

KIV/CPP – Programming in C++

11. Code profiling, finding memory leaks

Martin Úbl

KIV ZČU

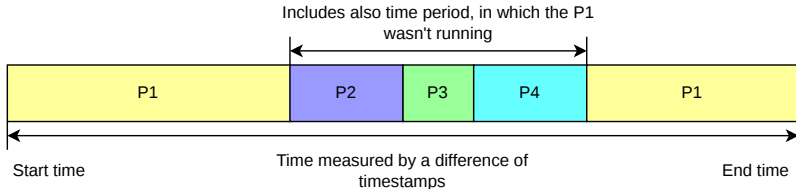
2025/2026

- *performance*
- (*phys.*) amount of work per unit of time
- number of instructions
 - ambiguous
 - each instruction may be executed within a different number of CPU cycles
 - vector instructions – numerically even fractions of CPU cycles
- another metric?
- we abstract to operations, and by extension to time
- in the end, we will mainly care about time anyway
- program performance analysis: profiling

- execution time
- affected by many aspects
 - CPU properties (frequency, instruction set, ...)
 - size of input data
 - amount of data for cache levels
 - RAM speed/throughput
 - architecture properties
 - OS scheduler
 - batch, real-time?
 - preemptive?
 - priority
 - parallelization
 - and many others...
- measurement should be independent of many of them
 - e.g. preemptive scheduling

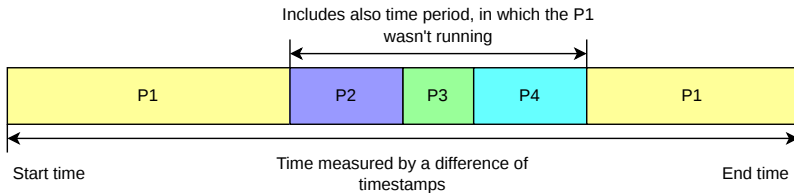
Performance

- execution time
- can be measured at the program level
- `std::chrono::steady_clock` + subtracting start and end time
- returns a timer value with certain properties
- does not filter out time when the process was not running
 - passive waiting
 - preemption by another process



Performance

- sometimes it is appropriate to include this time
- e.g. waiting for I/O operations can be a relevant part of program execution
- while waiting for an I/O operation, preemptive schedulers usually reschedule to other processes



- several ways of profiling
 - instrumentation
 - compiling diagnostic function calls into the program
 - manual vs. automatic
 - requires program modification, some slowdown
 - sampling
 - periodic/event-based sampling of the stack and CS:RIP
 - less precise, but minimal impact on performance
 - interpretation
 - binary code is treated as intermediate code and run in a virtual machine
 - “like Java and bytecode”
 - potentially very precise, but with a huge impact on performance
 - combination

Instrumentation

- instrumentation
- manual
 - collecting timestamps, subtracting times
 - tedious
- automatic
 - the compiler compiles additional calls during compilation
 - number of function calls, runtimes
- often combined with sampling

```
double transform(double a, double b)
{
    __mcount();
    return a*15.5+b;
}
```

- sampling
- performed either based on events or at a regular period
- a system timer can be configured
 - software
 - hardware
- the timer handler samples the program state
- system call `profil()`

- sometimes, therefore, another metric is appropriate
- *Hardware Performance Counters*
- special processor registers that increment their value on an event
- on overflow, they generate an interrupt (NMI)
- the operating system signals the collection program
 - it records the event
 - adds timestamps
 - sometimes a stack snapshot

- *Hardware Performance Counters*
 - on a program counter change (“instruction counter”)
 - on a cache-miss from the processor’s L1/2/3 cache
 - on branch misprediction
 - TLB miss
 - interrupts and system calls
 - and many others
- they are part of the PCB and process context
 - therefore they guarantee isolation from other processes
 - except for shared resources, such as CPU cache and TLB

