OS-based NUMA Optimization: Tackling the Case of Truly Multi-Thread Applications with Non-Partitioned Virtual Page Accesses

Ilaria Di Gennaro
Alessandro Pellegrini
Francesco Quaglia

High Performance and Dependable Computing Systems Group
Sapienza, University of Rome

CCGrid 2016

The Advent of NUMA Architectures

- Modern computing architectures have a large amount of RAM and a high count of cores
- Uniform Memory Access shows a latency which is no longer affordable
- Non-Uniform Memory Access (NUMA) is the de-facto reference organization of med/high end systems

- It has anyhow an effect on the *efficiency* of applications
    - Accessing memory areas has different costs
    - Concepts of *zones* and *distance*

How to optimize efficiency on NUMA?

1. Move threads around
   - It's a cheap operation
   - A thread can be moved close to its data
2. Move pages around
   - It's more costly
   - What if a thread has pages in all zones?
3. Move both pages and threads
   - Multiple threads can need the same pages
   - They can all be moved to the same zone

How to optimize efficiency on NUMA?

1. Move threads around
   - It's a cheap operation
   - A thread can be moved close to its data
2. Move pages around
   - It's more costly
   - What if a thread has pages in all zones?
3. Move both pages and threads
   - Multiple threads can need the same pages
   - They can all be moved to the same zone

You have to know *what to move*!

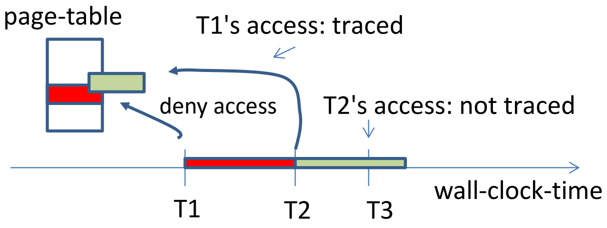
Runtime Determination of Access Patterns

- We want to know what the current *working set* is
  - at least a good estimation!
- We do not want to pay a high overhead for this

1. User-level approaches
   - Based on instrumentation and code injection
   - Ad-hoc code to monitor memory accesses
   - This can be intrusive, and cannot be fully disabled
2. System-level approaches
   - Based on memory protection
   - Segfaults are the materialization of memory accesses
   - This approach does not work at all with *non-partitioned vitual page accesses* within the same address space

Multi-thread Applications and Page Protection

- All threads live in a same process, so all threads have the same page table!

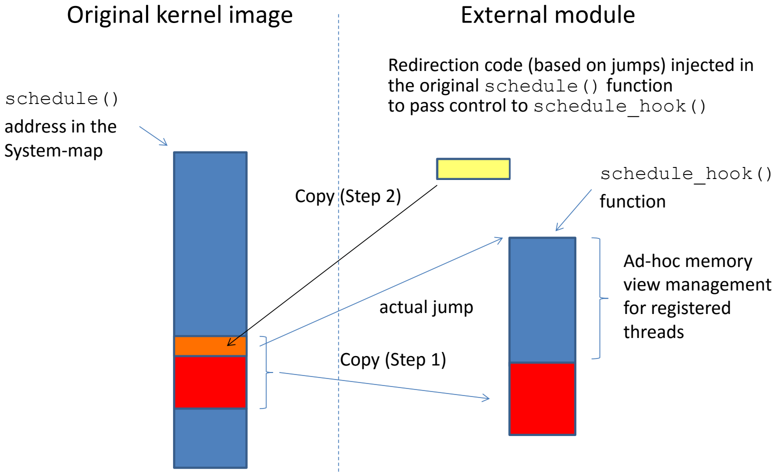


- Increasing the frequency of denial is not an option
    - The same thread might fault multiple times on the same page

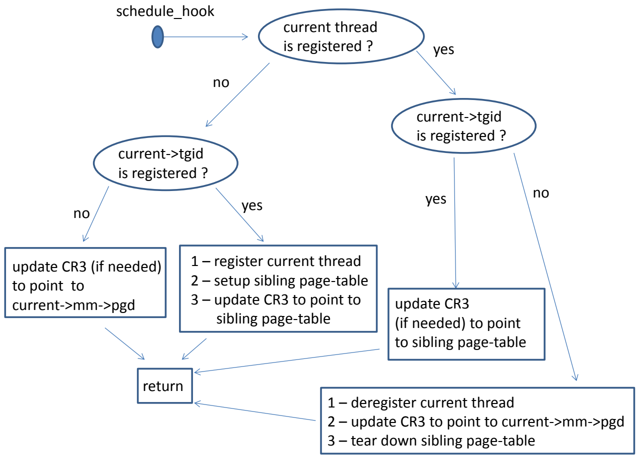
MVAS: *MultiView Address Space*

- Each thread has its memory view on the page table
  - We call it a *sibling page table*
- Minor faults can be used to detect accesses to pages
  - An ad-hoc fault handler is used
  - We don't have to use the (costly) chain of supports for a `SIGSEGV`
- Everything is based on a Linux Kernel Module
  - A device file is used to interact with the module
  - `ioctl` calls can be used to activate/deactivate tracing on a PID
  - A shell program is provided to manage the interaction
  - ...the scheduler "must know" what we are doing

