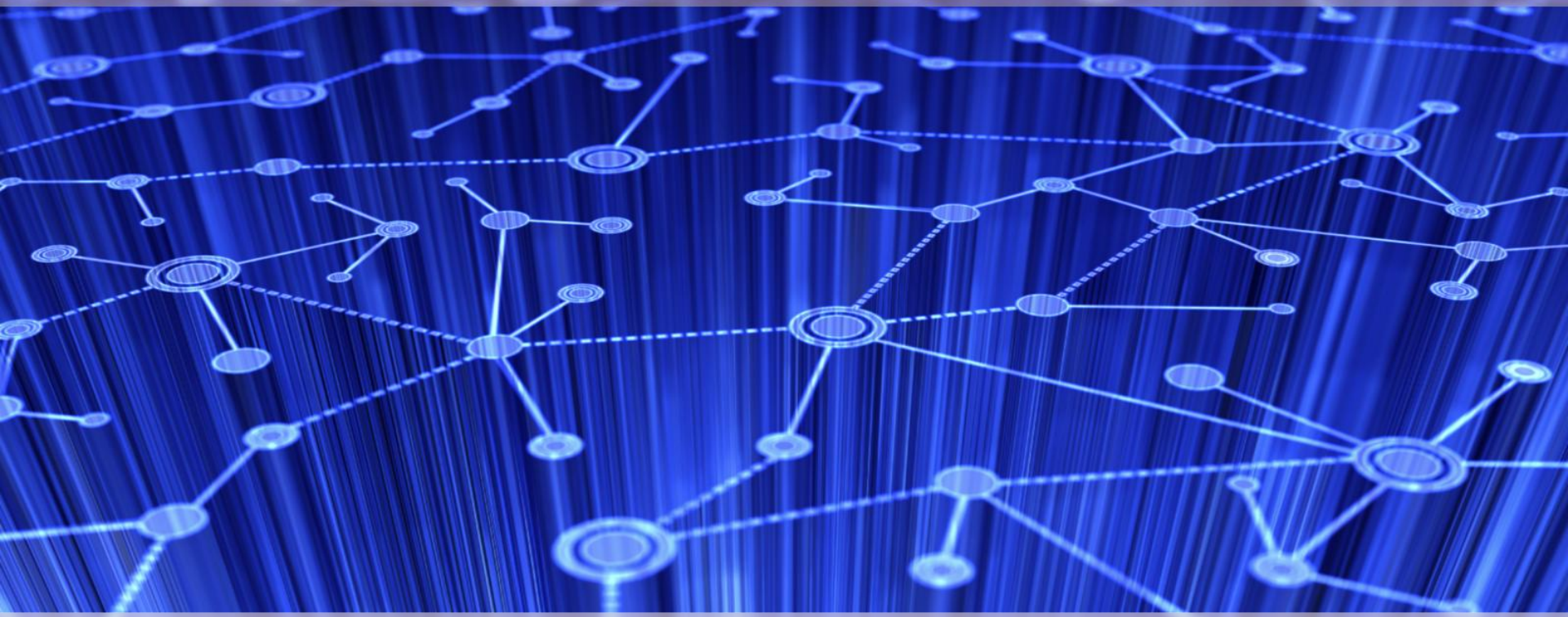


Foundations of High Performance Computing

Code optimization

Part II – Where to start from

Luca Tornatore – I.N.A.F

Code Optimization, part II – Outline

- Mapping the code: **call tree**
- Mapping the code: **code coverage**
- Profiling the code: **tools**
- Profiling the code **behaviour**
  - Hotspots
  - Bottlenecks
  - Resources utilization
- Profiling the code **efficiency**
  - Instructions/cycles
  - L1/L2/RAM hits
  - Branch-misses
- The **Roofline model**

Recap from Part I

If :

- You have a **clean** code
- You have a **good design**
- You have ruminated enough your **algorithms**
- You have **verified** and **validated** the code
- You have clear **ideas** about your needs: memory, time-to-solution, energy, …

… you can start “optimizing”

Where to start from

What would you do to start optimizing your code ?

… let's put some idea on the blackboard before going on…

Of course.. you need a map!

Of course.. you need a map!

