streaming stats toolbox

Kevin Pouget

May 22, 2020

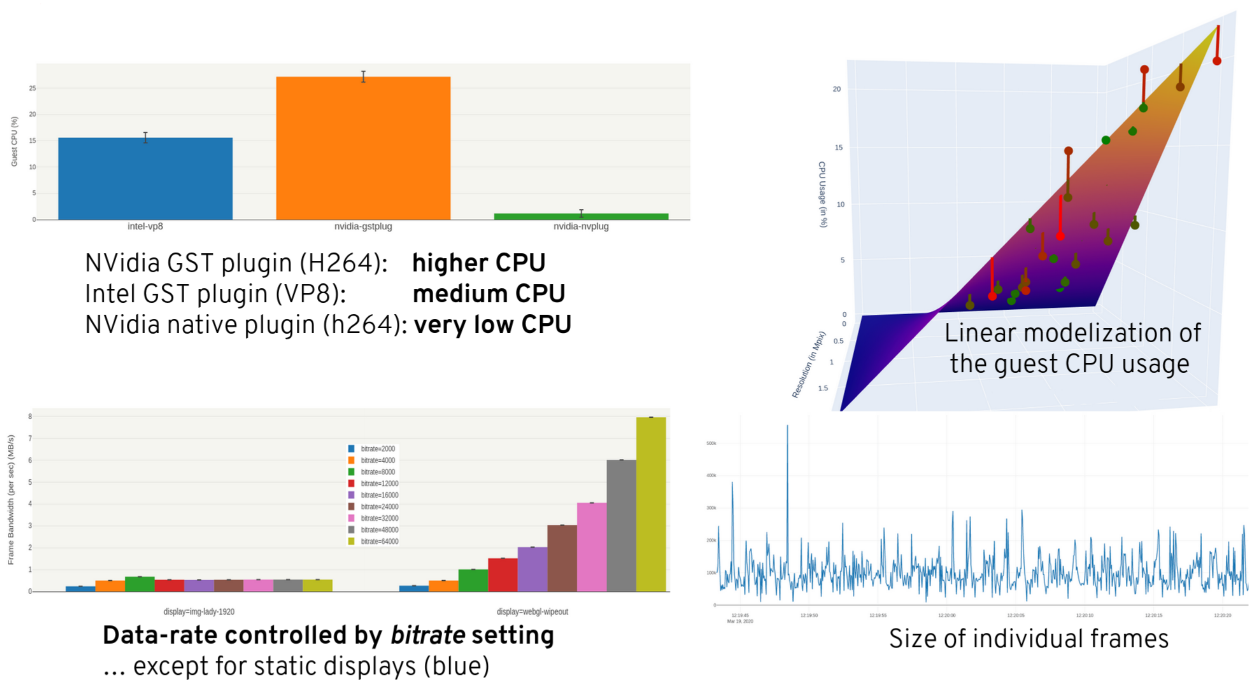
CONTENTS

The `streaming_stats` project is a modular toolbox for performance recording, benchmarking and visualization. It was originally developed for an in-depth study of SPICE distributed video streaming pipeline: video encoding inside the VM, frame forwarding in the host and video decoding in the client. This is the `adaptive` plugin. It was then refactored into a tool more modular. The `specfem` plugin illustrates the usage of the toolbox for a more classical benchmarking.

The toolbox is not designed to be used off-the-shelf in a new environment, but rather provides a framework for:

1. C code instrumentation (cf. the `agent interface`);
2. local collection of performance indicators (the `local agent`), such as the system CPU/memory load; the CPU usage of a particular process, etc;
3. a live view and interactive control of benchmarking (the `perf collector`);
4. an automated matrix benchmarking, ie, running the application with all the combinations of a set of parameters, and/or scripts influencing the system (eg, background noise, busy network, . . .)
5. a matrix-benchmark visualizer, that computes various statistics for every benchmark record, and plots them in a web interface. The web interface allows studying the impact of the various settings (eg, fix A and B values, and plot all the results for C and D settings/scripts).

In the following, we'll present each of the different steps. See the *Gallery* for screenshots.



CHAPTER

ONE

DEPENDENCIES

- Agent Interface (C):
  - Glib2
    - `CFLAGS += $(shell pkg-config glib-2.0 --cflags)`
    - `LINK_FLAGS += $(shell pkg-config glib-2.0 --libs)`
- Local agents:
  - Python process
    - Python 3.7 (does not work with 3.8)
    - PyYaml
      - `pip install --user pyyaml`
  - Plugins
    - VMStat, MPStat, PIDStat (system load)
      - https://github.com/sysstat/sysstat
      - `dnf install systat`
    - Intel_GPU_Top (GPU load)
      - https://cgit.freedesktop.org/xorg/app/intel-gpu-tools/
      - `dnf install igt-gpu-tools`
  - Matrix benchmarking, live view, control center, matrix visualizer
    - Dash+Plotly framework
      - https://plotly.com/dash/
    - YQ
      - yq is a portable command-line YAML processor
      - https://mikefarah.gitbook.io/yq/
    - See `Pipefile`
  - Documentation
    - Sphinx
      - General-purpose, high-level programming language supporting multiple programming paradigms.
      - https://www.sphinx-doc.org/en/master/

    - `sudo dnf install python3-sphinx`

- Read the Docs Sphinx Theme
    - https://sphinx-rtd-theme.readthedocs.io/en/stable/
    - `pip install sphinx_rtd_theme --user`

CHAPTER

TWO

THE AGENT INTERFACE

The agent interface is based on the Recorder interface. It borrows its `record()` concept and implementation, and adds custom configuration functions.

See the agent_interface/spice branch for the reference implementation.

2.1 Initialization

```
// launch the Agent-Interface server socket
extern void agent_interface_start(unsigned int port);
```

This call launches the agent interface on the given `port`, and creates a thread that will listen for messages incoming on the socket.

Example:

```
void init_agent_interface(...)
{
    uint port = 1236;
    agent_interface_start(port);
    ...
}
```

2.2 On_connect callback

```
typedef int (*on_connect_cb_t)(void *);
void agent_interface_set_on_connect_cb(on_connect_cb_t cb, void *data);
```

This call allows registering one (and only one) callback which will be executed every time a `local_agent` connects to the interface socket.

The callback returns `0` if the communication is accepted or any other value to refuse the connection.

Example:

```
int on__connect(void *data)
{
    ...

    return 0;
}
```

(continues on next page)

(continued from previous page)

```
void init_agent_interface(...)
{
    ...

    static int *connect_data = ...;

    agent_interface_set_on_connect_cb(on_connect, &connect_data);
}
```

2.3 Feedback callback

```
typedef void (*feedback_received_cb_t)(void *, const char *);
extern void agent_interface_set_feedback_received_cb(feedback_received_cb_t cb, void␣
↪*data);
```

The feedback callback allows the reception of text messages sent by `local_agent`. The messages are `NUL`-terminated and can be used for any purpose.

Example:

```
static void feedback_received_cb(void *data, const char *feedback)
{
    ...
}

void init_agent_interface(...)
{
    ...
    static void *feedback_data = ...;

    agent_interface_set_feedback_received_cb(feedback_received_cb, feedback_data);
}
```

2.4 Code instrumentation

```
RECORDER(recorder_name, int_not_used, "Description not used");
record(recorder_name, fmt, ...);
```

The code instrumentation interface is borrowed from the Recorder project, and reuses its generic macro functions.

- the `agent interface` recorders only have a name (`recorder_name`).
- the records are not buffered, they are either discarded if no :code: *local_agent* is currently connected, or immediately sent through the socket.
- the recorder interface uses the usual `printf` syntax, and sends a string message to the `local agent`, along with a timestamp and the code location of the `record` call.

Example:

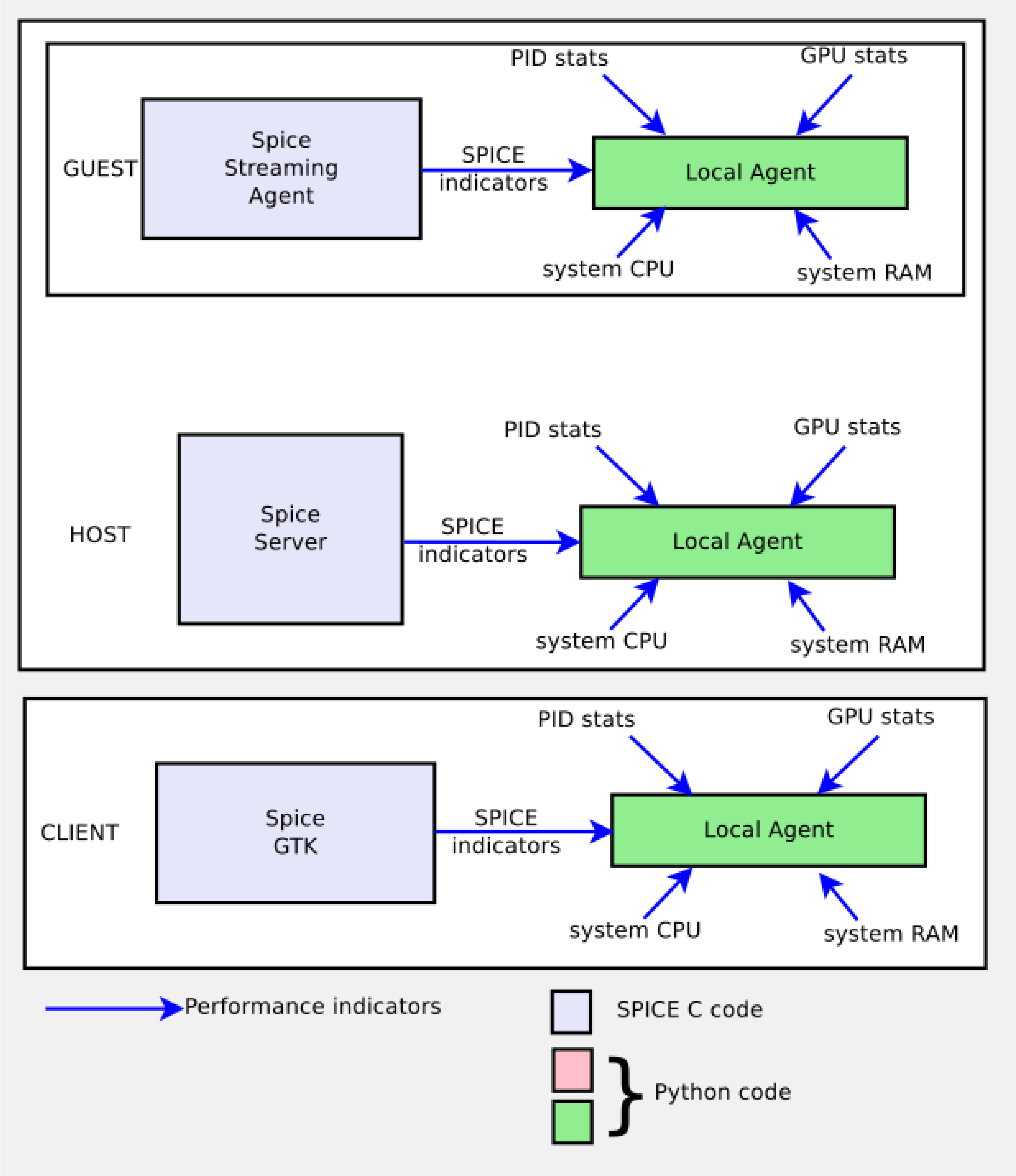
```
/* 1: timestamps + event identifiers */
record(frame, "Capturing frame...");
FrameInfo frame = capture->CaptureFrame();
record(frame, "Captured frame");

/* 2: internal stage monitoring */
record(frame, "Frame of %zu bytes", frame.buffer_size);
record(streaming_info, "resolution: height=%u width=%u", width, height);
```

CHAPTER

THREE

THE LOCAL AGENT

The local agent is a Python process running on each of the systems to benchmark. It collects raw performance indicators, aggregates them into tables and shares them with the performance collector.


