# Micro-optimizations in C

## By example

sudden6 at ToxCon 2018 12.-14.10.2018

# Why C over Assembler?

- Portable between architectures
- Don‘t want to learn Assembler
- Compiler has better instruction scheduling for more CPUs than me
- Compiler can optimize for different (micro-) architectures easily

# Example progam: m-queens

- Counts solutions for the general N-Queens problem
- Small: ~100 lines of C code for the solver
- Embarrassingly parallel problem
- Easy to benchmark

# Example: m-queens

- Algorithm and data fit in Cache
- Limited by branch prediction and arithmethics
- About O(n!) complexity using backtracking search
- State fits into registers of x86_64 CPU
- Backtracking stack fits into L1 data cache

# Profiling: perf

Reads CPU performance counters

- Find bottlenecks
- Important values
  - Branch count & branch-misses
  - Total number of instructions
  - Cache misses
  - Stalled instructions on CPU frontend & backend

# Profiling: Manual branch statistics

- An x86_64 CPU has many many registers

  → use them for branch counters

- Minimal performance hit
- Numbers exactly where you need them from
- Might be tools to do this for you

# Example: Manual re-odering

- Old

```
68      // The standard trick for getting the rightmost bit in the mask
69      uint_fast32_t bit = ~posib & (posib + 1);
70      uint_fast32_t new_diagl = (bit << 1) | diagl_shifted;
71      uint_fast32_t new_diagr = (bit >> 1) | diagr_shifted;
72      uint_fast32_t new_posib = (cols[d] | bit | new_diagl | new_diagr);
73      posib ^= bit; // Eliminate the tried possibility.
74      bit |= cols[d];
```

- New

```
68      // The standard trick for getting the rightmost bit in the mask
69      uint_fast32_t bit = ~posib & (posib + 1);
70      posib ^= bit; // Eliminate the tried possibility.
71      uint_fast32_t new_diagl = (bit << 1) | diagl_shifted;
72      uint_fast32_t new_diagr = (bit >> 1) | diagr_shifted;
73      bit |= cols[d];
74      uint_fast32_t new_posib = (bit | new_diagl | new_diagr);
```

# Example: Manual re-odering

- From 868G instructions to 847G instructions
- Executed instructions reduced by ~2%
- Benchmark run time reduction by 0,6%
- Rest was probably hidden by CPU pipelines already
- No logic changes, compiler should be able to do it?

# Example: Algorithm tradeoff

- Good branch prediction is crucial for performance
- Idea:
  - make branches more likely to take one direction
- Pro:
  - Better usage of CPU
- Con:
  - Probably not possible for all algorithms

