# Performance comparison between cumbia flavors

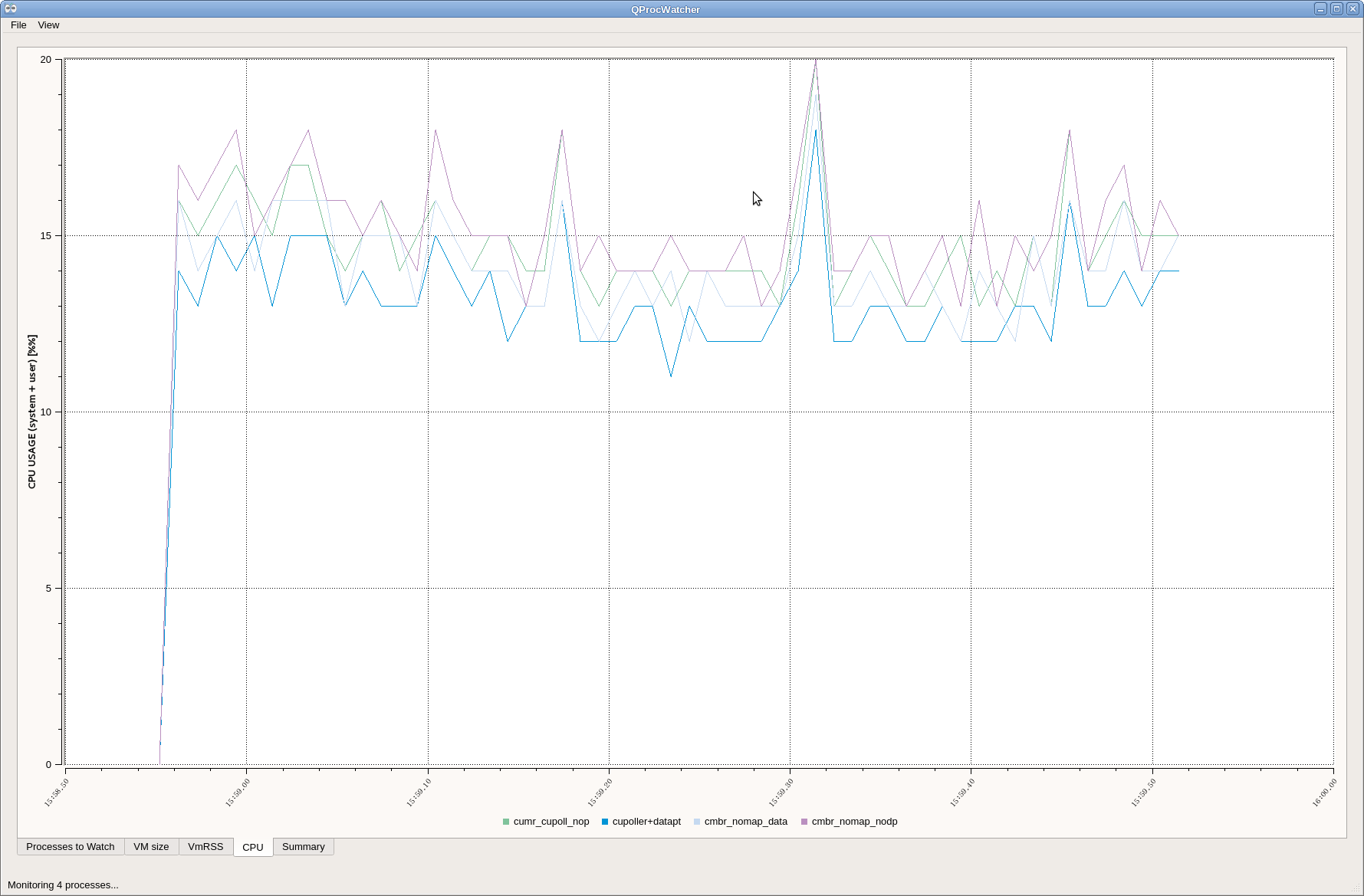


Figure 1.

