NUMA Page split

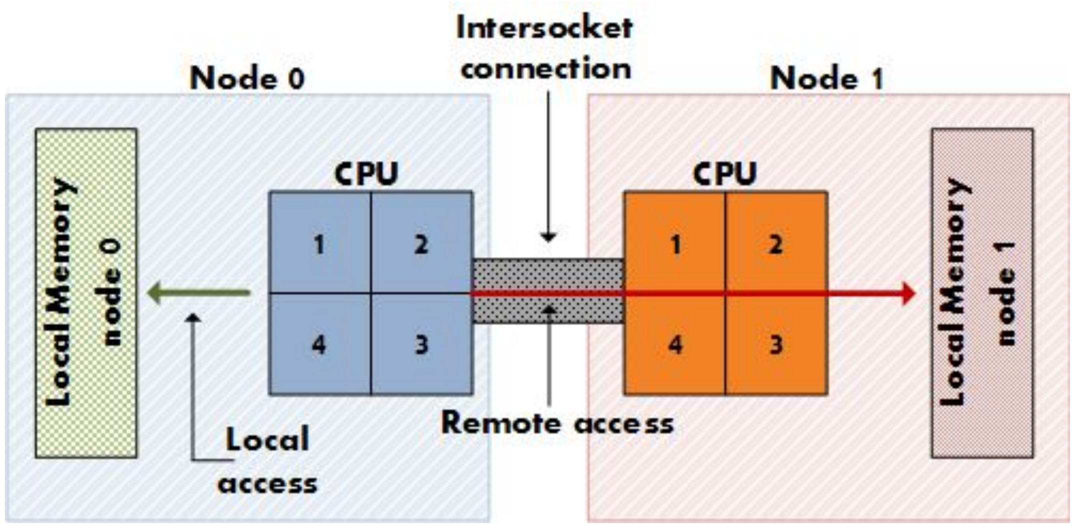
Improving Memory Management in the Linux Kernel for NUMA Architectures

Clément Gachod
Advisors: Julia Lawall, Jean-Pierre Lozi
Chair of Computer Systems
https://dse.in.tum.de/

15.01.2024 – 15.07.2024

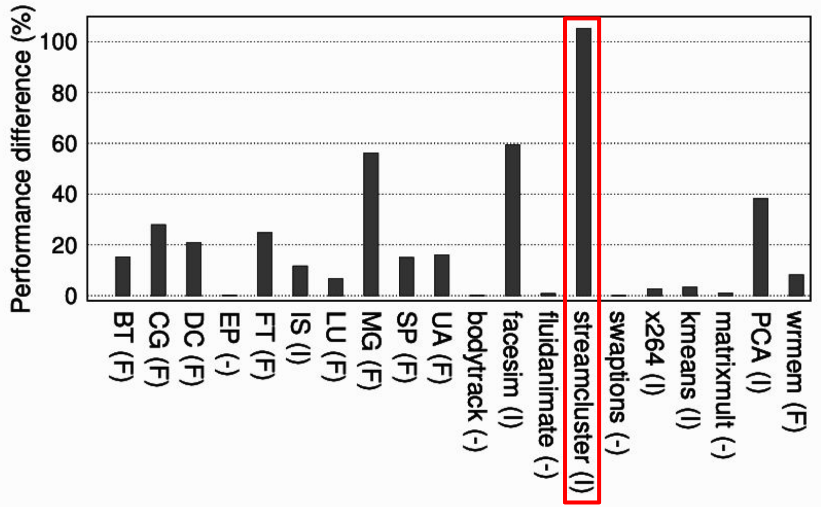
Non Uniform Memory Access (NUMA)

- Hardware resources are split into physically disjoint nodes
- Can be within a single chip or on a machine with several CPUs
- Essentially horizontal scaling for hardware



State-of-the-art

- Remote wire delay have limited performance impact, usually below 30%[1]
- Memory congestion on the other hand can cause severe slowdowns[1]



(b) Absolute perf. difference for multi-thread versions of applications between First-touch (F) and interleaving (I).

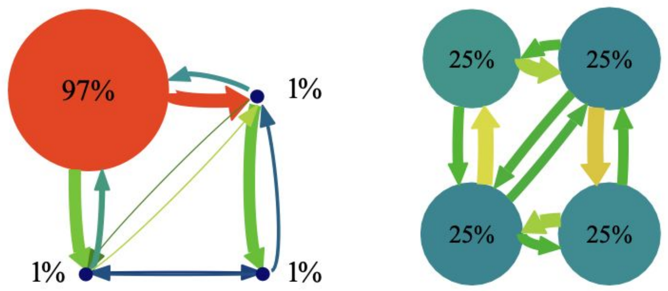


Figure 3. Traffic imbalance under first-touch (left) and interleaving (right) for *Streamcluster*. Nodes and links bearing the majority of the traffic are shown proportionately larger in size and in brighter colours. The percentage values show the fraction of memory requests destined for each node. The figure is drawn to scale.