# Example: Algorithm tradeoff

- Old

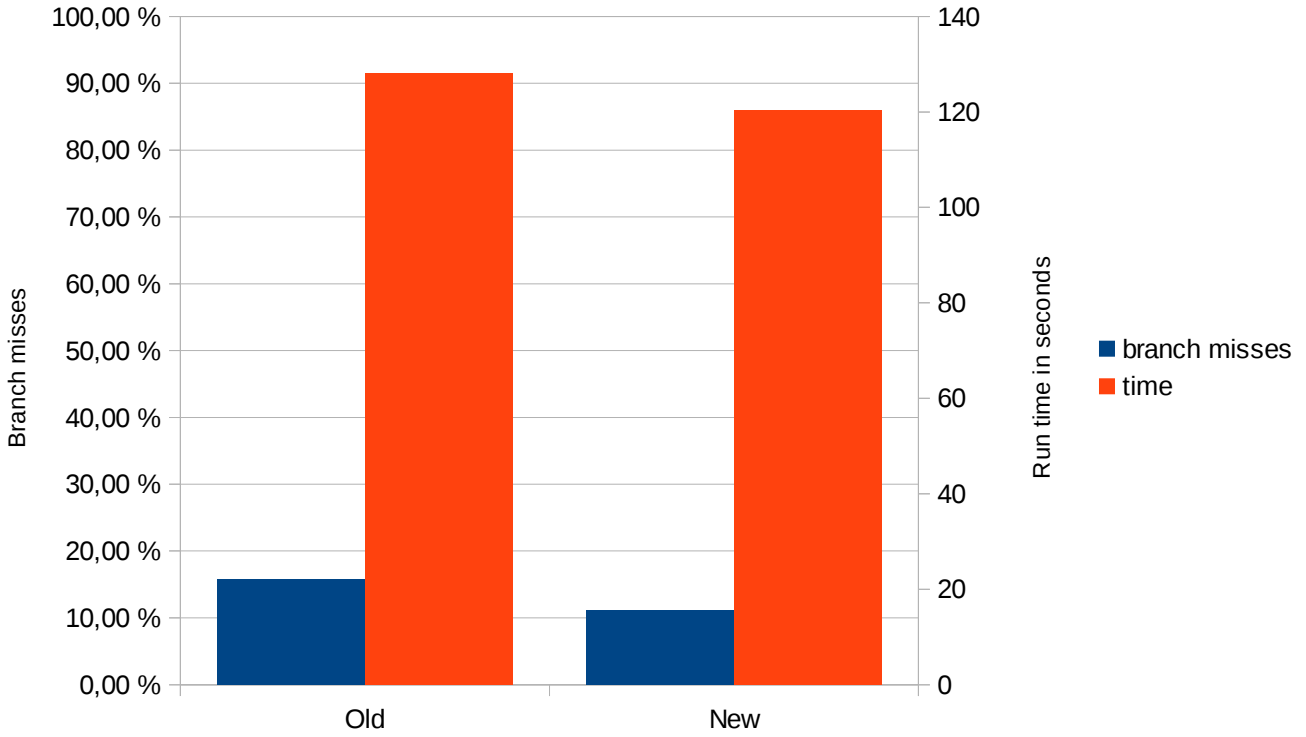
```
79          uint_fast32_t allowed1 = l_rest >= (int8_t)0;
80          uint_fast32_t allowed2 = l_rest > (int8_t)0;
81
82          if(allowed1 && (lookahead1 == UINT_FAST32_MAX)) {
83              continue;
84          }
85
86          if(allowed2 && (lookahead2 == UINT_FAST32_MAX)) {
87              continue;
88          }
```

- New

```
79          uint_fast32_t allowed2 = l_rest > (int8_t)0;
80
81          if(allowed2 && ((lookahead2 == UINT_FAST32_MAX) || (lookahead1 == UINT_FAST32_MAX))) {
82              continue;
83          }
```

# Example: Algorithm tradeoff

- Run time decreased from 128s to 120s or about 6%
- Branch misses decreased from 13,3% to 11,1%




# Overall improvements

- Started with code from the Internet:

  https://rosettacode.org/wiki/N-queens_problem#C

- Previous run time for N=18 was 173s
- Lots of optimizing later, runtime dropped to 120s

  → Runtime reduced by 30%

# What didn‘t work for me

- __likely(…) and __unlikely(…) from Linux kernel
  - Always made things worse
- Profile guided optimizations
  - No reliable improvement

# Why do I do this?

- Hardware limit reached
- Getting the last 1% of performance
- It‘s fun tinkering around and seeing benchmarks improve
- Personal hate for unoptimized Software

# Holes I have fallen into

- Run Firefox in the background during benchmarking
  - Huge variation in benchmark runtime, results become unusable
- Expected the same branch counts on every run
  - Other programs can change branch prediction
  - Hello Spectre/Meltdown!
- Benchmark runtime too short for meaningful results

# Future projects

- Port m-queens to the GPU using OpenCL
  - Try if optimization strategies work there too
  - Port already done
  - Promising first results
  - Cleanup for easier benchmarks
- Try using a superoptimizer

# References

- All benchmarks executed on a AMD Ryzen 5 2400G
- Benchmark command:

  gcc-7 -o m-queens.bin main.c -O2 -march=native -mtune=native -

  std=c99 && perf stat -d -r 3 ./m-queens.bin 18

# <getting in touch ...>

- Github

  https://github.com/sudden6/m-queens

- Email

  sudden6@gmx.at