Improving GStreamer Quality

Jason DeRose

27 August 2012

novacut


gstreamer

open source multimedia has a quality issue

but it's
not an
engineering
problem

it's a

DATA

problem

multimedia quality ebb & flow

1. fix something for one type of file
2. which inadvertently breaks another type
3. months pass before regression is found

proprietary multimedia
has the same issue

but it's a bigger problem
in open source multimedia
simply because of
broader format support

consequences of ebb & flow

1 too much time is spent chasing down regressions, time better spent solving hard engineering problems (or catching up on sleep)

consequences of ebb & flow

2 code shipped to users has serious quality issues for at least some formats, and quality is inconsistent over time for any given format

so how do we fix this?

provide
engineers
the DATA
they need

how to get the DATA:

continuous integration testing against a rather exhaustive set of real world files.

how to get the DATA quickly:

use the cloud!

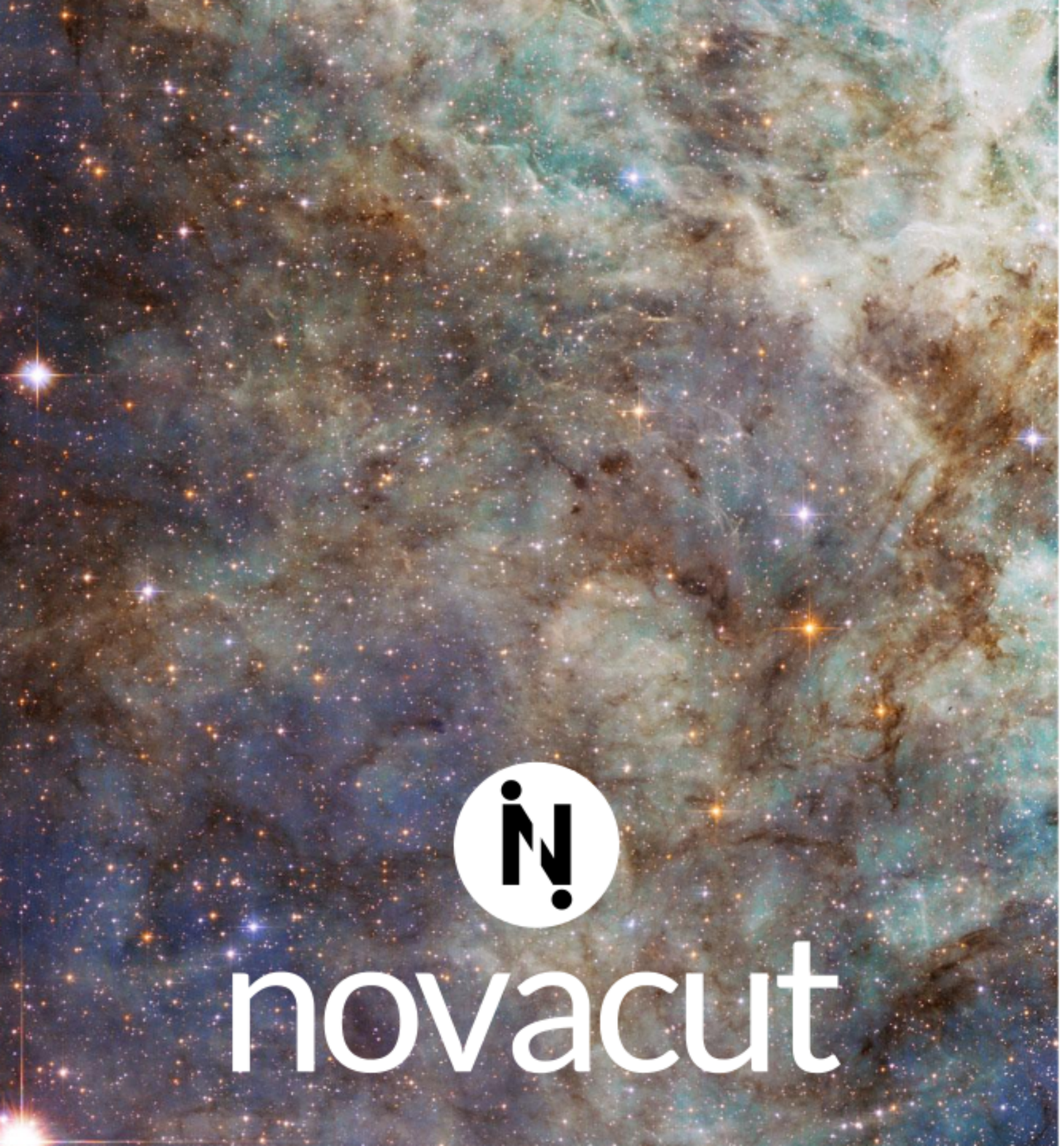
say you need 1000 hours of compute

1 node for 1000 hours

costs the same as

1000 nodes for 1 hour

we need regression feedback
in hours rather than months



big picture

1. media database
2. playback scenarios
3. NLE scenarios


1
media database

store media files in Dmedia

* we can host files on public cloud, but easily pass around on HDD also

why Dmedia?