[1] Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems - Mohammad Dashti et al. - 2013

State-of-the-art

- In an asymmetric hardware topology, task placement can have a substantial impact on performance[2]
- It can even be faster to go through a 3rd node rather than directly to the target node[2]

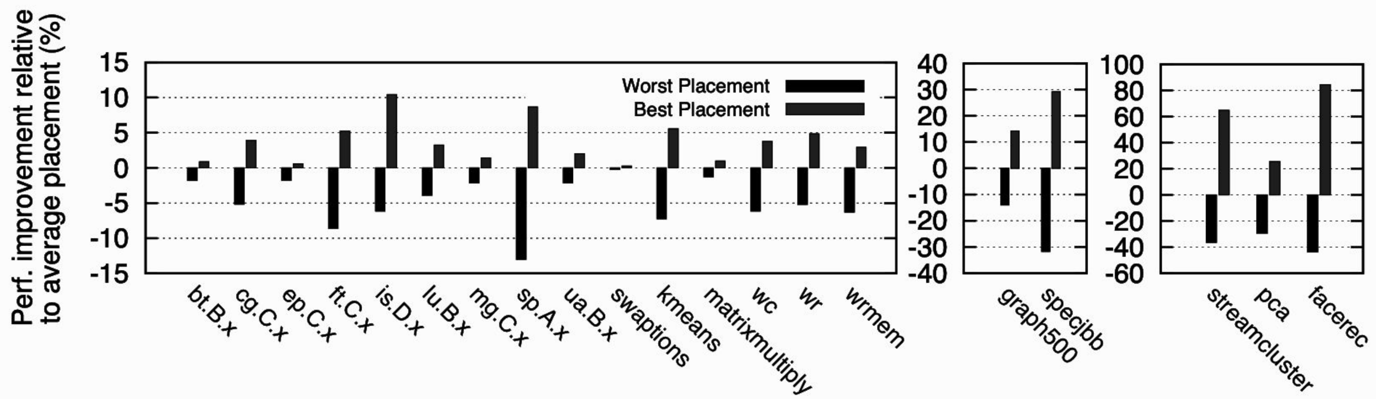
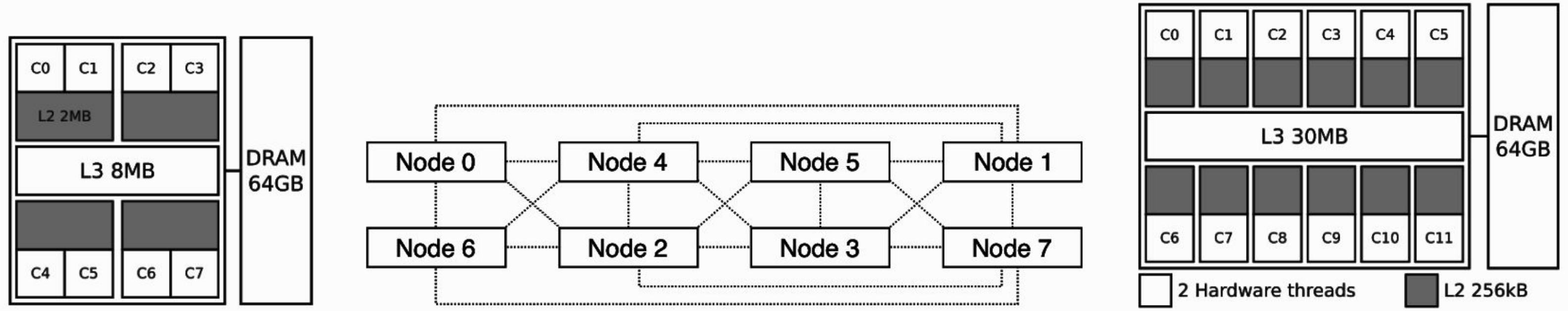


Figure 2: Performance difference between the best, and worst thread placement with respect to the average thread placement on Machine A. Applications run with 24 threads on three nodes. Graph500, specjbb, streamcluster, pca and facerec are highly affected by the choice of nodes and are shown separately with a different y-axis range.

[2] Thread and Memory Placement on NUMA Systems: Asymmetry Matters - Baptiste Lepers et al. - 2015

State-of-the-art

- More generally, optimal task placement can vary widely between architectures[3]
- Can be almost impossible to predict the best placement without prior measurements[3]



(a) AMD Opteron 6272 node (b) AMD interconnect (c) Intel Xeon E7-4830 v3 node

Figure 2: The two systems used in our study. The first is a quad AMD Opteron 6272. It has eight NUMA nodes (schematically shown in Figure 2a) connected with an asymmetric interconnect (Figure 2b) and a total of 64 cores. Pairs of cores share the instruction front-end, L2 cache, and floating point units. The second system is a quad Intel Xeon E7-4830 v3 with four NUMA nodes (Figure 2c) and 96 hardware threads (12 physical cores per node with SMT). The interconnect (not shown) is symmetric.

