HOW TO OPTIMIZE

How to Optimize
Overview

- Programmer's guide
  - General coding advice
  - C++ coding pitfalls
  - Target specific issues

- How to (profile on) Lauterbach (target debuggers)
  - Setup
  - Interpreting results

- Advanced topics
  - Optimization strategy
  - Local: reading the assembly code
  - Global: processor objects

Internal | Chassis Systems Control | Stefan Fuhrmann CC-DA/EAR1 | 3/06/2018

Don’t optimize just yet,
clean code is usually fast.

Programmer's Guide
General coding advice (1/2)

- Golden Rule: Code gets faster if it has less to do
  - Your code does lots of repeated work → get rid of that
  - Sometimes, a better algorithm using more of the available information is necessary.

- Factor out common code
  - In loop headers: `size()`, `end()` etc. are often needlessly repeated
  - Any (sub-)expression that does not depend on inner loop state should be moved out
  - Common sub-expressions can also be found in sequential code, particularly function results

```
x = foo()[0]; y = foo()[1];           // foo constructs an expensive object twice
v = foo(); x = vec[0]; y = vec[1];    // only 1 expensive operation
```


Internal | Chassis Systems Control | Stefan Fuhrmann CC-DA/EAR1 | 3/06/2018

Programmer’s Guide
General coding advice (2/2)

- “Big-O” runtime complexity: beware of nested loops
  - Number of operations is the product of all iteration counts,
    e.g. O(numLocations * numObjects * numNewLocations)
  - Cutting the iterations short by skipping iterations with if() often less effective in “dynamic situations”
    → extreme peaks while average runtimes look good
  - The inner loop(s) may be “hidden” in sub-functions
    Some container classes do a full scan for e.g. a `get_constObjectRef(id)`.
  - Most can be fixed by sorting before matching

- Order of checks in hot loops is important
  - Put the most restrictive check first
  - Make sure that code gets actually inlined (in case the check is in some sub-function)
  - Simplify the hot loop + main check combination as much as possible (ideal: pointer walk + comparison)

Programmer's Guide
C++ coding pitfalls (1/2)

- The following are rules, make exceptions at your discretion

- Function parameters
  - Pass non-trivial parameters by const reference (basically everything that is not a single number or pointer)
  - Pass trivial parameters by value
  - Avoid default values for non-trivial parameters

- ***Small*** virtual functions will usually not be inlined → see that you can drop the "virtual" for those

Programmer's Guide
C++ coding pitfalls (2/2)

- Function results
  - Getters for internal state should return non-trivial data by const reference.
  - Avoid intermediate copies

```
x = SomeType(MyFunction());      // usually, a temporary object will be created
x = MyFunction();                // no additional object, also robust against signature changes
```

  - Take advantage of the RVO (Return Value Optimization)
    Kicks in, if the receiving / calling side is constructing a new object

```
SomeType x;                      // construct x
x = MyFunction();                // creates a temp object, copies it to x, temp object is destroyed
SomeType y = MyFunction();       // RVO: function constructs its result directly in y
```

Programmer's Guide
Target specific issues

- Float vs. double
  - Target has 32 bit float support in hardware (FPU)
  - 64 bit floats are software-only
  - Be sure to use `float`, `float32_t` instead of `double`, `float64_t`
  - `0.5` is a double, `0.5f` is a float

- No inlining of loops
  - Any function containing a `for()` loop will not be inlined, even if the optimizer can completely eliminate the loop as it often happens in templates.
  - Maybe use recursion instead but only for loops that are ***short and*** have a ***fixed*** number of iterations

- Optimizer will not factor out repeated calculations, especially floating point functions, e.g. `pow` in `for(i=0;i<4;++i) {x[i] += pow(globalVariable);}`

Wer misst, misst Mist.
(He who measures, measures rubbish.)

How to Lauterbach
Setup

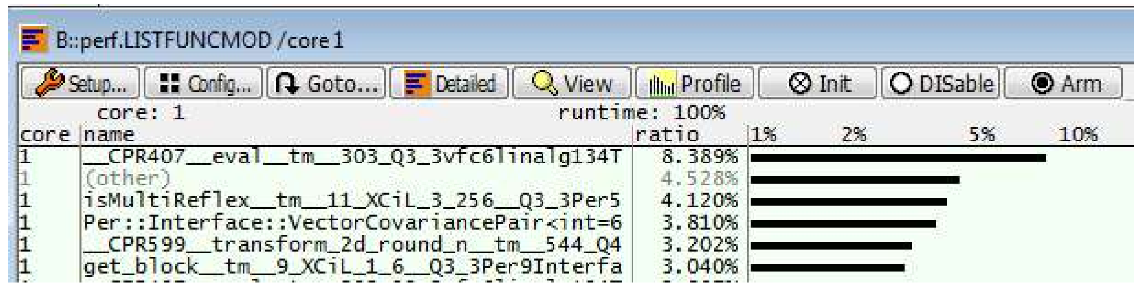
- Make performance tests repeatable: automate input and output
  - Restbus simulation may be sufficient for testing COM stack performance
  - Sensor data might be feed in through HiL but that is complicated
  - Bypass as alternative: extract data from MDF, store it in flash, replay by filling Daddy ports (see copy runnables in `jobCore1_T_BG()` in `per_profiling` branch in `pj_vwmqb37w.git`)