GUEST
Spice Streaming Agent
SPICE indicators
Local Agent
PID stats
GPU stats
system CPU
system RAM
HOST
Spice Server
SPICE indicators
Local Agent
PID stats
GPU stats
system CPU
system RAM
CLIENT
Spice GTK
SPICE indicators
Local Agent
PID stats
GPU stats
system CPU
system RAM
Performance indicators
SPICE C code
Python code

3.1 Configuration

The local agent is configured via `cfg/<plugin>/agents.yaml`, using multiple blocks:

3.1.1 The overall settings

These settings are share by all the agents:

Machine Setup

```
setup:
  machines: <machine_set_name>

machines:
  <set_name_A>:
    <machine_name1>: <addr_A1>
    <machine_name2>: <addr_A2>

  <set_name_B>:
    <machine_name1>: <addr_B1>
    <machine_name2>: <addr_B2>
```

These configuration keys allows configuring IP addresses of the machines to benchmark. The multiple sets allows configuring multiple similar environments.

Example

```
setup:
  machines: home

machines:
  home:
    Host: <host_home_addr>
    Guest: <guest_home_addr>
    Client: <client_home_addr>

  server:
    Host: <host_server_addr>
    Guest: <guest_server_addr>
    Client: <client_server_addr>
```

This example allows switching easily between the `home` and `work` setups.

3.1.2 The measurement sets

The measurement-set defines list of measurement plugin to use for a given agent:

```
measurement_sets:
  <set_name>:
    - <plugin 1>
    - <plugin 2>
    - <plugin 3>:
        <key1>: <value1>
        <key2>: <value2>
```

This creates a measurement set with 3 plugins. Plugins `1` and `2` do not require any configuration, and plugin `3` has two configuration keys.

Example

```
measurement_sets:
  guest:
    - VMStat
    - MPStat
    - Intel_GPU_Top
    - SpiceAgentInterface:
        port: 1236
        mode: guest

  collector:
    - Perf_Collect:
        port: 1230
        host: vm
        mode: guest

    - Perf_Collect:
        port: 1230
        host: server
        mode: server
```

3.1.3 The agent configuration

The agent configuration is used to configure the behavior of the local agent:

```
<local_agent_name>:
  <key1>: <val1>
  <key2>: <val2>
   measurement_sets:
     - <set_name1>
     - <set_name2>
```

Examples

Run a local agent with the `guest` measurement set, and listen for the performance collector on port `1230`.

```
measurement_sets:
  guest:
    - VMStat
    - MPStat
    - Intel_GPU_Top
    - SpiceAgentInterface:
        port: 1236
        mode: guest

guest_agent:
  port_to_collector: 1230
  measurement_sets:
    - guest
```

Run the performance collector (for live view) and collect data from the `guest` / `server` / `client` agents.

```yaml
measurement_sets:
  collector:
    - Perf_Collect:
      port: 1230
      host: vm
      mode: guest

    - Perf_Collect:
      port: 1230
      host: server
      mode: server

    - Perf_Collect:
      port: 1231
      host: client
      mode: client

collector:
  run_as_collector: True
  measurement_sets:
    - collector
```

Run the benchmark performance collector (for scripted benchmarking) and collect data from the `guest` / `server` / `client` agents.

```yaml
benchmark:
  run_as_collector: True
  run_as_benchmark: True
  measurement_sets:
    - collector
  benchmark:
    script: matrix
```

Run in viewer mode, for the matrix visualizer and the offline record viewer.

```yaml
viewer:
  run_as_viewer: True
```

3.2 Plugin mechanism

The local agents rely on measurement plugins, that collect data, pre-process it if necessary, then add them row by row into a table. The table content is directly shared with the remote performance collector, if it is connected, or discarded.

3.2.1 Loading

From the `agents.yaml` configuration file, the measurement plugins are loaded by name: `<Plugin_Name>` should be stored inside `measurement/plugin_name.py`, and it should define a class named `Plugin_Name`. The `measurement` package can be at the top level or inside the `plugins.<mode_plugin>` package.

3.2.2 Class definition

```
import measurement

class Plugin_Name(measurement.Measurement):
    def __init__(self, cfg, experiment):
        measurement.Measurement.__init__(self, experiment)
        self.prop_1 = cfg["prop 1"]
        self.prop_2 = cfg["prop 2"]

    def setup(self): pass
    def start(self): pass
    def stop(self): pass
```

3.2.3 Defining and populating tables

```
field = ["<tbl_name>.<tbl_field1>", "<tbl_name>.<tbl_field2>", ...]
self.table = self.experiment.create_table(fields)

self.table.add(field1, field2, ...)
# or
self.table.add(field1=...,
               field2=...,
               ...)
```

Currently, the table name (`tbl_name`) must be repeated for each field (for legacy reasons), and the field `time` (without table table) is allowed.

3.2.4 Asynchronous input parsing

The data collection is done with asynchronous input reading and parsing, with the `asyncio` package:

- `utils.live.LiveSocket` reads asynchronously from a socket,
- `utils.live.FollowFile` follows a file line by line
- `utils.live.LiveStream` follows a process output line by line

These classes rely on a `process` callback, provided by the plugin, to parse new data and fill it into the experimentation tables.

3.2.5 Agent-interface entry parsing

The plugin stub `measurement.agentinterface.AgentInterface` provides an abstract class that can be extended to process messages arriving from the Agent Interface:

```
import collections

# agent-interface entry definition:
Entry = collections.namedtuple("RecorderEntry", "name function loc time msg")

class SpiceAgentInterface(measurement.agentinterface.AgentInterface):
    def setup(self):
        table = agent.experiment.create_table([...])
```

(continues on next page)

(continued from previous page)

```
        state = collections.namedtuple('StateName', 'state1 state2 ...')

        def process(entry):

            if ...:
                update(state)

            else:
                populate_table(table, state)

        agent.processors["recorder_name"] = process
```

3.2.6 Feedback messages

Local agent to Perf-collector feedback

The `feedback` table is a particular text-based channel. It is intended for sharing human-readable information between the local agent and the performance collector, where they are displayed onscreen. The local agent stores the feedback messages it receives and share all of them when a `perf_collector` connects to its socket.

Example:

```
def register_feedback(agent):
    agent.feedback_table = \
        agent.experiment.create_table([
            'feedback.msg_ts',
            'feedback.src',
            'feedback.msg',
        ])

    def process(entry):
        src, _, msg = entry.msg.partition(": ")

        agent.feedback_table.add(entry.time, src, msg.replace(", ", "||"))
        if msg.startswith("#"):
            msg = msg[:20] + "..." + msg[-20:]
        print(f"Feedback received: '{src}' says '{msg}'")

    agent.processors["feedback_interface"] = process
```

Perf-collector to Local agent feedback

The channel is bi-directional, so the agent can also receive feedback messages. These messages can be used to perform local actions, such as launching or killing the process to benchmark.

Example:

```
from measurement.feedback import feedback
def remote_ctrl(_msg):
    msg = _msg[:-1].decode('ascii').strip()
    action, _, action_params = msg.partition(":")
    ....

obj = types.SimpleNamespace()
obj.send = remote_ctrl
```

(continues on next page)

(continued from previous page)

```
feedback.register("remote_ctrl", obj)
```

CHAPTER

FOUR

THE LIVE VIEW AND INTERACTIVE CONTROL CENTER

The live view and interactive control center is a Dash+Plotly web interface.

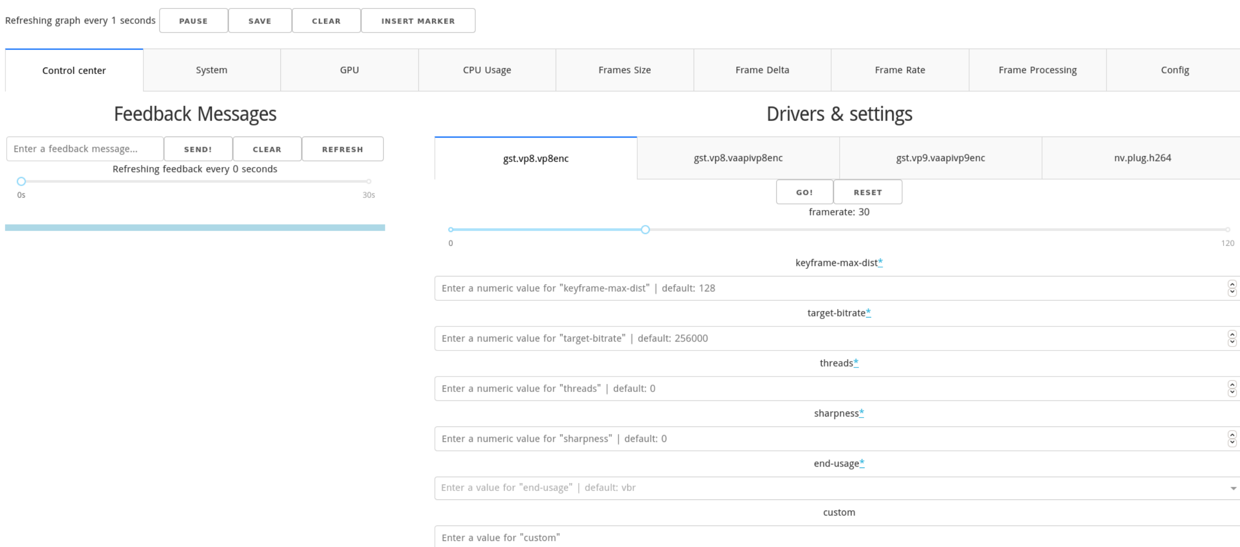
To run it, start a local agent with `run_as_collector=True`, eg:

```yaml
measurement_sets:
  collector:
    - Perf_Collect:
      port: 1230
      host: server
      mode: server

    - Perf_Collect:
      port: 1231
      host: client
      mode: client

collector:
  run_as_collector: True
  measurement_sets:
    - collector
```

4.1 Control Center

The control center interface allows an interactive control of the application to benchmark. It has two sides:

1. The feedback messages
2. The drivers & settings



4.1.1 The feedback messages

The feedback panel is used to exchange (send and receive) feedback messages with the local agents. Currently, the messages are sent to all the agents currently connected, and the agents should parse the message to decide if they should process it or not.

