FOSDEM 2020
Tracking Performance of a Big Application from Dev to Ops

Classification: TLP: green

Philippe WAROQUIERS
NM/TEC/DAD/TD/Neos
Objectives of Performance Tracking?

- Evaluate/measure resources needed by new functionalities
  - To verify the estimated resource budget (CPU, memory)
  - To ensure the new release will cope with the current or expected new load

- Avoid performance degradation during development e.g.
  - Team of 20 developers working 6 months on a new release
  - A developer integrates X changes per month
    - If one change on X degrades the performance by 1%:
      - Optimistic: new release is 2.2 times slower : 100% + (6 months * 20 persons * 1%)
      - Pessimistic: new release is 3.3 times slower : 100% * 1.01 ^ (6 * 20)

- => do not wait the end of the release to check performance
- => daily track the performance during development

Development Performance Tracking Objective: Reliably Detect Performance Difference of <1%
Eurocontrol

- European Organisation for the Safety of Air Navigation
  - International organisation with 41 member states
  - Several sites/directorates/…
  - Activities: operations, concept development, European-wide project implementation, …
  - More info: www.eurocontrol.int

- Directorate Network Management
  - Develop and operate the Air Traffic Management network
  - Operation phases: strategical, pre-tactical, tactical, post-operation
  - Airspace/route data, Flight Plan Processing, Flow/Capacity Management, …

- NM has 2 core mission/safety critical systems:
  - IFPS: flight plan processing
  - ETFMS: Flow and Capacity Management
IFPS and ETFMS

- Big applications: IFPS+ETFMS is 2.3 million lines of Ada code
- ETFMS Peak day:
  - > 37,000 flights
  - > 11.6 million radar position, planned to increase to 18 millions Q1 2021
  - > 3.3 million queries/day
  - > 3.5 million messages published (e.g. via AMQP, AFTN, ...)
- ETFMS hardware:
  - On-line processing done on a linux server, 28 cores
  - Some workstations running a GUI also do some batch/background jobs
- Many heavy queries, complex algorithms, called a lot, e.g.
  - Count/flight list e.g. “flights traversing France between 10:00 and 20:00”
  - Lateral route prediction or route proposal/optimisation
  - Vertical trajectory calculation
- …
Horizontal Trajectory
Vertical Trajectory
Performance needs and ETFMS scalability

- **Horizontal scalability: OPS configuration**
  - 10 high priority server processes handle the critical input (e.g. flight plan, radar position, external user queries, ...)
  - 9 lower priority server processes (each 4 threads) handle lower priority queries e.g. “find a better route for flight AFR123”
  - Up to 20 processes running on workstations, executing batch jobs or background queries e.g. “every hour, search a better route for all flights of aircraft operator BAW departing in the next 3 hours”

- **Vertical scalability, needed e.g. for “simulation”:**
  - Simulate/evaluate heavy actions on the whole of European data such as: “close an airspace/country and spread/reroute/delay the traffic”
  - Starting a simulation implies e.g. to
    - clone the whole traffic from the server to the workstation
    - re-create in-memory indexes (~20,000,000 entries)
  - Time to start a simulation: < 4 seconds (multi-threaded)
    - 1 task decodes the flight data from the server, 1 task creates the flight data structure, 6 tasks are re-creating the indexes
Track Performance during Dev: “Performance Unit Tests”

- "Performance unit tests": useful to measure e.g.
  - Basic data structures: hash tables, binary trees, ...
  - Low level primitives: pthread mutex, Ada protected objects, ...
  - Low level libraries performance e.g. malloc library
  - Performance Unit tests are usually small/fast
    - and reproducible/precise (remember our 1% objective)

<table>
<thead>
<tr>
<th>Function</th>
<th>Time (NSEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT 16384:</td>
<td>90.85 NSEC</td>
</tr>
<tr>
<td>INSERT 32768:</td>
<td>99.82 NSEC</td>
</tr>
<tr>
<td>INSERT 65536:</td>
<td>107.54 NSEC</td>
</tr>
<tr>
<td>INSERT 131072:</td>
<td>115.58 NSEC</td>
</tr>
<tr>
<td>INSERT 262144:</td>
<td>122.38 NSEC</td>
</tr>
<tr>
<td>INSERT 524288:</td>
<td>130.50 NSEC</td>
</tr>
<tr>
<td>INSERT 1048576:</td>
<td>140.84 NSEC</td>
</tr>
<tr>
<td>INSERT 2097152:</td>
<td>145.91 NSEC</td>
</tr>
</tbody>
</table>
Pitfalls of “Performance Unit Tests” 
A real life example with malloc

