MEMORY MANAGEMENT

AND OS PAGING

FOR HIGH PERFORMANCE COMPUTING.

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES

www.cern.ch

PhD. thesis work done at CEA in 2010-2014

Sébastien Valat - CERN

Conférence MiNET - 24 may 2018

Plan

I. Introduction

II. Analysis of OS paging policy

III. NUMA allocator for HPC applications

IV. Page zeroing in Linux first touch handler

V. Conclusion

INTRODUCTION

A little bit of bibliography

- **Interesting to read :**

- What every programmer should know about memory (Urlich Drepper)
  https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

- For all details from this presentation : look on my PhD. thesis **:**

- Contribution à l'amélioration des méthodes d'optimisation de la gestion de la mémoire dans le cadre du Calcul Haute Performance
  https://hal.archives-ouvertes.fr/tel-01253537

Context : HPC

- **Supercomputers** for numerical simulations
- At CEA, **Tera 100 :**
  - **6e** from TOP 500 in 2010
  - **140 000** cores, **1.05** Pflops.
- **Massively parallel** machines (**19 million cores**)
  - Gyoukou, Japan
- More and **more cores** !

http://www.cea.fr/multimedia/Pages/galeries/defense/Tera-100.aspx

Architecture	Proc.	Cores	Threads
Tera 100 thin nodes	**4**	**32**	32
Tera 100 large nodes	**16**	**128**	128
Intel KNL	1	64	256
PEZY-SC2	1	**2048**	**16 384**

Context : HPC

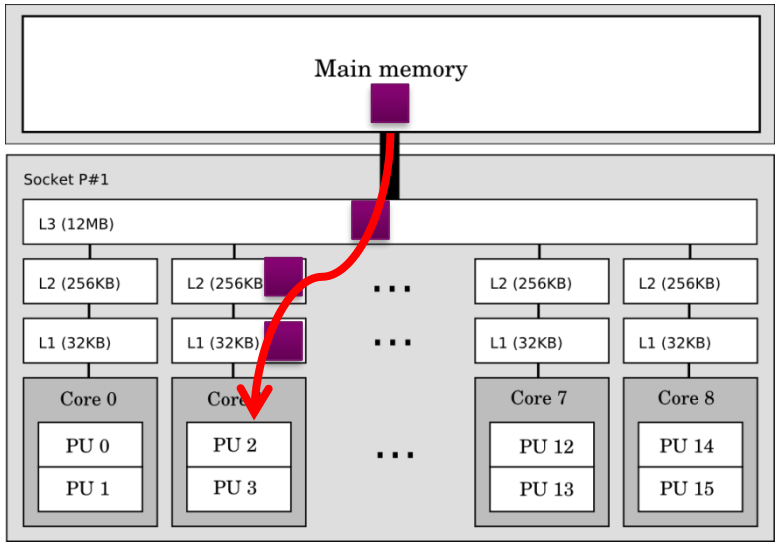
- **Memory** becomes a **critical resource**
- Growing impact on **performance**
- **Data movements :** speed gap CPU / RAM, **memory wall**.
- **Management** : now have to handle close to **TB** of memory
- Decreasing **memory per core**

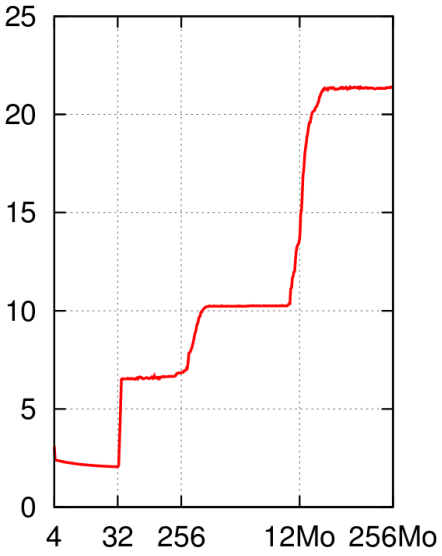
http://www.cea.fr/multimedia/Pages/galeries/defense/Tera-100.aspx

Architecture

- Computer science : **operations** & **data**
- Multiple **memory levels**
- Hierarchical **caches**
- *Pre-fetcher*

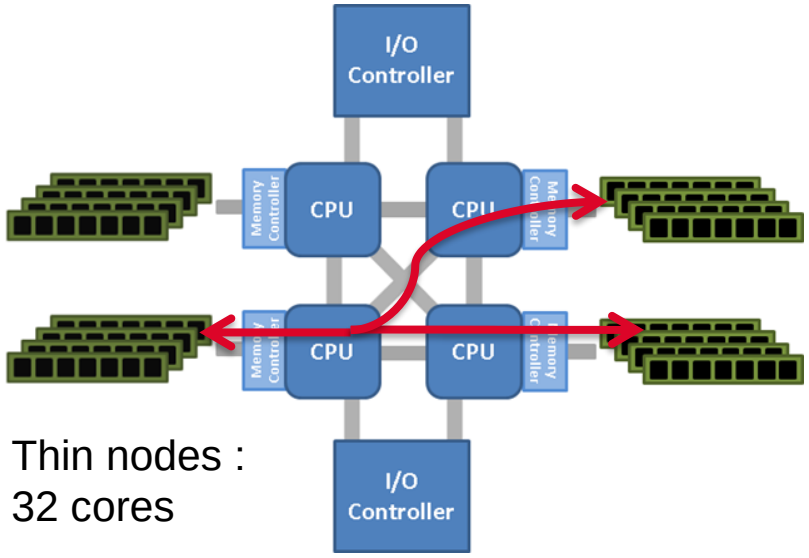
Processor : 8 cores



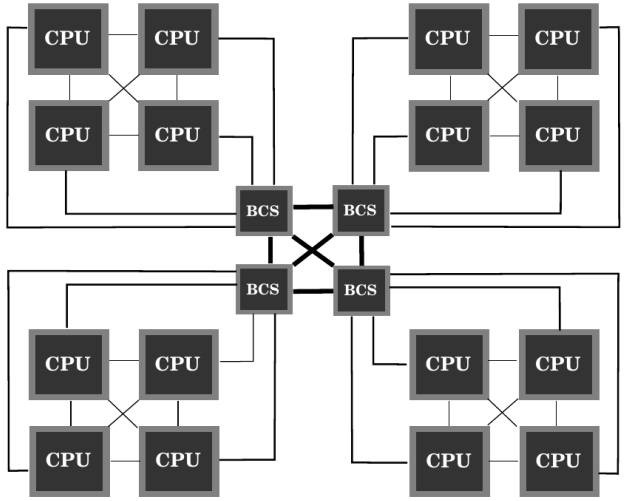


Architecture

- Hierarchical **memory**
- **Remote** / **local** memories **(NUMA : Non Uniform Memory Access)**



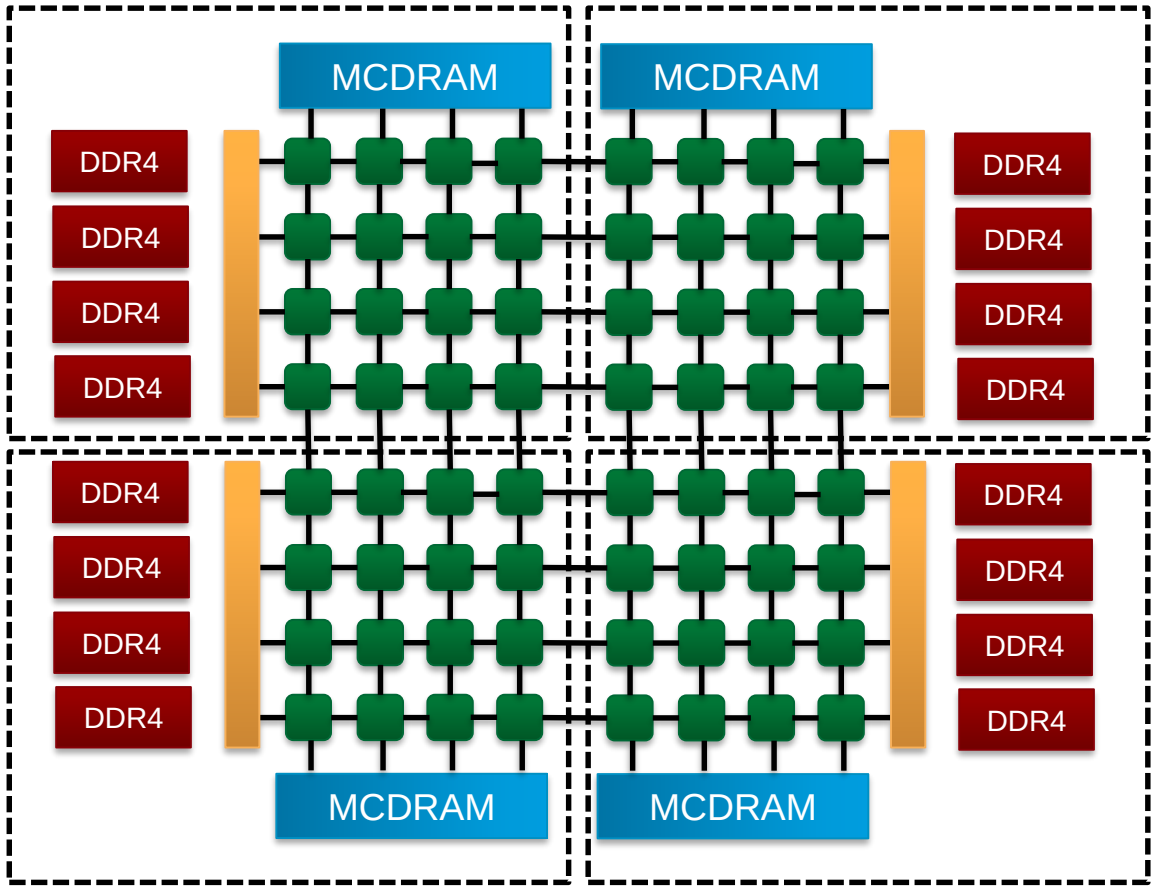
Thin nodes :
32 cores



Large nodes : 128 cores (BCS)

Now also inside the CPU – Intel KNL

- Intel KNL (64 cores) can be configured in **2 or 4 NUMA domains**
- Also add **MCDRAM** (similar idea than GPU GDDR5) **viewed** as a **NUMA** node



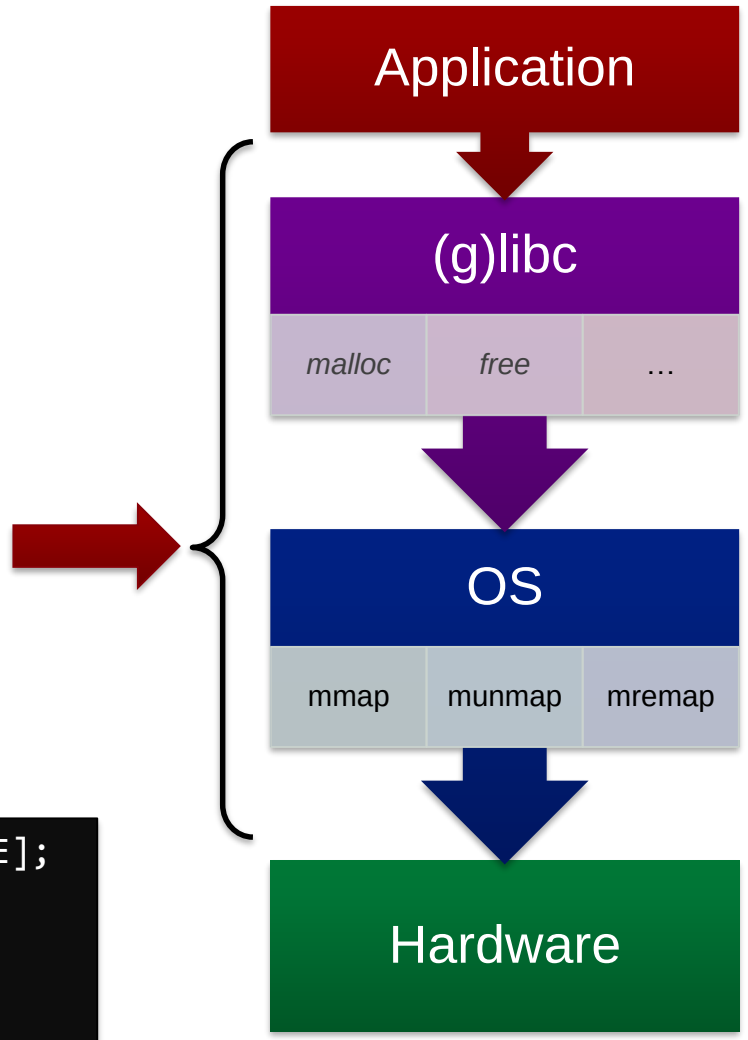
- Or on **AMD** CPUs

Software memory management layer

- **Impact of memory management mechanisms ?**
- Involving **two components** :
  - User space ***memory allocator*** (malloc)
  - **Operating System** (OS)
- Focus on :
  - Impact on **allocation time**
  - Impact on **access efficiency** (placement)
- Malloc C or C++ interface :