See [The local agent]/[Plugin mechanism]/[Feedback messages] for further information.

4.1.2 Drivers & settings

The Drivers & settings tabs are configured by `cfg/<mode_plugin>/driver_settings.yaml`:

```yaml
template_group:
  _group: true
  opt_name:
    desc: the option description
    url: link for more information

    type: int[start:stop:step]=default
    # OR
    type: int # free range
    # OR
    type: enum
    values: val_1, val_2, ...


_all:
  _group: true
  framerate:
    type: int[0:120:5]
    default: 30

grp_gst-vp8_vp9_intelvaapi_encoding:
  _group: true
  bitrate:
    type: uint
```

(continues on next page)

(continued from previous page)

```
    desc: "The desired bitrate expressed in kbps (0: auto-calculate)"
    url: "https://gstreamer.freedesktop.org/documentation/vaapi/vaapivp8enc.html?gi-
↪language=c#vaapivp8enc:bitrate"
    default: 0

  keyframe-period:
    type: uint
    default: 30
    desc: "Maximal distance between two keyframes (0: auto-calculate)"
    url:
    "https://gstreamer.freedesktop.org/documentation/vaapi/vaapivp8enc.html
↪#vaapivp8enc:keyframe-period"

gst.vp8.vaapivp8enc:
  _group: grp_gst-vp8_vp9_intelvaapi_encoding
  encoder-reload:
    type: enum
    values: yes, no
```

In this example:

- `gst.vp8.vaapivp8enc` is a driver name (here, a codec/encoder). It will be in a dedicated tab, with the settings from the `grp_gst-vp8_vp9_intelvaapi_encoding` group and `encoder-reload` enum.
- `grp_gst-vp8_vp9_intelvaapi_encoding` is a group of settings with two settings (`bitrate` and `keyframe-period`).
- `template` is a group never used.
- `_all` is a special group always included

Mind that the `default` field is only an indicator for the user, it is never actually used.

The `type` field can be:

- `int[start:stop:step]=default` for a slider
- `int|uint|ufloat` for a numeric input field
- `enum`, along with a `values` field, for a drop-down list.

4.2 Live View

The rest of the web interface is the live view tabs.

The performance collector receives performance indicators and plots them in multiple graph. These tabs and graphs are configured by `cfg/<mode_plugin>/dataview.yaml`:

```
CPU Usage:
  Server:
    table: server.server-pid
    x: time | as_timestamp
    y: server-pid.cpu_user
    y2: server-pid.cpu_system
    y_max: 100
    y_title: '% of CPU usage'

  Client:
    table: client.client-pid
    x: time | as_timestamp
    y: client-pid.cpu_user
    y2: client-pid.cpu_system
    y_max: 100
    y_title: '% of CPU usage'

Frame Delta:
  Guest generation rate:
    table: guest.guest
    x: guest.msg_ts | as_us_timestamp
    y: guest.msg_ts | as_delta | as_s_to_ms > Generation time delta

  Server forward rate:
    table: server.host
    x: host.msg_ts | as_us_timestamp
    y: host.msg_ts | as_delta | as_s_to_ms > Forwarding time delta
```

This example show two tabs (*CPU Usage* and *Frame Delta*), with two graphs in each of them.

The graphs are specified with the following fields:

- *table: <mode>.<table_name>*, where *<mode>* is the *mode* setting of one of the *Perf_Collect* measurement plugins, and *<table>* is the name of a table generated by a local agent measurement plugin.
- *x*, *y*, *y2* are the axis of the graph. They may be modified with | *modifier*, that apply custom transformation to the table values. See *ui.graph.GraphFormat* for the current modifiers, and add custom ones in *plugins.<mode_plugin>.graph.GraphFormat*.
- *y_max* and *y_title* are custom properties of the *y* axis.

CHAPTER

FIVE

THE SCRIPTED BENCHMARKING

In addition to the live view/interactive control interface, the toolbox provides a scripting interface. This scripting interface can be used to design custom benchmarks relying on the *streaming_stats* infrastructure. The reference implementation is the Adaptive Streaming Matrix Benchmarking (*plugins.adaptive.scripting.matrix*), that can be easily extended to adapt it to different environments (see *plugins.specfem.scripting.matrix*).

```yaml
_engine: matrix
_nb_agents: 3

_name: matrix

record_time: 40

script_config:
  display:
    web_still: https://news.ycombinator.com/
    webgl_wipeout: https://phoboslab.org/wipeout/
    webgl_aquarium: https://webglsamples.org/aquarium/aquarium.html
    webgl_sprites: https://webglsamples.org/sprites/index.html

run:
  - study_fps
  - study_resolution
  - study_bitrate
  - study_kfr
  - study_display

expe:
  study_fps:
    driver: gst.vp8.vaapivp8enc
    settings:
      framerate: 20, 25, 30, 35, 40, 45, 50, 120
      bitrate: 8000, 16000, 32000
      keyframe-period: 128
      rate-control: cbr
    scripts:
      display: webgl_wipeout, img_lady_1920
      resolution: 1920x1080, 1280x720
```

5.1 Generic Scripted Benchmarking

5.1.1 Loading a Scripting Benchmark

The scripting benchmarking is launched with a `run_as_benchmark` local agent:

```
benchmark:
  run_as_benchmark: True
  measurement_sets:
    - collector
  benchmark:
    script: matrix
```

The `measurement_sets` indicate how to collect data from the local agents, and `benchmark.script=matrix` specifies the `name` of the script that will be executed.

The scripts are defined in `cfg/<mode_plugin>/benchmark.yaml`. The file can define multiple script configuration, separated by the YAML — document separator. Every script *must* defined 3 properties:

- `_name`: a unique name identifying the scripting configuration;
- `_nb_agents`: an integer telling how many local agents must be connected before starting the benchmark;
- `_engine`: the name of the Python module implementing the scripting interface (see below).

The benchmark will run in `dry` mode unless the argument `run` is found in the command-line arguments (`dry = "run" not in sys.argv`).

5.1.2 Extending the Script Engines

The script engine is selected from the `_engine` property of the benchmark file. It will be loaded from the class named `<_engine.capitalize()>` inside the module `plugins.<mode_plugin>.scripting.<_engine>`.

The module must expose a `configure(expe)` method:

```
def configure(expe): pass
```

The class must extend `ui.script.Script` class and define the following methods:

```
class Name(script.Script):

    def __init__(self, yaml_desc):
        script.Script.__init__(self, yaml_desc)

   def do_run(self, exe): pass
```

The construct provides the YAML document `yaml_desc` describing the benchmark to run, and the method `do_run` is in charge of running the benchmark. Parameter `exe` is a helper object that calls back the toolbox for various operations:

```
class Exec():
    def log(self, *args, ahead=False): pass

    def execute(self, cmd): pass

    def request(self, msg, **kwargs): pass
    def apply_settings(self, driver, settings, force=False): pass
```

(continues on next page)

(continued from previous page)

```
    def append_feedback(self, msg): pass
    def clear_feedback(self): pass

    def save_record(self, fname): pass
    def clear_record(self): pass

    def reset(self): pass
```

The `execute` method takes either a shell command, or, if the command starts with `/py/`, a command to delegate to the python helper. For instance: `/py/ <helper_name> <args...>` will do:

1. load `plugins.<mode_plugin>.scripting.<helper_name>` module
2. call `<helper_name>.<helper_name>(state, exe, machines, args)` function, where:

- `state` is a `types.SimpleNamespace()` object created for each execution of the script
- `exe` is the `Exec()` instance described above
- `machines` is the map of machines defined in `agents.yaml`
- `args` is the `<args...>` string of the command

5.1.3 The Control Plugin

The control plugin is loaded from `plugins.<mode_plugin>.control`. It is used by to apply settings from the scripting benchmarks and the control center. It must expose the following functions:

```
def configure(plugin_cfg, machines): pass
def apply_settings(driver_name, settings): pass
def request(msg, dry, log, **kwargs): pass
def reset_settings(): pass
```

5.2 Matrix Scripted Benchmarking

The Matrix Benchmarking is a scripting engine developed for SPICE Adaptive Streaming (`plugins.adaptive.scripting.matrix`) and made reusable for Specfem3D plugin (`plugins.specfem.scripting.matrix`).

The results of this engine can be directly loaded in the Matrix Visualizer (see next section).

The idea behind matrix benchmarking is to define a set of parameters (`A`, `B`, `C`), that can take multiple values (`A1/A2/A3`, `B1/B2`, `C1/C2`). The engine will then iterate over the product: `A1+B1+C1`, `A1+B1+C2`, `A1+B2+C1`, etc, and measure the performance of this configuration. Once the datasets have been collected, the Matrix Visualizer allows studying the impact of each parameter, by plotting one or multiple values at the same time for each parameter.

The parameters can be either application settings (eg, command-line arguments, env variables, configuration-file values, ...), or scripted properties. The `settings` are passed along with the `driver` name to `plugins.<mode_plugin>.control.apply_settings(driver_name, settings)`; and the scripted properties are activated/stopped before/after the execution of a benchmark. The starting/stopping scripts are specified in the YAML document, along with overall startup/teardown scripts that are executed at the beginning/end of the overall benchmark.

5.2.1 Configuration

A matrix benchmark configuration looks like this:

```yaml
_engine: matrix
_nb_agents: ...
_name: ...

run:
  - expe1
  - expe2
  - _disabled_expe_1
  - _disabled_expe_2

expe:
  expe1:
    driver: my_driver

    settings:
      A: A1, A2, A3
      B: B1, B2
      C: C1, C2

    scripts:
      script1: value_1a, value_1b, value_1b
      script2: value_2a, value_2b

  expe2: ...

scripts:
  setup:
    - command_to_setup

  teardown:
    - command_to_teardown

  script1:
    before:
      - script1 before $script1
    after:
      - script1 after $script1

  script2:
    before:
      - script2 before $script2
    after:
      - script2 after $script2
```

This configuration file will run `expe1` and `expe2`. Expe `_disabled_expe_1` and `_disabled_expe_2` are skipped because they start with a _.