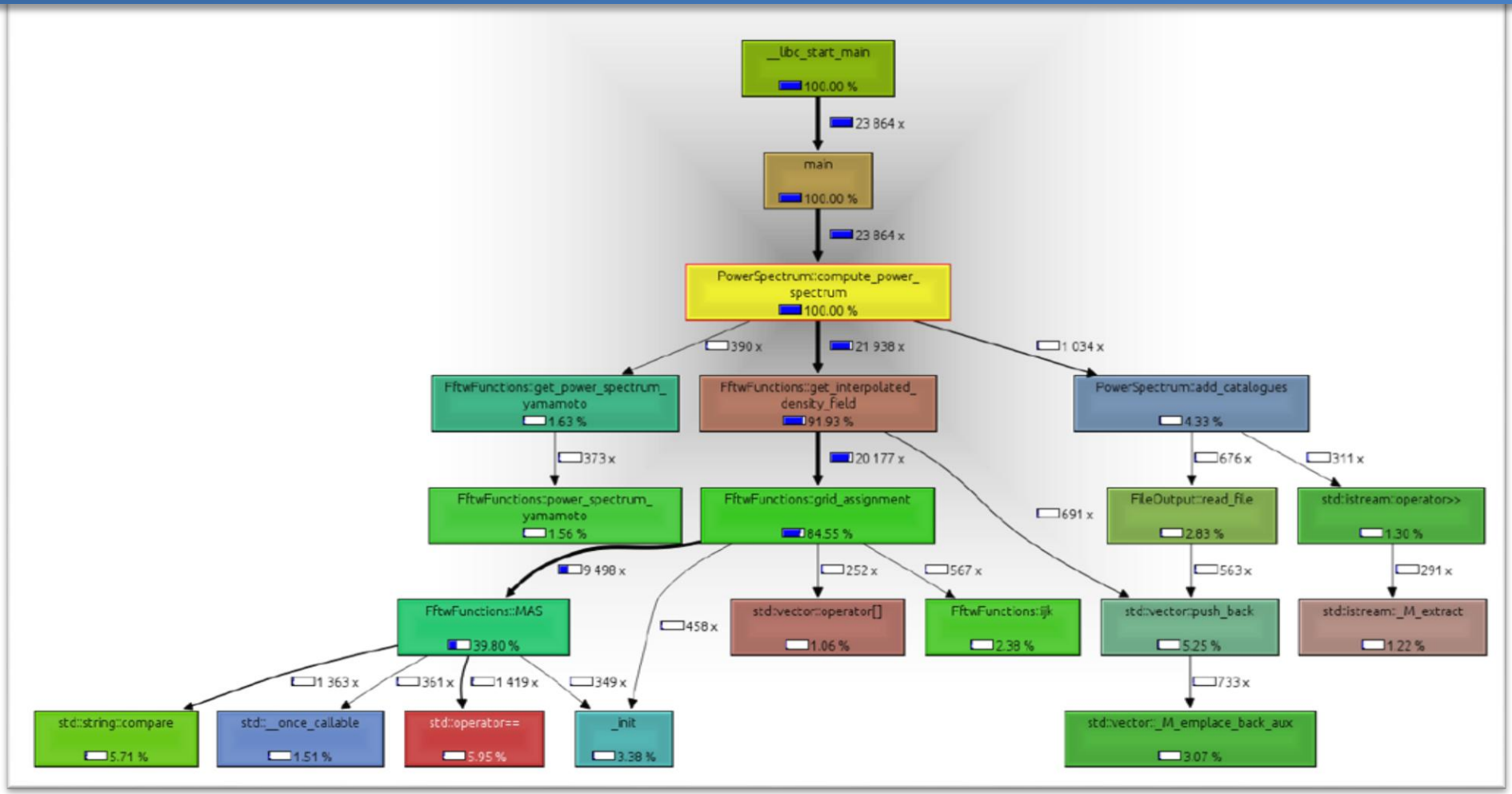
Mapping the workflow

A call tree (more precisely, a **call graph**) is a control flow graph that exhibits the calling relationships among routines in a program.
A *node* in the graph represent a routine, while an *edge* represents a calling relationship.

We'll concentrate on ***dynamic* call graphs** – i.e. the **records of program executions**.