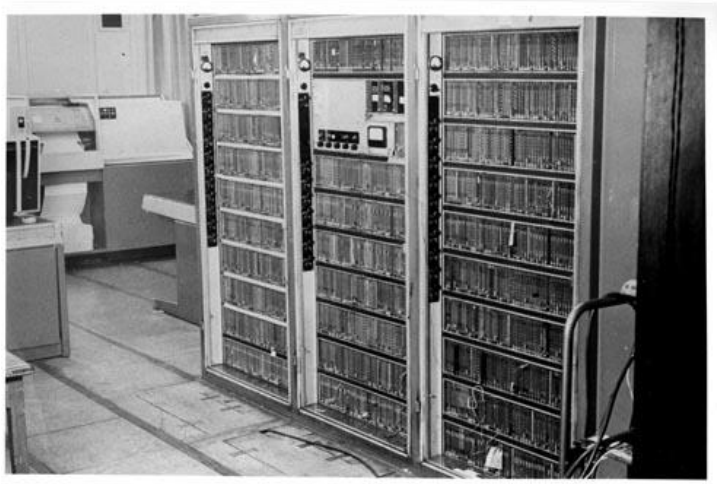
```
float * ptr = malloc(SIZE);
…
ptr = realloc(ptr,NEW_SIZE);
…
free(ptr);
```

```
float * ptr = new float[SIZE];
…
…
…
delete [] ptr;
```

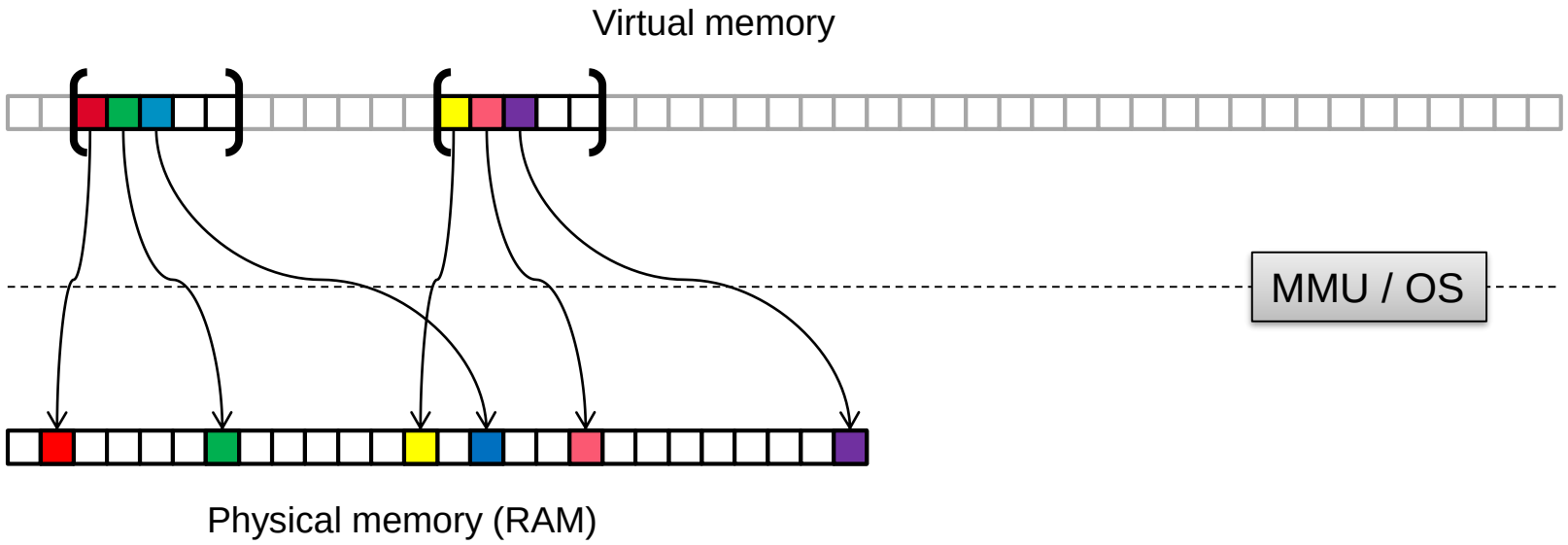


OS virtual / physical address spaces

- Two address spaces : **physical** + **virtual**
- Description of the **memory mapping** in blocks of **4 KB (pages)**
- **Paging** was first used in **1962** on the **ATLAS computer**
- **Area** creation with syscalls : **mmap / munmap / mremap**
- **Malloc** has the responsibility to **hide the pages to developers**

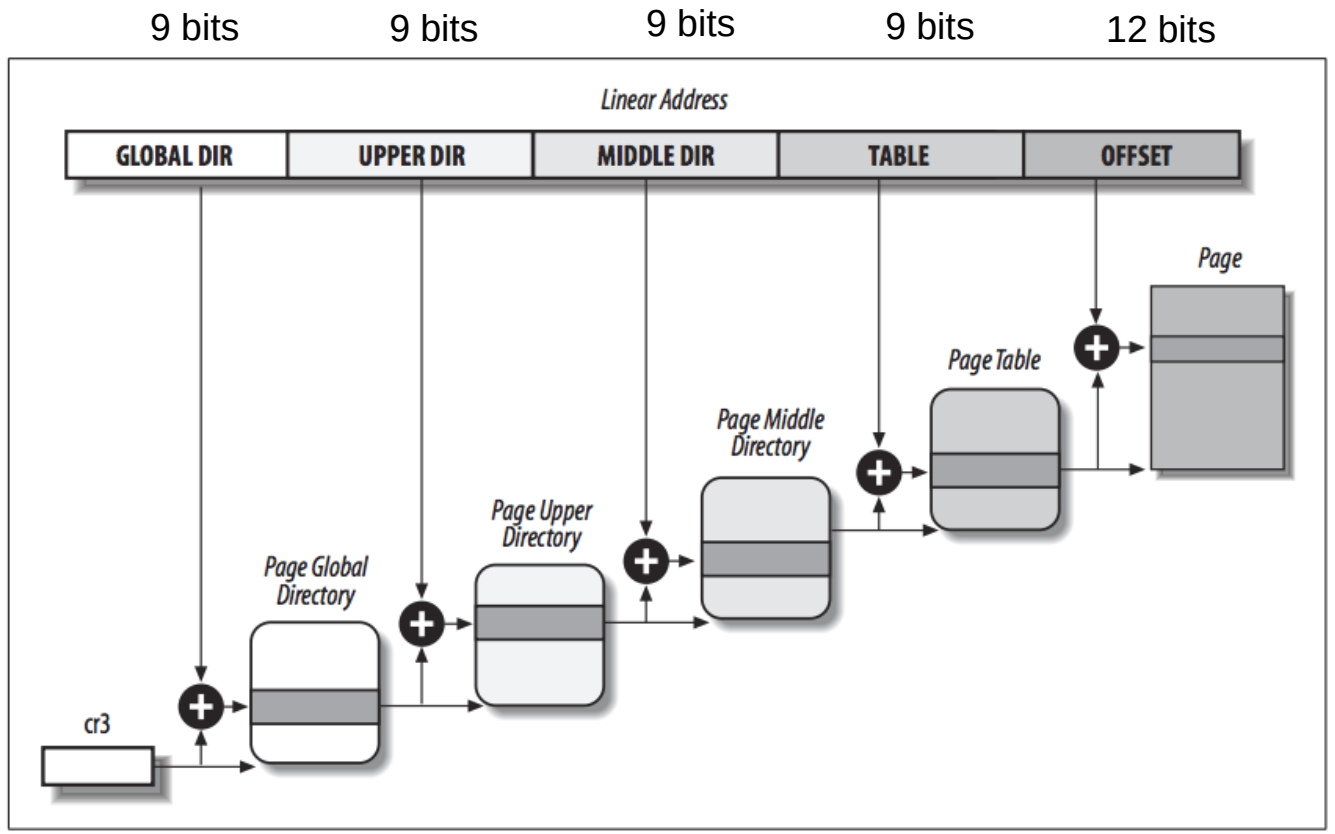


http://www.computerhistory.org/collections/catalog/102698470



32 bits = 4 GB
48 bits = 256 TB
57 bits = 128 PB
~~64 bits = 16 EB~~

Page table and TLB

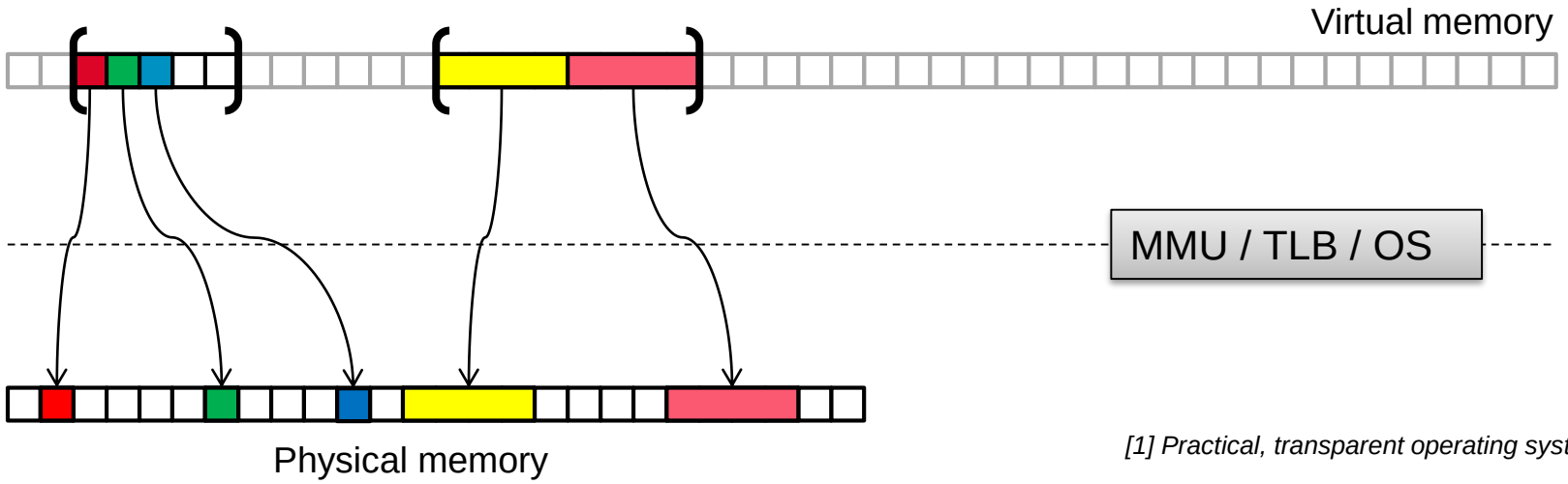


* From : Linux Kernel Driver

- We **don't** want to **walk** over the page table for **every access**
- CPU has **a cache** (**TLB** : Translation Lookaside Buffer)

Huge pages

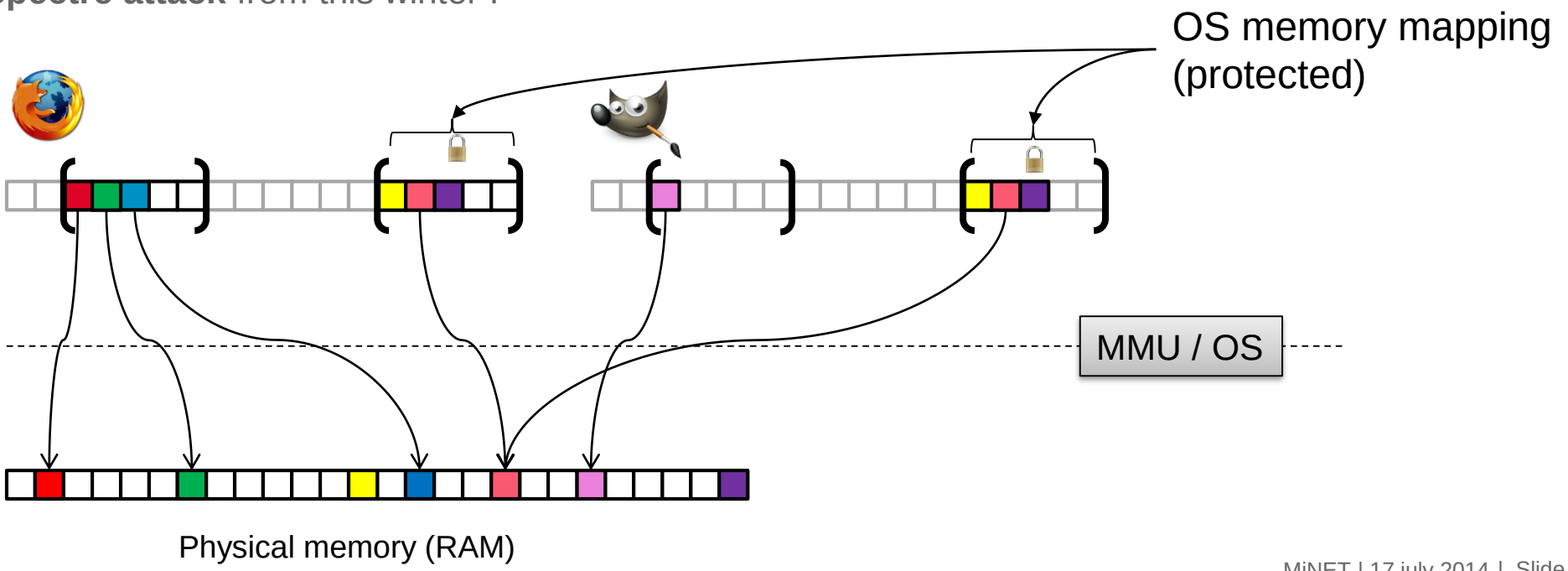
- With **4K pages**, intel CPU TLB has **1024 entries** => address **4 MB**
- **x86_64** processors also support **2 MB** or **1 GB** pages **(Huge pages)**
- With **2M pages**, TLB address **2 GB**
- **First real support : FreeBSD (superpages, 2002)** [1]
- Support **Linux : *old HugeTLBfs*** then now **Transparent Huge Pages (THP), 2011**