Experiment `expe1` has three application settings and two scripted properties, making a total of `3 x 2 x 2 x 3 x 2 = 72` benchmarks to run.

The following commands will be executed for this benchmark:

```
$ command_to_setup
$ script1 before value_1a
```

(continues on next page)

(continued from previous page)

```
$ script2 before value_2a
% apply_settings(my_driver, {A: A1, B: B1, C: C1})
% apply_settings(my_driver, {A: A1, B: B1, C: C2})
% apply_settings(my_driver, {A: A1, B: B2, C: C1})
## continue with all the settings matrix
$ script2 after value_2a
$ script2 before value_2b
## run all the settings matrix
$ script2 after value_2b
$ script1 after value_1a
$ script1 before value_1b
## for all the values of script2, run the settings matrix
$ script1 after value_1b
$ script1 before value_1c
## for all the values of script2, run the settings matrix
$ script1 after value_1c
$ command_to_teardown
```

5.2.2 Customization to another application

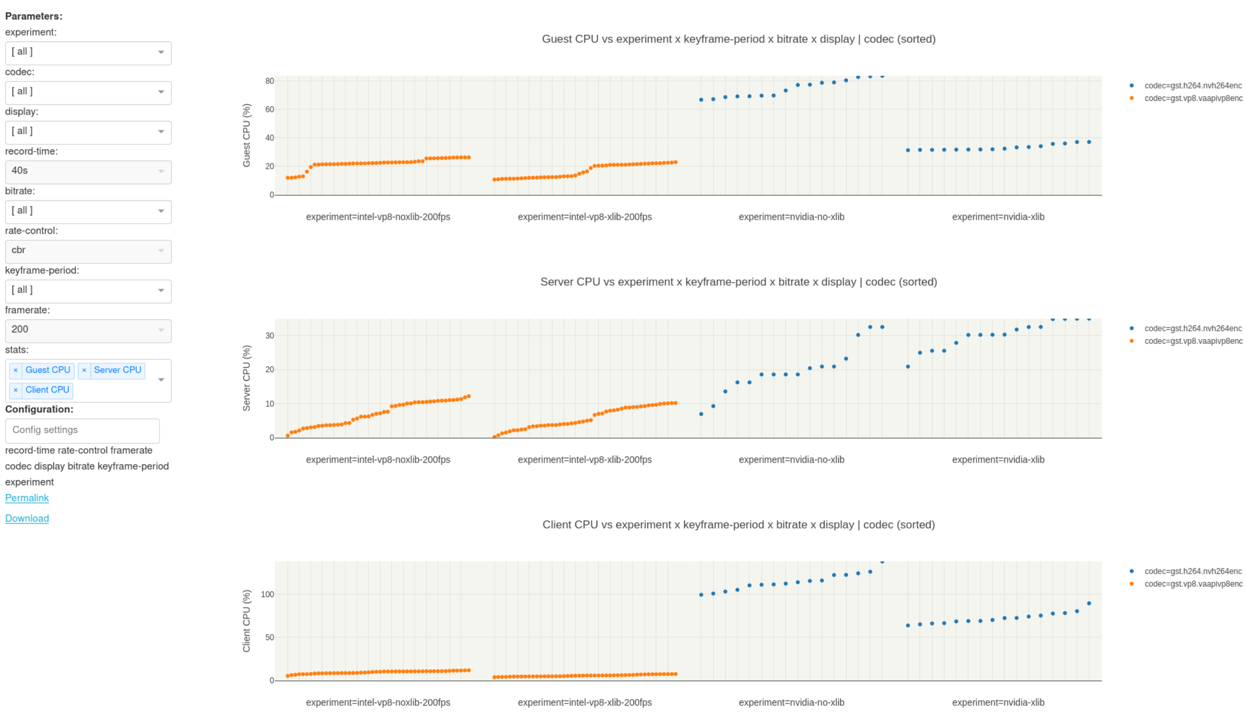
In addition to the control plugin customization, the Adaptive matrix benchmarking must be customized to be used with another application. See `plugins.specfem.scripting.matrix` for an illustration.

```
from plugins.adaptive.scripting import matrix as adaptive_matrix

class CustomMatrix():

    @staticmethod
    def add_custom_properties(yaml_desc, params): pass # nothing

    @staticmethod
    def get_path_properties(yaml_expe): return [...]

    @staticmethod
    def prepare_new_record(exe, context, settings_dict):
        ...

        exe.apply_settings(context.params.driver, settings_dict)

        exe.clear_record()
        exe.clear_feedback()

    @staticmethod
    def wait_end_of_recording(exe, context):
        while running:
            time.sleep(1)

def configure(expe):
    adaptive_matrix.configure(expe)

    adaptive_matrix.customized_matrix = CustomMatrix
```

THE MATRIX VISUALIZER

The matrix visualizer is an offline viewer of the matrix benchmarking results.

6.1 Simple Aggregation Plots



The UI part of the code is located in `ui.matrix_view`, and the code f:code:or visualizing the results with simple aggregation functions is inside *ui.table_stats*. To setup such aggregation functions, create a module `plugins.<mode_plugin>.matrix_view`, with a `register` function.

If you are using the matrix benchmarking script from the `adaptive` plugin, you can reuse `parse_data/all_records/get_record` functions.

See `ui.table_stats.TableStats` for the existing aggregation functions.

Example:

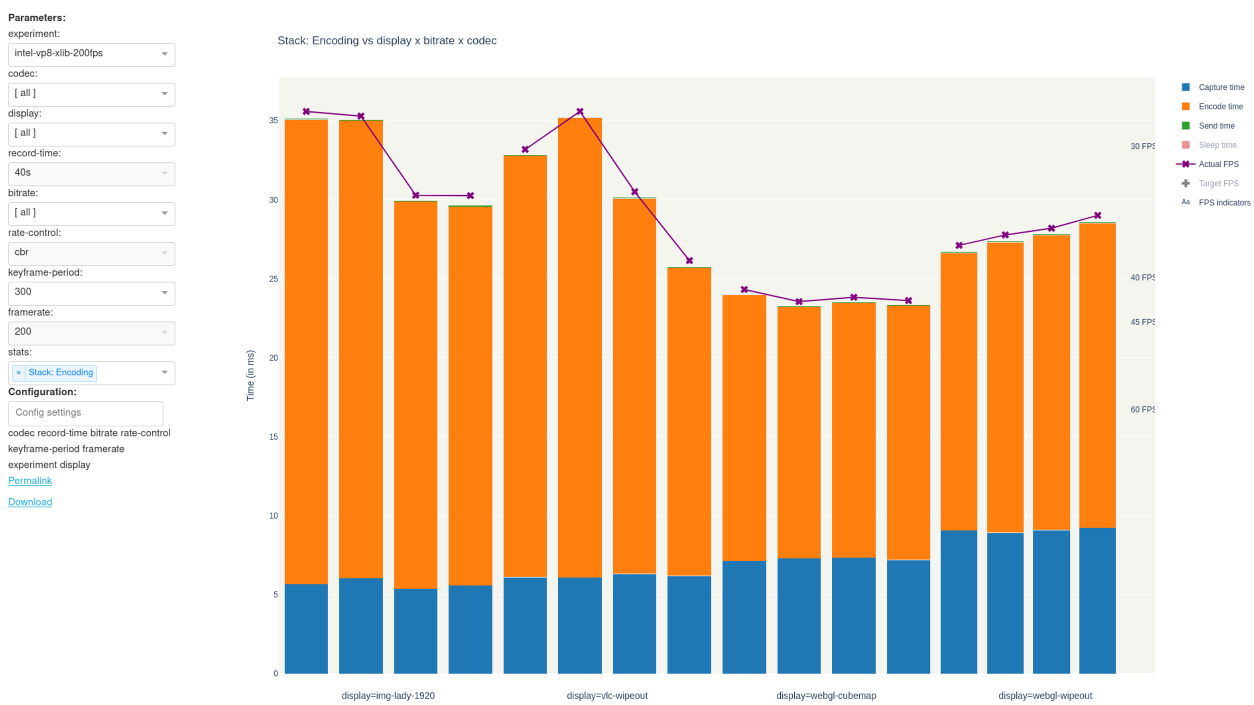
```
from ui.table_stats import TableStats
```

(continues on next page)

(continued from previous page)

```
import plugins.adaptive.matrix_view
from plugins.adaptive.matrix_view import parse_data, all_records, get_record

plugins.adaptive.matrix_view.rewrite_properties = lambda x:x

def register():
    TableStats.Average("sys_mem_avg", "Free Memory (avg)", "?.mem",
                       "mem.free", ".0f", "MB")

    TableStats.Average("sys_cpu_avg", "System CPU Usage (avg)", "?.cpu",
                       "cpu.idle", ".0f", "%")

    TableStats.Average(f"spec_cpu", f"Specfem CPU Usage (avg)", f"?.local-pid",
                       f"local-pid.cpu", ".0f", "%")

    TableStats.Average(f"general_time", f"Overall time", "?.general",
                       "general.specfem_time", ".0f", "sec")
```

6.2 Custom Plots



It is possible to write full-customized plots, instead of using the simple aggregation. See `plugins.adaptive.matrix_view` package for examples.

The custom plot class should implement this interface and return `plotly` figures.

```
import plotly.graph_objs as go

from ui.table_stats import TableStats
from ui import matrix_view

class PlotClassname():
    def __init__(self, ...):
        self.name = ... unique human-readable name
        self.id_name = ... unique name for html-id
        TableStats._register_stat(self)
        self.no_graph = True or False # ignores do_plot/graph if True

    def do_hover(self, meta_value, variables, figure, data, click_info):
        return hover_div_content # content of the hover-div below the graph when the user clicks on the graph

    def do_plot(self, ordered_vars, params, param_lists, variables, cfg):
        return fig, caption_div_below # figure and caption below
```

6.3 Custom Plot Settings

The interface of the matrix visualizer focuses on selecting the dataset that will be plotted, and the relative order of the parameters. However, when building custom graphs, one may need more configuration hooks. For this purpose, we added a simple text-based mechanism to pass custom *configuration* parameters to the plotting code.

These settings are received via the `cfg` dictionary passed to the `do_plot` method. It contains a key-value map, that can be freely used to tweak the plot (the basic format must be `<key>=[<value>]`).