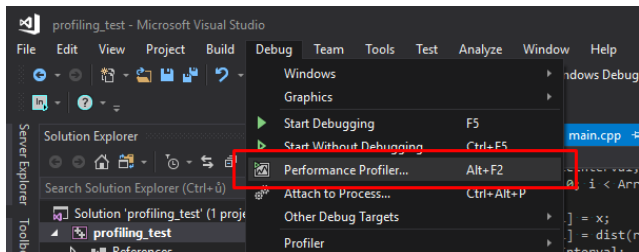
- a program for collecting profiling data – profiler
- usually just a program that uses OS kernel features
- “attaches” to system events
 - timer
 - HPC overflow (NMI)
 - other optional events
- receives the required data structures from the OS
- stores them in some format
 - usually in memory
 - every once in a while, “dumping” to a file

- MS Visual Studio Performance Profiling
 - MS Windows, part of Visual Studio
 - instrumentation, sampling, HPC
- gprof
 - GNU/Linux, macOS, part of gcc
 - instrumentation, sampling, (HPC)
- perf
 - GNU/Linux
 - sampling, HPC
- Intel VTune Amplifier
 - see KIV/PPR

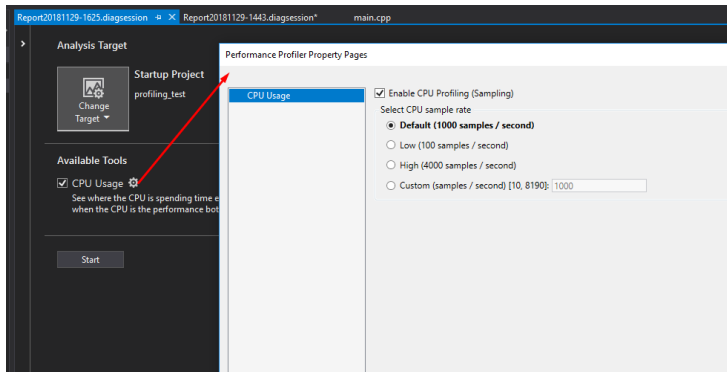
- sometimes outputs can be problematic because of certain optimizations
 - implicit function inlining
 - `-fno-inline-small-functions` (GCC)
 - `/Ob0` (MSVS)
 - not using frame pointers
 - `-fno-omit-frame-pointer` (GCC)
 - `/Oy-` (MSVS)

- time/samples
 - inclusive
 - time/samples in the function and in all functions it calls
 - exclusive
 - time/samples only within the instruction flow of the given function

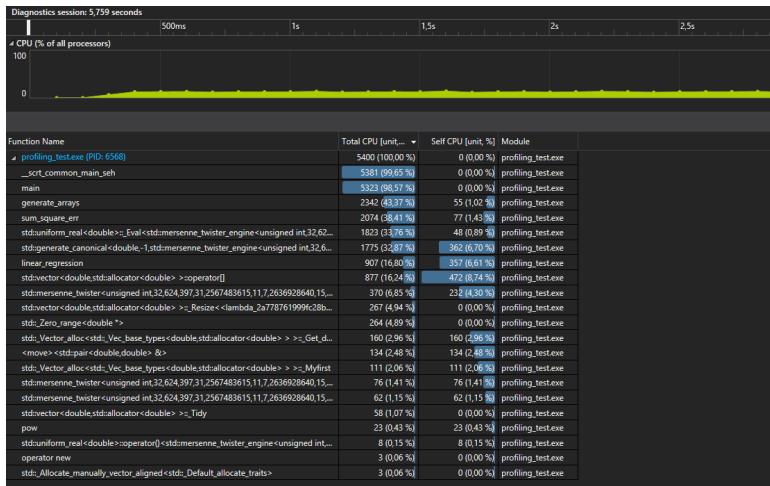
- MS Visual Studio Performance Profiling



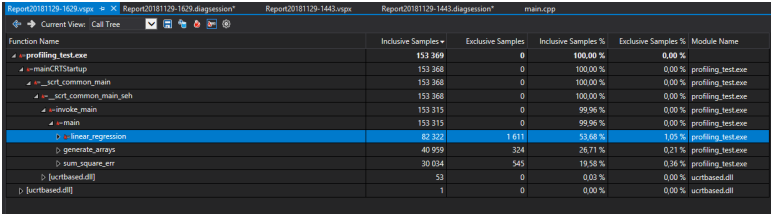
- MS Visual Studio Performance Profiling