[1] Practical, transparent operating system support for superpages, 2002

Play with memory mapping

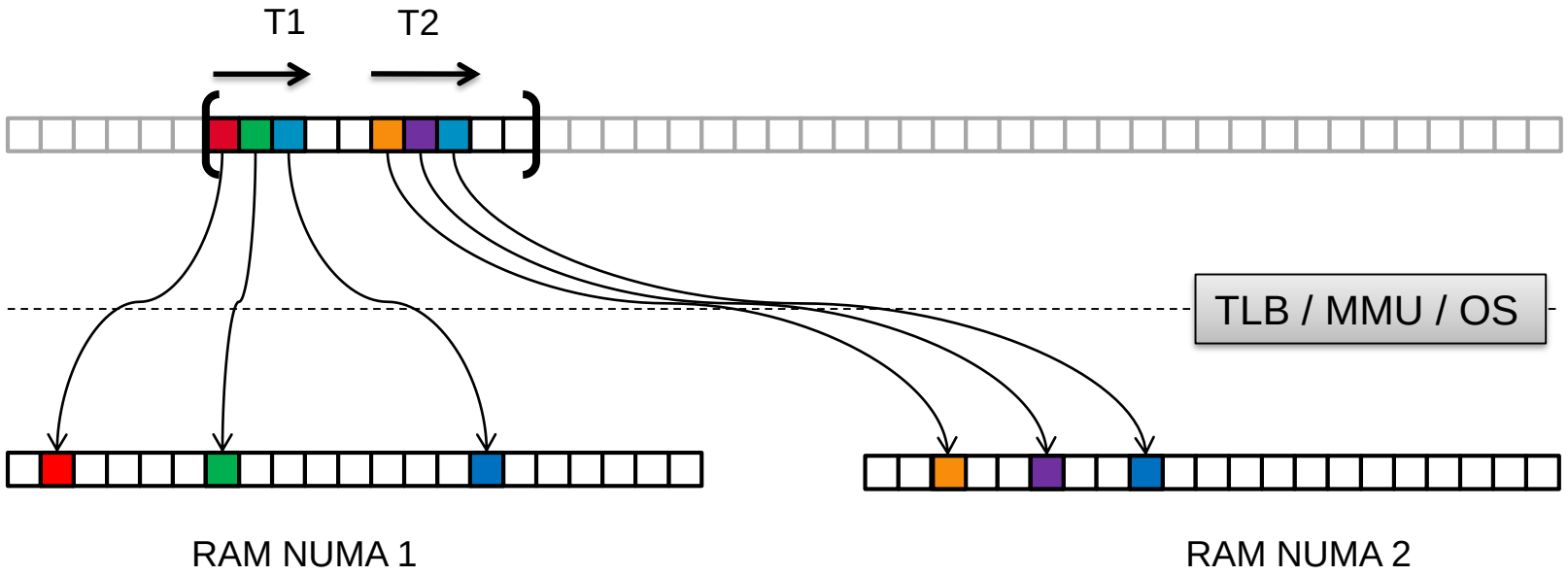
- Virtual memory **isolate** each process
- We can do **shared memory,** mapping **the same memory twice**
- **Most OS** also use a **trick** by **mapping** the **OS memory in each process**
  - At the **end of the address space**
  - **Protected**
- Issue with **Spectre attack** from this winter !



Lazy page allocation

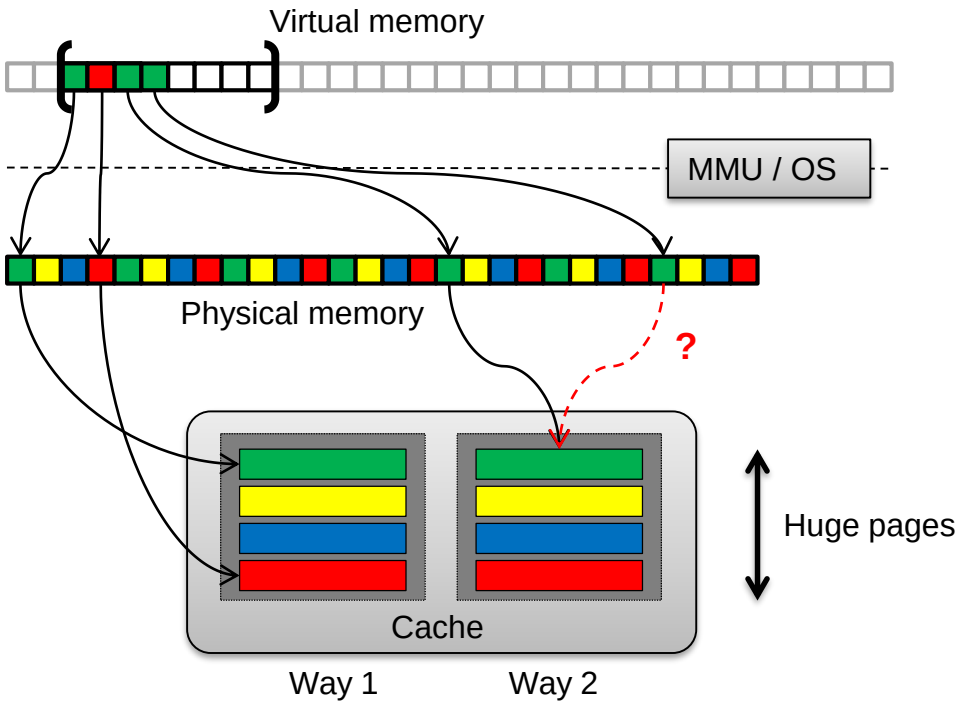
- **mmap** creates **pure virtual** area
- First touch creates a **page fault** for each virtual page
- OS provides **physical pages** on **first touch**
- **First touch <u>implicitly</u>** determines **NUMA placement** of the page

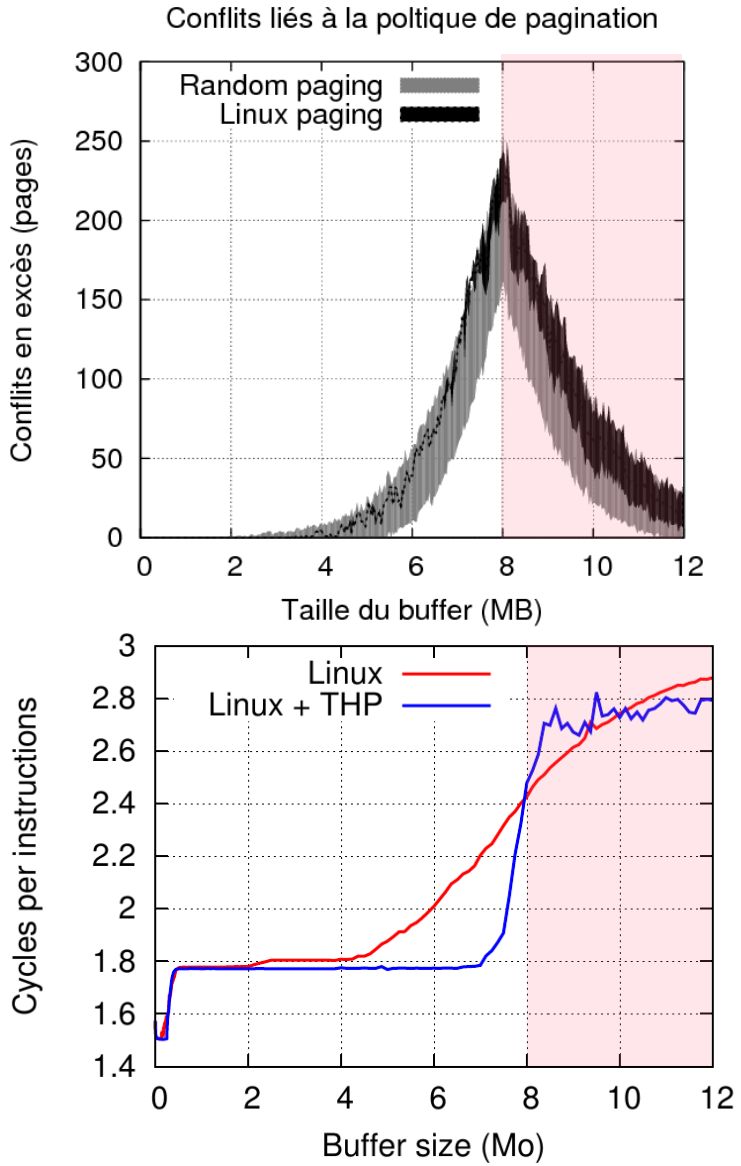
```
ptr = mmap(…,SIZE,…);
#pragma omp parallel for
for (i = 0 ; i < SIZE ; i++)
                ptr[i] = 0;
```



Cache associativity

- Data can only be placed in one of the **N lines associated to the address**
- Can create **conflicts** depending on the OS
- Linux "**randomly" chooses** the pages

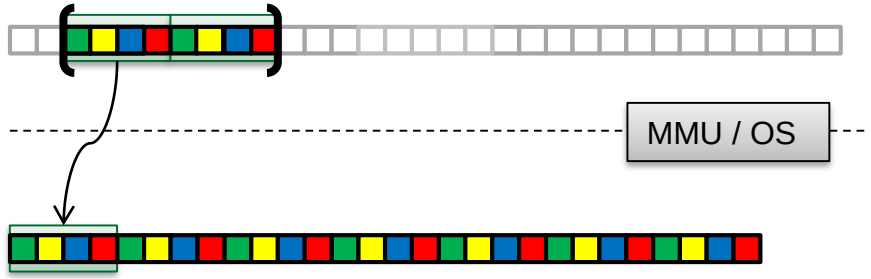




Existing solutions

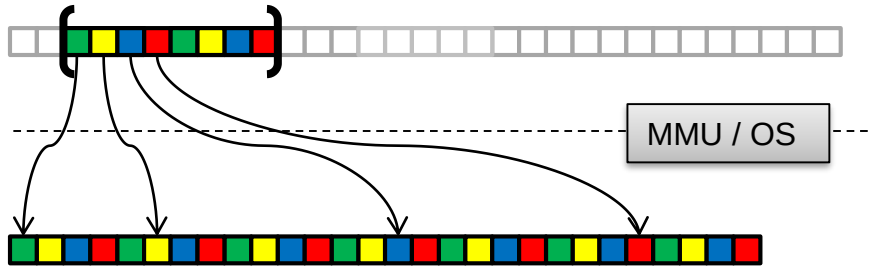
Huge pages

- **Larger than cache ways**
- **Native** support on **FreeBSD**
- **Extended** support on **Linux** / **OpenSolaris**



Page coloring

- 4K pages by **taking care of associativity**
- Available on **OpenSolaris**
- **Color** based on **virtual address** (modulo)
- **Regular coloring** : coloration with **repeated patterns**



ANALYSIS OF OS PAGING POLICY

OS strategies comparison

- Each **system** has its default paging **strategy**:

OS	Strategy
Linux	4K random
OpenSolaris	Page coloring
FreeBSD	Huge pages

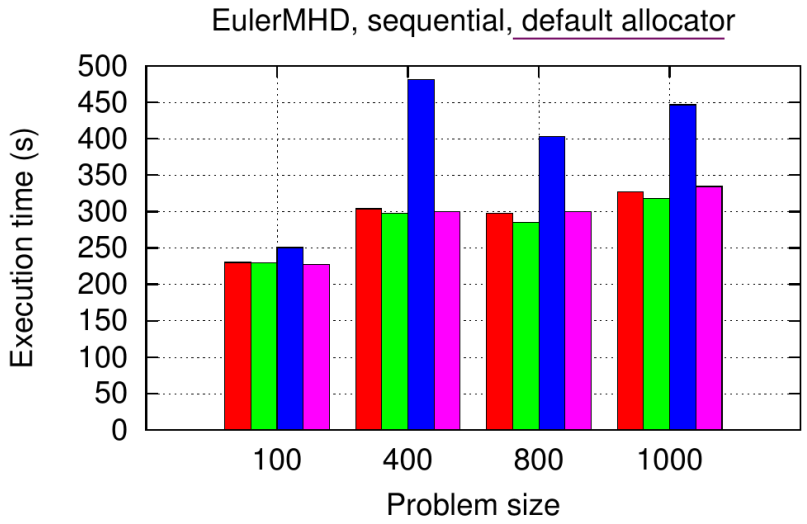
- Is **Linux** slower due to **random paging** ?
- Tested architecture : Intel **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**
- **Focus a pathological case**

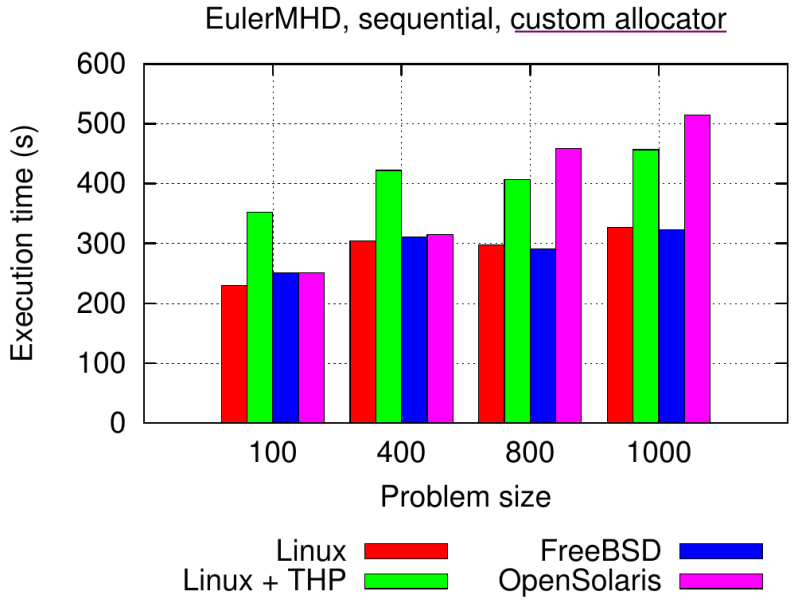
EulerMHD issue

- **EulerMHD (CEA) :**
  - **C++ /MPI**
  - Magnéto-hydrodynamic **stencil code**
- **FreeBSD :** slowdown of **1.5x,** up to **3x** in **parallel**
- Impacted function only do compute.
- Function with **9 arrays pre-allocated** at init. :

```
for (i = 0 ; i < SIZE ; i++)
        x1[i] = x2[i] + x3[i] … + x9[i]
```

- Change between OS's :
  - **User space memory allocator** (malloc).
  - **OS paging policy**
  - ~~*(Scheduler)*~~
- Effect can be controlled by **changing the allocator.**

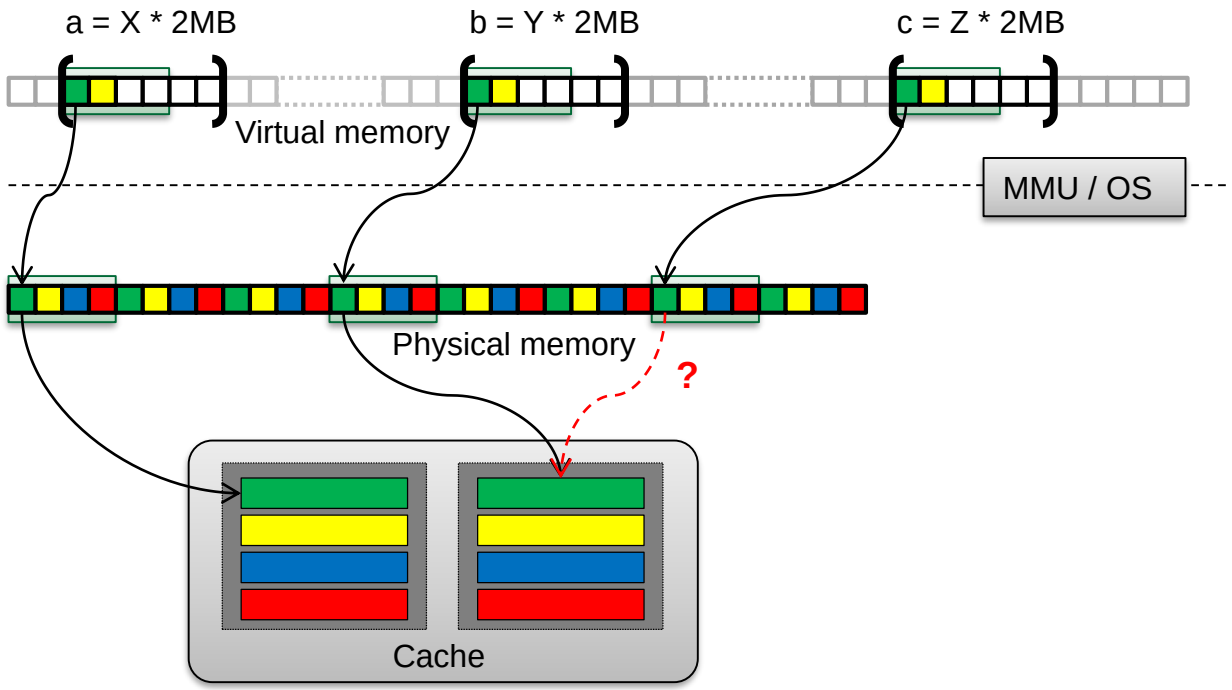




Alignment effect on regular coloring

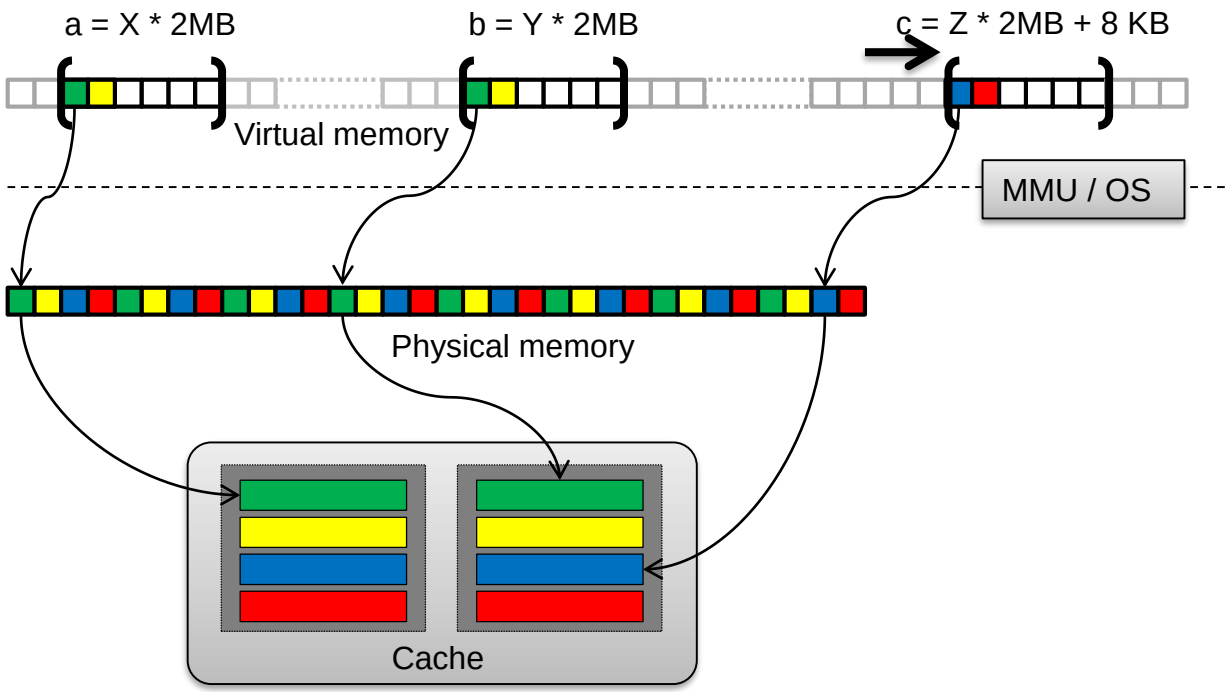
- Each **malloc** (OS) produces different **alignments**
- **FreeBSD** align **large segments** on **2 MB**
- **It interferes** with **regular patterns** generated by :
  - OpenSolaris coloration method (modulo)
  - Huge pages

```
for (i = 0 ; i < SIZE ; i++)
        a[i] = b[i] + c[i];
```



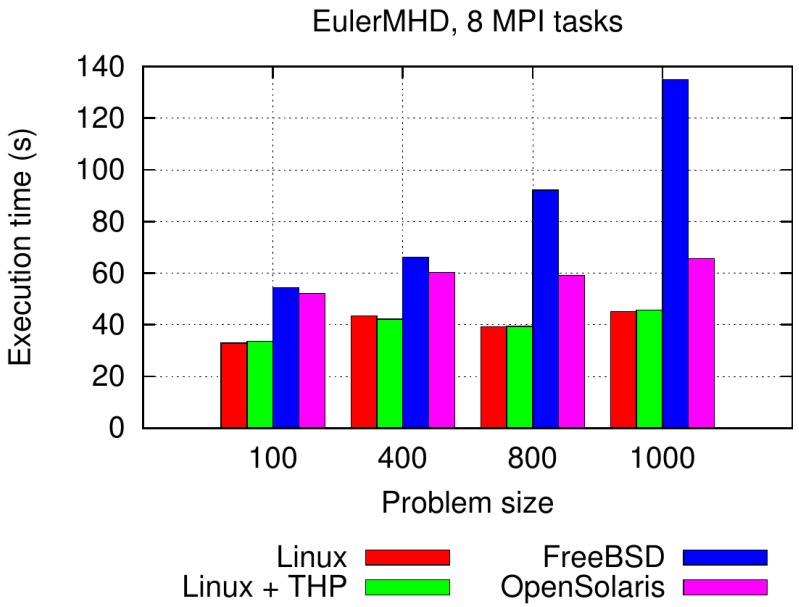
Solution

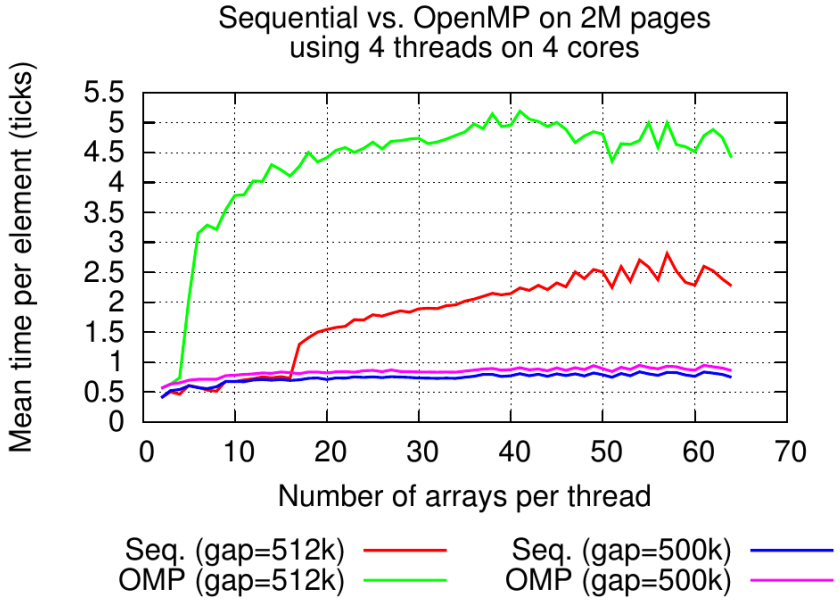
- Avoid segment **alignments** on **cache way size** (mmap / malloc)**.**
- The **Linux random** approach **prevents pathological cases**
- Do not use **regular patterns** for **page coloring** (eg. **single modulo**)
- **Huge pages** are **regular** by **hardware definition**