The most complete graph is context-sensitive, which means that every call stack of a procedure is recorded as a separate node (the resulting graph is called *calling context tree* instead of *call tree*).
However, that requires a larger amount of memory for large program, it is useful in case of code reuse (the same code being executed at different points by different call paths).

Obtain the call-tree



The resulting dynamic call tree from the sampling of a code run.
It is a context-insensitive graph, reporting the cumulative # of calls.

Obtain the call-tree

There are several way to obtain a dynamic call tree.
The main open source alternatives are:

1. Using gcc and **gprof**
2. Using linux **google perftools**
3. Using **valgrind**
4. Using **perf**
5. Some others, mostly non-free (among significant free: CodeXL by AMD). We'll mention them later.

Many IDE uses the aforementioned tools or their own plug-ins (eclipse, netbeans, code::block, codelite, …)

Obtain the call-tree | gcc + gprof

Using **gcc** + **gprof**

Just compile your source using **–pg** option.
You should also profile turning on the optimizations you're interested in.

```
gcc –[my.optimizations] –pg –o myprogram.x myprogram.c
```

Note: don't use the option `-p`*. It provides less information than* `-pg`*.*

After that, run your program normally. Profiling infos will be written in the file `gmon-out`.

- You can read (read the man pages for options and details) the informations by

```
gprof myprogram.x
```

- You can visualize the call graph by (read the man pages…)

```
gprof myprogram.x | gprof2dot.py | dot –T png –o callgraph.png
```

➔ **pay attention to the example to understand how to read results**

Obtain the call-tree | gcc + gprof

Well, now that you just met, say goodbye to the glorious **`gprof`** ?

He's a dinosaur from the past decades..

- lacks real multithread support
- lacks real line-by-line capability
- does not profile shared libraries
- need recompilation
- may lie easier than other tools (see later..)

You may still consider it for some call counts business

Using **gperftools**

That is the CPU profiler used at Google's. It provides a **thread-caching malloc**, a **heap-checker**, **heap-profiler**, **CPU profiler**.

As for the latter, basically there are 3 phases:

1. linking the library to the executable
2. running the executable
3. analyzing results

[1] LINKING

There are two options:

- Link `-lprofiler` at the executable
- Adding the profiler at run time
  `LD_PRELOAD="/usr/lib/libprofiler.so" /path/to/exec`

This does not start the CPU profiling, though.

Using **gperftools**

[1] LINKING

There are two options:

- Link `-lprofiler` at the executable
- Adding the profiler at run time
  `LD_PRELOAD="/usr/lib/libprofiler.so" /path/to/exec`

This does not start the CPU profiling, though.

Using **gperftools**

[2] RUNNING

- Define

```
env CPUPROFILE= exec.prof /path/to/exec
```

You may define a signal, too

```
env CPUPROFILE= exec.prof /path/to/exec \
CPUPROFILESIGNAL= XX exec.prof &
```

so that to be able to trigger the start and stop of the profiling by

```
killall -XX exec
```

- In the code, include `<gperftools/profiler.h>` and encompass the code segment to be profiled within

```
ProfilerStart("name_of_profile_file")
…
ProfilerStop()
```

`CPUPROFILE_FREQUENCY=x` modifies the sampling frequency

Using **gperftools**

[3] ANALYZING RESULTS

`pprof exec exec.prof`	"interactive mode"
`pprof --text exec exec.prof`	output one line per procedure
`pprof --gv exec exec.prof`	**annotated call graph via 'gv'**
`pprof --gv --focus = some_func …`	restrict to code paths including "*some_func*"
`pprof --list = some_func …`	per-line annotated list of some_func
`pprof --disasm= some_func …`	annotated disassembly of some_func
`pprof -callgrind exec exec.prof`	**output call infos in callgrind format**

Obtain the call-tree | perf

Using **perf**

Mapping a code

- Mapping the code: call tree
- Mapping the code: **code coverage**
- Profiling the code: tools
- Profiling the code behaviour
  - Hotspots
  - Bottlenecks
  - Resources utilization
- Profiling the code efficiency
  - Instructions/cycles
  - L1/L2/RAM hits
  - Branch-misses
- The Roofline model