- Malloc Performance Unit Test: glibc malloc <> tcmalloc <> jemalloc
  - 7 years ago: switched from glibc to tcmalloc : less fragmentation, faster
  - But parallelised ‘start simulation’ had not understandable 25% perf variation
    - Performance was varying depending on linking a little bit more (or less) not called code in the executable.
    - Analysis with ‘valgrind/callgrind’ : no difference. Analysis with ‘perf’: shows tcmalloc slow path called a lot more
  - => malloc perf unit test: N tasks doing M million malloc, then M million free
    - glibc was slower but consistent performance
    - jemalloc was significantly faster than tcmalloc
    - But the ‘real start simul’ was slower with jemalloc
- => more work needed on the unit test
Pitfalls of "Performance Unit Tests"
A real life example with malloc

- After improving unit test to better reflect ‘start simulation’ work:
  - tcmalloc was slower with many threads but became faster when doing L loops of ‘start/stop simulation’
  - With jemalloc, doing the M millions free in the main task was slower
  - Unit test does not yet evaluate fragmentation

- Based on the above, we obtained a clear conclusion about malloc:
  - We cannot conclude from the malloc “Performance Unit Test“
  - => currently keeping tcmalloc, re-evaluate with newer glibc in RHEL 8
Pitfalls of Performance “Unit Tests”

- Difficult to have a Performance unit test representative of the real load
  - Malloc: no conclusion
  - pthread_mutex timing: measure with or without contention?
    - And is the real load causing a lot of contention?
  - Hash tables, binary trees, ...:
    - Real load behavior depends on the key types/hash functions/compare functions/distribution of key values/...
- If difficult for low level algorithms, what about complex algorithms:
  - E.g. have a representative ‘trajectory calculation performance unit test’?
    - With which data (nr of airports, routes, airspaces, ...)?
    - With what flights (short haul ? long haul) flying where?
- Performance unit tests are (somewhat) useful but largely insufficient
- => Solution: measure(track) performance with the full system and real data: ‘Replay one day of Operational Data’
Replay Operational Data

- The operational system records all the external input:
  - Messages modifying the state of the system, e.g. flight plans, radar positions, ...
  - Query messages, e.g. “Flight list entering France between 10:00 and 12:00”

- ETFMS Replay tool can replay the input data
  - New release must be able to replay (somewhat recent) old input format

- Some difficulties:
  - Several days of input are needed to replay one day
    - E.g. because a flight plan for the D day can be filed some days in advance
  - Elapsed time needed to replay several days of operational data?
  - Hardware needed to replay the full operational data?
  - How to have a (sufficiently) deterministic replay in a multi-process system?
    - (to detect difference of <1%)
Replay Operational Data
Volume of Data to Replay

- Replaying the full operational input is too heavy
- => Compromise:
  - Replay the full data that changes the state of the system
    - Flight plans, radar positions, ...
  - Replay only a part of the query load:
    - Replay only one hour of the query load
      - And only a subset of the background/batch jobs

- Replaying in real time mode is too slow
  - But an input must be replayed at the time it was received on ops!
  - Many actions happen on timer events
  - => “accelerated fast time replay mode”:
    - The replay tool controls the clock value
    - Clock value “jumps” over the time periods with no input/no event
- Fast time mode: replaying one day takes about 13 hours on a (fast) linux workstation
Replay Operational Data
Sources of non Deterministic Results

- Network, NFS, ....
  - Replay on isolated workstations: local file system, local database, ...
- System Administrators
  - Are open to discussions to disable their jobs on replay workstations
- Security Officers
  - Are (somewhat) open to (difficult) discussions to disable security scans ;)
- Input/Output past history
  - Removing files and clearing the database was not good enough
  - => completely recreate the file system and database for each replay
- Operating System usage history
  - => Reboot the workstation before each replay
Replay Operational Data
Remaining Sources of non Deterministic Results

- Time-control replay tool serialises “most” of input processing
  - “most” but not all: serialising everything slows down the replay
    - E.g. radar positions at the same second are replayed “in parallel”
- Replays are done on identical workstations
  - Same hw, same operating system, ...
  - Still observing systematic small performance difference between workstations

- We finally achieved a reasonably deterministic replay performance, with 3 levels of results:
  - Global tracking: elapsed/user/system cpu for complete system
  - Per process tracking: user/system cpu, “perf stat” results, ...
  - Detailed tracking: we run one hour of replay under valgrind/callgrind
    - This is very slow (26 hours) but very precise
Replay Operational Data
Global Tracking
Replay Operational Data
Per Process Tracking