Impact on threads

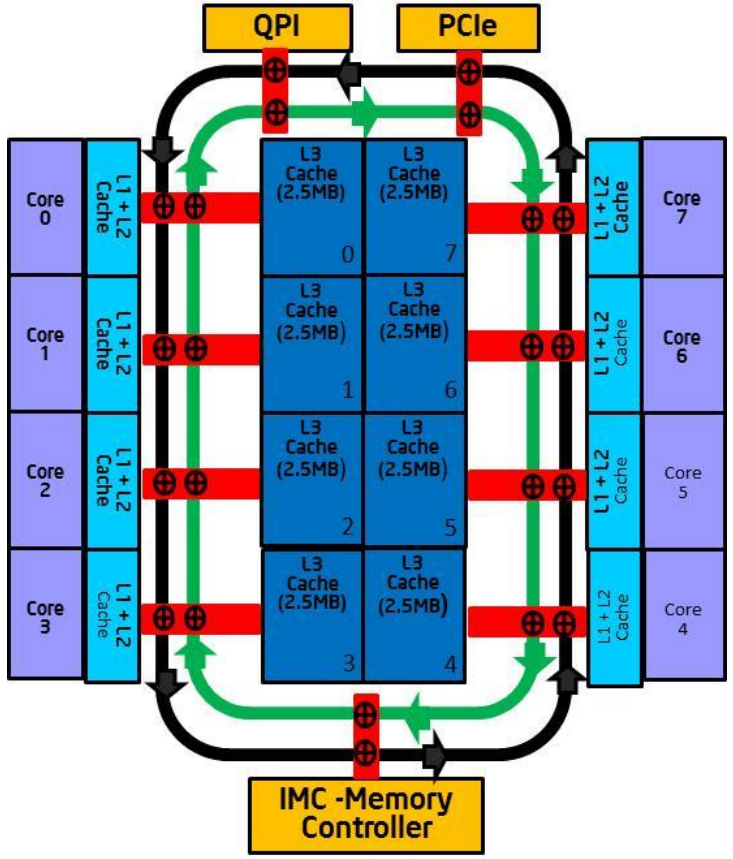
- **Larger effects** on **shared caches** with threads/processes (Nehalem)
- EulerMHD : **Slowdown** up to **3x** on **FreeBSD**
- **16** ways L3 cache implies a maximum of **4 aligned arrays** per core
- **No limit** on concurrent arrays for **unaligned allocations**





New intel L3 cache slices

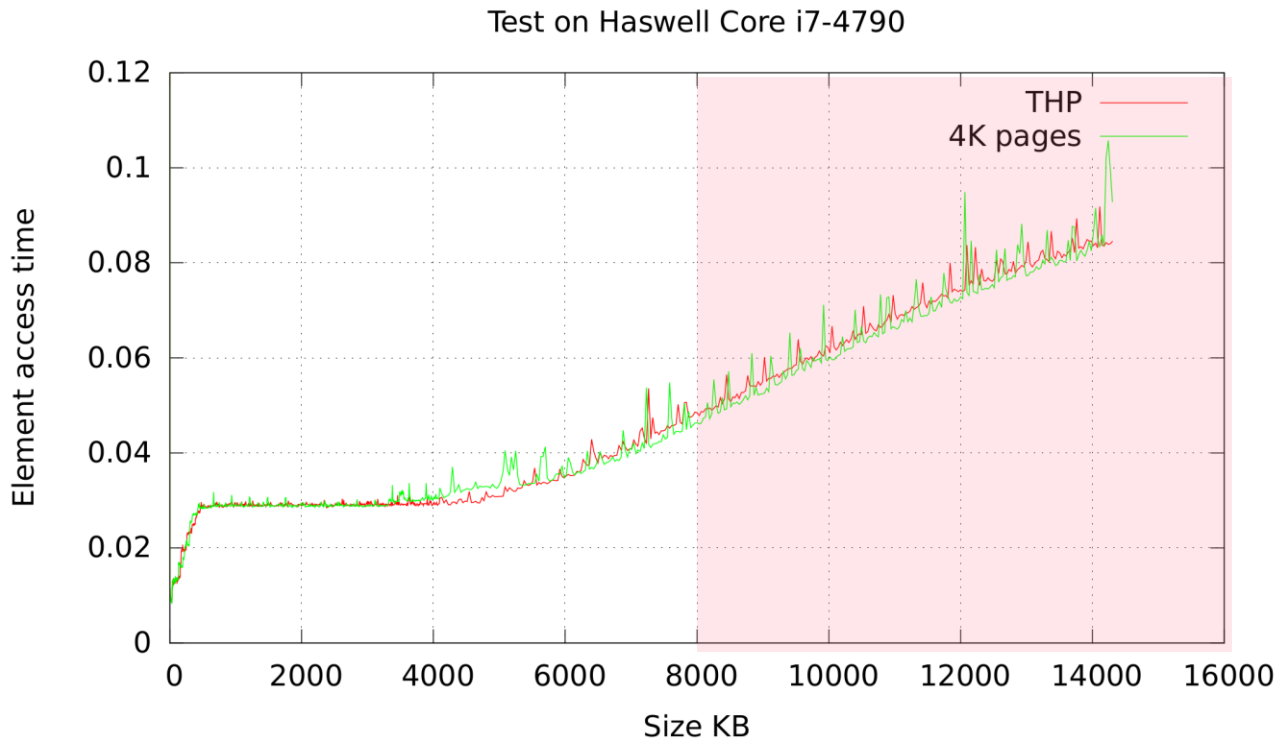
- Since **Sandy Bridge**
- L3 splits in **slices**
- **Slice** is selected by **hashing the address**
- **Each slice** has associativity with **16 ways**
- This **fix** the **coloring/alignment issue**



https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview

On today CPUs

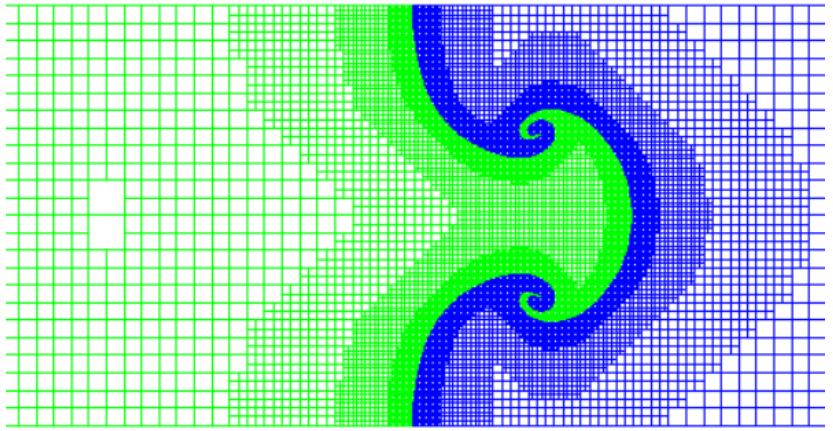
- **Not anymore** an issue **for Intel** L3 caches
  - **Change of topology**
- **AMD Zen (Ryzen)**
  - Now also use **slices**
  - Should solve the issue
- **Still** an issue on **IBM power 8**
  - L3 cache has 8 ways for 8 MB
  - **Issue present**
  - **Power 9 ?** Also “regions” in LLC ?
- For **ARM** (v7/v8) ?
  - **L2** shared associative cache
  - Issue should be present
  - But I never tested
- Issue **for L2 of all processors** !
  - Think hyperthreading with 8 ways !

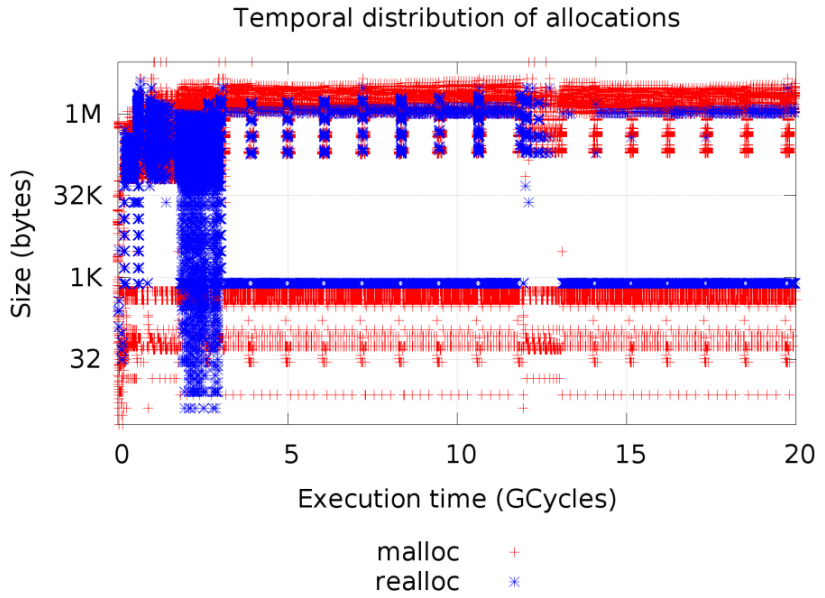


NUMA ALLOCATOR FOR HPC APPLICATIONS

Allocator performance on HPC applications

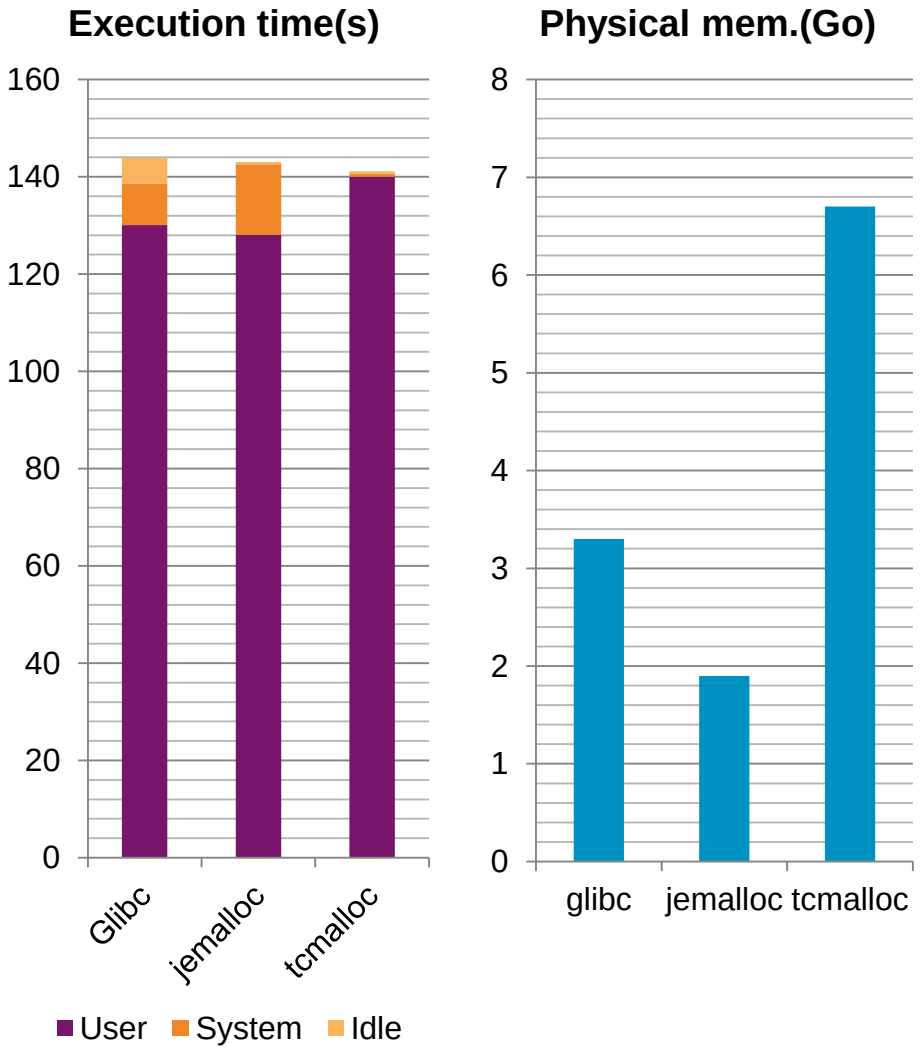
- Main interest : **malloc time cost**
- Test case : **Hera (CEA)**
  - **Adaptive Mesh Refinement** (AMR)
  - **Massive C++/MPI code** (~1 million lines).
- **Large number** of **memory allocations**
  (~75 millions / 5 minutes on 12 cores)
- **Large number** of **alloc/realloc** around **~20 MB**
- **Available allocators :**
  - **Doug Lea** / **PTMalloc** : libc Linux
  - **Jemalloc** : FreeBSD / Firefox / Facebook
  - **TCMalloc** : Google
  - *Hoard*



