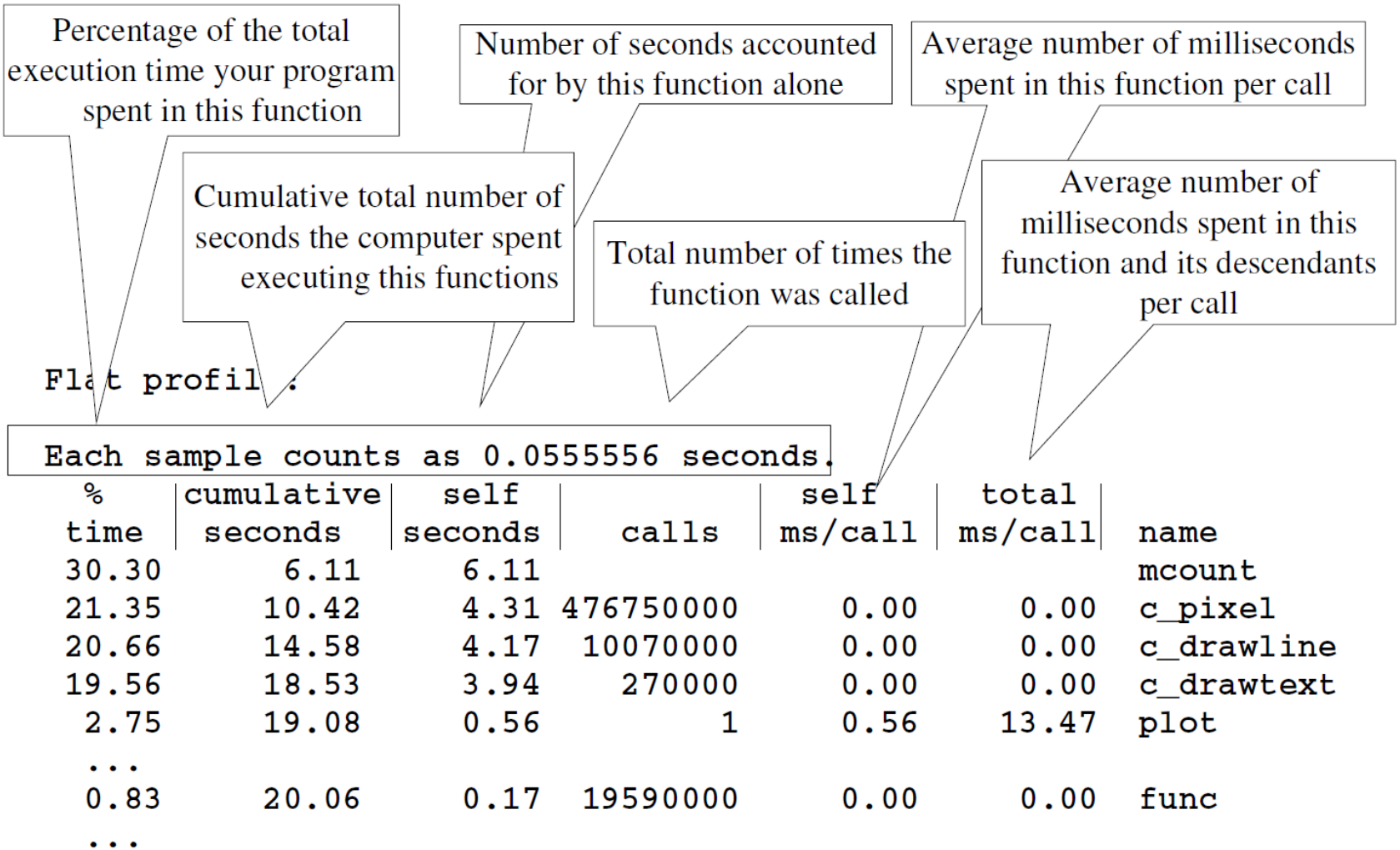
Information Provided by `gprof`

Flat profile shows how much time the program spent in each function, and how many times that function was called