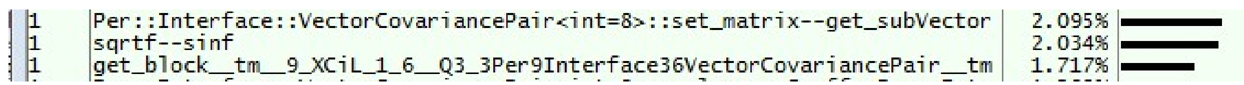
- Make outputs verifiable, e.g. the following way:
  - At runtime, store key info like DEP & SEP object counts in an array; one entry per task cycle
  - Dump the data to file, import in Excel and quick check for deviations from previous runs
  - Lauterbach script to dump the `scom::perfLog` array:

```
PRinTer.FILE C:\sbx\PerfLog.csv CSV
WinPrint.var.fixedtable %LOCATION.off %hex.off scom::perfLog
```

How to Lauterbach
Interpreting results

- Use `B::perf.ListFuncMod /core 1`
  - All perf views show the same data
  - Use Arm / Disable buttons to start / stop
  - Often shows mangled C names
  - "`(other)`" are system functions like `memcopy`, `sqrt`, `__ll_div` for 64 bit float math
  - `B::perf.List /core 1` will show those in detail but as "code ranges"





- Results do not cover costs of sub-function execution
- You may stop the execution at any point, set breakpoints etc.
  - Double click on perf list entry will bring the respective function up
  - Use "mixed" view to be pointed to the correct template instance

BOSCH

If brute force is not the answer,
you are just not using enough of it.

BOSCH

Advanced Topics
Optimization strategy

- **Measure before you try to optimize!**
  - Successful optimization is an art form like good management

- Develop a Theory of Computation for the respective SW / HW combination
  - A naïve, simplistic mental model of what the system has to do, e.g.
    PER = read locations, group to objects, extrapolate movement, match with new locations
  - What would the minimal amount of operations be, that is needed to produce the desired result?
  - What is “core functionality” and what is “overhead”?

- Generic strategy: Finding and understanding issues is harder than fixing them
  - Gain system understanding: Identify frequently used low-level functions and “squeeze” them hard
  - Identify “big-O” mistakes: Use call counts (callgrind!) to identify high “fan-outs” to sub-functions
  - Reduce friction, improve SNR: Either no sub-functions after inlining or runtime dominated by sub-functions

Advanced Topics
Local: reading the assembly code (1/2)

- The Lauterbach debugger will show you the machine instructions
  - Often, you can identify hot spots and inefficiencies directly
  - Use the compiler-generated assembly listing to see whether new code is better

- Search the `.lst` files for the implementation of the code lines in questions

- Things check at assembly level:
  - Has a particular function been inlined or even completely eliminated?
  - Are there a virtual function calls on the hot path, e.g. `call [a2]`?
  - Is there repetitive / superfluous code (see next slide)?
  - What would be the minimal set of assembly operations for the hot path?
    Try to express those as C/C++ code.

Advanced Topics
Local: reading the assembly code (2/2)

- GHS has a tendency to repeat address calculations; use iterators in that case
- Example: one iteration in an unrolled loop of matrix operations

```
l_sum -= f_L(i,k) * f_L(j,k);
    msub.f     d4,d4,d0,d6
    ld16.w     d0,[a12]
    addsc.a    a12,a13,d1,0
    lea        a12,[a12]0x14
    ld16.w     d6,[a12]
    addsc.a    a12,a13,d5,0
    lea        a12,[a12]0x10
    msub.f     d4,d4,d0,d6
    ld16.w     d0,[a12]
```

actually part of the next iteration but the pattern still holds

```
l_sum -= *l_ik_it * *l_jk_it;
    msub.f     d4,d4,d0,d1
    ld.w       d1,[a3]-20
    ld.w       d0,[a2]-20
    msub.f     d4,d4,d0,d1
    ld.w       d1,[a3]-24
```

Advanced Topics
Global: processing objects

- Extension of the “factor out common code” tactics for this scenario
  - One or more deeply nested functions are repeatedly called by some function high up in the stack
  - Nested code does not “know” about earlier executions
  - Nested code needs to re-compute things over and over

- Solution: Add a parameter that preserves the context between calls
  - Could be a caching object to short-circuit lookups
  - Could be data pre-calculated in the higher up function

- Enhanced version: Processing objects
  - Turn the context object into something semantically meaningful, e.g. `TMyTransformation`
  - Move the logic of the nested function into methods of the new object
  - Optimize code further between those methods; they are probably tightly linked