Profiling, assembler, optimisations

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Profiling

- detection of slow code
- desire to know
  - what is the slow code
  - where the code is called from
  - what is the execution time

Profiling

- detection of slow code
- desire to know
  - what is the slow code
  - where the code is called from
  - what is the execution time

perf

- linux tool for profiling
- a set of utilities
- use hardware counters
  - small overhead (compared to `callgrind` etc.)
- sometimes needs extra permissions
  - not granted on FI
  - for seeing kernel space, multiple processes

Perf – how to use

- compile program with `-g -fno-omit-frame-pointer` and desired optimization level
  - keeps frame pointers so that `perf` can recover call graph
- run `perf record -a --call-graph fp` to gain a call graph
  - `-a` – record on all CPUs (not supported on FI computers)
  - `--call-graph fp` – record call graph based on frame pointers
  - produces `perf.data` file
- run `perf report --stdio` to show the call graph
  - or omit `--stdio` to see curses-based interactive UI (but long C++ symbol names are problem)

Assembler

Assembly language (symbolic machine code)

- low-level; closest to machine code
- commands – machine code instructions

Why do we want to know about it?

- debugging
- computer security
- examine optimisation done by compiler
- sometimes it is good to know what's "under the hood"

Our focus here: reading assembly, not writing it

Tools

Disassemble

- `clang++ -S`, `g++ -S`, etc.
- `gdb`
  - `disassemble`
  - `x/10i` address (such as `$rip`)
  - (`print`, `disp`)
- `objdump -d`

Show raw bytes

- `hexdump -C`
- `xxd`

Assembler notation

Intel

- operands in order *dest*, *src*
  - `mov rax, rbx` moves *from* `rbx` *to* `rax`
  - `add rax, 0x1f` adds `0x1f` *to* `rax`
- memory indexing `[base + index*scale + disp]`
  - `mov eax, [rbx + rcx*4 + 0x10]`

AT&T

- operands in order *src*, *dest*
  - `mov %rbx, %rax`
  - `add $0x1f, %rax`
- memory indexing `disp(base, index, scale)`
  - `movl 0x10(%rbx, %rcx, 4), %eax`
- size indicated in the instruction mnemonic
  - `movb`, `movw`, `movl`, `movq` (1, 2, 4, and 8 bytes)
- immediate values with $, registers with %

Assembler notation

How to use Intel syntax?

- `clang++ -S -masm=intel`
- `objdump -d -M intel`
- `gdb`
  - `set disassembly-flavor intel`

x86(-64) Architecture

Registers

- instruction pointer: `ip` (16 bit), `eip` (32 bit), `rip` (64 bit)
- stack pointer: `sp` (16 bit), `esp` (32 bit), `rsp` (64 bit)
- general purpose: `ax`, `bx`, `cx`, `dx` (`eax`, `rax`, ...)
  - lower 8 bits: `al`, `bl`, `cl`, `dl`
- source/destination: `si`, `di` (`esi`, `rsi`, ...)
- stack frame base pointer: `bp` (`ebp`, `rbp`)
- 64 bit general purpose: `r8`, `r9`, ..., `r15`
  - low 32 bits: `r8d`, ...
  - low 16 bits: `r8w`, ...
  - low 8 bits: `r8b`, ...

x86(-64) Architecture

Stack

- memory area given by OS to programs
- LIFO data structure; x86 stack grows towards lower addresses
- `esp` (`rsp`) points to the top of the stack
- main use: return address, function arguments, local variables, temporary storage

`PUSH` value

- decrements `esp` (`rsp`) and then stores given value at the memory address given by (new) `esp` (`rsp`)

`POP` register

- copies the value from the memory address given by `esp` (`rsp`) into given register and then increments `esp` (`rsp`)

x86(-64) Architecture

How does function call work?

- parameters are stored somewhere (see below)
- `call` address
  - push address of next instruction on stack
  - jump to address
- `ret` (return from function)
  - pops address from stack and jumps to it

Calling conventions

- 32bit: many different possibilities
  - *cdecl*: arguments passed on the stack in reverse order
- 64bit: two main approaches (Microsoft x64, System V AMD64)
  - both use registers to pass (some of) the arguments

x86(-64) Architecture

Function frames (standard entry/exit sequence)

- at beginning of function:

  ```
  push rbp
  mov rbp, rsp
  sub rsp, 0x10
  ```
  (allocate 16 bytes on stack for local variables)
- `rbp` is the base frame pointer
  - local values referenced as `[rbp + 0x08]`, ...
  - note that `[rbp]` holds the value of previous `rbp`
- at end of function:

  ```
  mov rsp, rbp
  pop rbp
  ```

Note: Optimisations (frame pointer omission optimisation) may eliminate this.

x86(-64) Instructions

Move instruction

- `MOV` – copy value from *src* to *dest*

Arithmetic and logic instructions

- `ADD`, `SUB`, `MUL`, ...
- `AND`, `OR`, `XOR`, ...

Test instructions

- `CMP` – performs `SUB`; does not save the result, only sets *flags*
- `TEST` – similar to `CMP`, performs `AND`

Jump instructions

- `JMP` – unconditional jump
- `Jxx` – conditional jump, reacts to *flags*
  - `JZ` – jump if zero
  - `JBE` – jump if below or equal
  - ...

Optimisations

What can compiler optimize for us?

Optimisations

What can compiler optimize for us?

- speed
  - rearranging memory accesses
  - inline functions
  - tail recursion
  - loop unrolling
  - else-if to switch
- space
  - collapse common code into sections
- *obvious*
  - constant propagation

Optimisations

- add padding (on stack)
  - to profit from cache lines
- start load operations in advance
- postpone store operations
- group memory access operations
  - make the access sequential

Advanced transformations

- remove or optimize variables
  - merge them, avoid repeated load, store if no-one reads...
  - make them register-only variable
- replace loops by intrinsics
- to prevent branch prediction failure
  - swap cycles
  - place conditions outside the cycle

Optimisations

Inline functions

- probably the most important optimisation
- put the code of called function directly into the caller
- inline small or heavily called functions
  - complicated heuristics to decide which function should be inlined
  - call to a function is expensive
- big profit with combination of other transformation
- `inline` keyword does not force compiler to inline
  - compiler usually knows better then programmer (unless you profile heavily)

Tail recursion

- transform recursion into cycle
- no duplication on stack
- remove function calls

Optimisations

- loop unrolling
  - repeat cycle N times each iteration
  - reapply memory transformation
- transform else-if to switch
- implement switch by lookup table
- constant propagation
  - compute constants at compile time
  - propagate constant parameters into functions
  - may create specialized functions without some parameters
- copy elision
  - avoid use copy-ctor when not necessary
- return-value optimization
  - use directly the variable in which the result is assigned
- remove references

Task: Profiling

`06_smallvector.h`

- defines `brick::data::SmallVector`
- a vector which need not allocate memory dynamically if it is small
- a piece of real-world C++ code, don't get scared by all the templates (you will eventually learn what they mean)
- together with some assertion helpers

`06_smallvector_bench.cpp`

- a benchmark which compares `SmallVector` to `std::vector`
- `SmallVector` is slower (which is expected)
- but it is much slower when memory is-pre allocated (which is not expected)

- try to profile the problematic benchmarks (separately)
- try to find out what is the problem, think about the fix
- try other containers (`deque`, vector from 1st and 4th lecture)