experiment

August 7, 2023

1 Test Inspektor Gadget scaling

The goal of this experiment is to test until which number of nodes Inspektor Gadget scales and to also collect its CPU usage.

1.1 Experimental environment

1.1.1 Hardware environment

Inspektor Gadget will be tested on an AKS cluster using `Standard_DS2_v2` as node Azure image (*i.e* 2 CPU cores and 7 GB of DRAM). The number of nodes will vary over the experience with the following values: 2, 12, 25, 37 and 50.

1.1.2 Software environment

AKS runs kubernetes 1.25 and relies on kernel 5.15.

To test scaling, the idea is to deploy a pod on each node which will generate a lot of `exec()` and to count the number of `exec()` events reported by Inspektor Gadget. So, on each node, a pod running `stress-ng` is deployed to generate as many `exec()` it can during 1 second on all the node CPU (in our case 2). Before that, Inspektor gadget is deployed on the cluster to monitor all these pods (which belong to the same namespace). The experiment is described in `script.sh`.

1.1.3 Performance statistics

For each number of node, the above experiment is run 30 times to compute some statistics.

1.1.4 Resource statistics

For each Inspektor Gadget pod, we will collect the following statistics:

1. The whole CPU usage in microseconds, before the experiment is run.
2. The whole CPU usage in microseconds, after the experiment is run.
3. The current memory footprint in bytes, after the experiment is run.

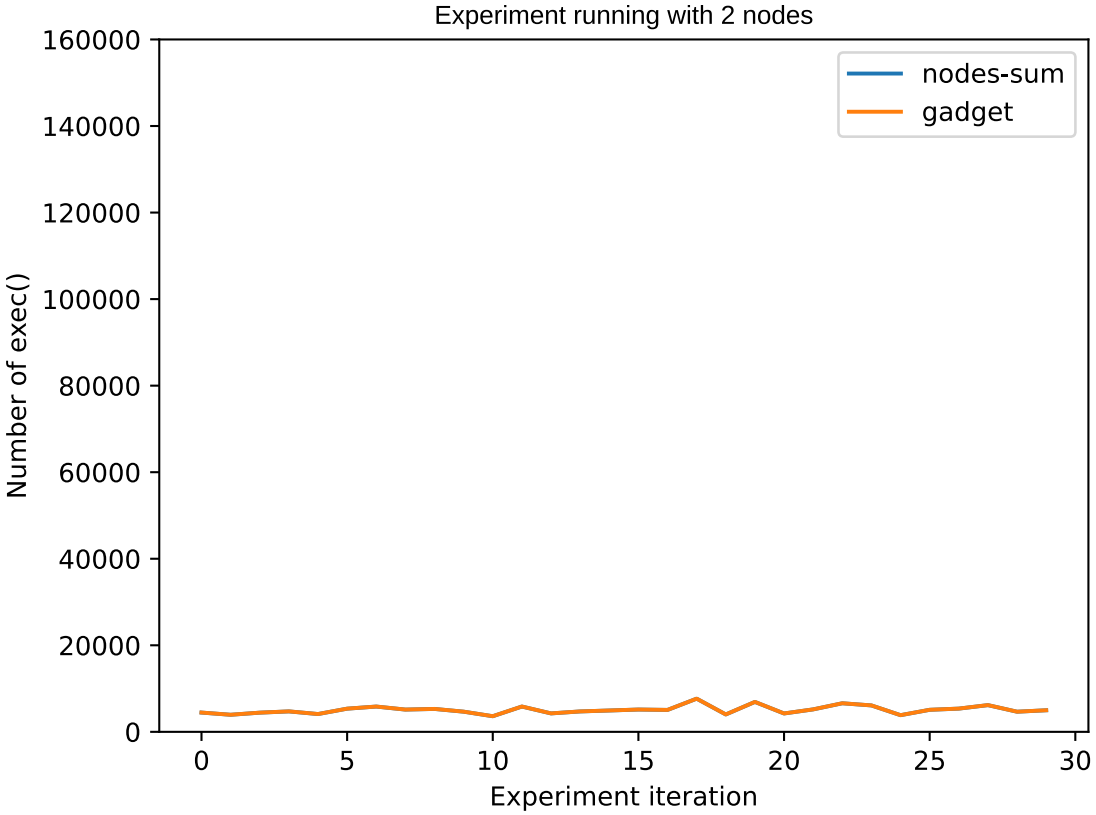
All these statistics are collected using `cgroupv2` `cpu.stat` and `memory.current` file. By substracting the CPU usage after to the CPU usage before, we would get the CPU usage for the experiment. Note that, this is not possible to get the peak memory footprint, as `memory.peak` is only available in kernel 5.19.