Code coverage

Using **gcc** + **gcov**

Just compile your source using **–g --coverage** option (equals to **`-ftest-coverage -fprofile-arcs`**)
You should also profile turning on the optimizations you're interested in.

```
gcc -[my.optimizations] -g --coverage -o myprog.x myprog.c
```

After that, run your program normally. Coverage infos will be written in the files `*.gcda` in the source directory

- You can produce the `*gcov` files that contains informations by

```
gcov *.c
```

- read the man page to learn how to use `-a, -b, …` options of `gcov`

➔ **pay attention to the example to understand how to read results and use `gcovr` to produce a better-looking html report**

Mapping a code

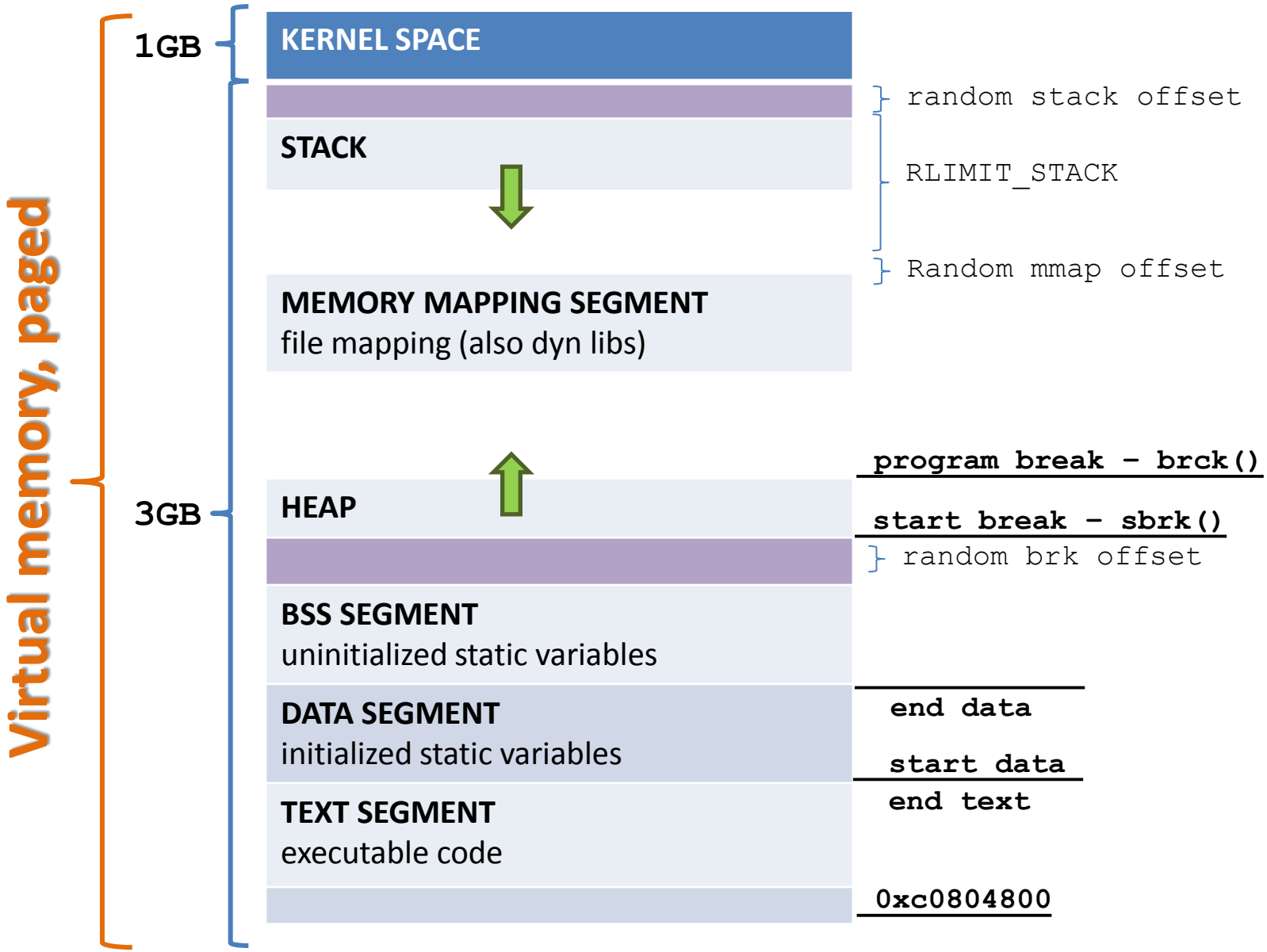
- Mapping the code: call tree
- Mapping the code: code coverage
- Profiling the code: **tools**
- Profiling the code behaviour
  - Hotspots
  - Bottlenecks
  - Resources utilization
- Profiling the code efficiency
  - Instructions/cycles
  - L1/L2/RAM hits
  - Branch-misses
- The Roofline model