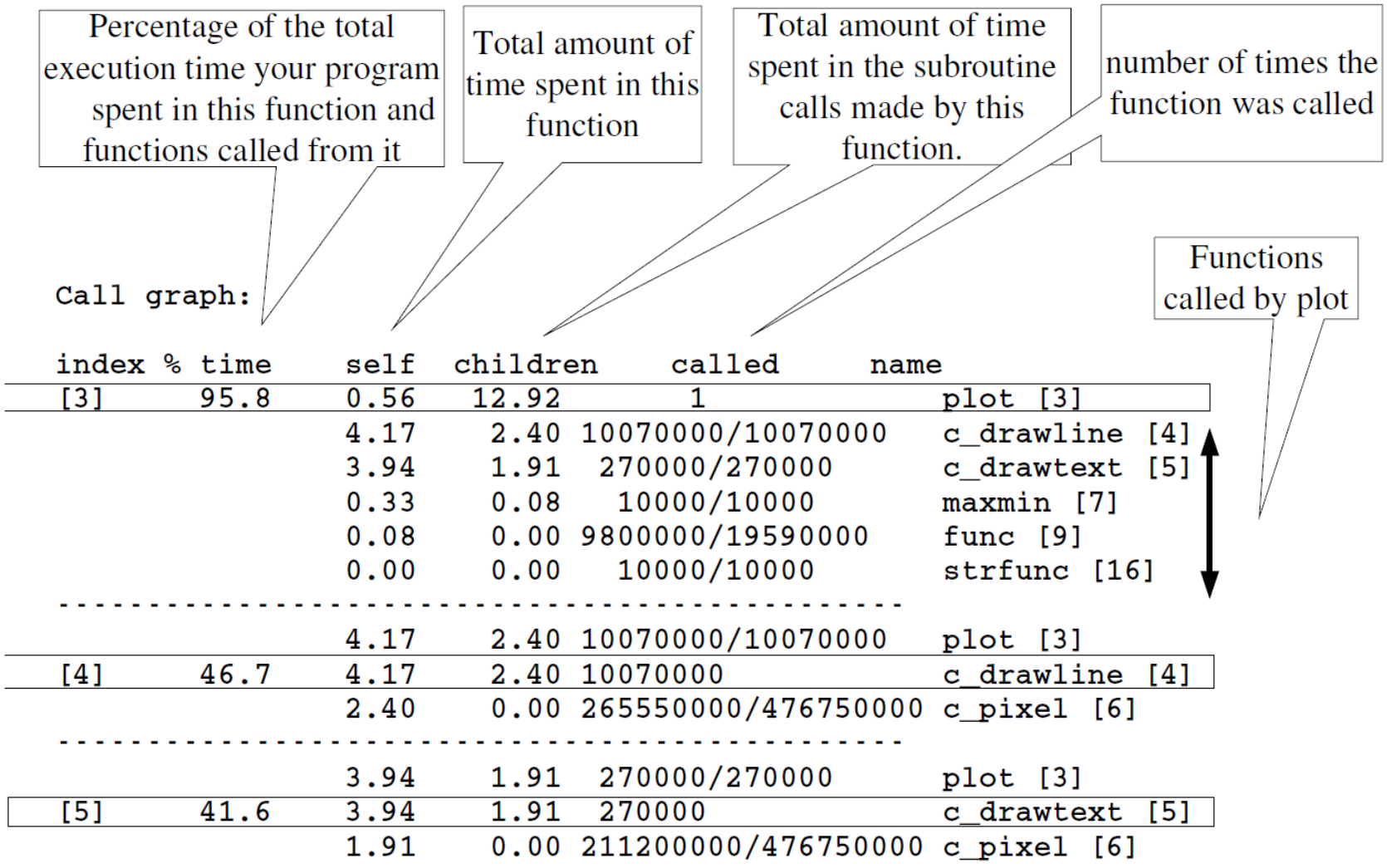
Call graph shows, for each function, which functions called it, which other functions it called and how many times. It also includes an estimate of how much time was spent in each function

Annotated source is a copy of the program's source code, labeled with the number of times each line of the program was executed

Flat Profile



Call Graph


Percentage of the total execution time your program spent in this function and functions called from it
Total amount of time spent in this function
Total amount of time spent in the subroutine calls made by this function.
number of times the function was called
Functions called by plot
Call graph:
index % time self children called name
[3] 95.8 0.56 12.92 1 plot [3]
4.17 2.40 10070000/10070000 c_drawline [4]
3.94 1.91 270000/270000 c_drawtext [5]
0.33 0.08 10000/10000 maxmin [7]
0.08 0.00 9800000/19590000 func [9]
0.00 0.00 10000/10000 strfunc [16]
4.17 2.40 10070000/10070000 plot [3]
[4] 46.7 4.17 2.40 10070000 c_drawline [4]
2.40 0.00 265550000/476750000 c_pixel [6]
3.94 1.91 270000/270000 plot [3]
[5] 41.6 3.94 1.91 270000 c_drawtext [5]
1.91 0.00 211200000/476750000 c_pixel [6]

Contents

Optimization: Example 1

- Let's consider a program that uses the functions `drawline()`, `drawtext()` and `pixel()`
- Some approaches to optimize these functions are as follows:
  1. Write the functions in assembly
     - Try first `pixel()`
  2. Define `pixel()` as a macro (using `#define`) or as an `inline` function
     - Eliminates the overhead of the prolog and epilog of a function call
     - Tradeoff between speed and size of executable
- Compile with a higher (more aggressive) optimization level

Summary of Results

- Execution times in seconds for 100,000 repetitions of a part of a program that calls `drawline()`, `drawtext()` and `pixel()`

	`drawline`	`drawtext`	`pixel`	Total
all in C	41	39	43	123
pixel macro	92	29	-	121
pixel asm	85	35	(28)	120
all in asm	78	16		94
all in C with -O3	82	15		97

`pixel()` in C

```
void c_pixel(int x, int y, int color, char *base, int h_res) {
    *(base + hres*y + x) = color;
}
```