In the UI side, the `configuration` is a text field. Its current value is always passed to the plotting code. By clicking on the `configuration` label, the config-string is stored, so that multiple settings can be passed. To remove a configuration setting, set it's value to empty.

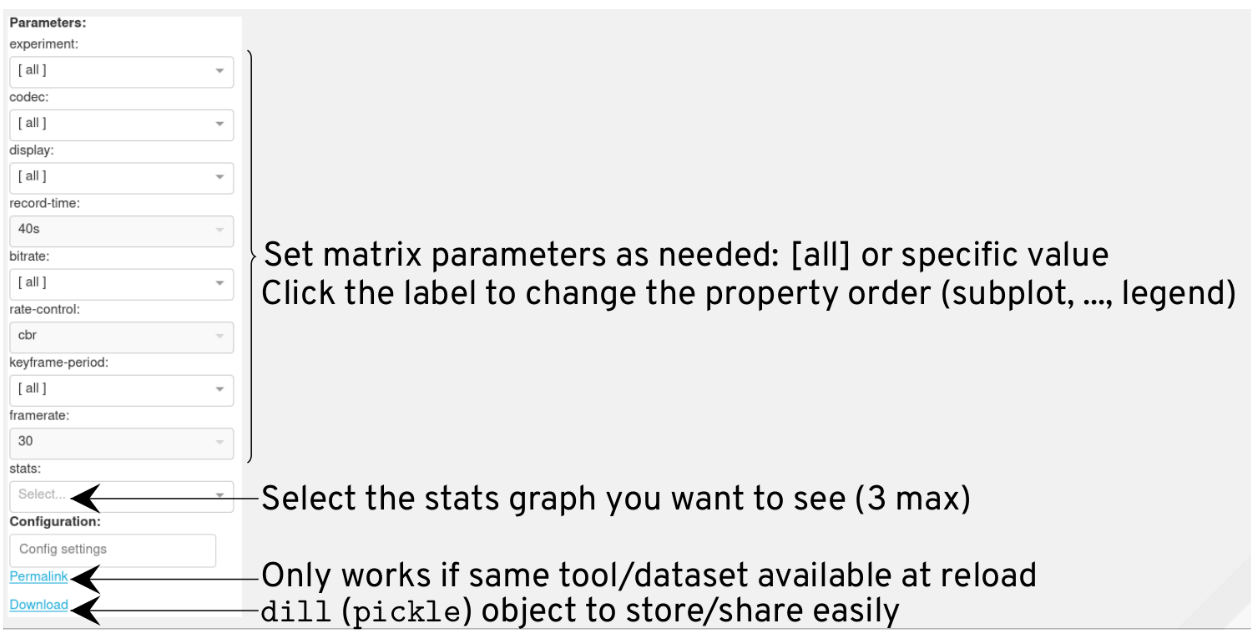
6.4 Permalink and Download

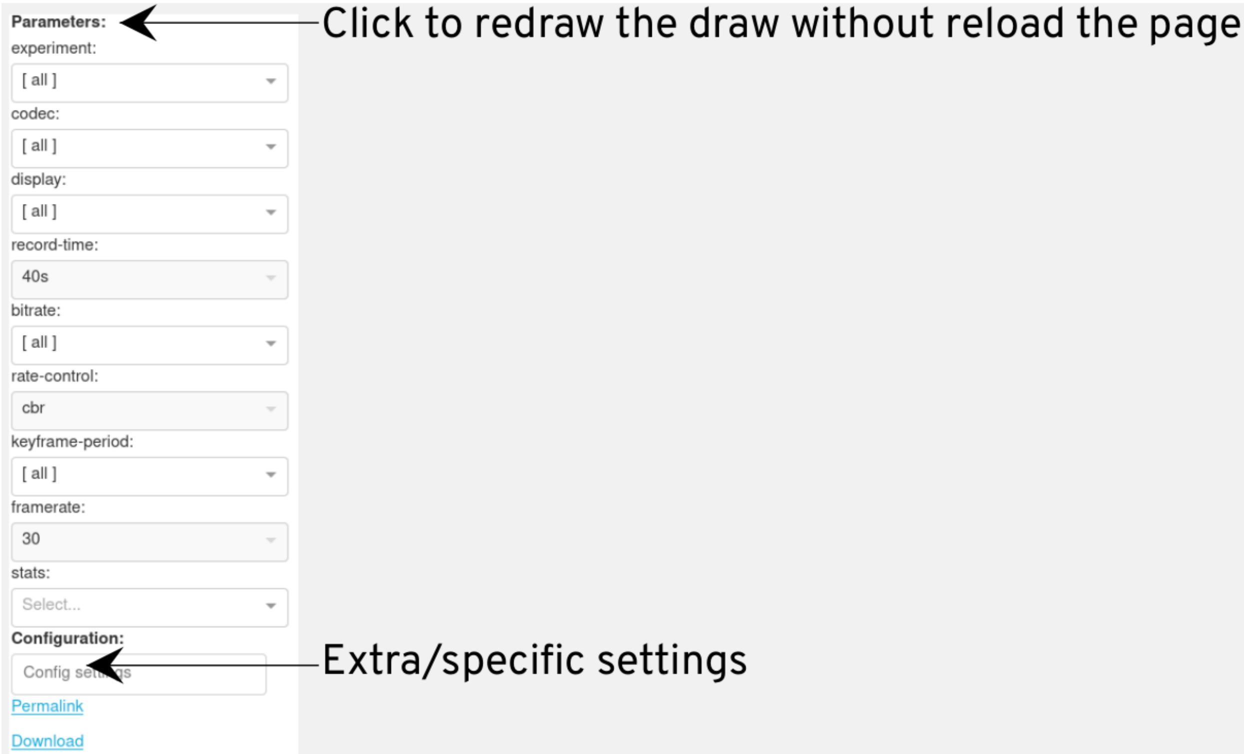
To share a custom view of the matrix visualizer, there exists multiple ways:

1. take a screenshot. That's the easiest way, but it looses the interactive info available with Plotly's graph. So we had to design more advance ways.
2. share a permalink. This link allows reloading the UI with its current set (parameters selected, graphs to display, custom settings . . .). For the permalink to work as expected, the viewer dataset must be identical when one tries to access the document. This is a lightweight way of sharing, but the content may change over the time, when records are added or removed.
3. download an offline version of current UI (dill format). This will generate a dill file containing the serialized dash object of the current layout. Then the dill file just has to be deserialized and return from a dash callback. Within the toolbox, the dill file can be placed in <base_dir>/saved/.../my_file.dill, and reloaded with http://.../saved/.../my_file.dill.

6.5 Interface Details

The interface of the Matrix Visualizer is focused on the selection of the data-sets to visualize. The images below detail the interface buttons and clickable labels:



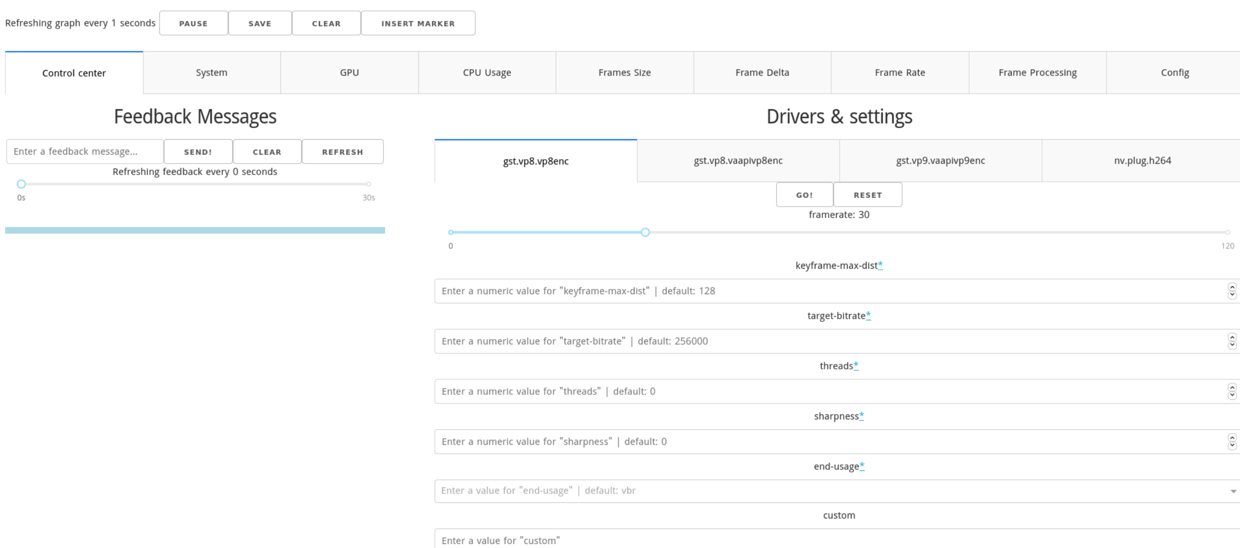

Parameters:
Click to redraw the draw without reload the page
experiment:
[all]
codec:
[all]
display:
[all]
record-time:
40s
bitrate:
[all]
rate-control:
cbr
keyframe-period:
[all]
framerate:
30
stats:
Select...
Configuration:
Extra/specific settings
Permalink
Download