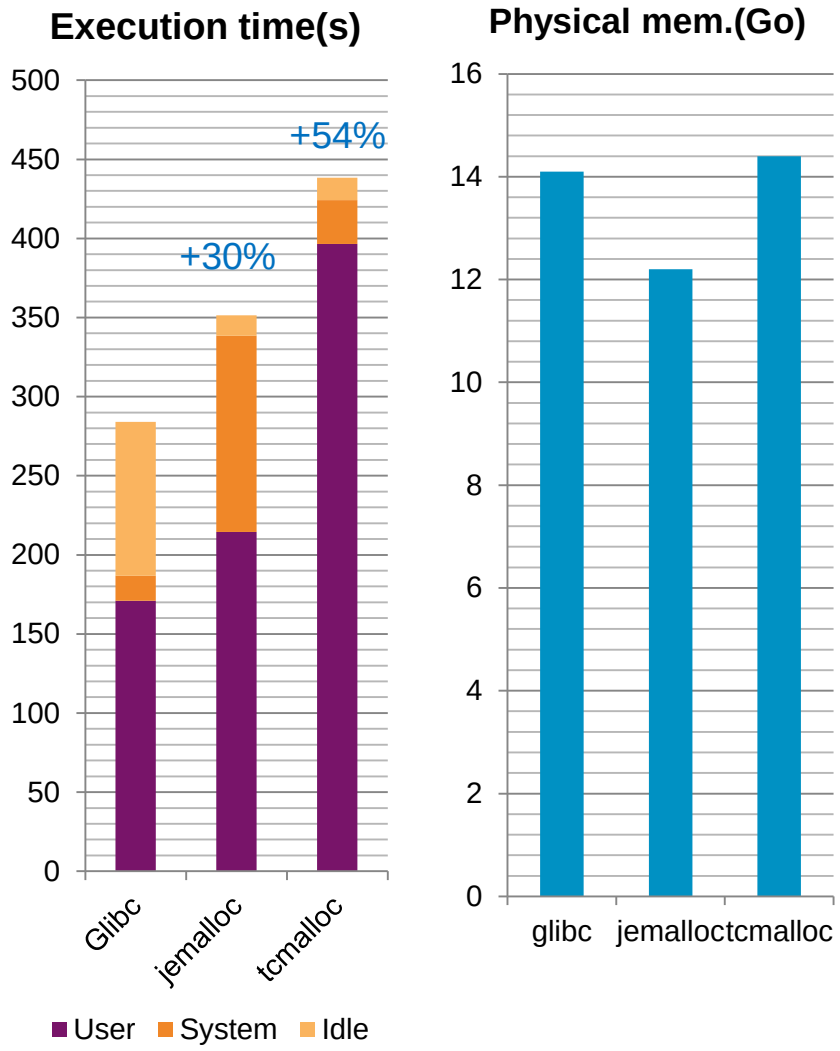


Hera preliminary results

12 cores



128 cores



How to measure malloc time

- Measurement method :

```
T0 = clock_start();
ptr = malloc(SIZE);
T1 = clock_end();
```

- Ok for **small blocks**, but not for **large** one :

```
T0 = clock_start();
ptr = malloc(SIZE);
for ( i = 0 ; i < SIZE ; i += PAGE_SIZE)
          ptr[i] = 0;
T1 = clock_end();
```

- **Lazy page allocation**.
- **Page faults** on first access.

For 4GB	Malloc	First access
Time (M cycles)	0,008	1 217

Large allocations

- Small allocation **well handled** by most allocators, **best is jemalloc / tcmalloc**.
- Cost for **large allocation : page faults.**
- **Commonly neglected**, literature mainly discuss small allocations
- Direct call to **mmap/munmap**
- **HPC applications** (expected to) use **large arrays**

Large allocations

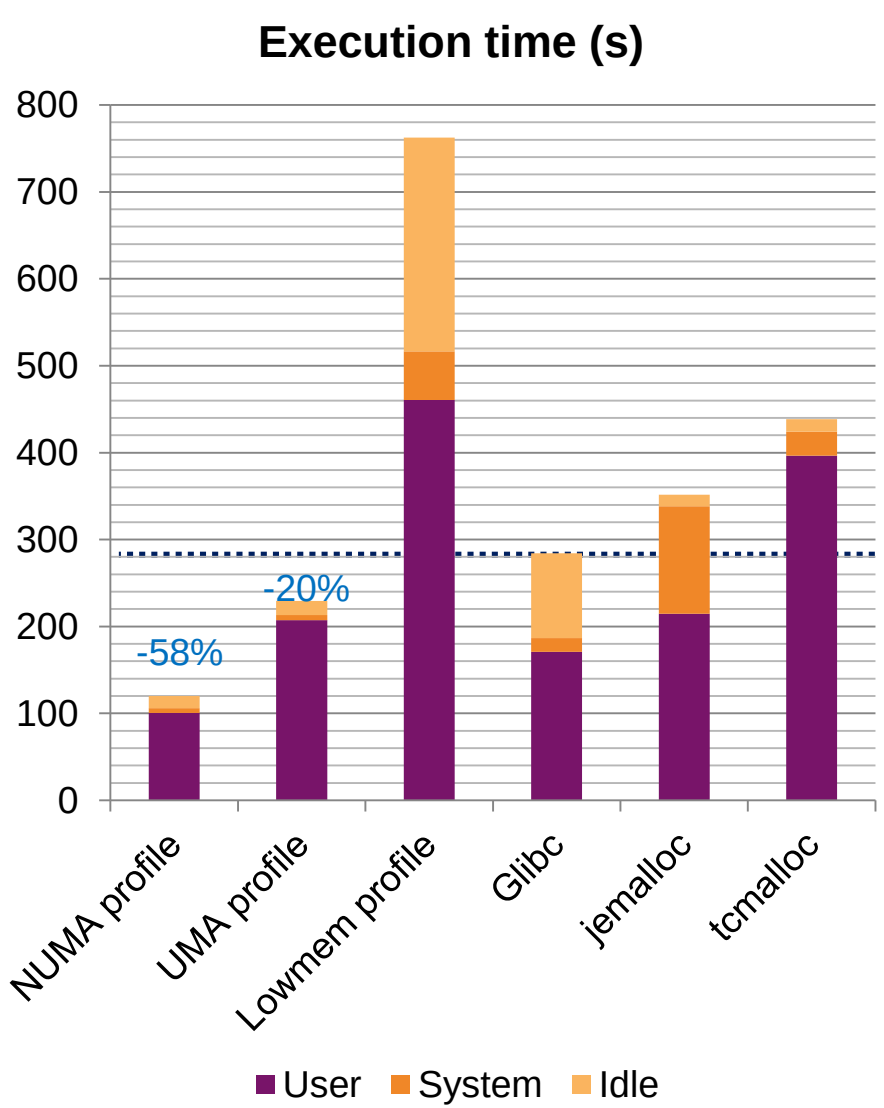
My goals :

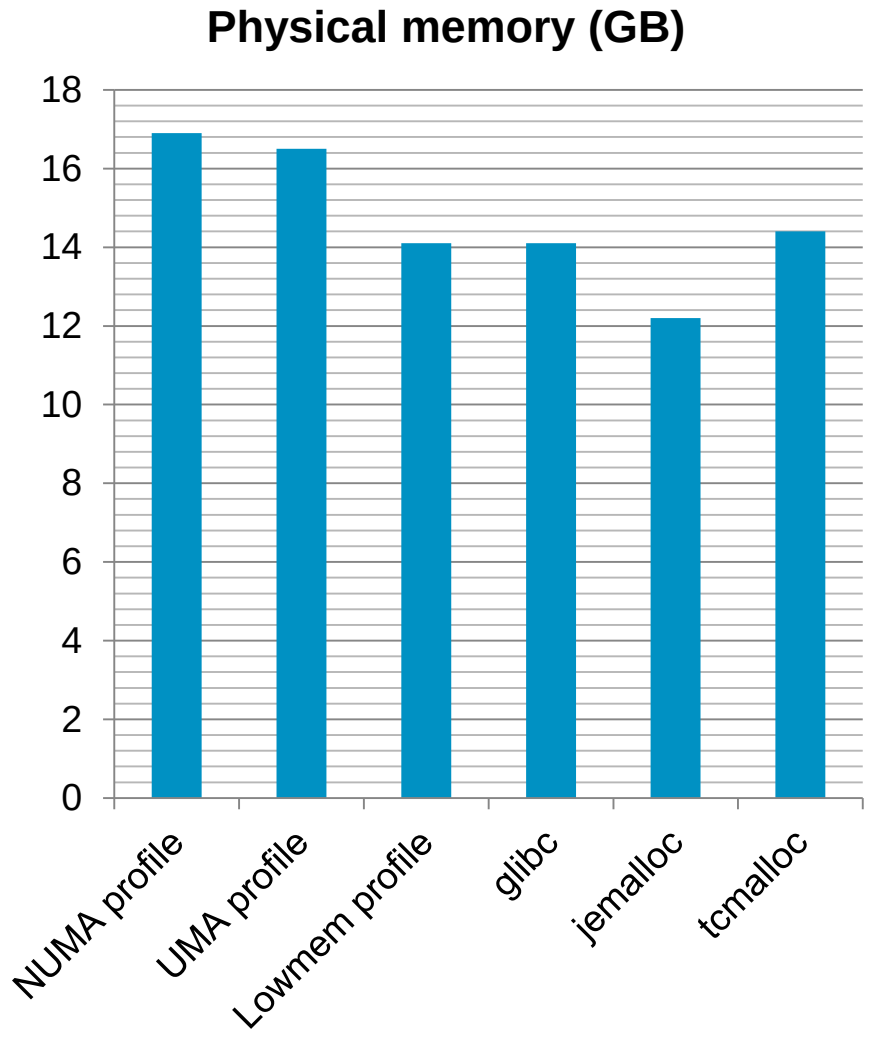
- **Recycle** large arrays
- *Avoid **fragmentation** on large segments*
- Take care of **NUMA**
- *Limit locks*

Allocator Profiles

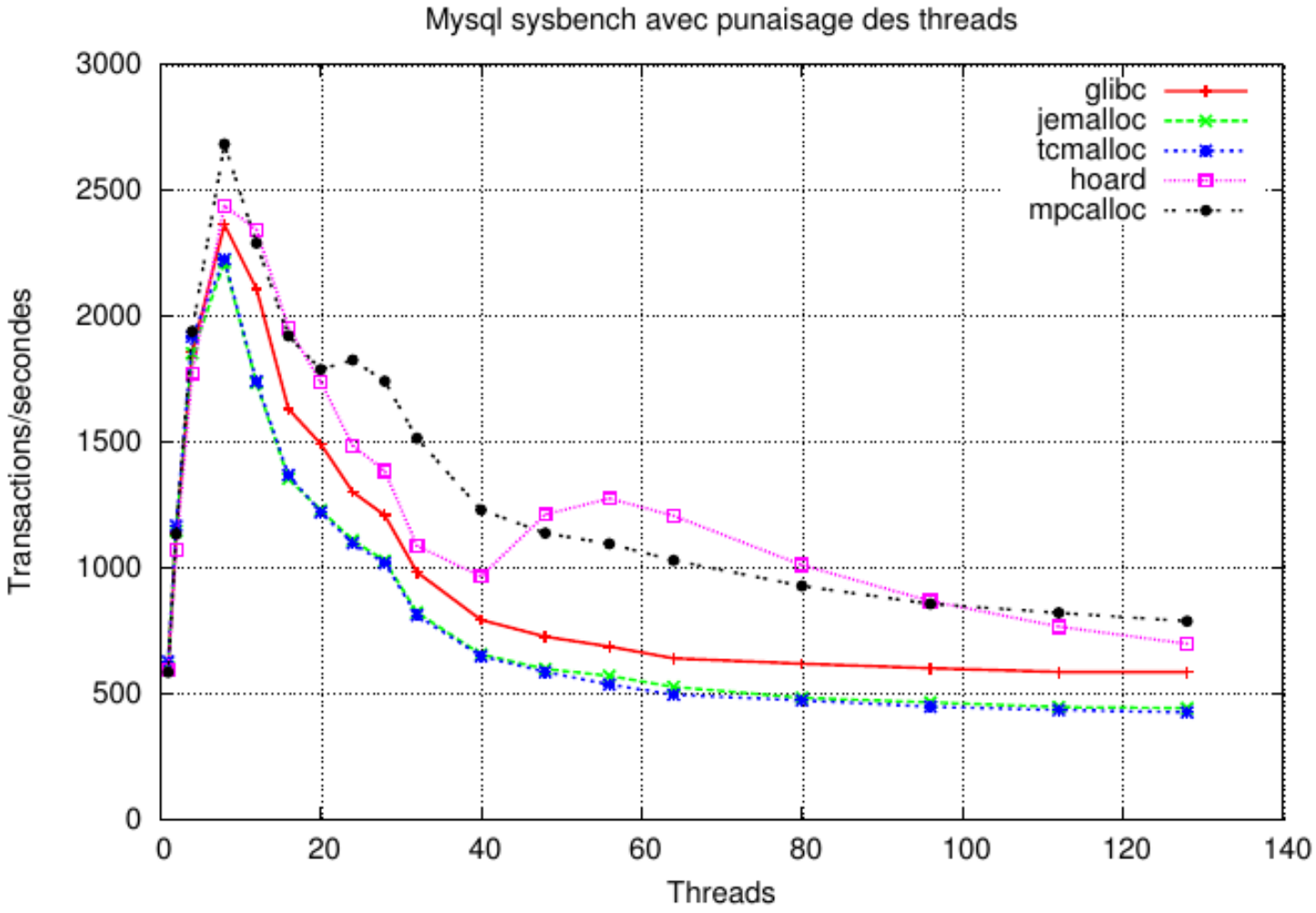
- Test allocator with **multiple profiles**
- **Lowmem** profile
  - Return memory to the OS as soon as possible
- **UMA** Profile
  - Recycle large segments
  - Disable NUMA
  - Use only one common memory source
- **NUMA** profile :
  - Recycle large segments
  - Enable NUMA structures