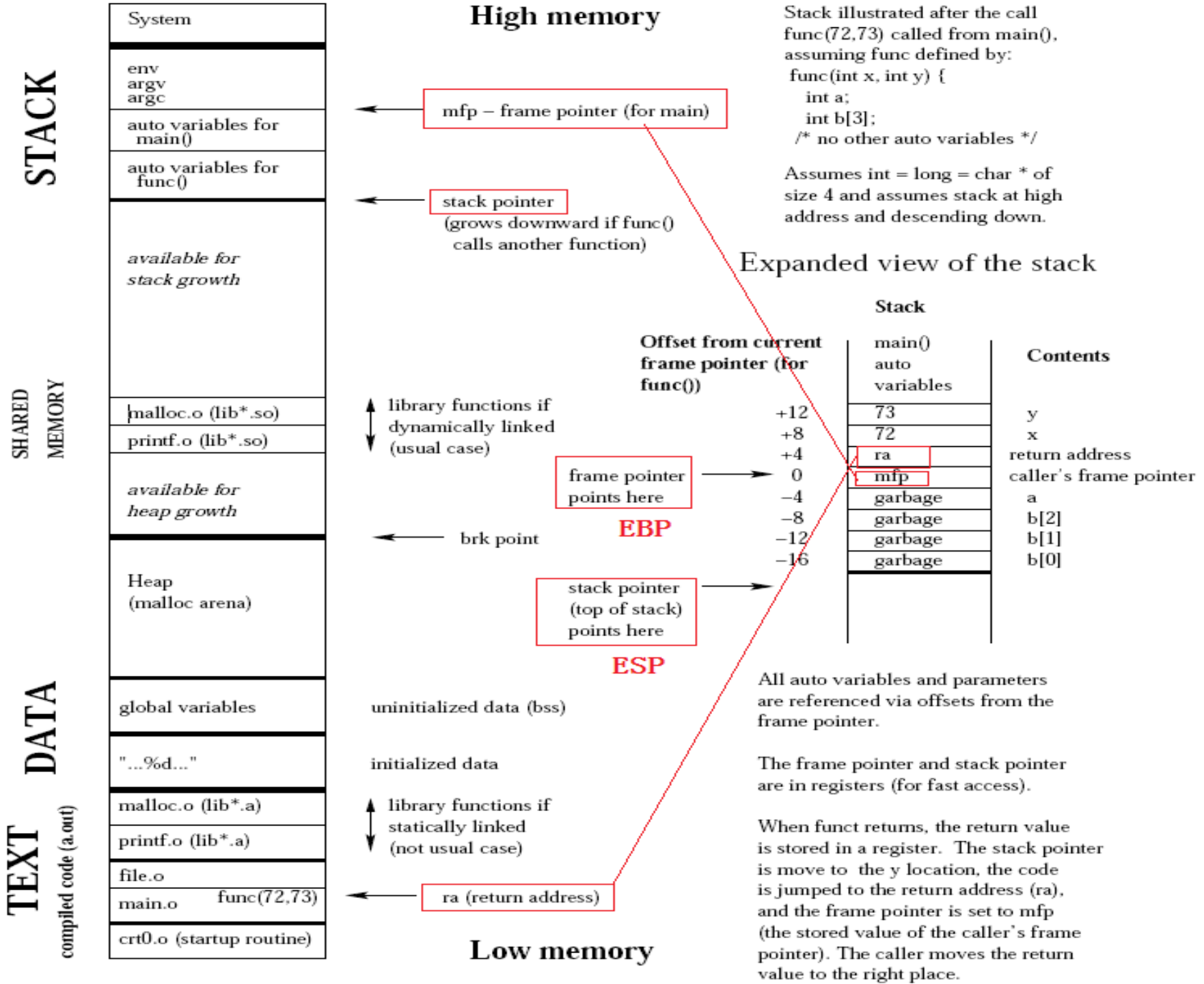
Understanding the code execution

For a basic understanding of **how a compiler works**, you also need a basic understanding of **how the memory and program execution works**.