Patching the Linux `schedule()`

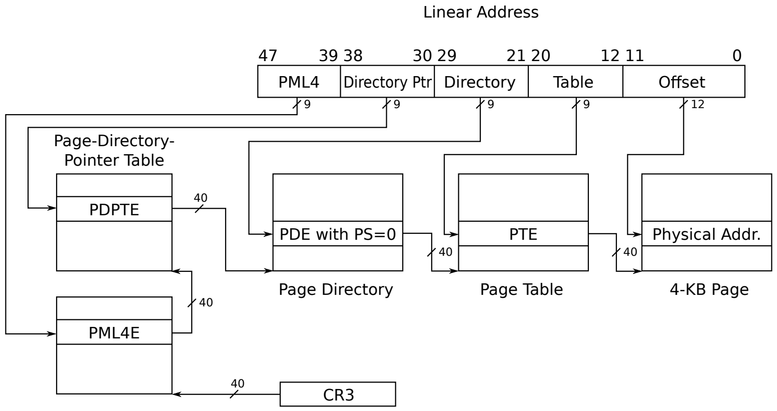

Original kernel image
External module
Redirection code (based on jumps) injected in the original schedule() function to pass control to schedule_hook()
schedule() address in the System-map
Copy (Step 2)
schedule_hook() function
actual jump
Ad-hoc memory view management for registered threads
Copy (Step 1)

Scheduling Operations



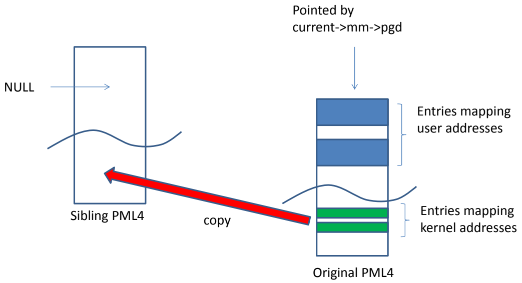
schedule_hook
current thread is registered ?
no
yes
current->tgid is registered ?
current->tgid is registered ?
no
yes
yes
no
update CR3 (if needed) to point to current->mm->pgd
1 – register current thread
2 – setup sibling page-table
3 – update CR3 to point to sibling page-table
update CR3 (if needed) to point to sibling page-table
return
1 – deregister current thread
2 – update CR3 to point to current->mm->pgd
3 – tear down sibling page-table

Recall on x86-64 Paging Scheme

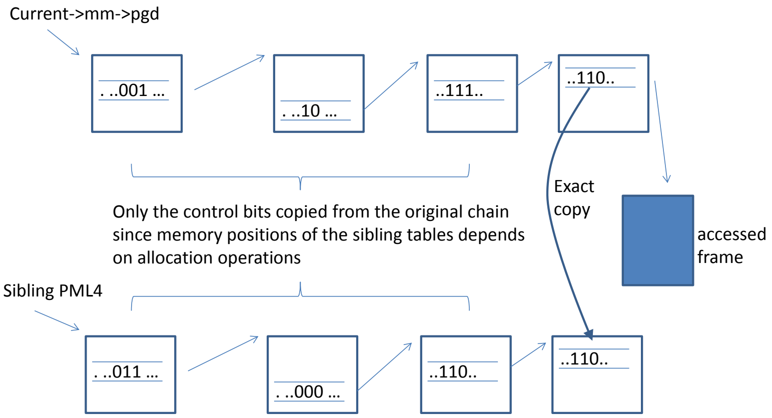


Setting up a Sibling Page Table

- A Sibling Page Table is setup by partially copying PML4 entries
- `NULL`'ed entries generate a page fault upon access
- Our IDT is modified: the fault call's an ad-hoc handler



How to Manage a Fault



How to Manage a Fault (2)

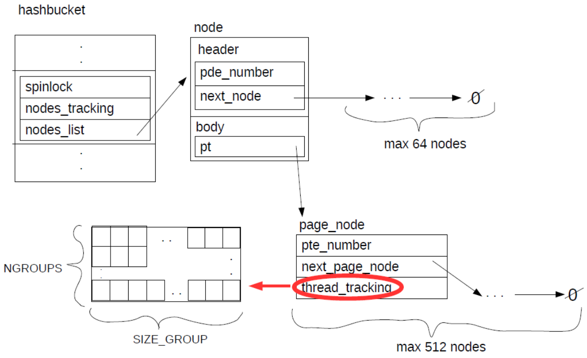
- In case we had a real fault (minor or not) the traversal fails
- In this case, we call the original fault handler
  - The original handler takes the table from `current->mm->pgd`

- If it had been minor, control returns to the original code
  - The thread runs on the parallel view
  - It has no information on how the fault was solved

- A new fault is generated, but it can now be resolved by our handler
- At most 2 faults are required to open the access

Logging the Faulting Accesses

- We use a scalable hash table in kernel memory
- An additional system call is offered by the module to dump the tracked access info to userspace