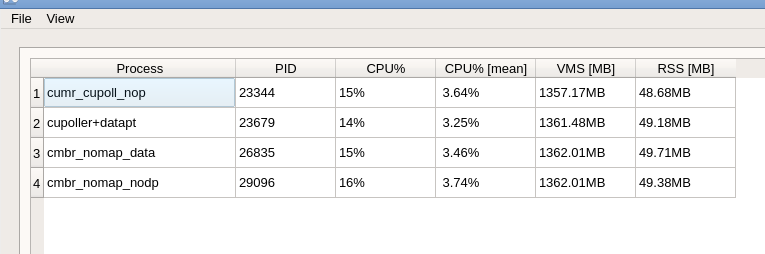


Figure 2.

Four branches have been created for the test:

1. cupoller+nodataptr
2. cupoller+dataptr
3. cupoller+nomap+nodataptr
4. cupoller+nomap+dataptr

All make use of *cupoller*, a centralised poller that groups attributes and commands per device *and* per period. This means one timer only per device and period. When the timer elapses, read\_attributes is used and command\_inout is called on the list of attributes and commands grouped per device and timer interval, as mentioned above.

## Command line used for the test

Example from branch cupoller+nomap+nodataptr:

*#!/bin/bash*

*LD\_LIBRARY\_PATH=/usr/local/cupoller+nomap+nodataptr:$LD\_LIBRARY\_PATH ./bin/cmbr\_nomap\_nodptr "inj/diagnostics/rtbpm\_inj.01/VaPickup" \*

*"inj/diagnostics/rtbpm\_inj.01/VbPickup" \*

*"inj/diagnostics/rtbpm\_inj.01->GetHorPos(0,1000)" \*

*"inj/diagnostics/rtbpm\_inj.01->GetVerPos(0,1000)" \*

*"inj/diagnostics/rtbpm\_inj.01/VcPickup" \*

*"inj/diagnostics/rtbpm\_inj.01/VdPickup" \*

*l00/diagnostics/rtbpm\_l00.01/VaPickup \*

*l00/diagnostics/rtbpm\_l00.01/VbPickup \*

*"l00/diagnostics/rtbpm\_l00.01->GetHorPos(0,1000)" \*

*"l00/diagnostics/rtbpm\_l00.01->GetVerPos(0,1000)" \*

*l00/diagnostics/rtbpm\_l00.01/VcPickup \*

*l00/diagnostics/rtbpm\_l00.01/VdPickup \*

*l00/diagnostics/rtbpm\_l00.01/VaPickup \*

*l00/diagnostics/rtbpm\_l00.01/VbPickup \*

*l00/diagnostics/rtbpm\_l00.01/VcPickup \*

*l00/diagnostics/rtbpm\_l00.01/VdPickup \*

*"l00/diagnostics/rtbpm\_l00.01->GetHorPos(0,1000)" \*

*"l00/diagnostics/rtbpm\_l00.01->GetVerPos(0,1000)" \*

*"l00/diagnostics/rtbpm\_l00.01->GetSum(0,1000)" \*

*l01/diagnostics/rtbpm\_l01.01/VaPickup \*

*l01/diagnostics/rtbpm\_l01.01/VbPickup \*

*l01/diagnostics/rtbpm\_l01.01/VcPickup \*

*l01/diagnostics/rtbpm\_l01.01/VdPickup \*

*"l01/diagnostics/rtbpm\_l01.01->GetHorPos(0,1000)" \*

*"l01/diagnostics/rtbpm\_l01.01->GetVerPos(0,1000)" \*

*"l01/diagnostics/rtbpm\_l01.01->GetSum(0,1000)" \*

*\*

*\*

*--truncate 10*

Remember to export TANGO\_HOST=srv-tango-srf:20000 before starting the script.

The performance is shown in the graph above.

The “*nodataptr*” versions use const std::vector<CuData>& to exchange data between the polling thread and the listeners in the main thread. This involves copying the vector of CuData **once**, when the data is passed from the background to the main thread.

The “*dataptr*” versions use std::vector<CuData> \* to exchange data. This **avoids copying** the vectors of data.

The graphs above are taken from a rough Qt application that reads CPU and memory usage as the *top* command would do.

### Considerations

From the Figure 1 and Figure 2 it looks like the *“dataptr”* versions perform *slightly* better than the *“nodataptr* counterparts, and that the *“nomap”* versions do not perform better than the respective “*array”* versions.