[3] Placement of Virtual Containers on NUMA systems: A Practical and Comprehensive Model - Justin Funston et al. - 2018

State-of-the-art

- More generally, optimal task placement can vary widely between architectures[3]
- Can be almost impossible to predict the best placement without prior measurements[3]

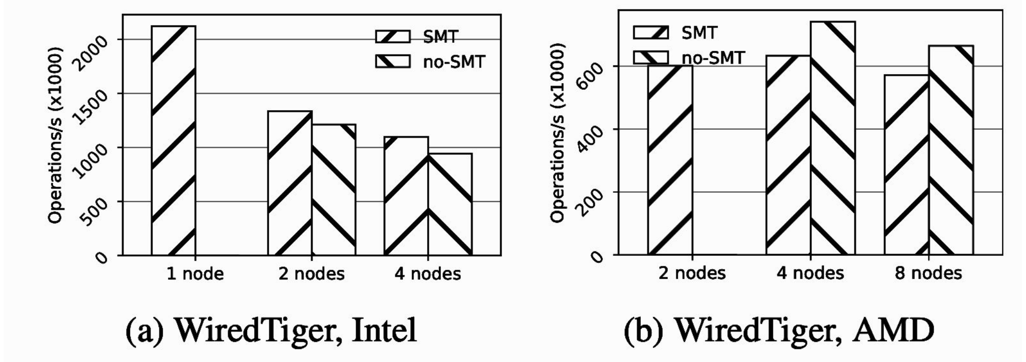


Figure 1: Throughput of the WiredTiger key-value store on two NUMA systems.

[3] Placement of Virtual Containers on NUMA systems: A Practical and Comprehensive Model - Justin Funston et al. - 2018

State-of-the-art

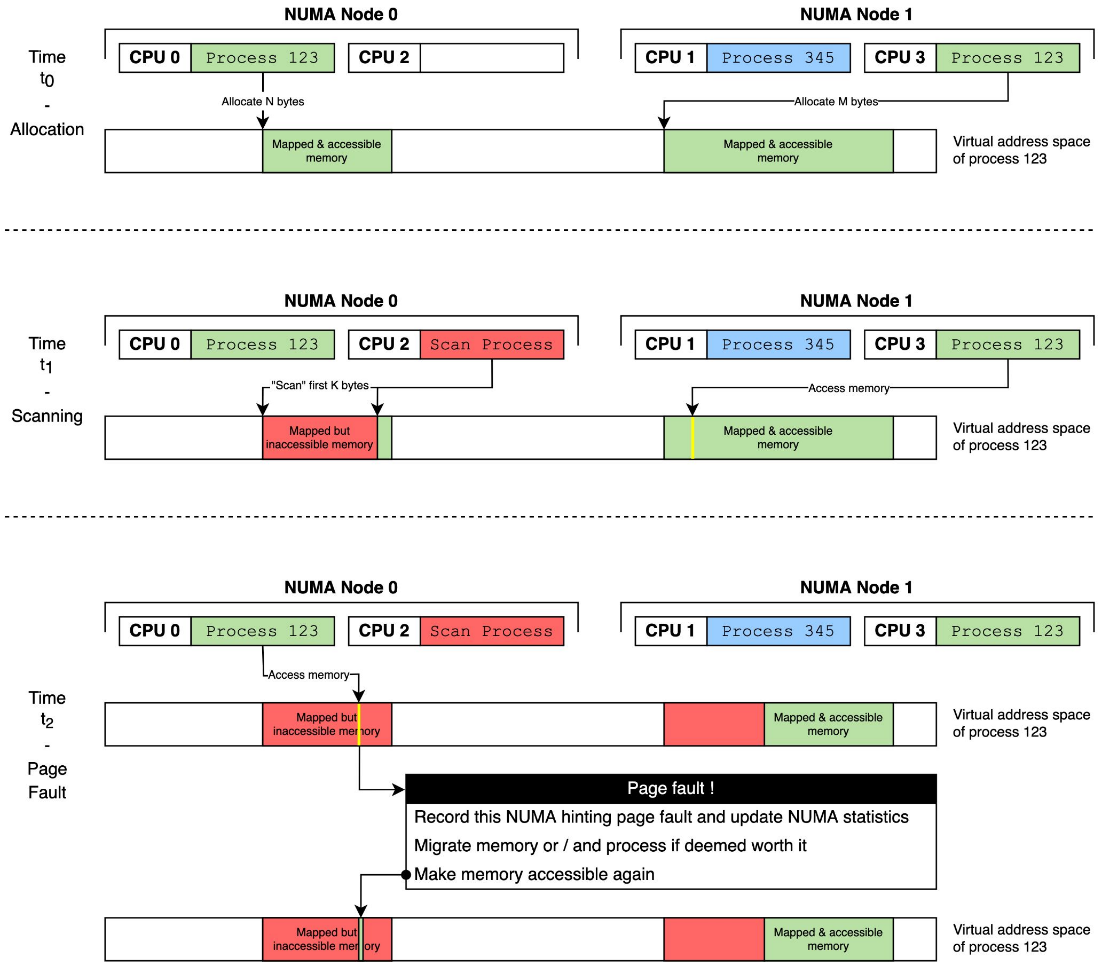
- Current solution in the Linux kernel : **automatic NUMA balancing**
- Aims at dynamically optimizing thread and memory placement on NUMA hardware
- 3 steps process :
  - Record memory accesses - referred to as “scanning”
  - Move memory closer to CPUs
  - Move tasks closer to memory

Moving memory and tasks happen as follows :

- Memory pages are migrated if they are accessed from the same (remote) node twice in a row
- Task migration on the other hand relies on NUMA hinting fault statistics : a per-process counter keeps track of the number of faults on each node
  - A task will try to migrate to the node that generates the largest number of page faults
  - However it still needs to find a cpu on this node, either idle or running a process that can be swapped with the current one