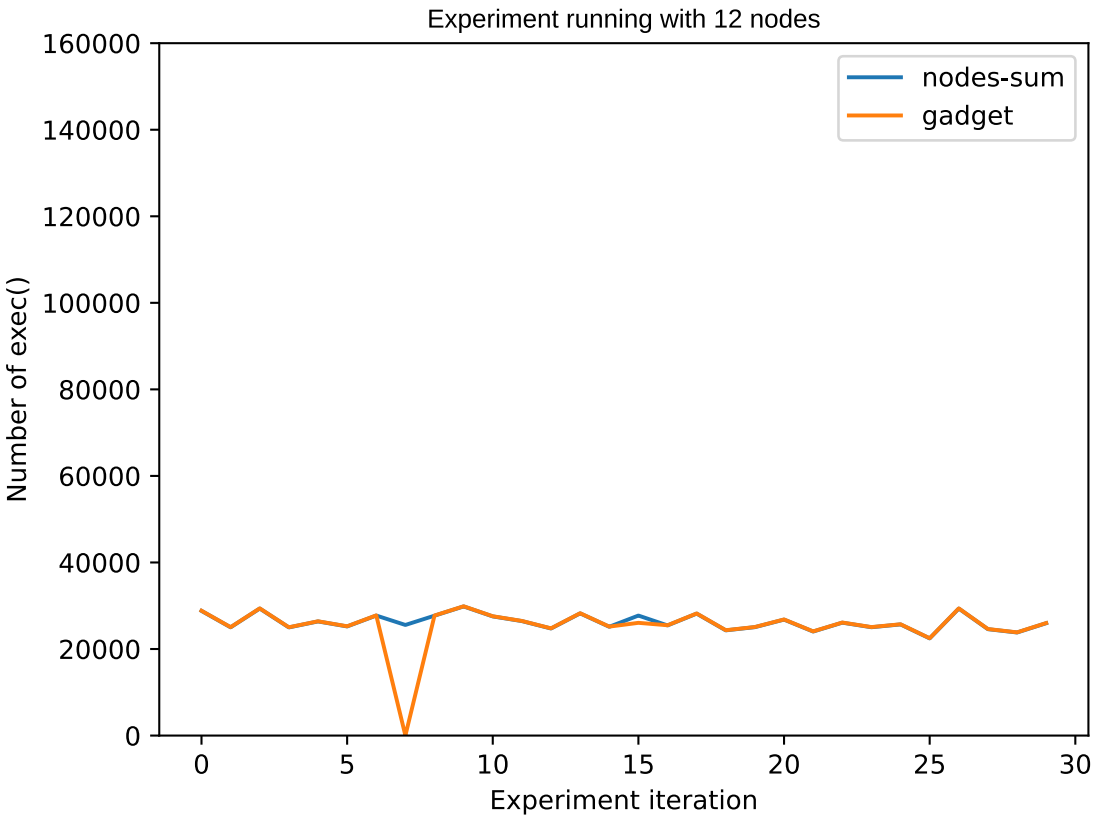
1.2 Results

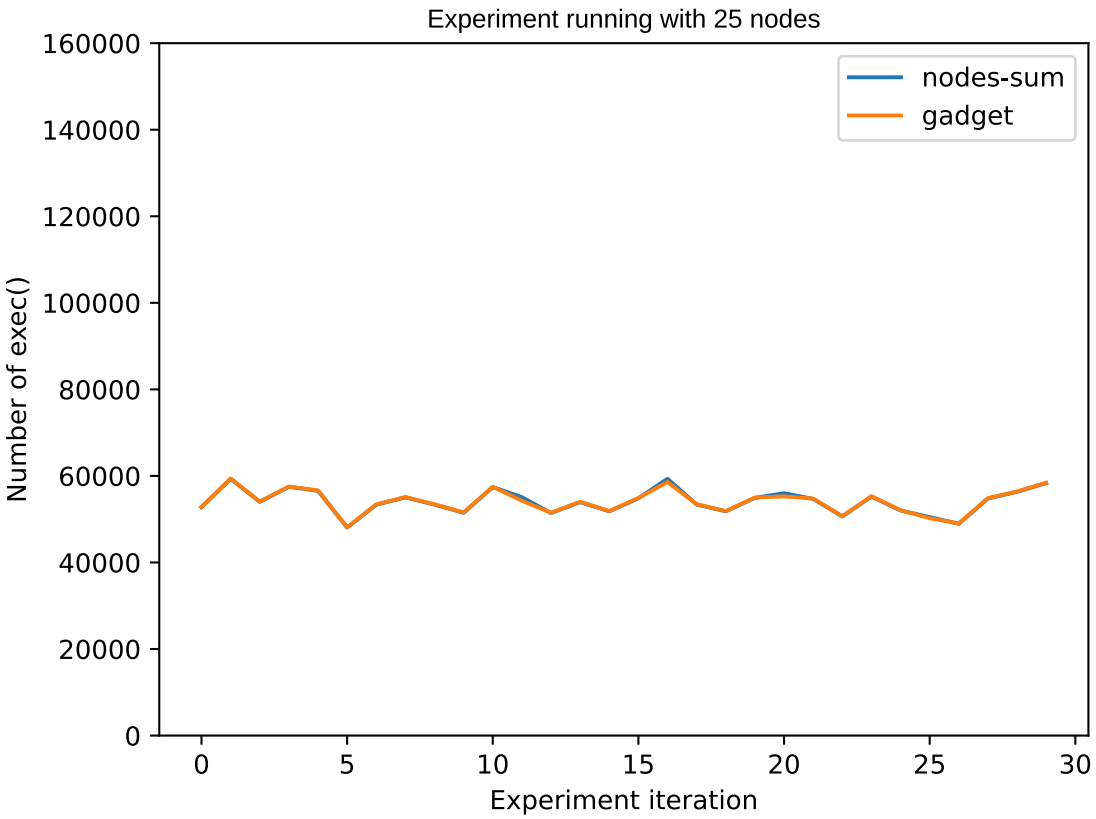
1.2.1 Scaling results

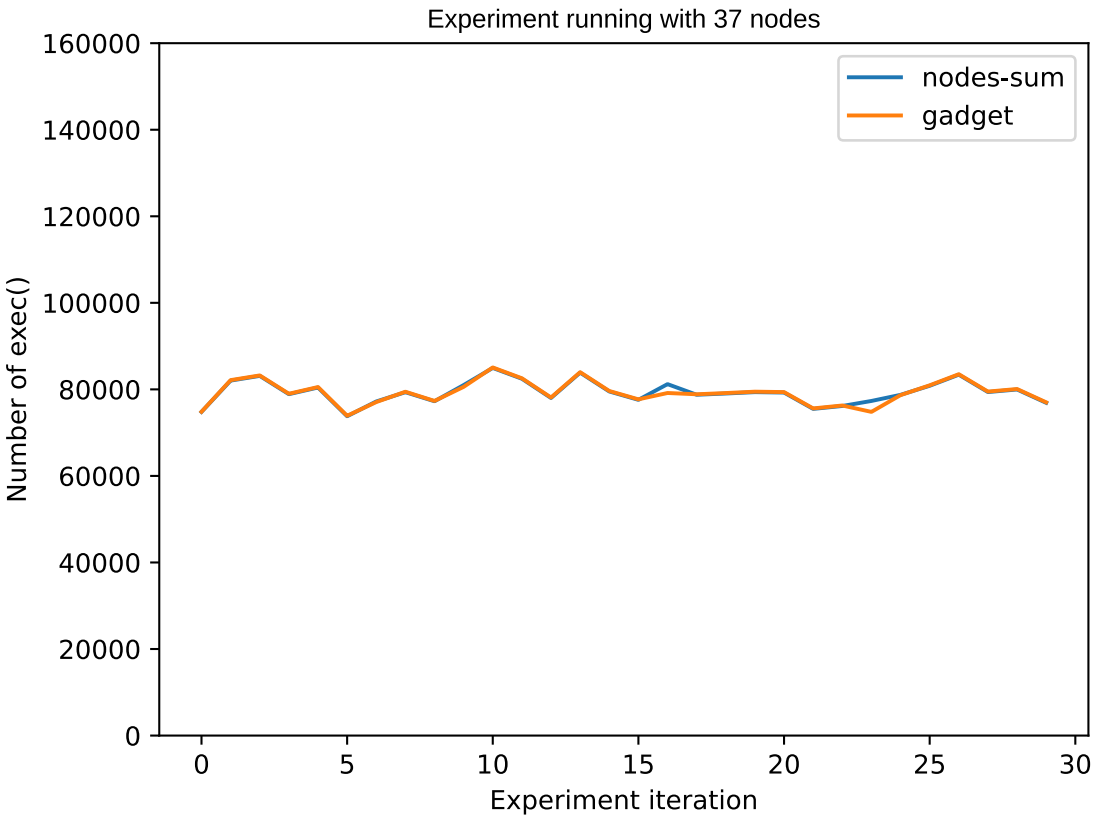
The results are depicted in the following graphs. The `x` axis indicates the experiment number from 1 to 30 while the `y` axis indicates the number of `exec()`. Each graph presents two curves:

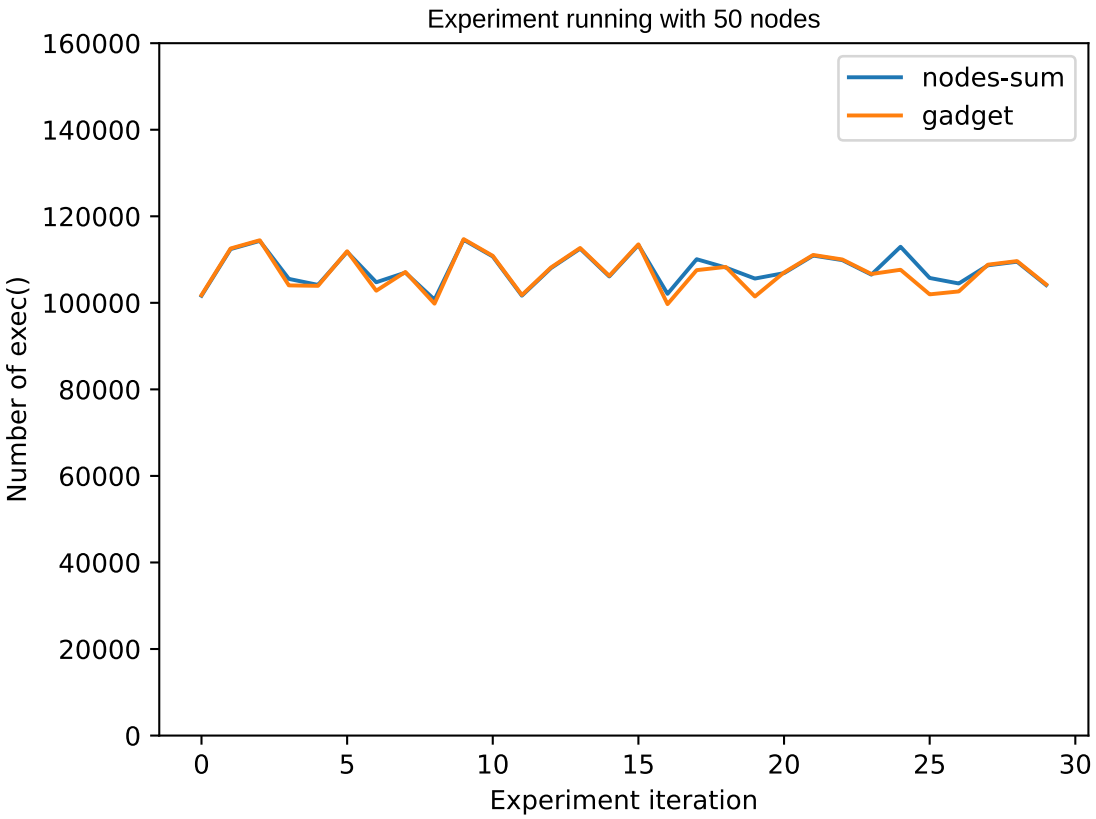
1. One which is the sum of all `exec()` generated for all nodes.
2. The other which is the number of `exec()` reported by Inspektor Gadget.




Experiment running with 12 nodes
nodes-sum
gadget
Number of exec()
Experiment iteration
160000
140000
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30


Experiment running with 25 nodes
nodes-sum
gadget
Number of exec()
160000
140000
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30
Experiment iteration


Experiment running with 37 nodes
nodes-sum
gadget
Number of exec()
Experiment iteration
160000
140000
120000
100000
80000
60000
40000
20000
0
0
5
10
15
20
25
30