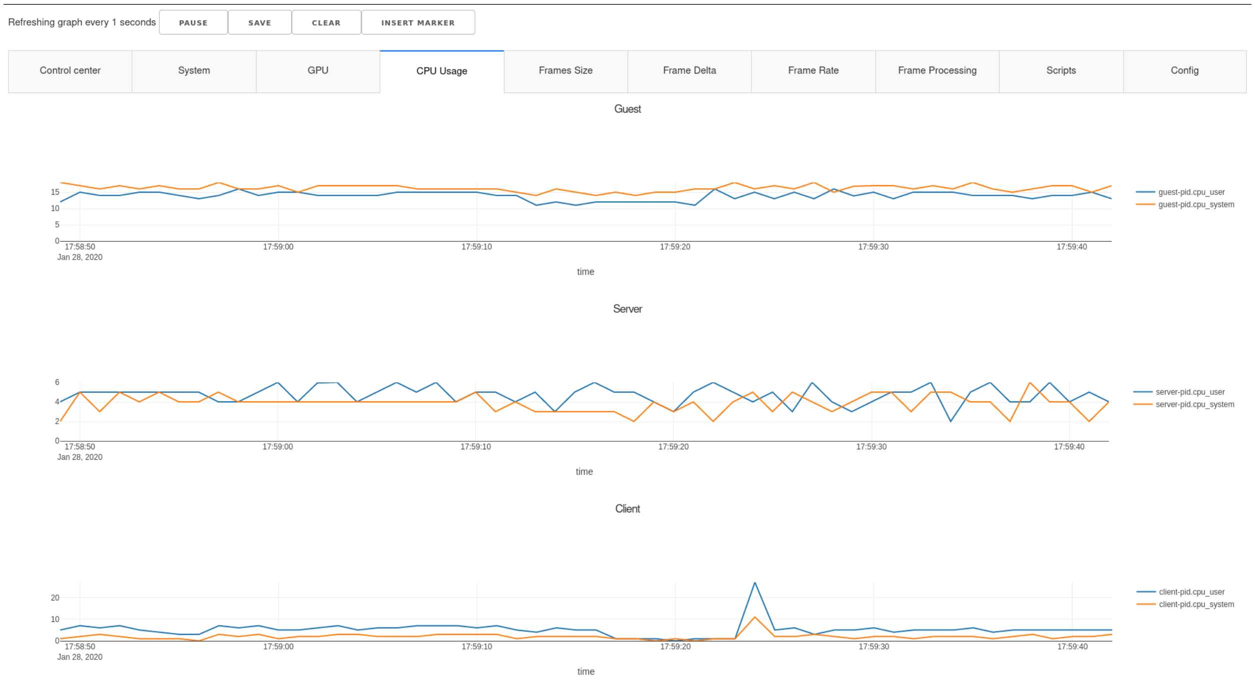
CHAPTER

SEVEN

GALLERY

7.1 Control Center and Live View

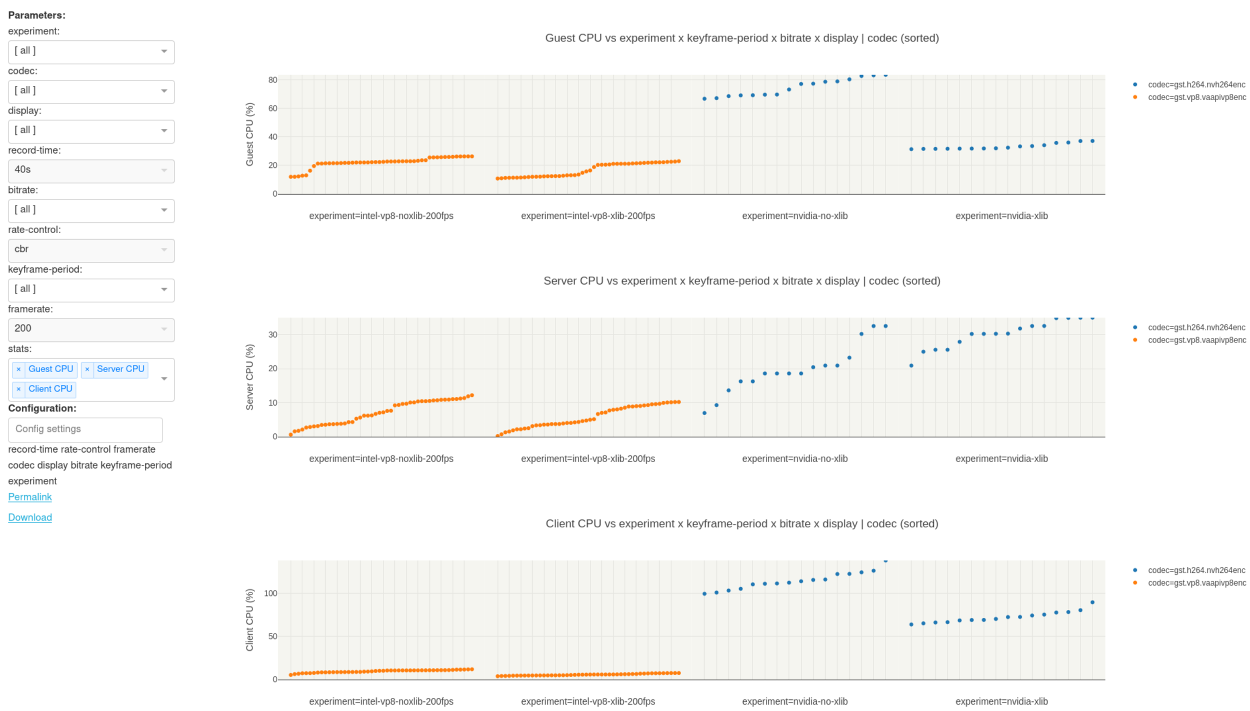


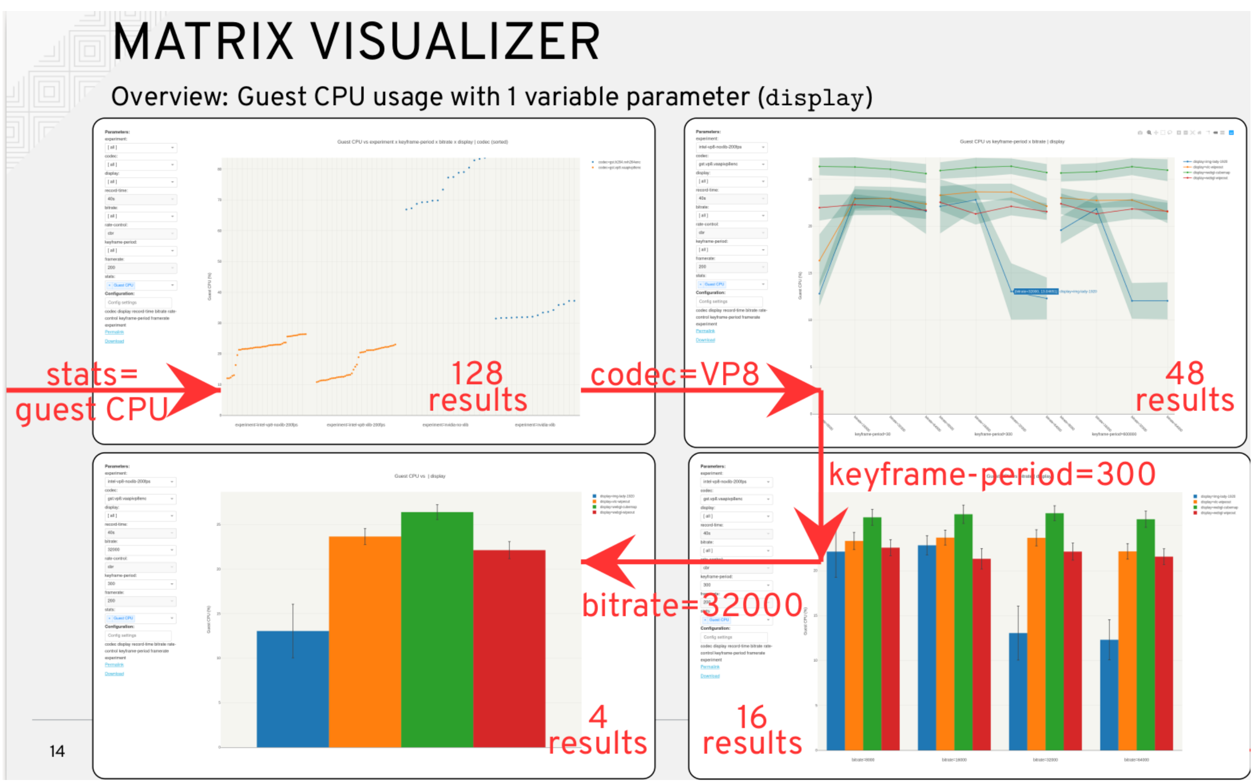

Refreshing graph every 1 seconds
PAUSE
SAVE
CLEAR
INSERT MARKER
Control center
System
GPU
CPU Usage
Frames Size
Frame Delta
Frame Rate
Frame Processing
Scripts
Config
Guest
guest-pid.cpu_user
guest-pid.cpu_system
time
Server
server-pid.cpu_user
server-pid.cpu_system
time
Client
client-pid.cpu_user
client-pid.cpu_system
time
Jan 28, 2020

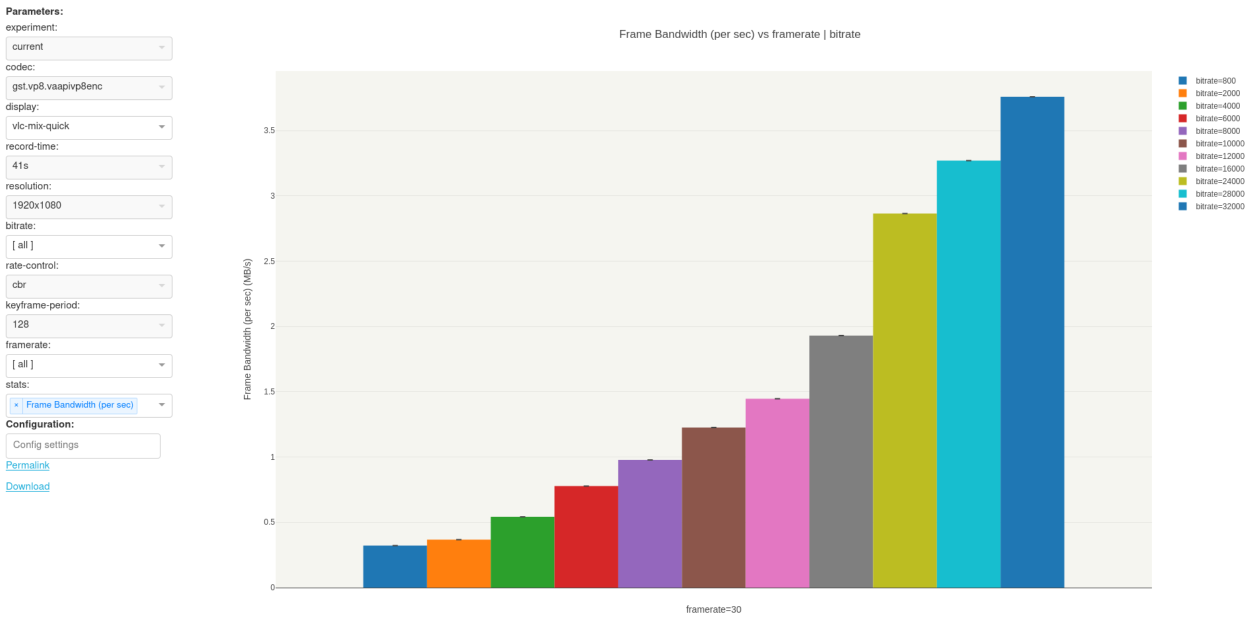

Refreshing graph every 1 seconds
PAUSE
SAVE
CLEAR
INSERT MARKER
Control center
System
GPU
CPU Usage
Frames Size
Frame Delta
Frame Rate
Frame Processing
Scripts
Config
Guest processing duration
guest.sleep_duration
guest.capture_duration
guest.encode_duration
guest.send_duration
guest.msg_ts
Client processing duration
client.decode_duration
per_second (over 5s)
per_second (over 20s)
client.msg_ts
Client processing per frame size
Frame processing time (in ms)
Frame size (in KB)
Jan 28, 2020