State-of-the-art

Diagram of the “scanning” phase of automatic NUMA balancing

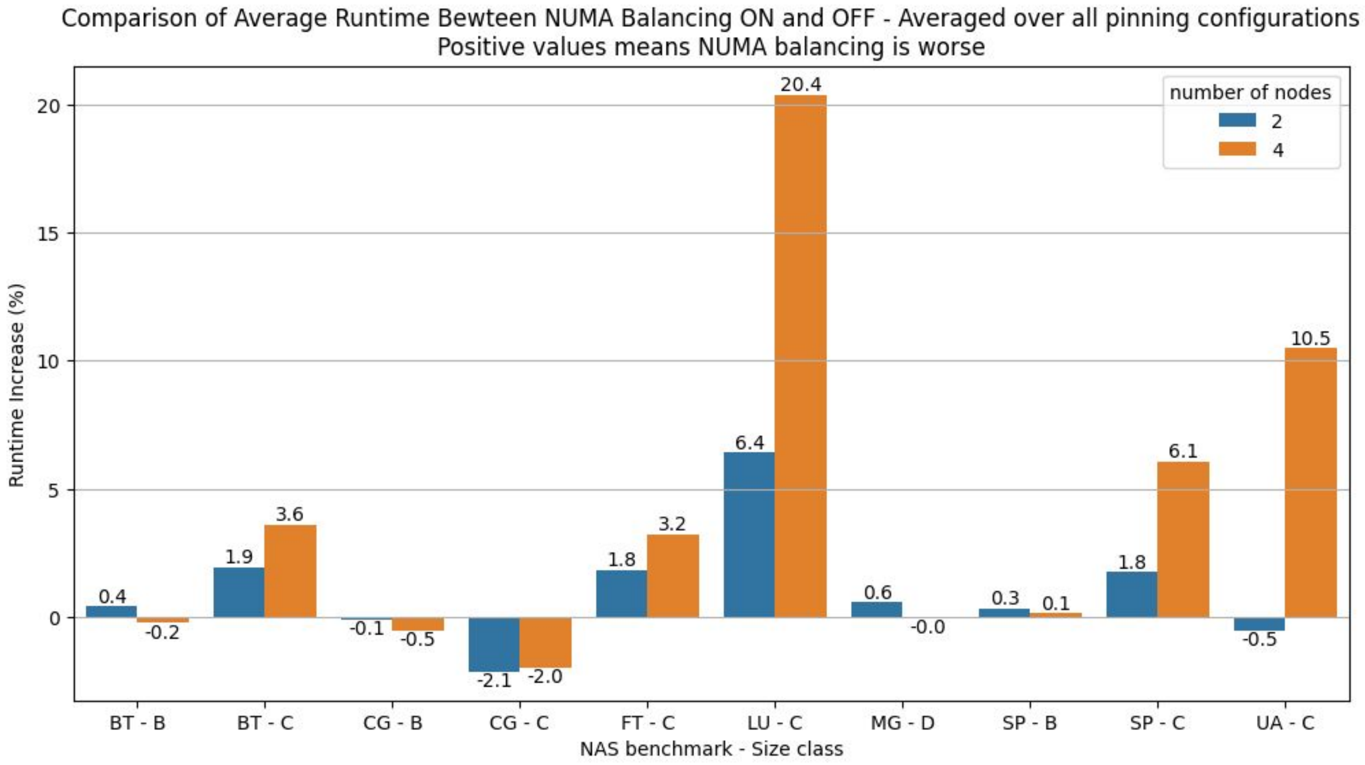


NUMA Node 0
NUMA Node 1
Time
t0
-
Allocation
CPU 0
Process 123
CPU 2
CPU 1
Process 345
CPU 3
Process 123
Allocate N bytes
Allocate M bytes
Mapped & accessible memory
Virtual address space of process 123
Time
t1
-
Scanning
Scan Process
"Scan" first K bytes
Access memory
Mapped but inaccessible memory
Mapped & accessible memory
Time
t2
-
Page Fault
Access memory
Page fault !
Record this NUMA hinting page fault and update NUMA statistics
Migrate memory or / and process if deemed worth it
Make memory accessible again