What I'll show you know is mainly valid for LinuX, although *mutatis mutandis* it broadly holds for all *niX systems and for Windows systems too.

When the program met the memory




STACK
SHARED MEMORY
DATA
TEXT
compiled code (a.out)
System
High memory
env
argv
argc
auto variables for main()
auto variables for func()
available for stack growth
malloc.o (lib*.so)
printf.o (lib*.so)
available for heap growth
Heap (malloc arena)
global variables
"...%d..."
malloc.o (lib*.a)
printf.o (lib*.a)
file.o
main.o
func(72,73)
crt0.o (startup routine)
Low memory
mfp – frame pointer (for main)
stack pointer (grows downward if func() calls another function)
library functions if dynamically linked (usual case)
frame pointer points here
EBP
brk point
stack pointer (top of stack) points here
ESP
uninitialized data (bss)
initialized data
library functions if statically linked (not usual case)
ra (return address)
Stack illustrated after the call func(72,73) called from main(), assuming func defined by:
func(int x, int y) {
int a;
int b[3];
/* no other auto variables */
Assumes int = long = char * of size 4 and assumes stack at high address and descending down.
Expanded view of the stack
Stack
Offset from current frame pointer (for func())
main() auto variables
Contents
+12
+8
+4
0
−4
−8
−12
−16
73
72
ra
mfp
garbage
garbage
garbage
garbage
y
x
return address
caller's frame pointer
a
b[2]
b[1]
b[0]
All auto variables and parameters are referenced via offsets from the frame pointer.
The frame pointer and stack pointer are in registers (for fast access).
When funct returns, the return value is stored in a register. The stack pointer is move to the y location, the code is jumped to the return address (ra), and the frame pointer is set to mfp (the stored value of the caller's frame pointer). The caller moves the return value to the right place.

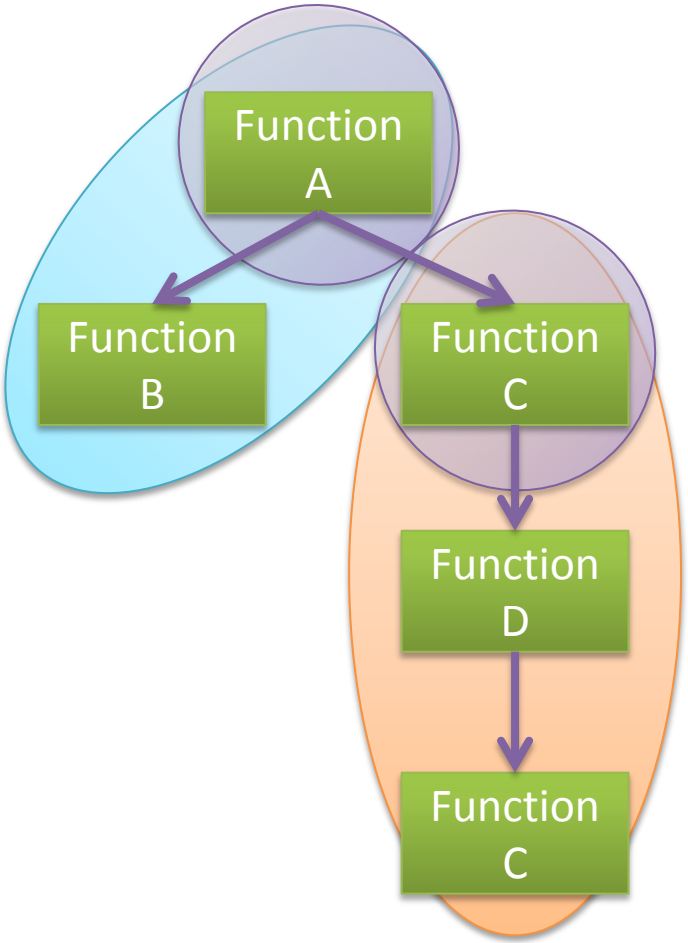
Basic concepts of profilers

Collect **program events**

- Hardware interrupts
- Code instrumentation
- Instruction set simulation
- tracing

Periodic sampling

- Top of the stack (exclusive)
- All stack (children inclusive)



Basic concepts of profilers

INSTRUMENTATION

Inserts extra code at compile time wrapping function calls to count how many times it calls / is called and how much time it takes to execute.

