OCaml as fast as C

Sylvain Le Gall, OCamlCore SARL,
OCaml Meeting 2009 – Grenoble, France

Plan

- Project workflow
- Optimize
- Tips and tricks

Foreword

This presentation is based on the project STSort which has been funded Talend.

- GNU sort like utility (POSIX/Windows)
- Performances
- Features
- 6 months
- 1 man
- 25 kLoC
- 10% of C

Goal

- This presentation will try to show how:
  - To integrate testing/benchmarking
  - To manage a project « designed for speed »
  - To convince your boss that OCaml is the right choice to do that

However this presentation is only self tested, most of the goal above are quite difficult to achieve (in particular the last point).

OCaml choice (criticism)

- OCaml is not a widely known
- Your boss prefer Java/C#/C++
- Documentation and examples are hard to find
- Garbage collection is slow

For all this reasons, your boss or even you, will avoid using OCaml for speed

This is an error!

OCaml choice

- Most of the time, code written in Java/COBOL/C++ won't never been read another time, if it works.
- OCaml allow to catch enough error to get a working program that nobody will read again
- OCaml and functional programming community is growing
- OCaml GC is efficient and can achieve a good speed

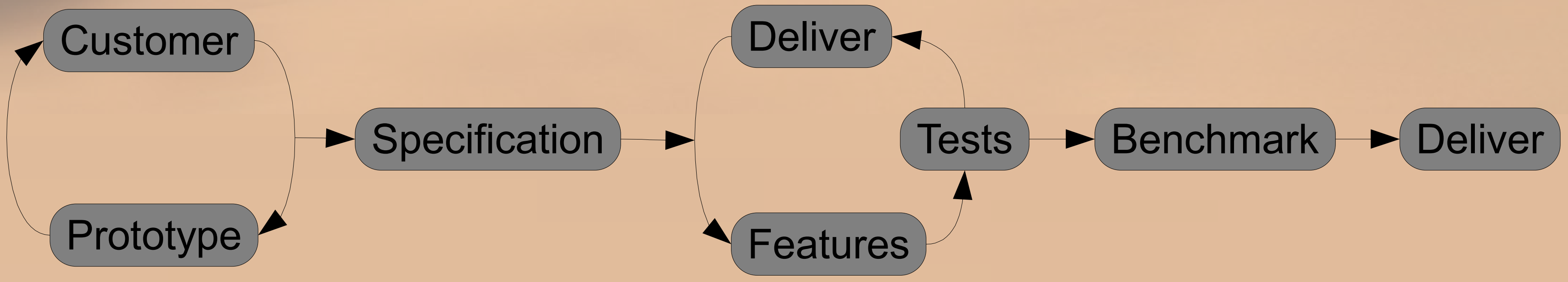
For all these reasons you can choose OCaml for speed

OCaml choice

- Multi-paradigm language
- Strict typing
- Compile-time checks
- Fast prototyping
- Static compilation
- Spend less time finding NULL pointer error
- Develop application faster

If OCaml is your most fluent programming language, just choose it

Programming



- OCaml allow fast prototyping:
  - Establish a very short circuit with customer
  - Find the root of what they need
  - Show them something that works quickly

When prototyping, « quick and dirty » is the rule of thumb

Programming (after prototype)

- Stop and think:
  - What is really useful in my prototype?
  - What can I reuse in it?
  - Are there things I can replace with Open Source libraries?
  - Draw the general architecture of your software
- Evaluate your need in term of time and workforce
- Begin to gather information on what has failed in your prototype!

Programming

- Try to deliver weekly/monthly, even if this is not finished, it gives you early feedback and show your progress (including to you)
- Use a good build tool:
  - OcamlMakefile: portable, small project
  - Omake: large project
  - Ocamlbuild: >= 3.10
- Use ocamldoc as soon as possible
- Don't try to program generic functions at the beginning. Only do it when required

Testing

- Write test before programming
- Test only high level behavior
- Plenty of test framework, my preferred one: OUnit
- Do small test or allow to skip big ones (OUnit.skip_if or OUnit.todo)
- Include running test in your default Makefile target

Even if OCaml allow to catch a lot of programming errors, tests is the only way to prevent more subtle error

Optimize

- Never try to optimize an unfinished software
- Extend your tests before beginning to optimize
- Use specific Makefile targets to produce different « flavor »:
  - Bytecode
  - Bytecode with debug
  - Native
  - Native with profiling

Keep in mind: optimizing any software can be a long task!

Optimize

- This is a balance between:
  - Being fast in a particular case
  - Being fast in all cases
  - Spending time to achieve the two points above
- General case: good but very expensive
- Particular case: cheap and cost/time effective
- In fact, this is just as the algorithm complexity:
  - Worst case (general)
  - Average (most used particular case)

Optimize

Optimizing should be about the « average optimization »

- Find your most common cases
- Identify the best algorithm for these cases
- Profile at low level for each case

Optimize (search)

- Find obvious problem:
  - non-terminal recursion
  - inefficient functions called too often
- Find bottleneck
- Find best algorithm
- Find low-level optimization

Optimize

- Optimization is also long work:
  - Time your initial case
  - Do a small change
  - Time your small change
  - Evaluate the win
  - Restart from beginning
- Always try to avoid introducing complicated programming pattern for an optimization
- It is highly improbable that you will be able to optimize a program in a single day

Samples

One very important task when doing optimization is to find your most common cases and transform it into benchmark samples

- Samples should allow you to cover:
  - All your most common cases
  - Different scale of it (from the smaller to the bigger)