We repeat the measurements other two times, each time restarting the four versions of the *cumbiareader* command line program.

As you can see, the results are equivalent:

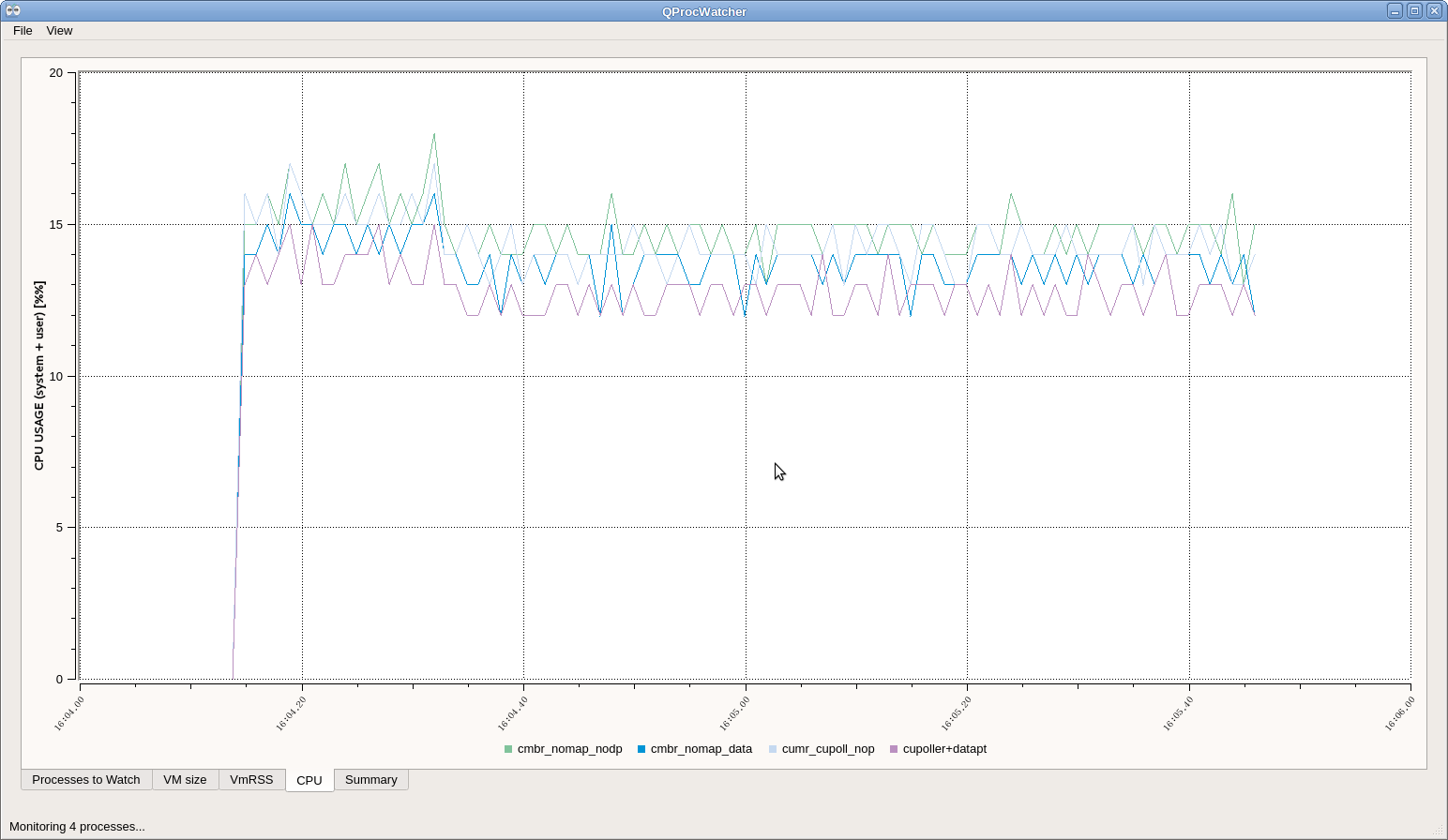


Figure 3. (repetition number 1)