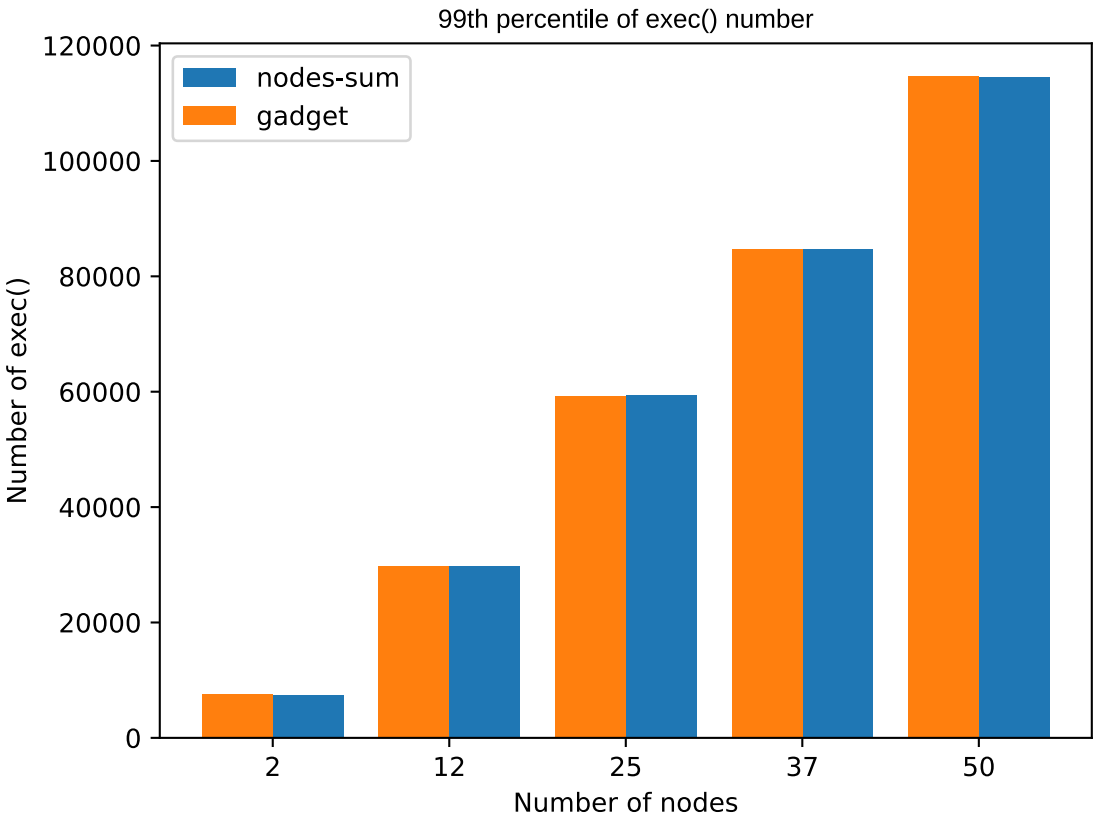


1.2.2 Interpretation

First of all, the curves are quite flat, so we can conclude there are not so much variation across the iterations. There are nonetheless one iteration with 12 nodes where Inspektor Gadget failed. We can also see there are some iterations where it collects less events than generated when run on 50 nodes but nothing serious.

The number of events reported by Inspektor Gadget follows the number of `exec()` generated.

Let's compute the 99th percentile for all number of nodes:



With the 99th percentile, we can clearly see that Inspektor Gadget scales.

Let's now take a look to the resource consumption of Inspektor Gadget.

1.2.3 Resources results

As the result across different nodes are quite stable, we will only compute statistics over resource consumption on the experiment with 50 nodes.

Let's compute the 99th percentile of CPU usage during the stress, the memory footprint after the stress for each node and also over all nodes:

node	CPU usage (s)	Memory footprint (MB)
node-1	1.29	100
node-2	1.32	95
node-3	1.36	96
node-4	1.33	96
node-5	1.28	96
node-6	1.29	96
node-7	1.35	96
node-8	1.26	97
node-9	1.36	97

node	CPU usage (s)	Memory footprint (MB)
node-10	1.30	96
node-11	1.32	97