* Dmedia lets you work locally with an arbitrary and changing subset of files

* which is important, because this media database will get BIG (100s of terabytes)

store metadata in DVCS

```
{
    "_id": "SM3GS4DUDVXOEU2DTTTWU5HKNRK777IWNSI5UQ4ZWNQGRXAN",
    "bytes": 20202333,
    "copyright": "2012 Jason Gerard DeRose",
    "license": "CC-BY-SA 3.0",
    "content_type": "video/quicktime",
    "width": 1920,
    "height": 1080,
    "frames": 107,
    "framerate": {"num": 30000, "denom": 1001},
    "samples": 171371,
    "samplerate": 48000
}
```

we want to store deep, per-buffer metadata


```
"video": [
    {
        "bytes": 3133440,
        "duration": 33366666,
        "md5": "291b7b738ad38fae8bdcd5e698417508",
        "pts": 0
    },
    {
        "bytes": 3133440,
        "duration": 33366667,
        "md5": "fb795cefb2abba5e51674c42aa64b69f",
        "pts": 33366666
    },
    {
        "bytes": 3133440,
        "duration": 33366667,
        "md5": "940c6860ffc79697189b24102a0c3e74",
        "pts": 66733333
    }
]
```

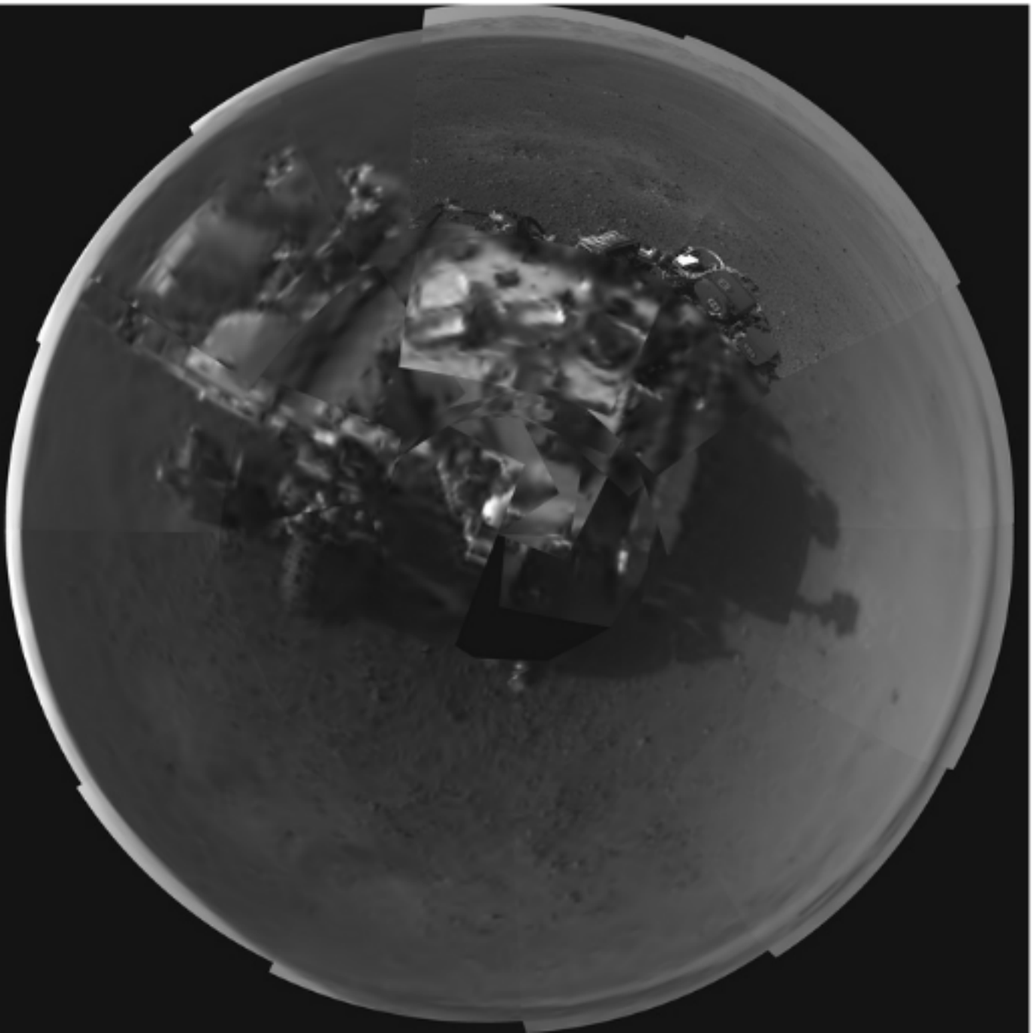
```
{
    "_id": "SM3GS4DUDVXOEU2DTTTWU5HKNRK777IWNSI5UQ4ZWNQGRXAN",
    "bytes": 20202333,
    "content_type": "video/quicktime",
    "perfect_timestamps": true,
    "video_bytes": 3133440
}
```