- flat view



- call tree



Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %	Module Name
profiling_test.exe	153 369	0	100,00 %	0,00 %	
mainCRTStartup	153 368	0	100,00 %	0,00 %	profiling_test.exe
__scrt_common_main	153 368	0	100,00 %	0,00 %	profiling_test.exe
__scrt_common_main_seh	153 368	0	100,00 %	0,00 %	profiling_test.exe
invoke_main	153 315	0	99,96 %	0,00 %	profiling_test.exe
main	153 315	0	99,96 %	0,00 %	profiling_test.exe
linear_regression	82 322	1 611	53,68 %	1,05 %	profiling_test.exe
generate_arrays	40 959	324	26,71 %	0,21 %	profiling_test.exe
sum_square_err	30 034	545	19,58 %	0,36 %	profiling_test.exe
[ucrtbased.dll]	53	0	0,03 %	0,00 %	ucrtbased.dll
[ucrtbased.dll]	1	0	0,00 %	0,00 %	ucrtbased.dll

- must be compiled with the `-pg` switch
- then run the program normally
- the file `gmon.out` is generated
- analysis with the `gprof` command

```
g++ -pg main.cpp  
./a.out  
gprof
```

- must be run as root
- run with the `perf record` command
 - since we will also want to record the call tree, we add the `-call-graph=fp` switch
- the file `perf.data` is generated
- analysis with the `perf report` command

```
g++ main.cpp  
perf record --call-graph=fp ./a.out  
perf report
```

- perf call tree

Samples: 864 of event 'cycles', Event count (approx.): 672421625

Children	Self	Command	Shared Object	Symbol
+ 97,51%	0,00%	a.out	[unknown]	[k] 0x220e258d4c544155
+ 97,51%	0,00%	a.out	libc-2.27.so	[.] __libc_start_main
+ 97,51%	0,00%	a.out	a.out	[.] main
+ 88,34%	2,28%	a.out	a.out	[.] generate_arrays
+ 39,52%	39,52%	a.out	a.out	[.] std::generate_canonical<double, 53ul, std::mersenne_twister
+ 34,37%	4,86%	a.out	libc-2.27.so	[.] __mcount
+ 30,44%	30,09%	a.out	libc-2.27.so	[.] __mcount_internal
+ 12,45%	2,31%	a.out	a.out	[.] std::vector<double, std::allocator<double> >::_M_default_app
+ 8,23%	1,64%	a.out	[kernel.kallsyms]	[k] async_page_fault
+ 6,59%	0,14%	a.out	[kernel.kallsyms]	[k] __do_page_fault
+ 6,45%	0,35%	a.out	[kernel.kallsyms]	[k] handle_mm_fault
+ 6,10%	1,06%	a.out	[kernel.kallsyms]	[k] __handle_mm_fault
+ 4,20%	4,20%	a.out	a.out	[.] linear_regression
+ 3,64%	3,53%	a.out	a.out	[.] sum_square_err
+ 3,18%	0,11%	a.out	[kernel.kallsyms]	[k] alloc_pages_vma
+ 3,15%	0,22%	a.out	[kernel.kallsyms]	[k] get_page_from_freelist
+ 3,15%	0,00%	a.out	[kernel.kallsyms]	[k] __alloc_pages_nodemask
+ 2,55%	2,55%	a.out	[kernel.kallsyms]	[k] clear_page_erms
+ 1,92%	0,00%	a.out	[kernel.kallsyms]	[k] entry_SYSCALL_64_after_hwframe
+ 1,92%	0,00%	a.out	[kernel.kallsyms]	[k] do_syscall_64
+ 1,77%	0,00%	a.out	[kernel.kallsyms]	[k] do_munmap
+ 1,77%	0,00%	a.out	[kernel.kallsyms]	[k] unmap_region
+ 1,73%	0,00%	a.out	libc-2.27.so	[.] __munmap
+ 1,73%	0,00%	a.out	[kernel.kallsyms]	[k] __x64_sys_munmap
+ 1,73%	0,00%	a.out	[kernel.kallsyms]	[k] vm_munmap
+ 1,40%	0,80%	a.out	[kernel.kallsyms]	[k] error_entry
+ 1,37%	0,68%	a.out	[kernel.kallsyms]	[k] unmap_page_range
+ 1,37%	0,00%	a.out	[kernel.kallsyms]	[k] unmap_vmas
+ 1,04%	0,23%	a.out	[kernel.kallsyms]	[k] release_pages