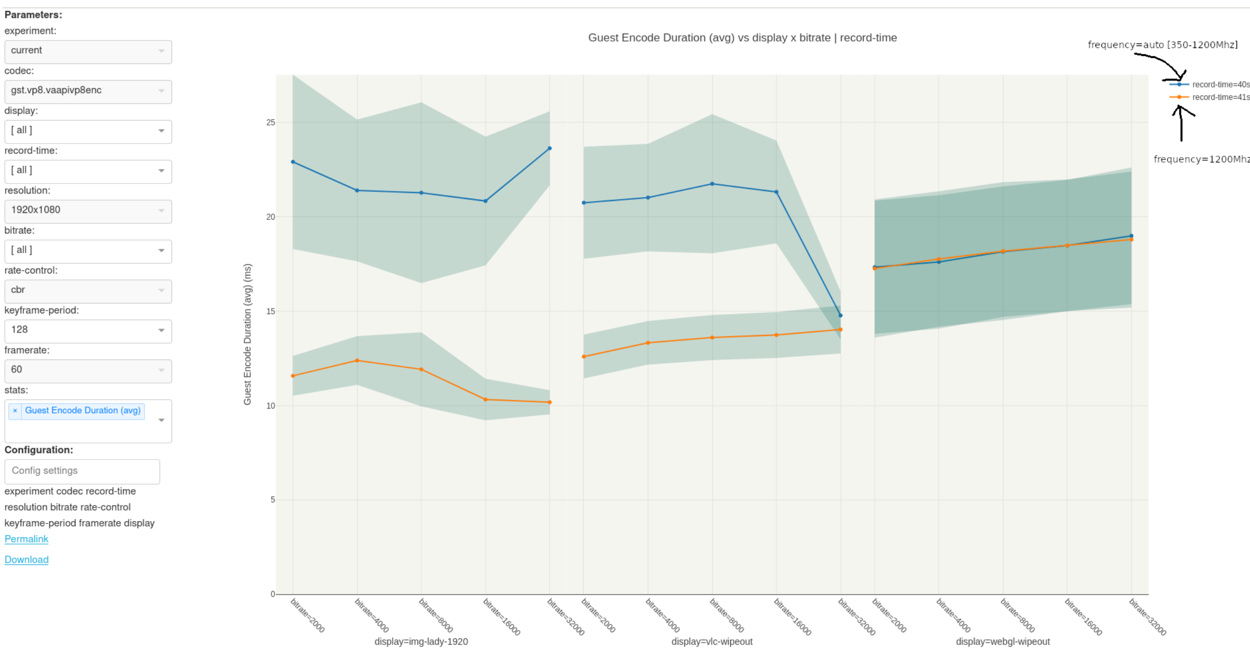
7.2 Matrix Benchmarking

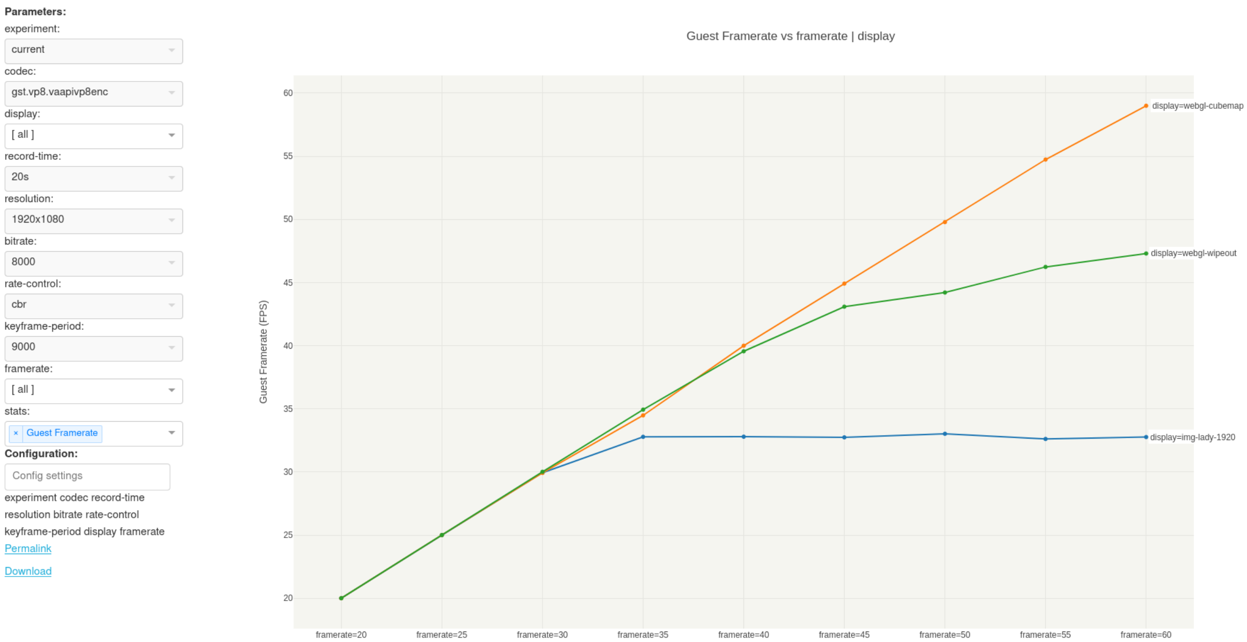
7.2.1 Simple Aggregation Plots

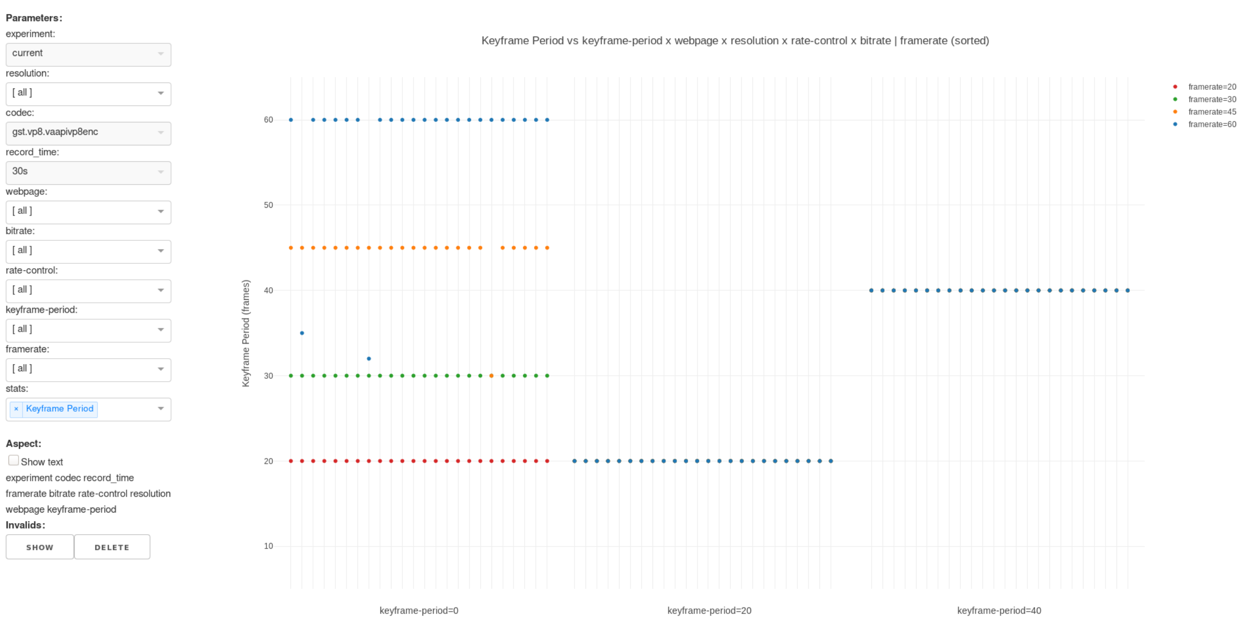




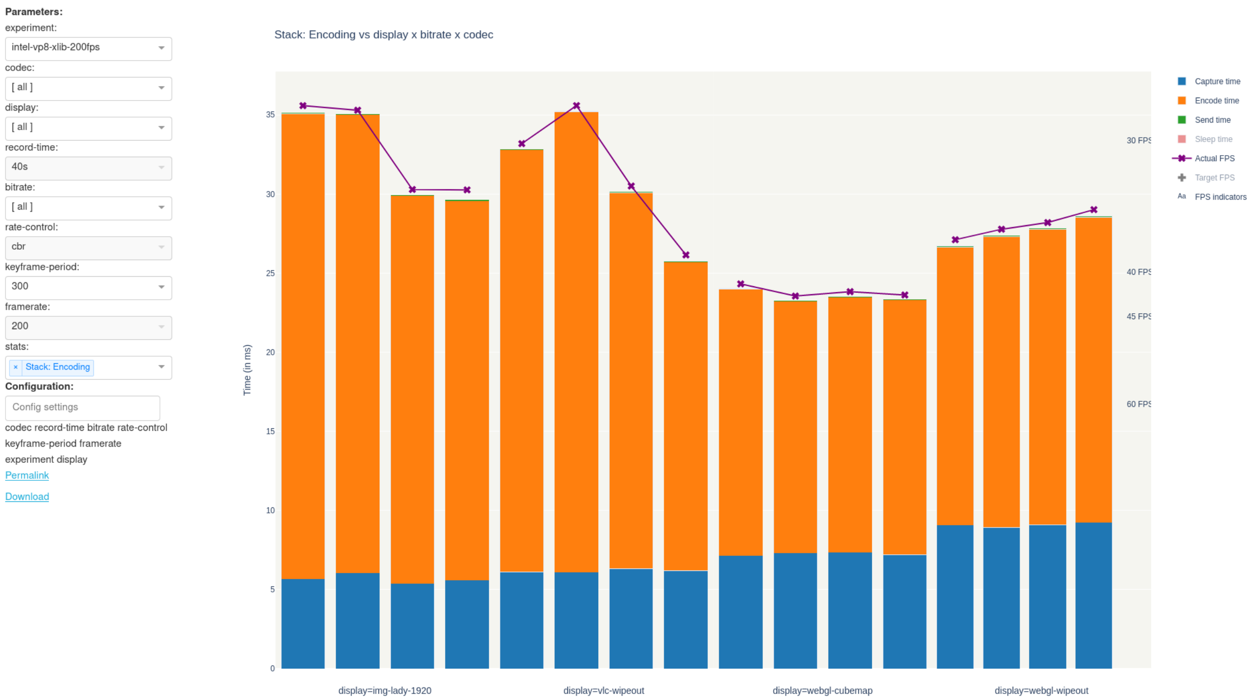

Parameters:
experiment:
current
codec:
gst.vp8.vaapivp8enc
display:
vlc-mix-quick
record-time:
41s
resolution:
1920x1080
bitrate:
[all]
rate-control:
cbr
keyframe-period:
128
framerate:
[all]
stats:
Frame Bandwidth (per sec)
Configuration:
Config settings
Permalink
Download
Frame Bandwidth (per sec) vs framerate | bitrate
Frame Bandwidth (per sec) (MB/s)
framerate=30
bitrate=800
bitrate=2000
bitrate=4000
bitrate=6000
bitrate=8000
bitrate=10000
bitrate=12000
bitrate=16000
bitrate=24000
bitrate=28000
bitrate=32000