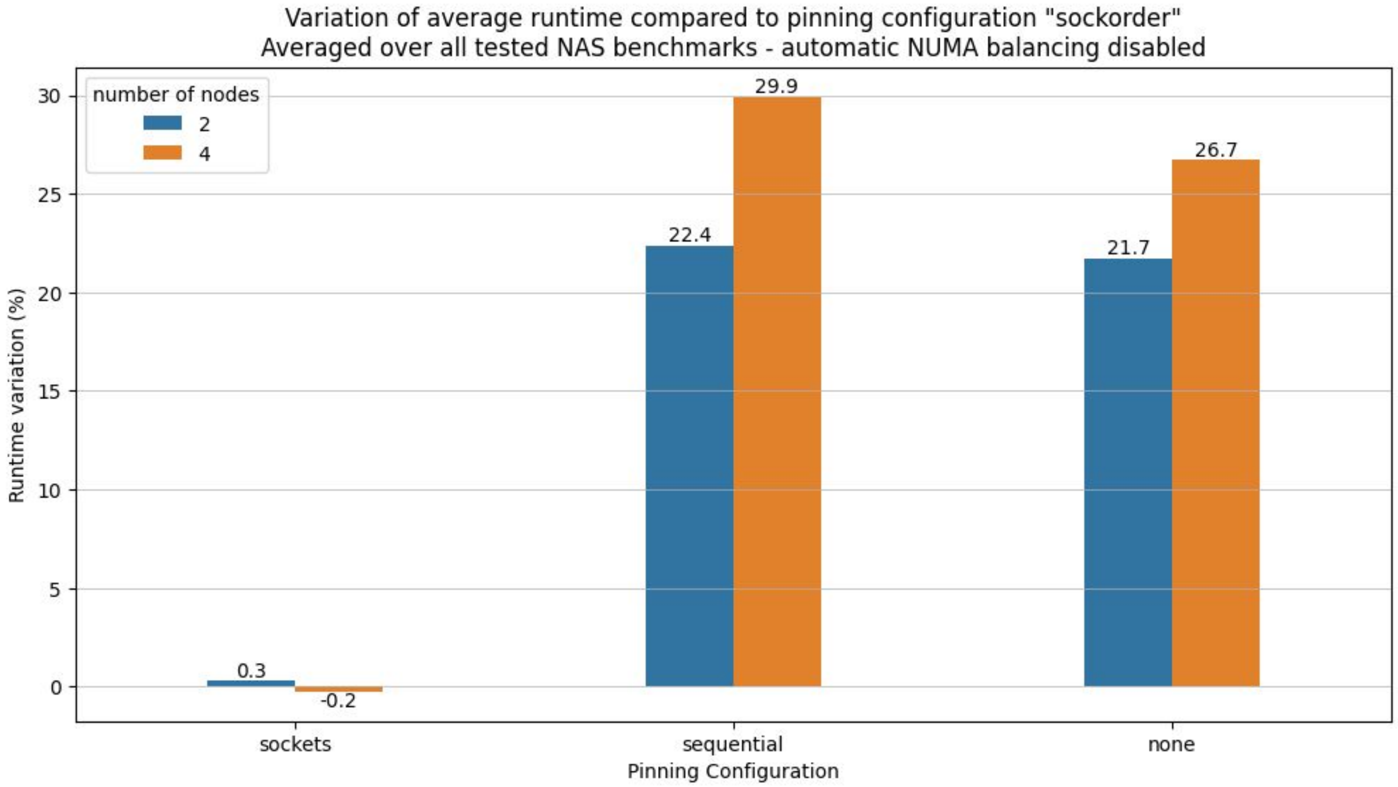
Exploratory Experiments

- **Hypothesis :** automatic NUMA balancing does not improve / degrades performance on certain workloads
- Testing protocol :
    - NAS benchmarks : multicore, wide range of memory access paterns
    - trace-cmd : record execution trace
- Parameters :
    - NAS benchmark application
    - Data size
    - Machine
    - Thread pinning configuration
- **1st set of results :** performance with vs without automatic NUMA balancing
- **2nd set of results :** performance variation between thread pinning configurations