`pixel()` in Assembly

Hand-written `pixel()`

```
proc asm_pixel
% arg x:dword, y:dword,
   color:dword, base: dword,
   h_res:dword
uses
     mov    ecx, [y]
     imul   ecx, [h_res]
     mov    edx, [base]
     add    edx, [x]
     mov    al,  [color]
     mov    [ecx+edx], al
endproc
```

`gcc` generated ASM

```
_c_pixel:
     pushl %ebp
     movl  %esp, %ebp      ; EOP
     movl  12(%ebp), %eax
     movl  %eax,     %ecx
     imull 24(%ebp), %ecx
     movl   8(%ebp), %eax
     movl  20(%ebp), %edx
     addl  %eax,     %edx
     movb  16(%ebp), %al
     movb  %al,     (%ecx,%edx);
     popl  %ebp            ; BOE
     ret
```

- `pixel()` is simple, and thus hard to optimize
- Only some programmers can hand-write speedy assembly
- Function epilog and prolog account for 1/3 of the instructions

Call to `pixel()` from C

- In addition, there is also an overhead to call `pixel()`:

```
pushl   32(%ebp)
pushl   8(%ebp)
pushl   28(%ebp)
pushl   -8(%ebp)
pushl   -4(%ebp)
call    _c_pixel
addl    $20, %esp
```

- Overall, the instructions required to call `pixel()` and `pixel`'s prolog and epilog represent about 50% of all instructions required to execute `pixel()`
- This overhead can be eliminated by using
  - Either C macros, i.e. `#define`, if every ns counts and you have plenty of time for debugging
  - Or inline functions, otherwise
    - Inline functions should be preferred because the compiler can detect errors

Example 2: `memcpy()`

Double Buffering

- Double-buffering refers to the use of a memory region (buffer) in addition to the VRAM region, which is displayed by the graphics adapter
- With a resolution of 1024 $\times$ 768, with 8 bits per color, we need a buffer of 768 KByte
- With 30 fps, we need a to transfer approximately: 30 $\times$ 768 $\times$ 2 = 46 MByte/s
- If our budget allocates 20% for this task, we'll need a memory bandwidth of 230 MByte/s

Platforms

	Pentium 3	Pentium-M	AMD 64	Pentium 4
Clock (MHz)	500	1600	2400	3200
FSB (MHz)	112	100	200	800
L1 Cache (MB/s)	4940	19000	19800	22735
L2 Cache (MB/s)	647	9000	4700	17389
Main memory (MB/s)	231	923	2300	1600

: Courtesy of Memtest86

- All CPUs use 32-byte cache lines

Optimization Techniques

Operand sizes

Loop unrolling

MMX registers

SSE

- Avoiding cache polution
- Prefetching

Varying Operand Sizes

One-byte at a time using a char pointer

```
void memcpy_charp(char *dst, char *src, int count) {
    int i;
    for( i = 0; i < count; i++ )
        *dst++ = *src++;
}
```

4-bytes at a time using long pointer (should be 4-byte aligned)

```
void memcpy_longp(char *dst, char *src, int count) {
    int i;
    long *d = (long *) dst,
         *s = (long *) src;
    for( i = 0; i < count/sizeof(long); i++ )
        *d++ = *s++;
}
```

8-bytes at a time using long long pointer (should be 8-byte aligned)

```
void memcpy_llongp(char *dst, char *src, int count) {
    int i;
    long long *d = (long long *) dst,
              *s = (long long *) src;
    for( i = 0; i < count/sizeof(long long); i++ )
        *d++ = *s++;
```

Operand Sizes: Results

- Transfer rates in MByte/s
- Average of 10 runs for blocks of 100 MByte

	P3 700 MHz	PentiumM 2.6 GHz	Athlon64 2.2 GHz	Xeon 2 GHz
charp	47	109	120	123
shortp	105	236	258	347
intp	168	389	461	687
longp	167	389	459	667
llongp	185	474	676	1077
memcpy	202	972	1208	2199

source: João Cardoso

Libc's `memcpy()`

`MOVSB` move (string) byte at a time from [ESI] to [EDI]

`MOVSL` move (string) a 4 bytes at a time from [ESI] to [EDI]

`REP` repeat instruction ECX times (for string operations)

```
...       ; prologue
...       ; setup ESI, EDI, ECX
rep movsb; move non-aligned bytes a byte at a time
...       ; setup ESI, EDI, ECX
rep movsl; move aligned longs a long at a time
...       ; setup ESI, EDI, ECX
rep movsb; move-byte non
...       ; epilogue
```

Naïve memcpy

```
void memcpy_movsd(char *dst, char *src, int count) {
    asm("pushl %%esi                                \n"
        "pushl %%edi                                \n"
        "cld              # clear direction flag    \n"
        "movl %0, %%esi # dst, 1st arg              \n"
        "movl %1, %%edi # src, 2nd arg              \n"
        "movl %2, %%ecx # count, 3rd arg            \n"
        "shrl $2, %%ecx # count /= 4                \n"
        "rep movsl                                  \n"
        "popl %%edi                                 \n"
        "popl %%esi                                 \n"
        : : "g" (src), "g" (dst), "g" (count));
}
```