SAMPLING

The profiler ask for interrupts $N_{samples}$ per sec + interrupts at function calls + interrupts at selected events, and records on a histogram the number of occurrences in every part of the program. Infers the call graph.

DEBUGGING

The profiler ask for interrupts at every line code and function call

Your profiler may mislead you

The consequences of the fact that `gprof` (`gcc -pg`) does not record the call stacks

```
#include <stdlib.h>
void loop(int n)
{
  int volatile i; // does not optimize out
  i = 0;
  while(i++ < n);
}

void light(int n) { loop(n); }
void heavy(int n) { loop(n); }

int main(void)
{
  light(100000);
  heavy(100000000);
return 0;
}
```

credits: this example and the following one were found by Y. Krenin

Your profiler may mislead you

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
101.30      0.20     0.20        2   101.30   101.30  loop
  0.00      0.20     0.00        1     0.00   101.30  heavy
  0.00      0.20     0.00        1     0.00   101.30  light
```

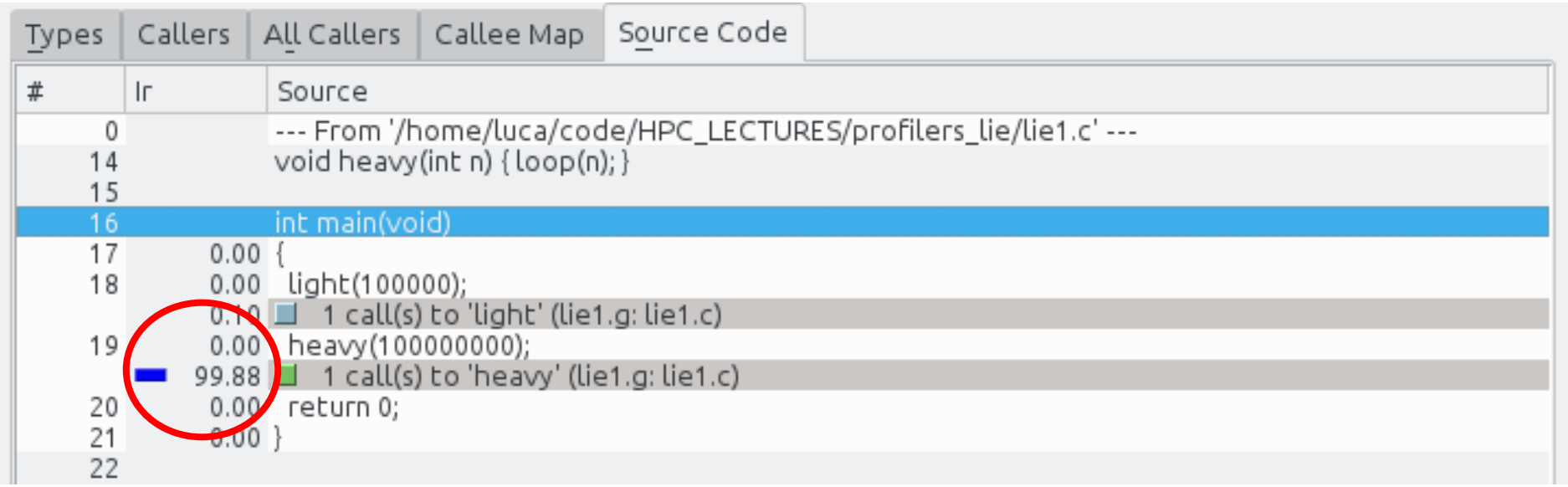
How could it happen ?

The `man page` for `gprof` does'nt lie at all:

We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called.

Thus the time propagated along the call graph arcs to the function's parents is directly proportional to the number of times that arc is traversed.

Your profiler may mislead you



VALGRIND seems to better understand the situation, because it doesn't record only the number of calls to a function but also the time spent in a function given a call path (which, at odds, gprof infers backwardly).
That's way the valgrind measure of time spent in a function plus its callees is reliable.

However, it may as well end up to a misleading picture if we stick in and additional layer of complexity, *unless* you explicit tells it to track separately different call stacks with the command-line option --separate-callers = *N*

➔ ***Let's go live to discuss this second case***

The message is : know your tool *(as always it is)*

- The bad news is that **there are no magic wands**.
  No tool (or compiler's flag) saves you from catching the tiny details of both your code and the tool you use to analyze it.
  Some of the profilers' errors are not fixable increasing the signal-to-noise ratio (i.e. profiling a bulk of runs instead of one): they may be logically, and so deadly, wrong (for instance: cost assignment to context).
  And.. read the documentation.

- The good news is that the same attitude that makes you know and understand **how and when you can trust your profiler**, makes you a good profiler yourself and a good optimizer.

- Learn how to choose your tool.
  Gprof is fast, require special compilation, is stuck to 100 samples/s, has precise call counts but bogus timings (for instance for code reuse).