Exploratory Experiments



Exploratory Experiments



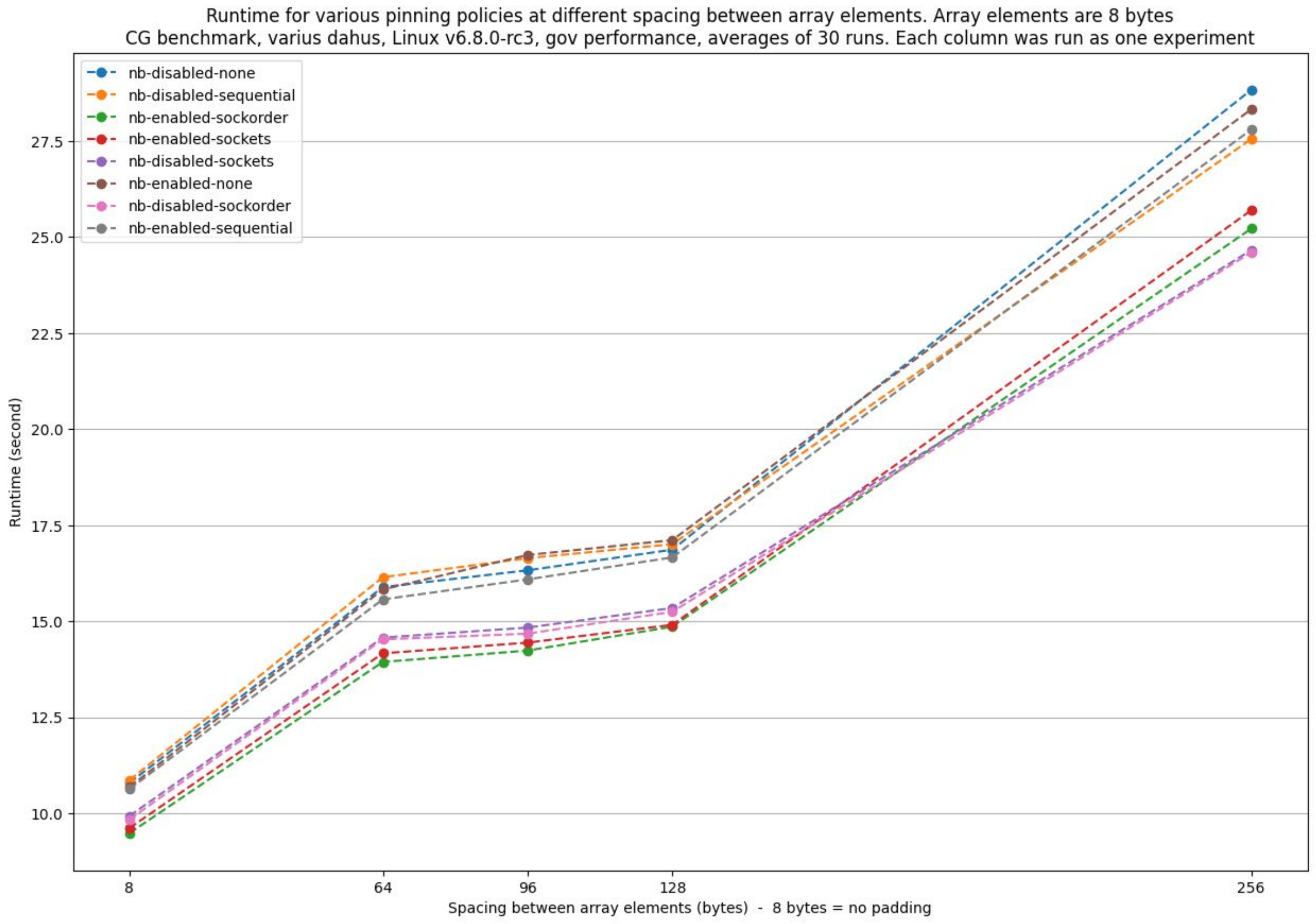
Exploratory Experiments

- **Conclusion :**
  - Automatic NUMA balancing **can** indeed **underperform** under certain workloads
  - Need for a more in-depth analysis to find the root causes

- **Other important takeaway :**
  - *Sequential* thread pinning configuration performs consistently worse than *sockorder*
  - No apparent reason for that
  - Easier to investigate because of better replicability compared to no thread pinning