- User and system cpu
- heap status: used/free, tcmalloc details, ...
- ...

```
43836.82s real  5762.10s user  895.71s system  counterp1_out_01
43836.83s real  7744.71s user 1280.37s system  flight1_out_01
43836.60s real  7716.72s user 1278.94s system  flight2_out_01
43836.78s real  7695.46s user 1263.85s system  flight3_out_01
43836.82s real  7762.09s user 1281.87s system  flight4_out_01
...

  process_name       C lib(used) | C lib(free)
counterp_process-28348-27-04  4026449248 | 59458208
flight_process-28400-27-04    650265680  | 153271216
flight_process-28428-27-04    648387288  | 160982312
flight_process-28429-27-04    643411488  | 211112416
flight_process-28432-27-04    643439200  | 210757024
...
```
Replay Operational Data
Detailed Tracking with valgrind/callgrind/kcachegrind
Dev Performance Tracking: Detection of a real life missed failed optimisation

Optimisation idea: decrease the number of rendez-vous by using lower level synchronisation based on **Volatile**

```plaintext
task body Monitor is
  Locks : Natural := 0;
  select
    accept Unlock;
    Locks := Locks - 1;
  or
    accept Get_Lock_Count (Lock_Count : out Natural) do
      Lock_Count := Locks;
      end Get_Lock_Count;
  or
    ...

Locks : Natural := 0 with Volatile;
function Get_Lock_Count return Natural is (Locks);
  task body Monitor is
    select
      accept Unlock do
        Locks := Locks - 1;
      end Unlock;
    or
      ...

Performance tracking detected this was a pessimisation: the compiler optimises the ‘no body’ rendez-vous, and the nr of **Unlock** calls is significantly bigger than the nr of **Get_Lock_Count** calls.

This should be faster: we will have the same number of **Unlock** rendez-vous but we will have much faster **Get_Lock_Count** calls.
```
Dev Performance Tracking: A Summary

- We have a good dev performance tracking, using a mix of:
  - Performance Unit Tests
  - Replay Operational Data in a as deterministic as possible setup
    - The replayed day is changed ~every year to match new usage patterns
  - Various tools: valgrind/callgrind + kcachegrind, perf, top, ...
  - Beware of blind spots of your tools e.g.
    - Valgrind/callgrind + kcachegrind is very easy to use but
      - very slow and serialises multi-thread applications
      - Limited system call measurement can be misleading
  - Have global indicators, zoom on the details when needed

- Some improvements to the tooling done or in the pipe-line:
  - callgrind next release can now measure system call CPU
  - working on developing “callgrind_diff” to help visualising differences
Dev Performance Tracking: Good Enough/Sufficient to Go Operational?

- What about: you are on-call, waken up Saturday 4:00 AM because “users are complaining that the system is slow”
  - You need something else than: “I will replay the day and get back to you Monday morning”
- What about: is the reference replayed day representative of what happens on OPS?
- What about: evolution of the OPS workload and capacity planning
  - E.g. what functionalities/queries/… are increasing?
  - E.g. what additional capacity is needed to support X times more queries of that type?

- Solution: “permanently activated response time monitoring and statistics”
On-line “TACT Response Time” Monitoring

- Application contains measurement code at “critical points” such as:
  - Remote Procedure Call invocation begin/end (i.e. “client side”)
  - Remote Procedure Call execution begin/end (i.e. “server side”)
  - Database access begin/end
  - Significant algorithms begin/end, such as: “calculate a vertical trajectory”
  - ...

- Measurements typically nested, e.g. inside a RPC execution begin/end

- The “TACT response time” package maintains:
  - A circular buffer with the last M measurements
  - For each begin/end measurement:
    - Elapsed time, Thread CPU time, optionally full Process CPU time
  - Statistics:
    - How many measurements
    - Histogram of Elapsed/Thread CPU
    - Details about the N worst cases

- Reasonable overhead ~1.7% CPU => always activated
### TACT Response Time

Last M Measurements Circular Buffer

<table>
<thead>
<tr>
<th>Info</th>
<th>main_task Acc</th>
<th>main_task Ent</th>
<th>driver_task Acc</th>
<th>driver_task Ent</th>
<th>env_data_monitor Acc</th>
<th>env_data_monitor Ent</th>
</tr>
</thead>
</table>