so we could
just store
per-frame md5

these cameras produce
"perfect" timestamps


```
"video": [
    "eec99ef4fe950037dd5c0d905e61a6d6",
    "06bd7617a881feca0d6b4337988602a2",
    "eaa9672ce36e173355e32812a9288339",
    "c766d6be8cd6e37e8c8f8eb07f3708ef",
    "7da8ff75b60ea3f194166b3917de3182",
    "16024c7000abb6be6b22efdc0c92de0b",
    "49bb77d81394787e4704b75c312b0a7c",
    "90f8267d712bfe8d26746fc3de651da1",
    "92bbe39ed01188dcdb7aa3c105d4aad5",
    "7be7832db0b6ba4863fd5aef533d66c9",
    "0fd5acece269f6971e91cf8c821aef6c",
    "a4cfffca706e25630d64508ba55fc39",
    "ba7ee74c79f7410887c1afefa92785b5"
]
```

think of the test scripts as compression functions.
they don't produce pass/fail, they produce DATA.



as we develop deep understanding of what the correct behavior is for specific media files, we want to store metadata rich enough to automatically verify correctness, release after release.

we want the data
produced by the tests
to help engineers
spot failure patterns

diff result data
against expected data

insights from failure patterns over time

* help identify failure hot-spots (areas that need more unit tests, re-achitecting, etc)

* shows which subset of files would give a high probability of detecting regressions (aka, how cheap can we go, how little compute can we get away with)

* we're looking for more than pass/fail; we're looking for patterns, opportunities for machine learning


2
playback scenarios

types of playback tests

* extraction

* play-through

* seeking

* demux-through?

extraction test

```
filesrc ! decodebin ! fakesink(s)

State.PAUSED
```

* everything we can get from caps (width, framerate, etc)

* duration in frames, samples, and nanoseconds

* content_type, etc

play-through test

```
filesrc ! decodebin ! fakesink(s)
```

```
State.PLAYING
```

* detailed per-buffer info from "handoff" callback(s)

* buffer pts, dts, duration

* buffer data md5, bytes

seek test

```
filesrc ! decodebin ! fakesink(s)

State.PAUSED/State.PLAYING
```

* random seeking, random segments

* random state change between PLAYING/PAUSED

* per-buffer results from "handoff" callback(s)

demux-through test

```
filesrc ! qtdemux ! fakesink(s)

State.PLAYING
```

* again, detailed per-buffer info from "handoff" callback(s)

* buffer pts, dts, duration

* buffer data md5, bytes

scripts are simple, provide "what is" data

```
extract FILE.MOV > DATA.JSON

play-through FILE.MOV > DATA.JSON

seek FILE.MOV SEEKS.JSON > DATA.JSON
```

data analyzed to determine pass/fail,
and to answer the "why"


