CPU, memory, latency

profiling and optimizing GStreamer

Edward Hervey
Senior Multimedia Architect
edward@collabora.com
Gstreamer Conference 2012

- No, I'm not giving this talk because I'm a gentoo user
- No, I will not bore you with -funroll-loops CFLAGS

Goal

- “Make GStreamer as efficient as possible”
  - lowest-overhead as possible
  - Make it possible to leverage as much as possible from the underlying hardware/software

Why is it needed ?

- We're a framework
  - Initial goal is to make sure as many use-cases as possible are doable with the provided API/design
  - Secondary goal is to make sure they can be done as efficiently as possible
- We can't test/profile all usages
  - Experience/Usage helps us improve GST

- What impacts performance
- How to profile it
- Tools available
- Optimizing
- Examples
- Lessons learnt

Collabora

Performance

- Issues can appear in a variety of way
  - CPU
  - Memory
  - Latency (internal and external)
  - I/O
- First goal is to understand and track those metrics

Metric 1 : CPU

1. Useless computation
   - Codepaths that could be avoided
   - Codepaths that are repeated
   - Computing that could be delayed or be made asynchronous
2. Algorithmic improvements
3. Better usage of CPU (SIMD, ...)

• Note : Not only main CPU

Metric 2 : Memory

1. High heap/stack usage
   - Problem for tight memory platforms
   - Hit swap on more powerful systems
   - You could run more on the same platform
2. Memory re-use
   - Avoid memcpy
   - Bandwith issues

Collabora

Metric 3 : Latency

- Internal Latency
  - Need future data to output past data
  - Waiting for preroll
  - Accurate latency reporting
  - Critical for best live playback
- External Latency
  - Storage/network/hardware latency

Metrics summary

- They are varied
- They impact each other
  - Memcpy => bandwith/io/latency/cpu
  - Async computation => latency/memory
- Let's measure them !

Profiling

1. Measuring those various metrics
2. Pinpoint the culprit
   - (Have a reproducible synthetic test)
3. *Optimize*
4. GOTO 1

   → **Prove you have improved the situation**

Methodology

- Just like for debugging
  - Smallest synthetic test that reproduces the same behavior.
    - pinpoint what element/file is the culprit
    - You will be running it many times
- Be careful to impact of profiling tools
  - Changes delays/races/...
  - Use options wisely

Tools

- Time
  - Trivial to use
  - Detect overall CPU regressions quickly
  - Low/zero impact
  - Low amount of information
- Top
  - Low/zero impact
  - Memory/cpu usage over time

Tools

- GST_DEBUG logs
  - (insanely) verbose
  - Plenty of metrics over time
  - You can add your own metrics
  - Medium to high impact
- Oprofile
  - Low-ish impact
  - No changes required

Tools

- Valgrind (here be dragons !)
  - Memcheck (memory usage)
  - Massif (heap profiling)
  - Callgrind/cachegrind
  - Pro: **very** verbose and detailed
  - Con: high overhead

A picture is worth...

- Use existing Uis
  - Kcachegrind
    - callgrind, cachegrind, but also other inputs
  - Massif-visualizer
  - gst-debug-log-viewer
- Create your own
  - matplotlib

Optimizing

- You know where your bottleneck, hotspot is, you're essentially done \o/
- Not going to go down in cpu/asm improvements
  - Let me google that for you ...
- A lot can be done by high/medium level improvements

CPU examples

- G_DISABLE_CAST_CHECKS
  - Expensive, detected through profiling
  - Enabled in releases
- Don't enforce behavior on caller
  - 1.0 caps function that don't require writable caps
    - If needed it will make a copy

CPU examples

- Give more hints/context
  - Might require API change !
  - query_caps(pad, **filter**) in 1.0
- Delay processing to later on
  - Downstream element could do the processing (1.0)
  - Lazy index parsing (qtdemux/avidemux)

CPU examples

- Expensive GstCaps check when linking
  - Exponential on number of elements
  - “Does this square plug fit in this round socket” is enough at link time (i.e. templates)
  - Check details at stream time
- GES timeline startup on N9 => 20+ seconds down to less than 1.

CPU examples

- Other ideas
  - Disable decoding for streams not used
  - Disable fetching data for streams not used
- And plenty more that don't require low-level optimizations!

Memory examples

- Decode into display memory
  - 0.10 bufferalloc
  - 1.0 GstMemory, pools, ALLOCATION_QUERY
- Avoid/reduce copies
  - References when possible
  - 1.0 GstMemory re-used in multiple buffers

Latency examples

- Avoid using certain formats in live
  - B-frames
- Limit latency in some elements
  - audioresample ?
- Playback/seeking
  - Avoid doing small sparse reads
  - Seek to the optimal position (ex: asfdemux)
  - Live mpeg-ts demuxing (Broadcast DVB)

Last thing about optimization...

- ALWAYS RE-RUN PROFILING
- ENSURE YOU HAVE IMPROVED THE SITUATION
- CHECK ON OTHER FILES/CASES
  - You might have improved one case...
  - ... and made all the other worse

Collabora

Lessons learnt

- Know and understand the code you are trying to optimize
  - Have a good idea of what the code ***should*** be doing
- Optimizing one bottleneck/hotspot will uncover the next one
  - Speed up data passing ...
  - ... and discover GTypeInstance creation/destruction isn't efficient :(

Lessons learnt

- Assume dependencies aren't the bottleneck...
  - ... but don't ignore them either *(Glib anyone ?)*
- Check your optimization doesn't impact other cases
  - Re-try, re-try, re-try, get metrics
  - Ex: gst-discoverer <*.filetype>

Collabora

Lessons learnt

- Preemptive optimization is mostly evil
  - Get things working first
  - You most likely don't know all usage
  - It'll be easier to optimize later on

Bored ?

- Profile your use-cases, files, ...
- You'll learn a lot
- You might find more things to optimize

Collabora

Thankyou

- Any questions ?

Collabora