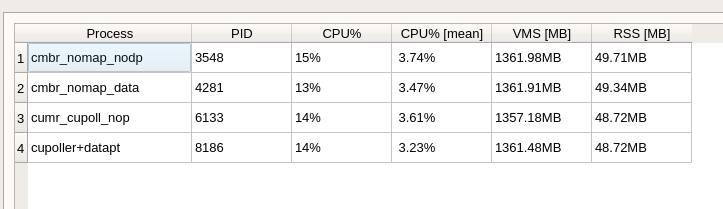


Figure 4. (repetition number 1)

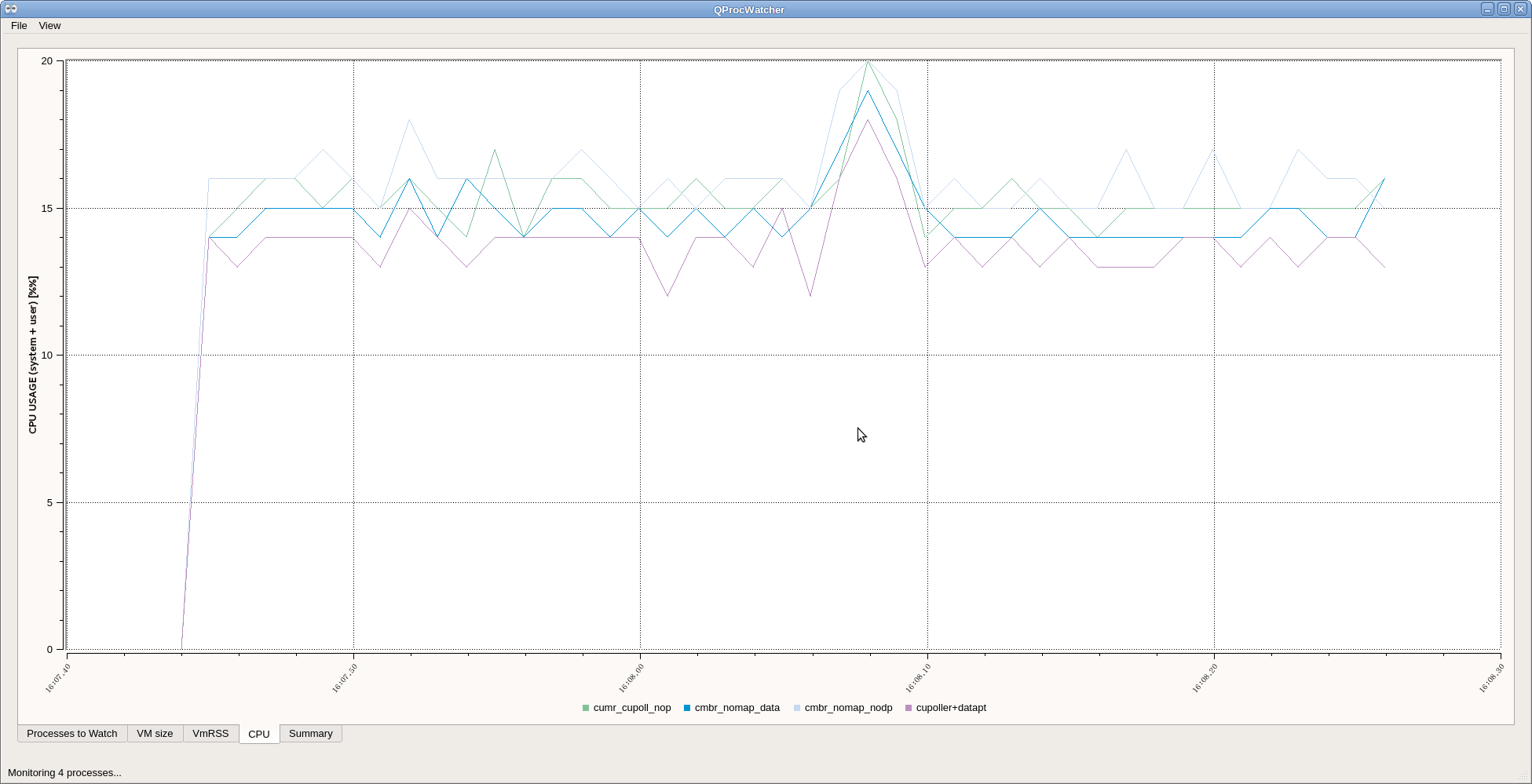


Figure 5 (repetition number 2)