  Callgrind is slower, requires no dedicated compilation, has very complex and refined capabilities, has precise call counts and events along call paths, may use thousands of samples/s
  Gperftools is fast, requires no special compilation, may use thousands of samples/s, logs complete call stacks

Examples of profilers

- **BASIC / SYSTEM tools**
  - gprof / gdb / perf / gperftools
  - Valgrind – cachegrind, callgrind, ..
- **HARDWARE COUNTER / PMU interface**
  - perf
  - PAPI
  - Intel PMI
  - Likwid
  - …
- **HPC Tools**
  - HPCtoolkit
  - OpenlSpeedShop
  - TAU
  - SCOPEP +
- **VENDOR tools**
  - ARM-Allinea
  - CodeXL (AMD)
  - Intel tools
  - …

Profiling: tools - Valgrind

An instrumentation framework for building dynamic analysis tools.
Valgrind basically runs your code in a virtual "sandobx" where a synthetic CPU (the same you have) is simulated.
Actually it converts x86 instructions in cleaner RISC-like Ucode and executes it appropriately instrumented.

There are various Valgrind based tools for debugging and profiling purposes.

- **Memcheck** is a memory error detector → correctness
- **Cachegrind** is a cache and branch-prediction profiler → velocity
- **Callgrind** is a call-graph generating cache profiler. It has some overlap with Cachegrind
- **Helgrind** is a thread error detector → correctness
- **DRD** is also a thread error detector. Different analysis technique than Helgrind
- **Massif** is a heap profiler → memory efficiency using less memory
- **DHAT** is a different kind of heap profiler → memory layout inefficiencies
- **SGcheck** (experimental tool) that can detect overruns of stack and global arrays

KCacheGrind is a very useful GUI

Profiling: tools - Valgrind

Memcheck helps you in highlighting some common memory errors:

- Unvalid memory access: overrunning/underrunning of heap blocks or top of stack, addressing freed blocks, …
- Use of variables with undefined values
- Incorrect freeing of heap memory
- Errors in moving memory (unwanted src/dst overlaps, …)
- Memory leaks

Cachegrind simulate the interaction of the code with a cache (you can model it).
It can report how many hits (L1, L2 and L3, I- and D-) and how many misses.
It can analyze CPU's branch prediction.

Ir, I1mr, LLmr, Dr, D1mr, DLmr, Bc, Bcm, Bi, Bim, …

Callgrind is a CPU profiler.
It collects the number of instructions executed, links them to source lines, records the caller/callee relationship between functions, and the numbers of such calls.
Optionally, it may collect data on cache simulation and/or branch as Cachegrind does

Mapping a code

- Mapping the code: call tree
- Mapping the code: code coverage
- Profiling the code: tools
- Profiling the **code behaviour**
  - Hotspots
  - Bottlenecks
  - Resources utilization
- Profiling the code efficiency
  - Instructions/cycles
  - L1/L2/RAM hits
  - Branch-misses
- The Roofline model

Profiling code behaviour