- Transfer rates in MByte/s
- Average of 10 runs for blocks of 100 MByte

	P3 700 MHz	PentiumM 2.6 GHz	Athlon64 2.2 GHz	Xeon 2 GHz
memcpy	202	972	1208	2199
movsd	200	972	1194	2149

source: João Cardoso

Using MMX Registers

`MOVQ` MMX instruction, 8 bytes at a time

```
void memcpy_mmx(char *dst, char *src, int count) {
    asm(...
        "movl $8, %%eax          # 8 bytes \n"
        "loop:                            \n"
        "    movq (%%ecx, %%esi), %%mm0   \n"
        "    movq %%mm0, (%%ecx, %%edi)   \n"
        "    subl %%eax, %%ecx            \n"
        "    jnz loop                     \n"
        "end:                             \n"
        ...
        : : "g" (src), "g" (dst), "g" (count));
}
```

- Transfer rates in MByte/s
- Average of 10 runs for blocks of 100 MByte

	P3 700 MHz	PentiumM 2.6 GHz	Athlon64 2.2 GHz	Xeon 2 GHz
llongp	185	474	676	1077
memcpy	202	972	1208	2199
mmx	185	507	1374	2099

MMX and Loop Unrolling

- The goal is to increase speed by reducing:
  - The overhead due to loop control instructions
  - Branch penalties (due to instruction prefetching)
  - Hiding latencies when reading from memory

```
"loop:                                  \n"
"    movq (0, %%esi), %%mm0             \n"
"    movq (8, %%esi), %%mm1             \n"
"    movq (16, %%esi), %%mm2            \n"
...
"    movq (56, %%esi), %%mm7            \n"
"    movq %%mm0, (0, %%edi)             \n"
"    movq %%mm1, (8, %%edi)             \n"
"    movq %%mm2, (16, %%edi)            \n"
...
"    movq %%mm7, (56, %%edi)            \n"
...
"    jnz loop                           \n"
```

MMX and Avoiding Cache Pollution (SSE)

- For caching purposes, data reference patterns can be:
  Temporal data will be used again soon
  Spacial data in adjacent positions will be used (soon)
  Non-temporal data is referenced but not reused (in the immediate future) – e.g. VRAM in double-buffering

`MOVNTQ` don't use cache when writing, i.e. write directly to VRAM

```
"loop:                                  \n"
"    movq (0, %%esi), %%mm0             \n"
"    movq (8, %%esi), %%mm1             \n"
"    movq (16, %%esi), %%mm2            \n"
...
"    movq (56, %%esi), %%mm7            \n"
"    movntq %%mm0, (0, %%edi)           \n"
"    movntq %%mm1, (8, %%edi)           \n"
"    movntq %%mm2, (16, %%edi)          \n"
...
"    movntq %%mm7, (56, %%edi)          \n"
"    jnz loop                           \n"
```

Summary of MMX Results

- Transfer rates in MByte/s
- Average of 10 runs for blocks of 100 MByte

Clock	P3 700 MHz	PentiumM 2.6 GHz	Athlon64 2.2 GHz	Xeon 2 GHz
FSB	112 MHz	100 MHz	200 MHz	800 MHz
memcpy	202	972	1208	2199
mmx	185	507	1374	2099
mmx_lu	185	502	1410	1932
mmx_nc	634	547	1618	3083

source: João Cardoso

- MMX registers alone do not beat string operations on Intel
- Loop unrolling has marginal benefits for memory blocks this large
- Avoiding cache polution (SSE) can have a very large effect on performance

Using XMM Registers with SEE2 Instructions

`MOVDQA` XMM instruction, 16 bytes (aligned) at a time

```
void memcpy_xmm(char *dst, char *src, int count) {
    asm(...
        "movl $16, %%eax          # 16 bytes \n"
        "loop:                               \n"
        "    movdqa (%%ecx, %%esi), %%xmm0   \n"
        "    movdqa %%xmm0, (%%ecx, %%edi)   \n"
        "    subl   %%eax, %%ecx             \n"
        "    jnz loop                        \n"
        "end:                                \n"
        ...
        : : "g" (src), "g" (dst), "g" (count));
}
```

Using XMM Registers and Prefecthing (SSE)

`PREFETCHNTA` L2 cache prefetch

`MOVNTQ` don't use cache when writing, i.e. write directly to VRAM

```
void memcpy_xmmx(char *dst, char *src, int count) {
    asm(...
        "# prefetch next loop, non-temporal \n"
        "prefetchnta 64(%%esi)              \n"
        "prefetchnta 96(%%esi)              \n"
        "movdqa (%%esi, %%ecx), %%xmm0      \n"
        "# to dest, avoid cache pollution   \n"
        "movntdq %%xmm0, (%%edi, %%ecx)     \n"
        ...
        : : "g" (src), "g" (dst), "g" (count));
}
```