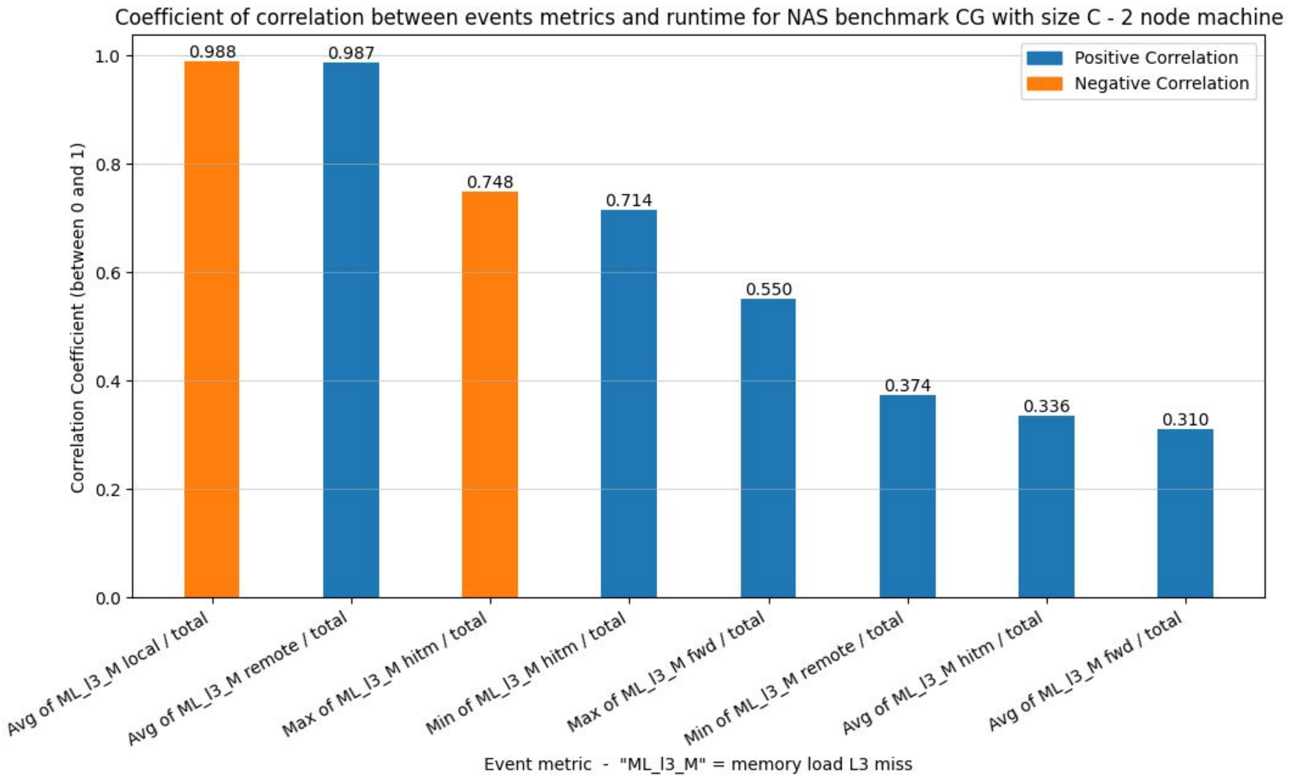
Profiling

- **Goal** : Account for difference in performance between *sequential* and *sockorder* pinnings
- Focus on CG with size C
- **1st hypothesis** : false cache sharing
  - Spaced out data to test the hypothesis
  - Showed it was **not** the reason for performance difference between pinning configurations
- Analyze correlation between runtime and various system events
  - 98+ % correlation with metric [L3 misses fetched from remote dram / all L3 misses]
- Identify patterns in memory accesses responsible for performance
  - Very large memory allocation blocks : **transparent huge pages**
  - False sharing of huge pages : several nodes access disjoint set of pages within the huge page
  - Many of the remote dram L3 miss events happen in false sharing configurations
- **Conclusion** : Need to address the false sharing of huge pages

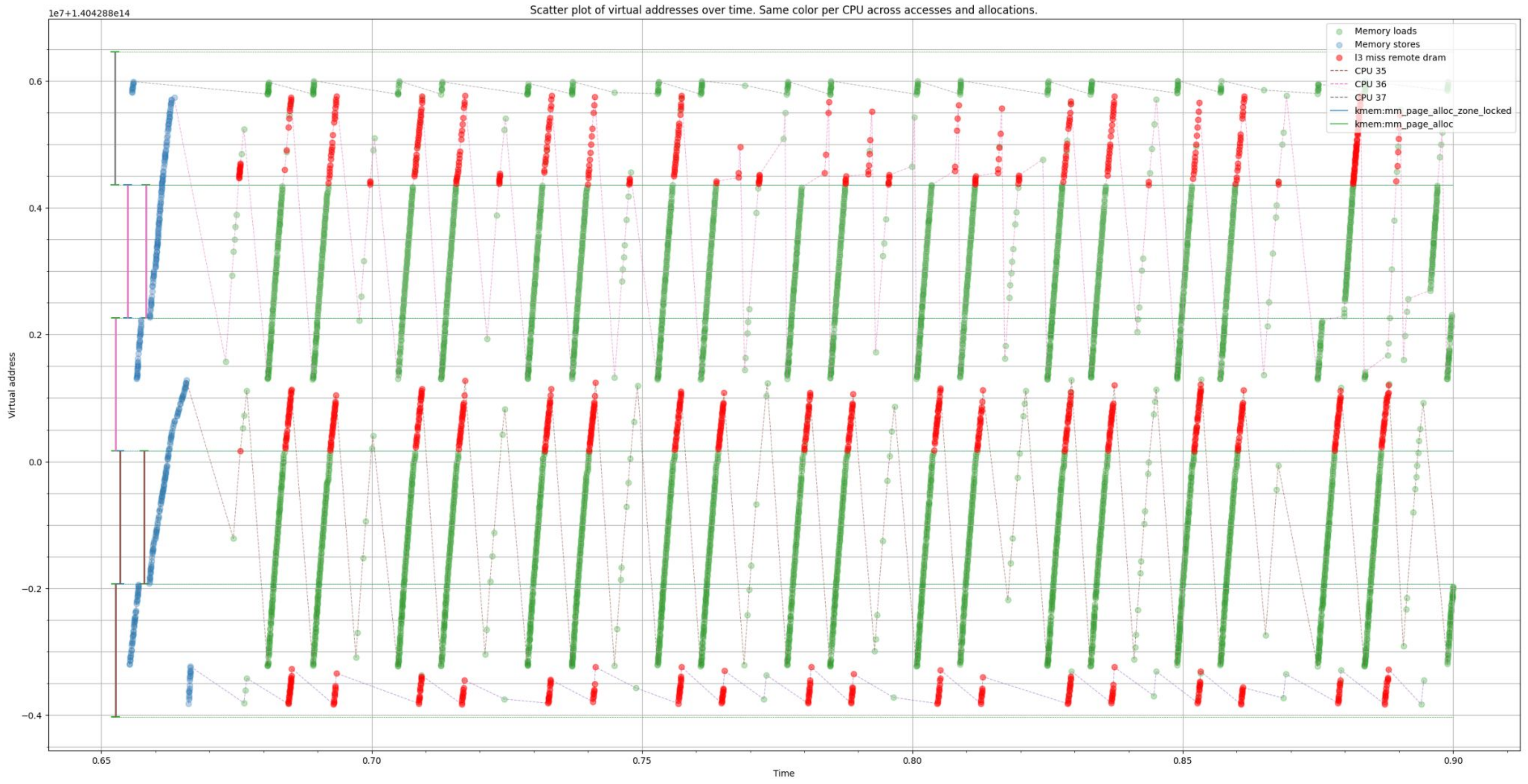


Runtime for various pinning policies at different spacing between array elements. Array elements are 8 bytes
CG benchmark, varius dahus, Linux v6.8.0-rc3, gov performance, averages of 30 runs. Each column was run as one experiment
nb-disabled-none
nb-disabled-sequential
nb-enabled-sockorder
nb-enabled-sockets
nb-disabled-sockets
nb-enabled-none
nb-disabled-sockorder
nb-enabled-sequential
Runtime (second)
27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
8
64
96
128
256
Spacing between array elements (bytes) - 8 bytes = no padding