Samples

- You should be able to integrate benchmark into your default Makefile target
- Use at least 2 set of benchmark samples:
  - « small »: day to day
  - « big »: nightly/weekly

Samples

- Gathering samples is a complex task:
  - Lack of data
  - Privacy
  - Too small or too big samples
- But OCaml is a fast prototyping language, so just create your sample generator!
  - Adapt to most case
  - Use your customer as seed
  - Include seed and sample generator in your source

Tools

- Don't be mislead by ocamlprof:
  - You must compile everything with the right option to know precisely where you spend time
  - It remains an approximation to what really need to be optimized
  - It has an heavy impact on processing speed, which in turn can change your bottleneck
  - Output is hard to read
- However, ocamlprof is a good tool:
  - For estimating what functions is called often
  - To know what is the time share of the GC

Tools

- A simple trace tools can help you easily
- ocaml-dbug:
  - Stderr output
  - Can be removed at compile time
  - Low impact on performance
- Allow you to find most problem « by eye »:
  - A non-tail recursion implies a quick slowdown in output
  - Time for each part of your program

Moreover, this is also a good tool to debug your program

Tools

- Another good tools is your process data:
  - Memory used
  - Time
  - Temporary file used
- ocaml-process-monitor:
  - Display memory and disk usage
  - Output on stderr (interlaced with ocaml-dbug)

Tools

- Create your benchmark suite:
  - For low level (function call): benchmark OCaml library
  - For high level (whole software on an input): build your own software (using OCaml)
- For low level, integrate your benchmark with your test suite:
  - Only benchmark if test is correct
  - Use the same type of functions (factor your code)
  - Help to dedicate, one test suite to one source module

Tools

For high level, allow to run it through a standard cron job:

- Takes a lot of time to run (15min to several hours)
- Will be launch out of your standard development environment

Benchmarking

- Use a reference implementation and do benchmark relative to it
- Find a way to always produce the same result in a specific context
- Use different scale, different computer architectures and OS
- Keep a log of your benchmark results
- Find a suitable presentation for your results!

Searching for speed

The first step in looking for speed is to understand what is the problem.

- Evaluate difference of performance in your sample and try to know why there is a difference
- Consider asymmetrical evolution of the performance of each sample (why this sample perform better than this one after a change)

Searching for speed (GC)

- Use ocamlprof to detect how much the GC is using
- Hook « benchmark » module to monitor GC allocation
- Avoid a lot of fast allocation/deallocation
  - This cause data to migrate from minor to major heap
  - Make the GC works a lot
- Try to reduce GC allocation by reusing datastructure

Searching for speed (GC)

- Avoid maintaining a lot of datastructures for very long time in memory
  - Each time you do a major collection, it will be scanned again and again
- Reuse, when possible, datastructures already allocated (Buffer, String,...)

Searching for speed (imperative)

- Try to use imperative aspect of OCaml when it worth it
- Purely functional programming doesn't apply well to some datastructure (Hashtbl, Array, Queue)
- The limit between purely functional versus imperative is hard to draw:
  - Only incrementing an integer in record is more efficient with purely functional
  - Manipulating string buffer is more efficient with imperative style

Searching for speed

- Non « inlined » function
  - Functor
  - Anonymous function (like in « List.iter »)

Searching for speed (solutions)

- If a lot of datastructure need to be maintained in memory:
  - Use a custom block/C datastructure
  - Use Ancient OCaml module
- A very good imperative datastructure to build your own: Buffer
  - It converges to maximal size of your input data
  - Once at this size, it stops allocating
  - All primitives are good (Buffer.clean, Buffer.reset...)
  - If needed, you can call Buffer.reset to do a « deep »-deallocation

Searching for speed (solutions)

- Play with the different GC call, from minor to compact:
  - Sometimes you don't have a lot of memory allocated, but it is fragmented
  - However, this is only to understand where the problem is, avoid doing this in your final product
- Don't try to play with GC settings, in most case, they are good for general purpose application.

Tips and tricks

- Imperative loop versus recursion
  - No difference of performance “while ...” and “let rec ...”
  - “let rec ..” for complex loop
  - “while ...” to ensure that you have no « non » tail recursion
- The case of int/Int32
  - 31 bits integer is just enough for common case
  - If you need to compute things to 32 bits (or 64 bits), compute using “int” until maximum and switch to Int64/Int32/Big_int when needed

Tips and tricks (32/63 bits)

- OCaml GC rely on pointer
- On amd64:
  - Pointer is twice the size
  - Can address more than 3GB of memory
- Choose what really worth:
  - Memory usage (32)
  - 64 bits number (64)
  - Using more than 3GB memory (64)

Tips and tricks

- Test before change
  - Testing value can be optimized
  - Changing value has always an overhead (caml_modify)
- C functions
  - Use C functions for really heavy arithmetic or OS specific task
  - Using C for processing string or unboxed array is almost useless
  - Don't use caml_register_global_root if they are many

Tips and tricks

- ocamlopt options
  - « -inline » has almost no effect if you don't have written a code with this in mind
  - « -ccopt -O9 » have a real impact on performance
  - « -unsafe » and « -noassert » only apply to compiled code
- « unsafe » functions
  - Whenever you check the bound in a function before processing – use it

OCaml as fast as C!

- OCaml has many advantages:
  - GC is efficient
  - Mixing imperative and functional style can help you to get the best from the two worlds
  - You can make OCaml code run really fast

However, you cannot go as fast as low level hand-made optimization in a language like C.

- But with OCaml:
  - You won't something that works quickly
  - You won't spend time debugging your segfault.

All in all, OCaml is a good bet for speed and cost effective solution

Questions

?