- memory leaks
- forgotten deallocation
- skipped deallocation due to incorrect program control flow
- for a small single-purpose program with minimal allocations, they *may not be a problem*
 - but we should still try not to introduce leaks
- for a large system, for example an interactive one, this is a major problem

- possible solutions
 - do not use dynamic allocation
 - use RAII-wrapped dynamic allocation
 - use whatever we want, but debug the program with a detection tool
- detection tools
 - AddressSanitizer (Clang, gcc, MSVS, ...)
 - CrtDebug (MSVS)
 - Visual Leak Detector (MSVS)
 - Dr. Memory (MS Windows)
 - Valgrind Memcheck (GNU/Linux, macOS)
 - not only memory leaks

- principle of operation
- wrapping allocations in a custom routine
- it records all allocations and deallocations
- anything not freed when the session ends is reported

- classes of memory leaks and memory usage problems
 - new/delete mismatch
 - allocation larger than the maximum heap size
 - calloc/alloca overflow
 - double free
 - use after free
 - global variable overflow
 - heap buffer overflow
 - invalid alignment
 - memcpy parameter overlap
 - stack buffer overflow (underflow)
 - use after scope
 - access to “poisoned” memory
 - and others

- AddressSanitizer (ASan)
- compiled directly into the binary
- standard tool, support in LLVM, gcc, MSVS, and others
- functionally similar to Valgrind/CrtDebug, but potentially much faster and more precise
- in MSVS, it can be enabled with `/fsanitize=address` (must be installed additionally)
- in LLVM/gcc, similarly with `-fsanitize=address`

- CrtDebug – setup

```
_CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF  
                | _CRTDBG_LEAK_CHECK_DF  
                | _CRTDBG_CHECK_ALWAYS_DF );  
_CrtSetReportMode( _CRT_WARN, _CRTDBG_MODE_FILE );  
_CrtSetReportFile( _CRT_WARN, _CRTDBG_FILE_STDOUT );
```

- listing of unfreed blocks

```
_CrtDumpMemoryLeaks();
```

- note: to identify the allocation location, new must be redefined

```
#define MYDEBUG_NEW new( _NORMAL_BLOCK, \  
                        __FILE__, __LINE__ )  
#define new MYDEBUG_NEW
```

- creating a heap snapshot, and therefore a “checkpoint” during program execution

```
_CrtMemState state;  
_CrtMemCheckpoint(&state);
```

- listing of unfreed blocks since the given “checkpoint”

```
_CrtMemDumpAllObjectsSince(&state);
```

VLD

- download, installation
 - <https://kinddragon.github.io/vld/>
- adding it to the project

```
#include "<path to vld>\include\vld.h"  
#pragma comment(lib, "<path to vld>/vld_x64.lib")
```

- start

```
VLDEnable();
```

- listing of unfreed blocks

```
VLDReportLeaks();
```

- note: the tool is no longer developed

Valgrind

- thanks to LD_PRELOAD, it overrides allocators without needing to change the code
- no special code or compilation is needed
 - it is only necessary to compile with debug info (the -g switch for g++)
- however, it significantly slows down program execution (20x and more)
- run

```
valgrind --tool=memcheck ./program param1
```

- sometimes it is useful to add flags for origin tracking and maximizing detection

```
valgrind --tool=memcheck --leak-check=full  
        --track-origins=yes ./program param1
```