Hera on Nehalem-EP (128 : 4*4*8 cores)





Mysql results



PAGE ZEROING IN LINUX FIRST TOUCH HANDLER

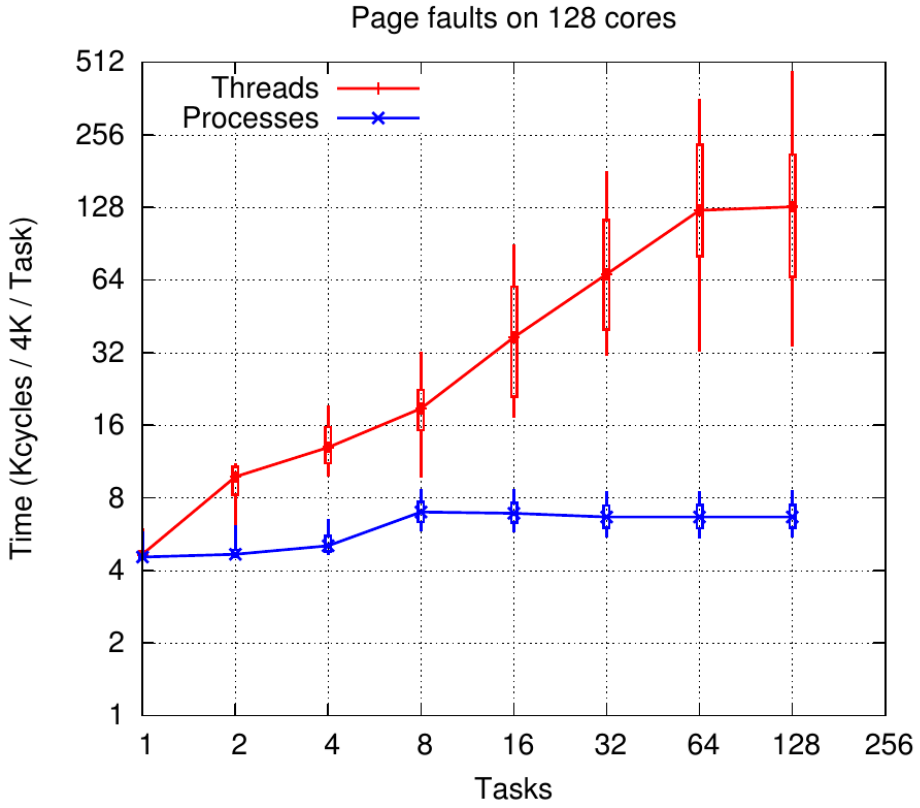
Benchmarking page faults

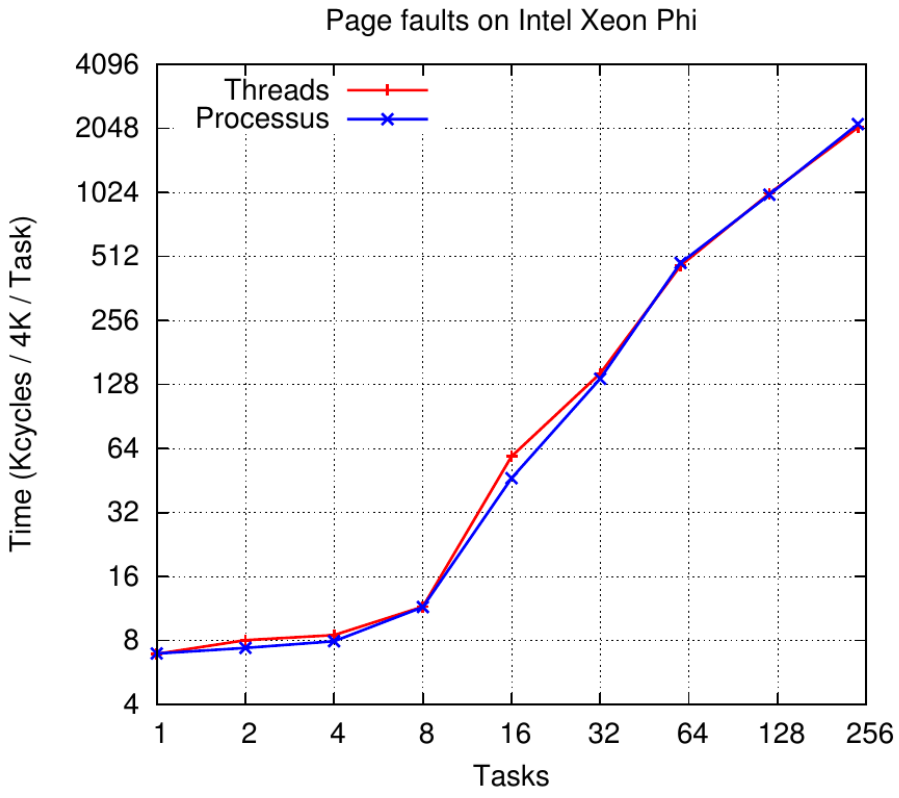
- **Page faults** are an issue for **allocation performance**
- We **previously limit them** with **large segment recycling**
- Can we **improve fault performance**?
- **Micro-benchmark** :

```
ptr = mmap(SIZE);
#pragma omp parallel for
for ( i = 0 ; i < SIZE ; i += PAGE_SIZE)
{
        TIME_DISTRIBUTION(ptr[i] = 0);
}
```

Page fault scalability

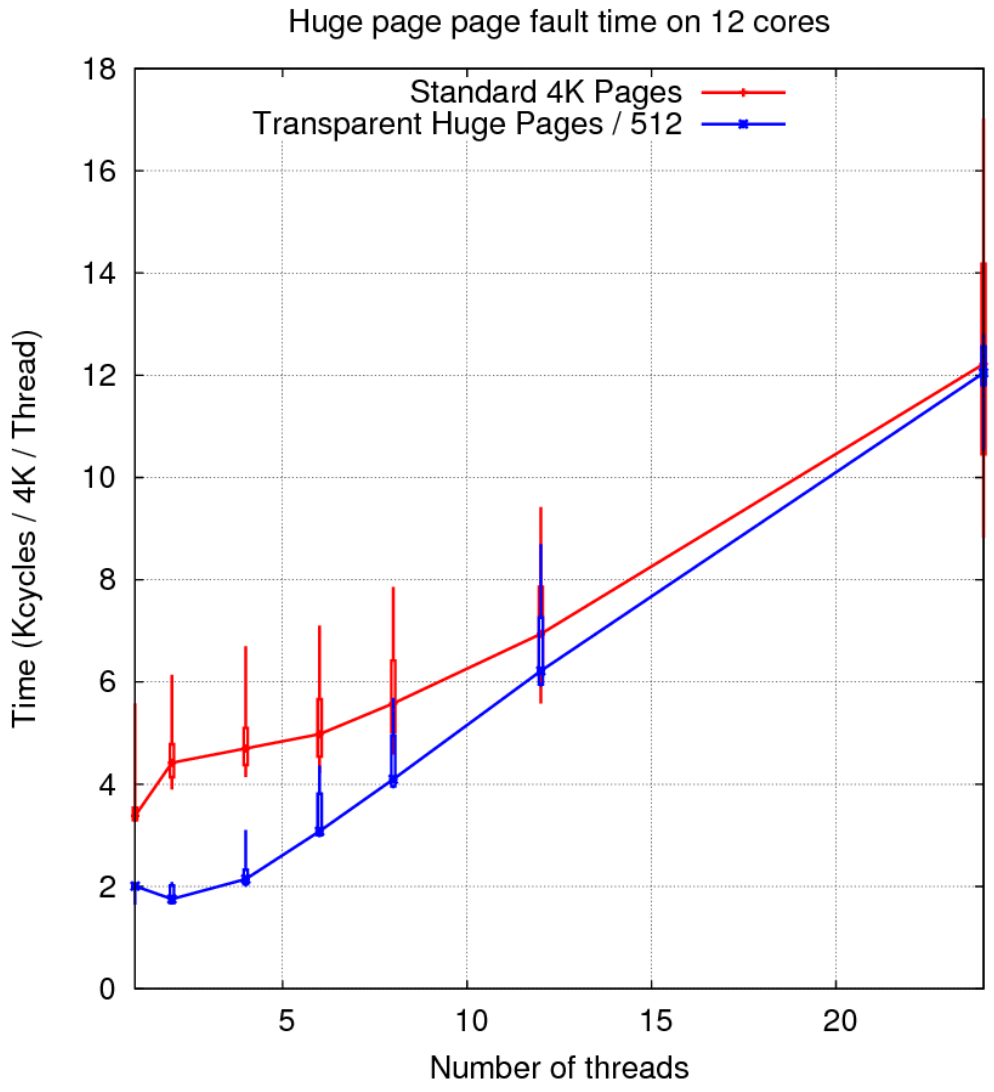
- **Are page faults scalable** ? Over threads or processes.
- Mesurement on **4*4 Nehalem-EP** (128 cores) and on **Xeon Phi** (60 cores)
- **Get scalability issue !**





Can huge pages solve this issue ?

- Standard pages: **4K**
- Huge pages (x86_64): **2M**
- **Divide number of faults by 512**
- Impact on performance ?
  - Sequential : **only 40%**
  - Parallel **: No**
- **Why ?**



What happens on first touch page fault ?

- Hardware generates an interruption to the OS
- Take **locks** on **page table**
- Check reason of the fault
- Is **first touch** from **lazy allocation**
- Request a free page to NUMA **free lists**
- **Clear the page content**
- Map the page, update the **page table**
- **Release locks**

Possible issue on Xeon Phi

~1400/3400 cycles 40%
99% for THP !

Locks, but hard to fix
(some work from
A.T. Clement ASPLOS12)

How to avoid page zeroing cost ?

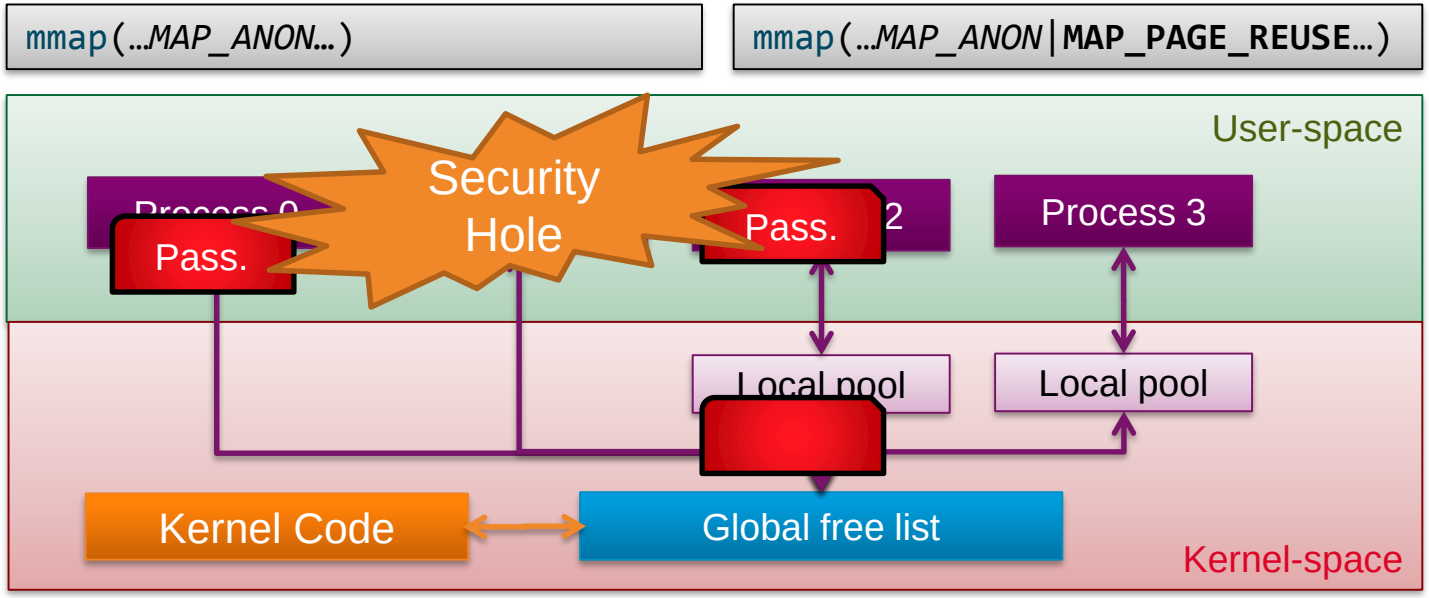
- Microsoft approach **:**
  - **Windows** uses a **system thread** to clear the memory
  - So its done **out** of **critical path**
- But **zeroing**:
  - Implies **useless work**
  - Consumes CPU **cycles** so **energy**
  - Consumes **memory bandwidth**
- **Allocation pattern** follow:

```
double * ptr = malloc(SIZE * sizeof(double));
for ( i = 0 ; i < SIZE ; i++)
        ptr[i] = default_value(i);
```

- Why not **avoid them** ?