3
NLE scenarios

if you can't deliver perfect frame accuracy, you can't do pro editing

types of NLE tests

* quick-test that checks buffers at a fakesink

* full render to file, then do per-frame SSIM test

quick test

```
gnlcomposition ! fakesink
```

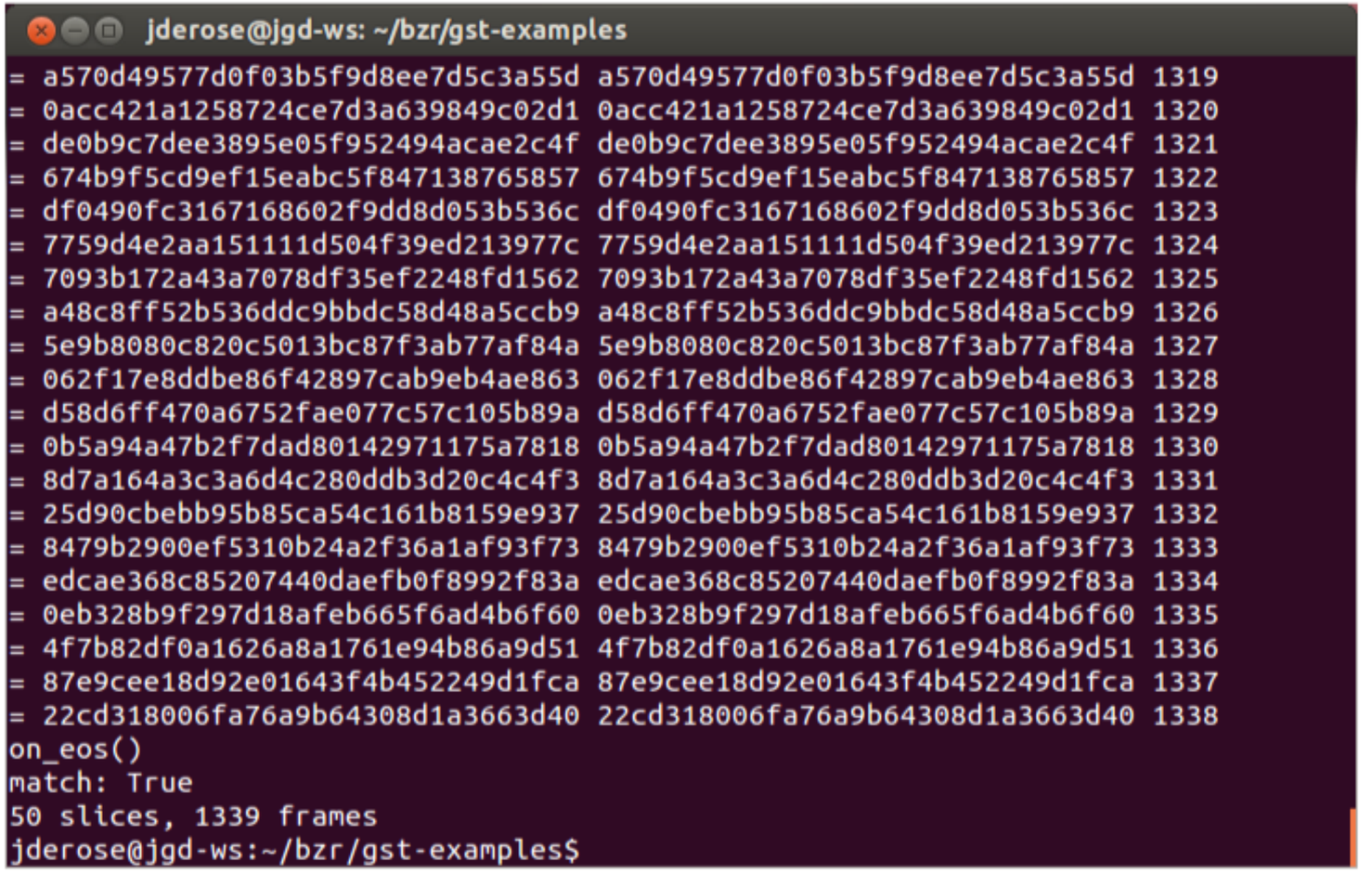
* do a play-through of all source videos to get md5 for each frame

* play-through composition, md5 each buffer in "handoff" callback

* this test is fast and brutally accurate

* and for HDSLR footage we've tested with...

gnonlin delivers perfect frame accuracy!




```
bzr checkout lp:~jderose/+junk/gst-examples
```

full render + SSIM test

```
gnlcomposition ! <encoder> ! filesink
```

* the quick test tells us a lot, but we need to verify renders

* first step is to count frames, check timestamps

* but to be certain, we need per frame SSIM compare

* we're not doing this yet... fingers crossed ☺

a note about nanoseconds and frames

```
def frame_to_nanosecond(frame, framerate):
    return frame * SECOND * framerate.denominator // framerate.numerator

def video_pts_and_duration(start, stop, framerate):
    pts = frame_to_nanosecond(start, framerate)
    duration = frame_to_nanosecond(stop, framerate) - pts
    return (pts, duration)

def video_slice_to_gnl(offset, start, stop, framerate):
    (pts1, dur1) = video_pts_and_duration(start, stop, framerate)
    frames = stop - start
    (pts2, dur2) = video_pts_and_duration(offset, offset + frames, framerate)
    return {
        'media-start': pts1,
        'media-duration': dur1,
        'start': pts2,
        'duration': dur2,
    }
```

modeling user intent

* frames and samples are a better way to model user intent (easier for UI designers, keeps us honest)

* easy to convert from frames/samples to nanoseconds when needed (convert as late as possible to avoid accumulating rounding error)

* there are some engineering problems using nanoseconds as the API for video editing (can't produce perfect outgoing timestamps without being frame and sample aware)

questions?

thank you!

novacut