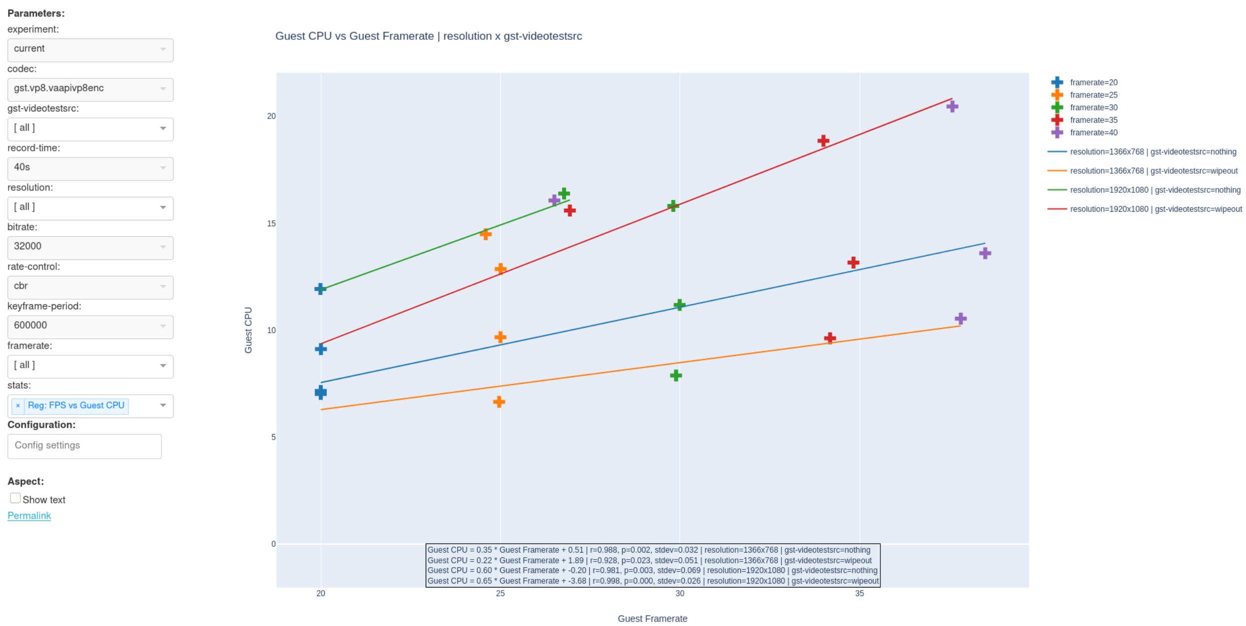

Parameters:
experiment:
current
codec:
gst.vp8.vaapivp8enc
display:
[all]
record-time:
[all]
resolution:
1920x1080
bitrate:
[all]
rate-control:
cbr
keyframe-period:
128
framerate:
60
stats:
Guest Encode Duration (avg)
Configuration:
Config settings
experiment codec record-time
resolution bitrate rate-control
keyframe-period framerate display
Permalink
Download
Guest Encode Duration (avg) vs display x bitrate | record-time
frequency=auto [350-1200Mhz]
record-time=40s
record-time=41s
frequency=1200Mhz
Guest Encode Duration (avg) (ms)
display=img-lady-1920
display=vlc-wipeout
display=webgl-wipeout

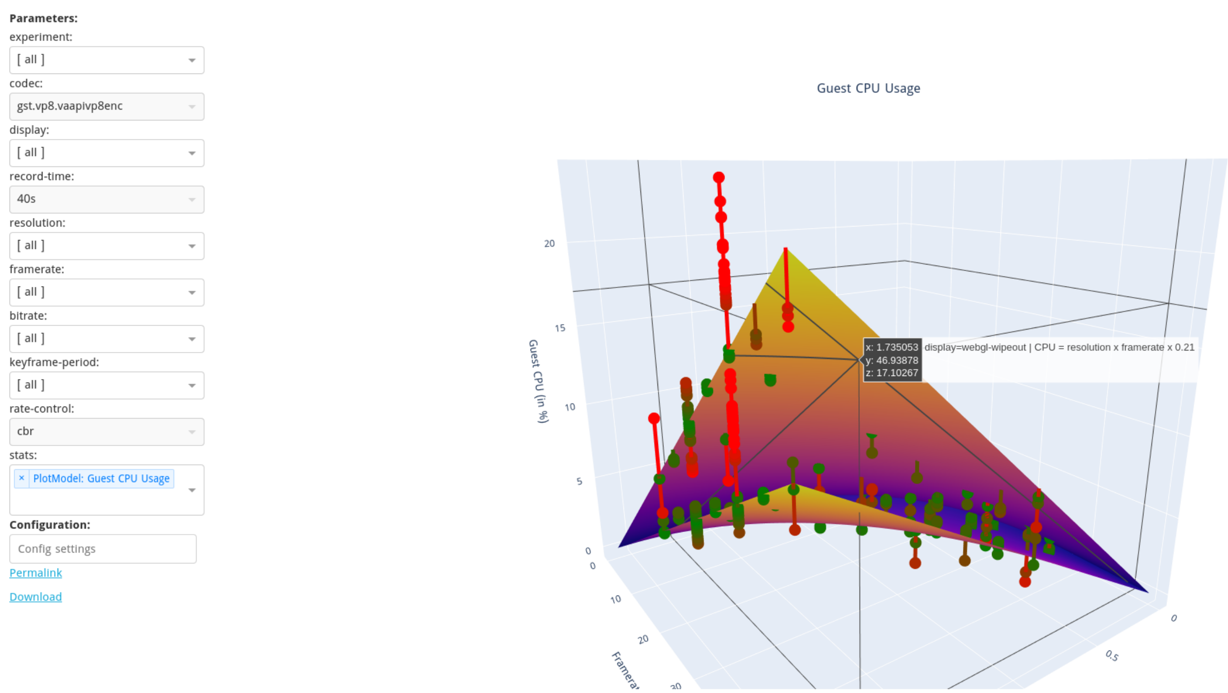

Parameters:
experiment:
current
codec:
gst.vp8.vaapivp8enc
display:
[all]
record-time:
20s
resolution:
1920x1080
bitrate:
8000
rate-control:
cbr
keyframe-period:
9000
framerate:
[all]
stats:
× Guest Framerate
Configuration:
Config settings
experiment codec record-time
resolution bitrate rate-control
keyframe-period display framerate
Permalink
Download
Guest Framerate vs framerate | display
Guest Framerate (FPS)
display=webgl-cubemap
display=webgl-wipeout
display=img-lady-1920
framerate=20
framerate=25
framerate=30
framerate=35
framerate=40
framerate=45
framerate=50
framerate=55
framerate=60


Parameters:
experiment:
current
resolution:
[all]
codec:
gst.vp8.vaapivp8enc
record_time:
30s
webpage:
[all]
bitrate:
[all]
rate-control:
[all]
keyframe-period:
[all]
framerate:
[all]
stats:
× Keyframe Period
Aspect:
Show text
experiment codec record_time
framerate bitrate rate-control resolution
webpage keyframe-period
Invalids:
SHOW
DELETE
Keyframe Period vs keyframe-period x webpage x resolution x rate-control x bitrate | framerate (sorted)
Keyframe Period (frames)
framerate=20
framerate=30
framerate=45
framerate=60
keyframe-period=0
keyframe-period=20
keyframe-period=40

7.2.2 Custom Plots




Parameters:
experiment:
current
codec:
gst.vp8.vaapivp8enc
gst-videotestsrc:
[all]
record-time:
40s
resolution:
[all]
bitrate:
32000
rate-control:
cbr
keyframe-period:
600000
framerate:
[all]
stats:
Reg: FPS vs Guest CPU
Configuration:
Config settings
Aspect:
Show text
Permalink
Guest CPU vs Guest Framerate | resolution x gst-videotestsrc
framerate=20
framerate=25
framerate=30
framerate=35
framerate=40
resolution=1366x768 | gst-videotestsrc=nothing
resolution=1366x768 | gst-videotestsrc=wipeout
resolution=1920x1080 | gst-videotestsrc=nothing
resolution=1920x1080 | gst-videotestsrc=wipeout
Guest CPU
Guest CPU = 0.35 * Guest Framerate + 0.51 | r=0.988, p=0.002, stdev=0.032 | resolution=1366x768 | gst-videotestsrc=nothing
Guest CPU = 0.22 * Guest Framerate + 1.89 | r=0.928, p=0.023, stdev=0.051 | resolution=1366x768 | gst-videotestsrc=wipeout
Guest CPU = 0.60 * Guest Framerate + -0.20 | r=0.981, p=0.003, stdev=0.069 | resolution=1920x1080 | gst-videotestsrc=nothing
Guest CPU = 0.65 * Guest Framerate + -3.68 | r=0.998, p=0.000, stdev=0.026 | resolution=1920x1080 | gst-videotestsrc=wipeout
Guest Framerate


Parameters:
experiment:
[all]
codec:
gst.vp8.vaapivp8enc
display:
[all]
record-time:
40s
resolution:
[all]
framerate:
[all]
bitrate:
[all]
keyframe-period:
[all]
rate-control:
cbr
stats:
PlotModel: Guest CPU Usage
Configuration:
Config settings
Permalink
Download
Guest CPU Usage
Guest CPU (in %)
x: 1.735053
y: 46.93878
z: 17.10267
display=webgl-wipeout | CPU = resolution x framerate x 0.21
Framerat

CHAPTER

EIGHT

REFERENCES

- Code repository
- SPICE Adaptive Streaming presentation/demo recording (full video / focused video)
- SPICE Adaptive Streaming presentation (slides)
- Specfem3D plugin (video)

CHAPTER

NINE

OVERVIEW