node-12	1.25	98
node-13	1.36	98
node-14	1.32	96
node-15	1.37	96
node-16	1.28	99
node-17	1.36	99
node-18	1.33	98
node-19	1.27	98
node-20	1.35	98
node-21	1.29	102
node-22	1.33	96
node-23	1.55	111
node-24	1.40	98
node-25	1.37	100
node-26	1.36	104
node-27	1.44	96
node-28	1.34	100
node-29	1.34	101
node-30	1.43	100
node-31	1.43	99
node-32	1.49	98
node-33	1.38	100
node-34	1.43	100
node-35	1.39	99
node-36	1.36	101
node-37	1.36	100
node-38	1.39	93
node-39	1.43	98
node-40	1.40	100
node-41	1.46	100
node-42	1.43	94
node-43	1.40	109
node-44	1.34	98
node-45	1.36	100
node-46	1.46	95
node-47	1.46	103
node-48	1.39	97
node-49	1.38	101
node-50	1.41	98
all nodes	1.52	110

1.2.4 Interpretation

As we can see, the 99th percentile CPU usage on all nodes is 1.52 seconds. This number is higher than the duration of the experiment, which was one second. This is totally normal and possible, as there the node has 2 CPU, so the whole time would be 2 seconds.

The memory fooprint is 110 MB, but it was collected at the end of the experiment when no gadget were working.

2 Conclusion

To conclude, Inspektor Gadget scales until 50 nodes. Its CPU usage may be high but this should be more investigated and its peak memory footprint will be collected once AKS support kernel 5.19.