- **Display Tools**

- **Performance of PROF 1**

- **Last Updated: 28-18:28**

- **TACT Response Time**

- **Last M Measurements Circular Buffer**

- **Network Manager**

  - **nominated by the European Commission**

  - **Eurocontrol**

  - **Page 23**
### TACT Response Time: Statistics

#### Performance of PROF 1

- **Path Finder NORMAL_GRAPH_CREATION**
- **SINCE_CREATION**
- **NR_SAMPLES**: 36566
- **TOTAL**: ELAPSED_REAL: 1548.67, THREAD_CPU: 1520.62 SEC
- **AVERAGE**: ELAPSED_REAL: 0.04, THREAD_CPU: 0.04 SEC

#### TIME RANGE (in SEC) ELAPSED_REAL THREAD_CPU

| <= 0.00 | 19741 | 19915 |
| <= 0.01 | 5178  | 5159  |
| <= 0.02 | 2489  | 2466  |
| <= 0.04 | 2369  | 2332  |
| <= 0.08 | 1832  | 1700  |
| <= 0.16 | 2177  | 2191  |
| <= 0.32 | 1978  | 1959  |
| <= 0.64 | 718   | 652   |
| 1.28   | 84    | 72    |

#### ELAPSED_REAL Maximum 10 1.10 SEC 27-12:45:07.71 (Path Finder NORMAL_GRAPH_CREATION [960866])
- Path Finder NORMAL_GRAPH_CREATION [997547]
- Path Finder NORMAL_GRAPH_CREATION [964446]
- Path Finder NORMAL_GRAPH_CREATION [985277]
- Path Finder NORMAL_GRAPH_CREATION [10614]
- Path Finder NORMAL_GRAPH_CREATION [985307]
- Path Finder NORMAL_GRAPH_CREATION [976715]
- Path Finder NORMAL_GRAPH_CREATION [10982]
- Path Finder NORMAL_GRAPH_CREATION [998811]
- Path Finder NORMAL_GRAPH_CREATION [16617]
- Path Finder NORMAL_GRAPH_CREATION [965277]
- Path Finder NORMAL_GRAPH_CREATION [975847]
- Path Finder NORMAL_GRAPH_CREATION [960866]
- Path Finder NORMAL_GRAPH_CREATION [976715]
- Path Finder NORMAL_GRAPH_CREATION [988811]
- Path Finder NORMAL_GRAPH_CREATION [985307]
- Path Finder NORMAL_GRAPH_CREATION [16617]
- Path Finder NORMAL_GRAPH_CREATION [10982]
- Path Finder NORMAL_GRAPH_CREATION [998811]
- Path Finder NORMAL_GRAPH_CREATION [985307]
- Path Finder NORMAL_GRAPH_CREATION [16617]
- Path Finder NORMAL_GRAPH_CREATION [10982]
- Path Finder NORMAL_GRAPH_CREATION [998811]
- Path Finder NORMAL_GRAPH_CREATION [985307]
- Path Finder NORMAL_GRAPH_CREATION [16617]
- Path Finder NORMAL_GRAPH_CREATION [10982]
- Path Finder NORMAL_GRAPH_CREATION [998811]
TACT Response Time
Used from Dev to Ops

- **Dev**
  - Helps to understand how the system works, e.g. to see messages exchanged between processes, algorithms executed, ...
  - Statistics used to analyse Performance Operational Data Replay
  - Compare the profile of the “replayed reference day” with OPS profile
  - Measure resource consumption for new functionalities
  - …

- **Ops**
  - On-line investigation of performance problems
  - Bug investigation:
    - Policy: exceptions are used for bugs, not for normal behaviour
    - In case of exception: take a core dump, drop input, process next message
    - => the core dump contains in memory the details of the last M measured actions
  - Post-ops analysis, trend analysis
  - Input for capacity planning
Performance Tracking of a Big Application

Summary

- (Reasonably) deterministic performance tracking during development:
  - Allows to detect performance regression on a daily basis
  - Allows to verify that optimisations really have the desired effect
  - Allows to plan capacity for demand growth and new functionalities
  - ...

- A mix of various techniques and tools are needed, e.g.
  - Performance unit test
  - Replay real data
  - Application self-measurement (“TACT response time”).
  - Avoid blind spots by using various tools: perf, valgrind/callgrind, ...

- Tooling can be used for various purposes e.g. Replay Tool:
  - Is also the (automated) testing tool
  - Is used by our users to analyse/optimise operational actions/procedures

- Performance tracking and statistics also on the operational system
Questions?