Summary of XMM Results

- Transfer rates in MByte/s
- Average of 10 runs for blocks of 100 MByte

Clock	P3 700 MHz	PentiumM 2.6 GHz	Athlon64 2.2 GHz	Xeon 2 GHz
FSB	112 MHz	100 MHz	200 MHz	800 MHz
memcpy	202	972	1208	2199
mmx	185	507	1374	2099
mmx_nc	634	547	1618	3083
xmm	184	510	1392	2149
xmm_nc_pref	459	642	1766	2916

source: João Cardoso

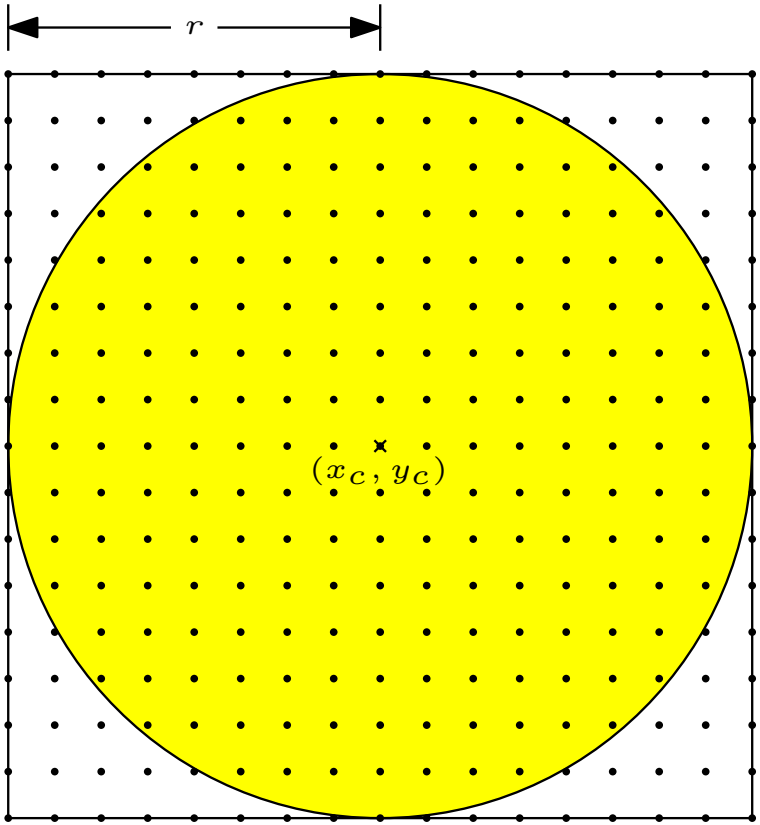
- There are not major differences between MMX and XMM
  - There should be no major impact on reads, which are performed a cache line at a time, independently of the size of the data read
  - The major difference should be on writes
- Again avoiding cache polution (SSE) (with prefetching) can have a very high effect on performance

Circle Drawing

- No sprites
- Based on the mathematical definition:

$$\sqrt{(x - x_c)^2 + (y - y_c)^2} \leq r$$

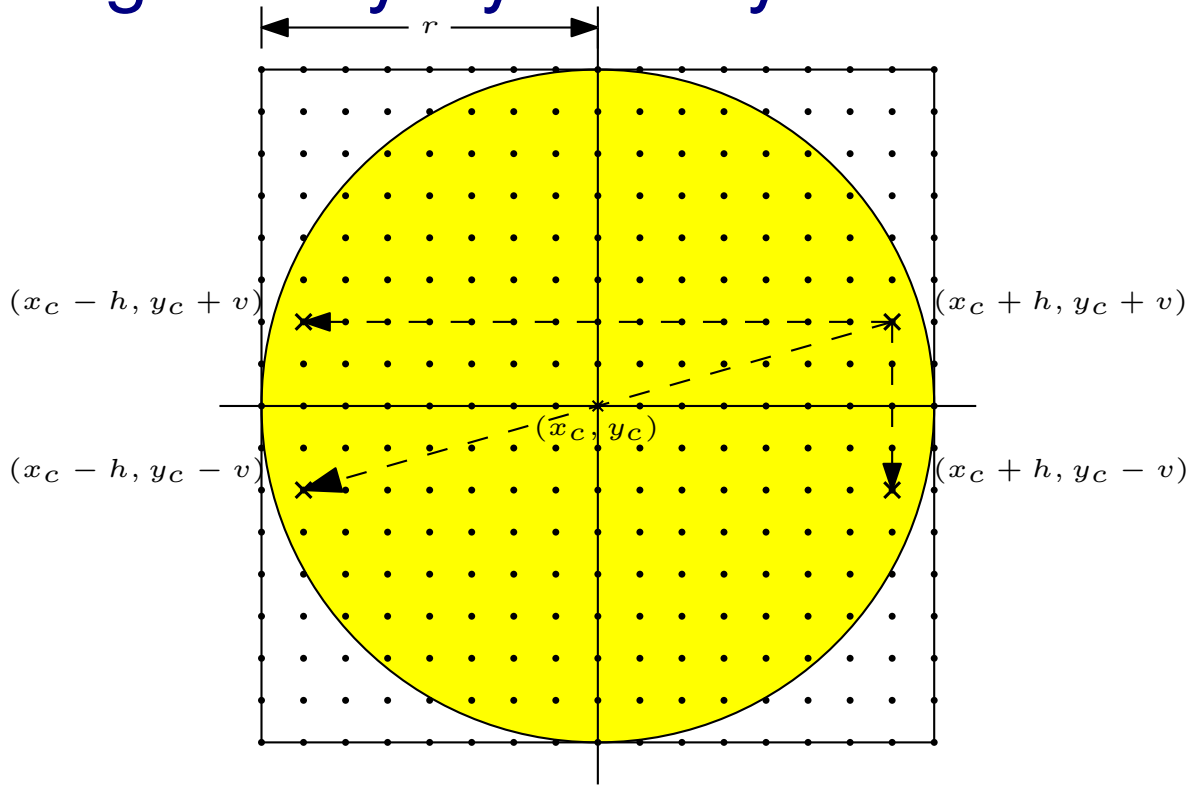
Circle Drawing: Naïve



$$\sqrt{(x - x_c)^2 + (y - y_c)^2} \leq r$$

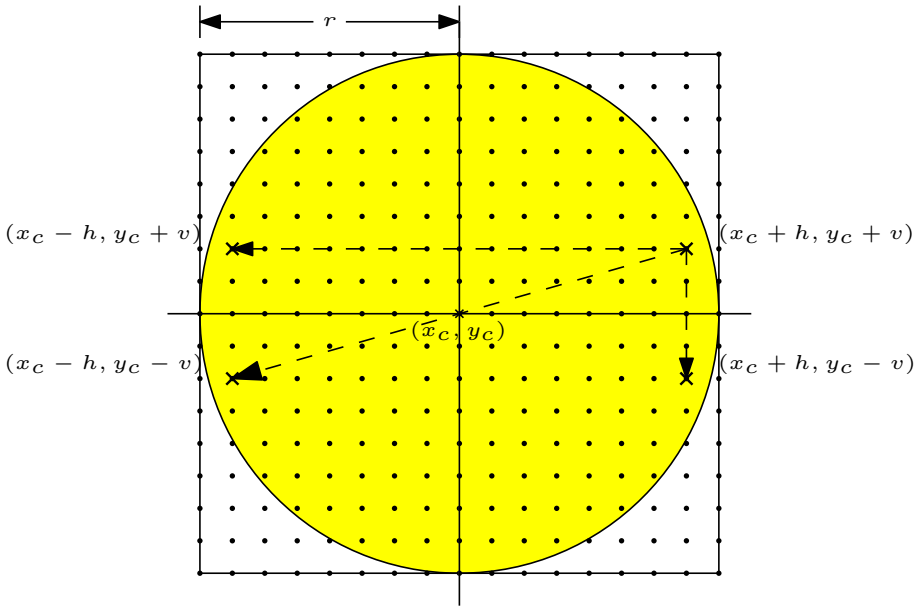
```
for(v = -r; v <= r; v++)
    for(h = -r; h <= r; h++)
        if( sqrt(pow(v,2) + pow(h,2)) <= r )
            vg_set_pixel(x+h, y+v, color);
```

Circle Drawing: 4-way Symmetry




```
for(v = 0; v <= r; v++)
    for(h = 0; h <= r; h++)
        if( sqrt(pow(v,2) + pow(h,2)) <= r ) {
            vg_set_pixel(x+h, y+v, color);
            vg_set_pixel(x+h, y-v, color);
            vg_set_pixel(x-h, y+v, color);
            vg_set_pixel(x-h, y-v, color);
        }
```

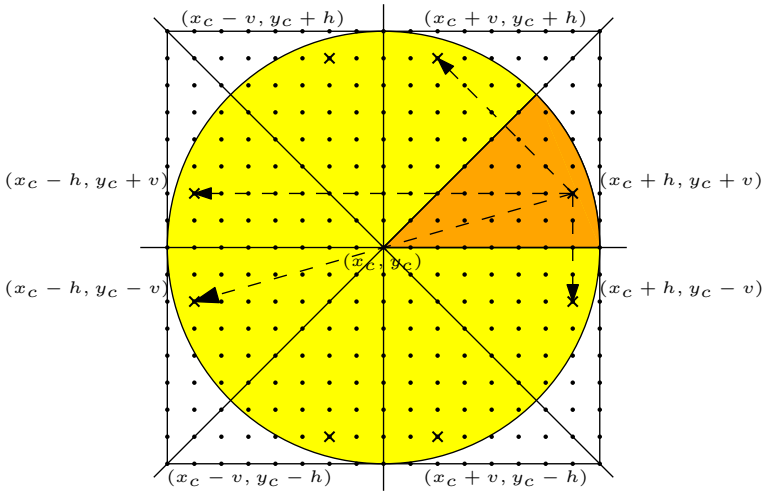
Circle Drawing: 4-way Symmetry without `libm`



$$(x - x_c)^2 + (y - y_c)^2 \leq r^2$$

```
for(v = 0; v <= r; v++)
    for(h = 0; h <= r; h++)
        if( v*v + h*h) <= r*r ) {
            vg_set_pixel(x+h, y+v, color);
            vg_set_pixel(x+h, y-v, color);
            vg_set_pixel(x-h, y+v, color);
            vg_set_pixel(x-h, y-v, color);
        }
```