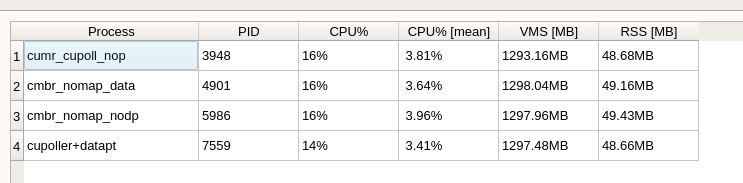


Figure 6 (repetition number 2)

In all the experiments, the plot shows that the *cupoller+dataptr* is the less CPU hungry version. As said, it avoids copying data from the background to the foreground thread (using pointers) and does not employ plain C *arrays* to optimize data access in CuData. This last observation is important because using std::map to access data in CuData lets us access it using strings rather than integers, thus decoupling *cumbia* from *cumbia-tango* and all other modules.

Let’s now compare the *“nomap+dataptr”* and the simple “*dataptr”* versions: the “nomap” avoids using std::map to pair keys and values into the CuData containers. It instead uses plain C arrays. We expect it to be faster.

## “nomap+dataptr” vs “dataptr”

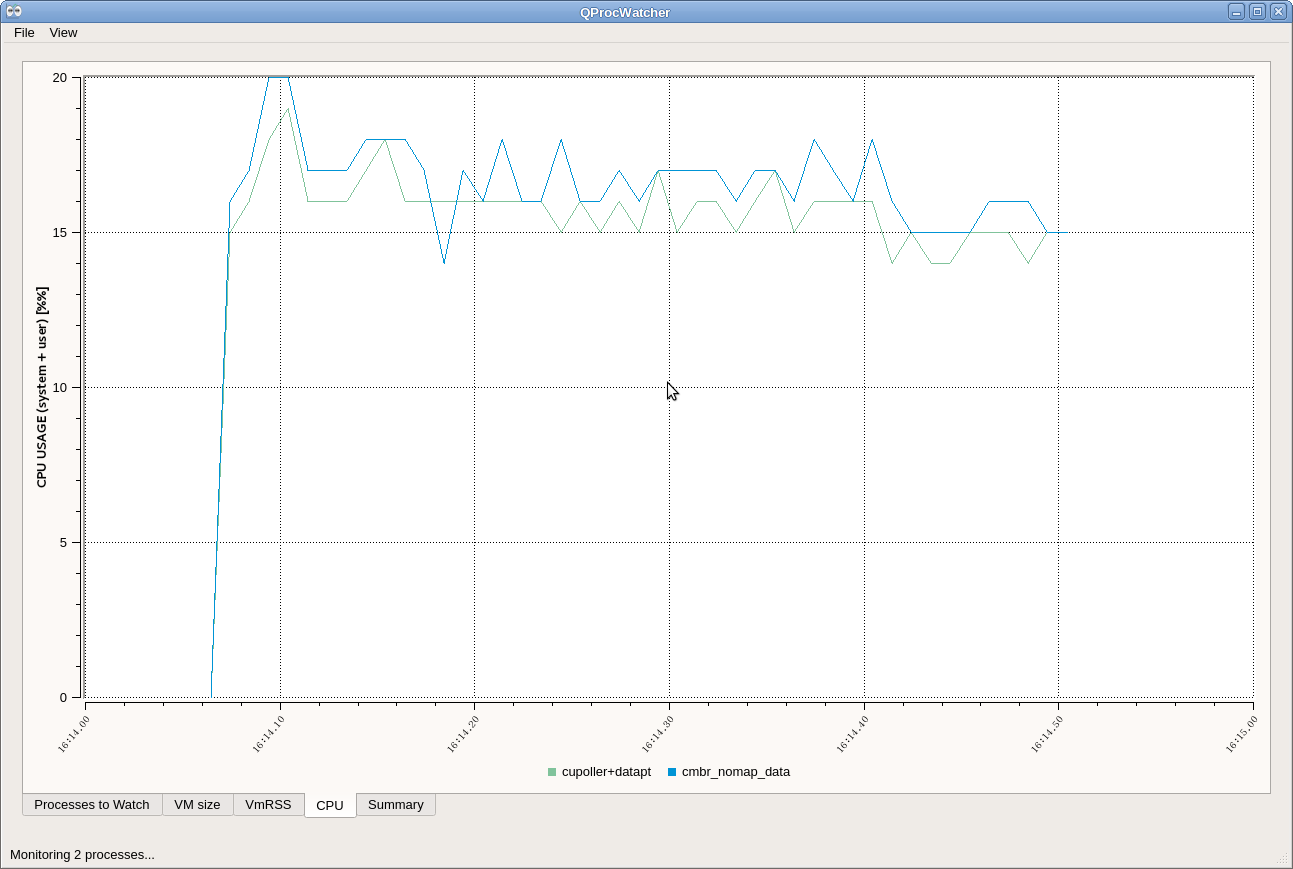


Figure 7.

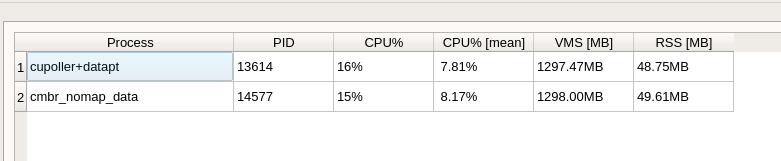


Figure 8.

Again, you can see that there is a *smaller CPU usage* by the application using std::map instead of plain C arrays.

## Competition against QTango

Last but not least, a comparison of the CPU usage with the equivalent command line application compiled with QTango:

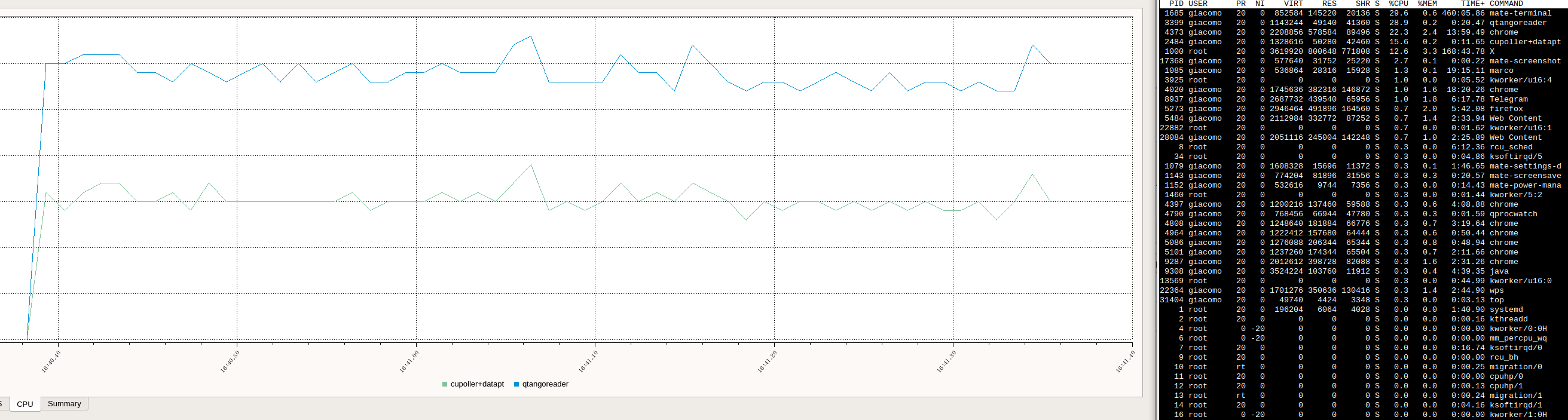


Figure 9. *Cumbia reader vs QTango reader*

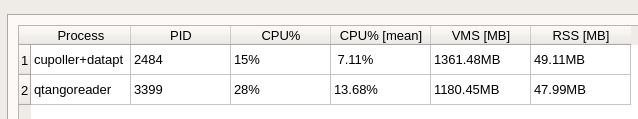


Figure 10. *Cumbia vs QTango*. CPU usage in QTango is almost two times cumbia’s.

You can see a drastic improvement brought by the *cumbia* library as far as CPU usage is concerned.