Profiling

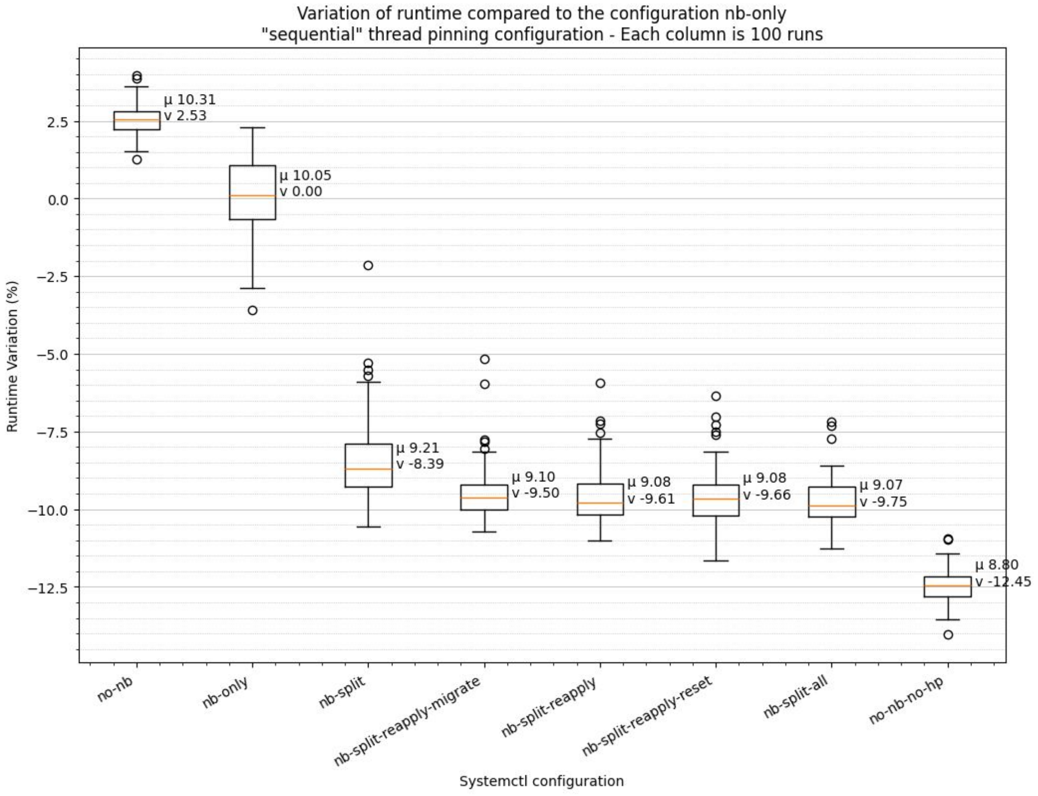

Coefficient of correlation between events metrics and runtime for NAS benchmark CG with size C - 2 node machine
Positive Correlation
Negative Correlation
Correlation Coefficient (between 0 and 1)
0.0
0.2
0.4
0.6
0.8
1.0
0.988
0.987
0.748
0.714
0.550
0.374
0.336
0.310
Avg of ML_l3_M local / total
Avg of ML_l3_M remote / total
Max of ML_l3_M hitm / total
Min of ML_l3_M hitm / total
Max of ML_l3_M fwd / total
Min of ML_l3_M remote / total
Avg of ML_l3_M hitm / total
Avg of ML_l3_M fwd / total
Event metric - "ML_l3_M" = memory load L3 miss

Profiling


Scatter plot of virtual addresses over time. Same color per CPU across accesses and allocations.
1e7+1.404288e14
Virtual address
Time
0.6
0.4
0.2
0.0
−0.2
−0.4
0.65
0.70
0.75
0.80
0.85
0.90
Memory loads
Memory stores
l3 miss remote dram
CPU 35
CPU 36
CPU 37
kmem:mm_page_alloc_zone_locked
kmem:mm_page_alloc

Proof of Concept

- Make use of automatic NUMA balancing logic to split transparent huge pages shared between nodes
- Optimizations :
  - Reapply NUMA page protection on split pages : trigger a page fault next time they are accessed
  - Reset NUMA counters of split pages : migrate on accessing node at next page fault
  - Migrate faulting page directly



Open questions

- Automatic NUMA balancing :
  - Is the efficiency of automatic NUMA balancing also poor on applications other than NAS benchmarks or is it just an exception ?

- Memory access patterns :
  - Are remote dram L3 cache misses also a cause of slowdowns on other benchmarks and / or other NUMA architectures ?
  - What proportion of remote dram L3 cache misses is caused by huge page false sharing ?
  - Is huge page false sharing the cause of other performance issues ?
  - How often does huge page false sharing happen ? Is it specific to the CG benchmark ?

Summary

Automatic NUMA balancing does not perform as good as expected

- Reduces performance for most of the NAS benchmarks

L3 misses fetched from remote dram seem to be the main performance bottleneck for certain applications on Intel machines

- Found a particularly high correlation with runtime
- In our case they were partly caused by false sharing of transparent huge pages

We have designed and implemented a Linux kernel patch that splits transparent huge pages shared between several NUMA nodes

- It has shown performance improvements but only in specific sptiatuin
- Is it applicable for a wider range of computing loads ?