Circle Drawing: 8-way Symmetry without `libm`



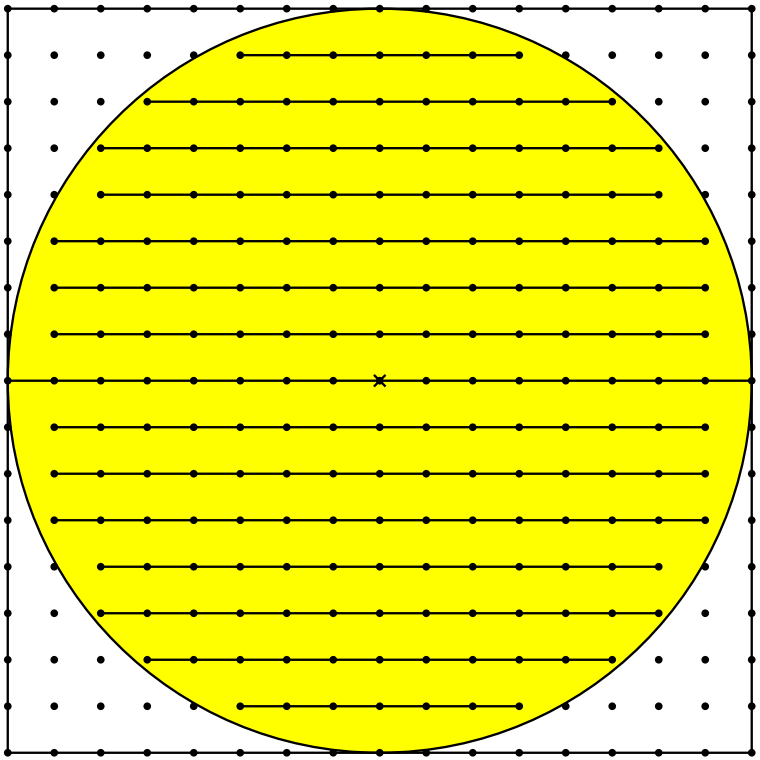
$$(x - x_c)^2 + (y - y_c)^2 \leq r^2$$

```
for(v = 0; v <= r; v++)
    for(h = 0; h <= v; h++)
        if( v*v + h*h) <= r*r ) {
            vg_set_pixel(x+h, y+v, color);
            vg_set_pixel(x+h, y-v, color);
            vg_set_pixel(x-h, y+v, color);
            vg_set_pixel(x-h, y-v, color);
            vg_set_pixel(x+v, y+h, color);
            vg_set_pixel(x+v, y-h, color);
            vg_set_pixel(x-v, y+h, color);
            vg_set_pixel(x-v, y-h, color);
        }
```

Circle Drawing: Using `draw_line()`

Idea If we know the points in the circumference we can use `draw_line()` instead of `set_pixel()`

- Need not check all the points inside the circle



Problem How do we find the points of a cicumference in an efficient way?

Circumference Drawing (?Filipe and Tiago?)

Idea

- Take advantage of 8-way symmetry
- Start from a known point, e.g. on the x-axys
- Increment y, unless out of circle, in which case you should increment x.