Keeping parallel view consistent

- The original page table can change at runtime:
  - Pages can be invalidated
  - Access privileges might change
  - Physical mapping might change
- Linux is synchronized to ensure consistency
  - System calls perform changes synchronously
  - IPI message invalidate TLBs on other cores
- We have wrapped all the system calls that change the memory map

NUMA migration rule: affinity estimation

- For each page, we generate an *access-count tuple*:

$$p_i = \langle n_0, n_1, \ldots, n_{T-1} \rangle$$

- We extract the *highest access count* $M_i = \max_j\{n_j\}$ to compute the *relative access-frequency tuple*:

$$\varphi_i = \left\langle \frac{n_0}{M_i}, \frac{n_1}{M_i}, \ldots, \frac{n_{T-1}}{M_i} \right\rangle$$

- We build a *per-page access-frequency matrix*:

$$\Phi^i_{l,m} = \Phi^i_{m,l} = \begin{cases} (\varphi_{i,l} + \varphi_{i,m})/2 & \text{if greater than } \alpha \\ 0 & \text{if } \varphi_{i,l} \vee \varphi_{i,m} = 0 \quad \forall l \neq m \\ 0 & \text{otherwise} \end{cases}$$

NUMA migration rule: affinity estimation (2)

- We then build the symmetric *access matrix*:

$$A_{l,m} = A_{m,l} = \sum_{i=0}^{P-1} \Phi^{i}_{l,m}$$

- This matrix tells whether two threads share a certain amount of pages in their working set
- From this matrix we extract elements to build a vector **v** of tuples $\langle t_l, t_m, a_{l,m} \rangle$
- Vector **v** is ordered descending by the elements $a_{l,m}$

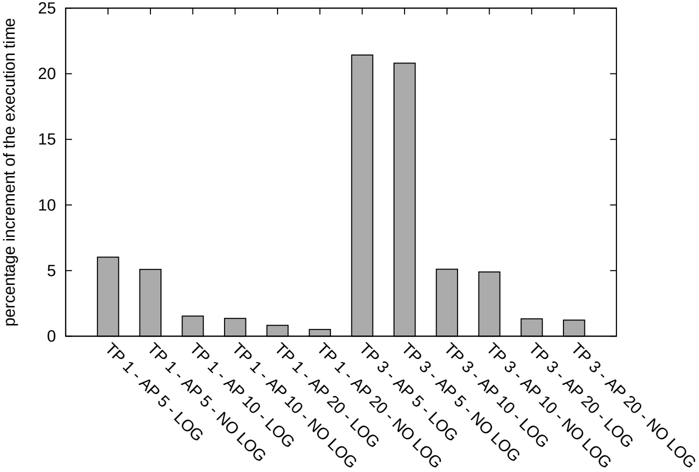
NUMA migration rule: grouping

- At this point, we want to map threads and pages to NUMA zones
- We pick threads t_l^0 and t_m^0 from $\mathbf{v}$, and we add them to the first zone
- We then iterate over the vector
  - If one of the two threads in the next element already belongs to a zone, we add the other thread to the zone if there is space
  - Otherwise we add it to the next zone
- Leftover threads are not bound to a zone
  - We let the OS decide what to do with them
- Pages are moved to the zone with the thread with the highest access count

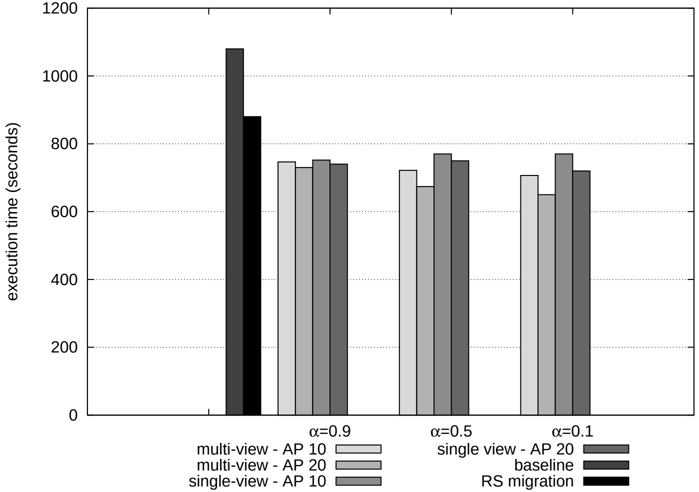
Experimental Environment

- 32-cores NUMA Machine
  - 8 NUMA zones
  - Two different distances (10 and 20) for each core
- The ROme OpTimistic Simulator (ROOT-Sim)
  - HPC application
  - Speculative execution: large usage of memory (around 30 GB)
  - CPU-bound threads: performance is affected by memory policies
  - Load-Sharing Policy: threads actually share a lot of data
  - It has an internal NUMA policy targeting the *resident set*
- A mobile phone simulation model has been run on top of ROOT-Sim

Experimental Results: Overhead Evaluation



Experimental Results: Performance Evaluation



Thanks for your attention

Questions?

pellegrini@dis.uniroma1.it
http://www.dis.uniroma1.it/∼pellegrini
https://github.com/HPDCS/MVAS