Proof

Assumptions

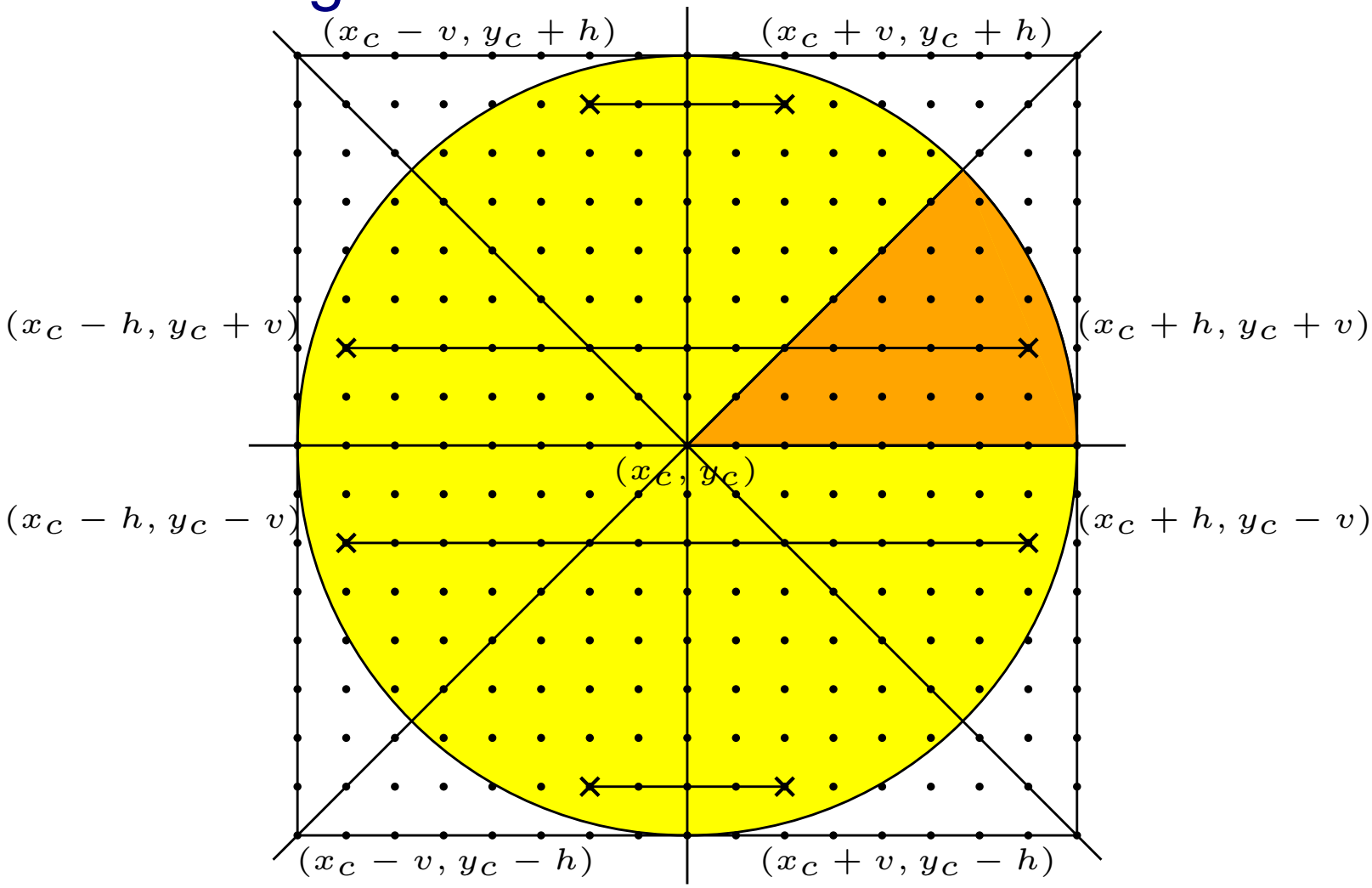
- (x, y) is on the circumference (with center $(0, 0)$)
- $(x, y + 1)$ is not

In the first octant $x \geq y$

Therefore $(x - 1, y + 1)$ is on the circumference

Verification Code algorithm and use `assert()`

Circle Drawing: From Circumference



- Not so straightforward
  - If only horizontal lines, may draw lines on the same $|y| > r/\sqrt{2}$
  - If use vertical lines, redraws points on $|y| < r/\sqrt{2}$

Results (Preliminary)

- Time to draw a circle with radius of 300 pixels (10 measurements)

	no-power	-O3	circle4	-O3	circle8	-O3	line	-O3
1	479952	340568	13450	17123	12693	2327	4278	1603
2	294016	301793	3700	1831	4822	2292	4132	1458
3	342395	322193	3747	1070	2796	1035	3435	577
4	346438	343001	3288	826	2709	2104	2959	1462
5	322981	338090	3389	816	3862	996	6500	586
6	301371	325957	192	843	4013	996	6593	599
7	292796	311701	2796	820	2827	1003	3963	583
8	330127	314924	3616	1678	2839	996	3085	590
9	307018	298670	3067	801	3940	1053	4155	622
10	321516	320263	2695	800	3019	1094	3146	574

- Unit is in μs
- Numbers should be used only for comparison purposes, may not be reliable
  - Obtained using the `RDTSC` (read time-stamp counter) on VMware

Conclusions

Writing code in assembly is rarely the right way to make your code faster

- To write efficient code a programmer needs to master the processor's architecture
  - Today's computer architectures are very complex
- Most of the time optimizing compilers produce better results than most programmers
  - "Sometimes its difficult to understand why the optimization techniques work, but they do . . . "
- Furthermore, more recent hardware does some optimizations
- If, anyway, you want to try your chance try the functions where your code spends most of the time
  - Use a code profiler like `gprof` to learn what these functions are
- You'll probably gain most investing time on optimizing application-related algorithms and data structures

Contents

Further Reading and Acknowledgments

Further Reading

- `gprof` Manual
- Intel 64 and IA-32 Architectures Optimization Reference Manual
- Michael Abrash Graphics Programming Black Book

Thanks to

- João Cardoso not only for the transparencies on which these are based, but also for the design and the results of all experiments but circle drawing presented herein
- Filipe Oliveira and Tiago Azevedo with whom I discussed circle drawing (and for proposing the algorithm based on the circumference)