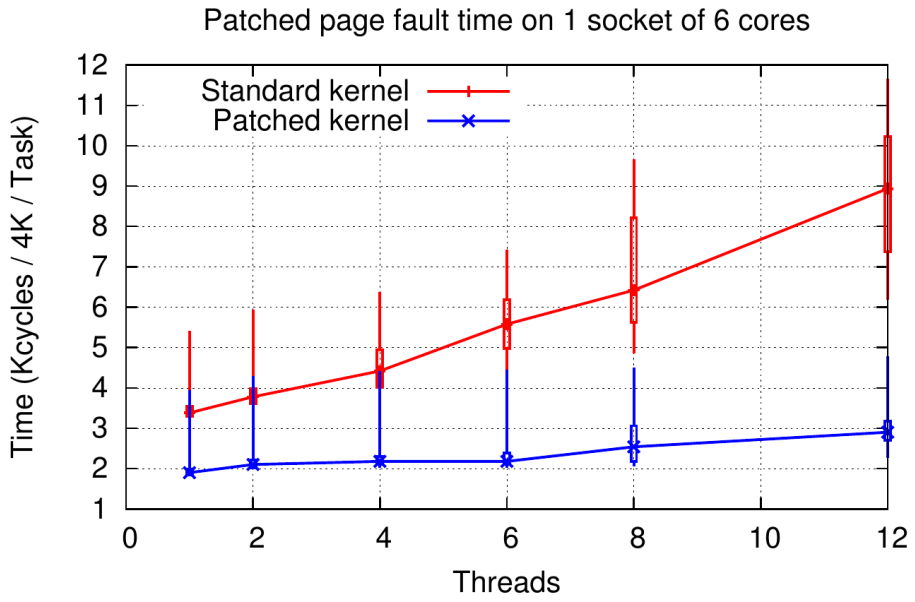
Reusing local pages to avoid zeroing

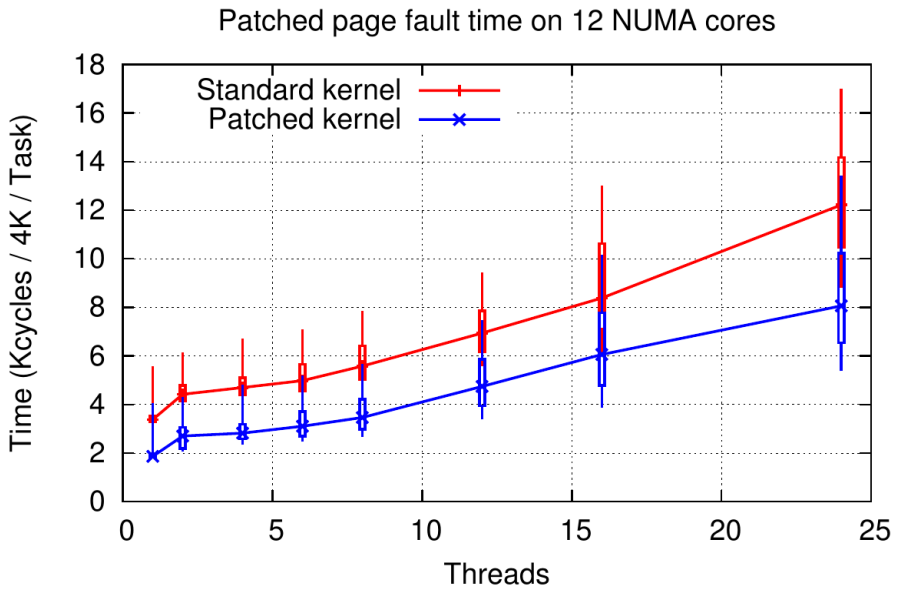
- Page zeroing is **required** for **security reason**
- It prevents information **leaks** from **another processes** or from the **kernel**.
- **But we can reuse pages locally !**
- Need to **extend** the **mmap semantic** :
- Usable by **malloc / realloc**.



Performance impact

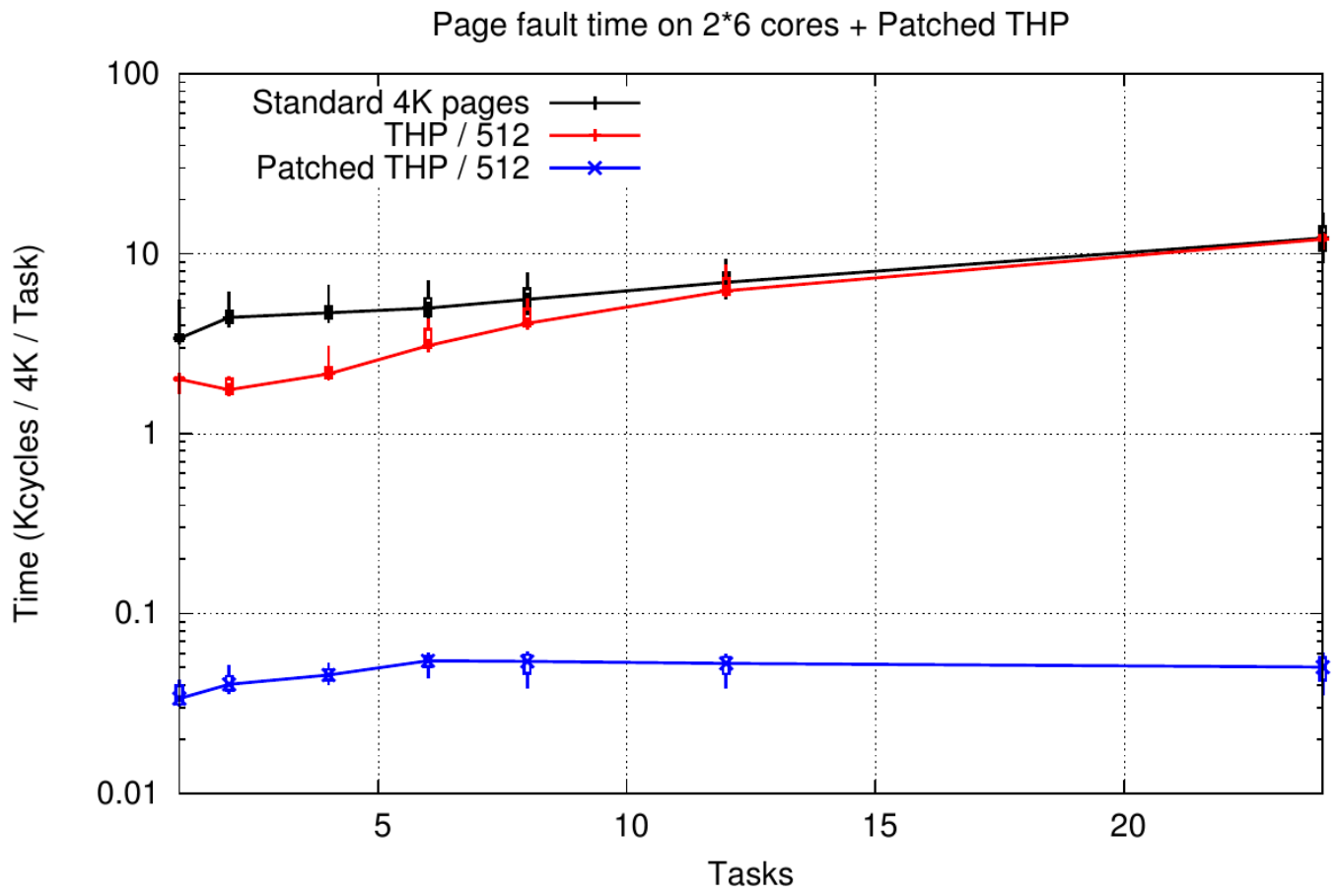
- Get the **expected improvement** on **4K pages** (40% for sequential).
- Also improve **scalability** on 1 socket
- On NUMA **locking effets become dominant for scalability**
- Get the constant improvement related to page zeroing.





Performance impact on huge pages

- **Huge pages** (2 MB) faults become **47** times faster, **60** in parallel.
- **New interest** for huge pages.



CONCLUSION

Conclusion

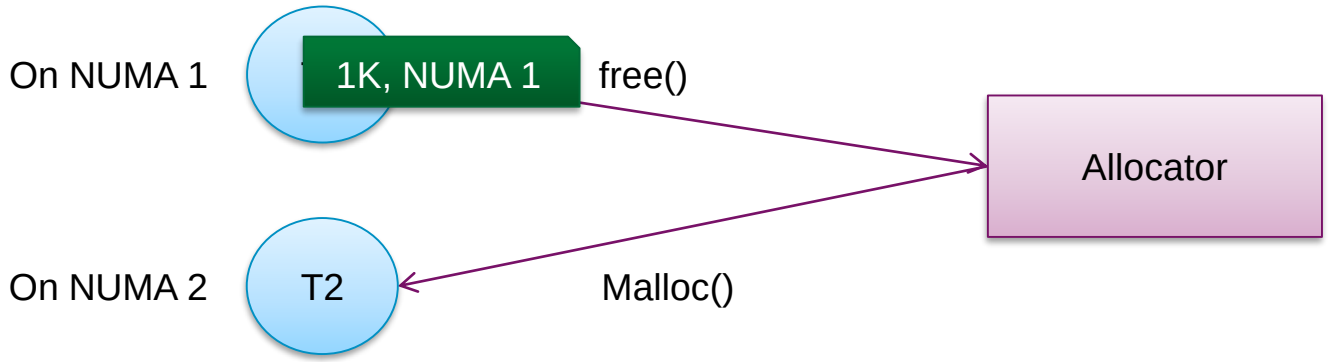
- Consider the **genius** of **peoples who invented** the **pagination** !
- Event after **60 years** of memory management we **can still do a lot !**
- Current operating systems **still have to digest** side effects of **multi-core** and **NUMA**
- Hope you know better **what is behind malloc** now !

QUESTIONS ?

BACKUPS

Malloc NUMA issue

- **Exchanges** between **NUMA nodes :**



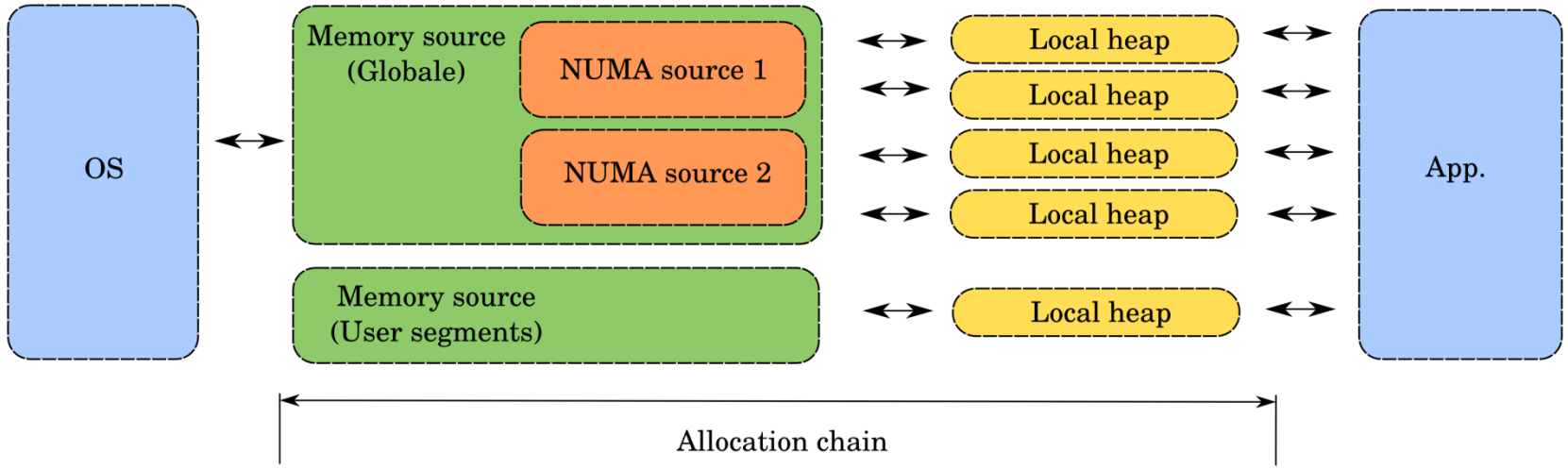
- Most **current allocators** are affected by this issue

- **Malloc** has **no information** about the **use** of allocated segments

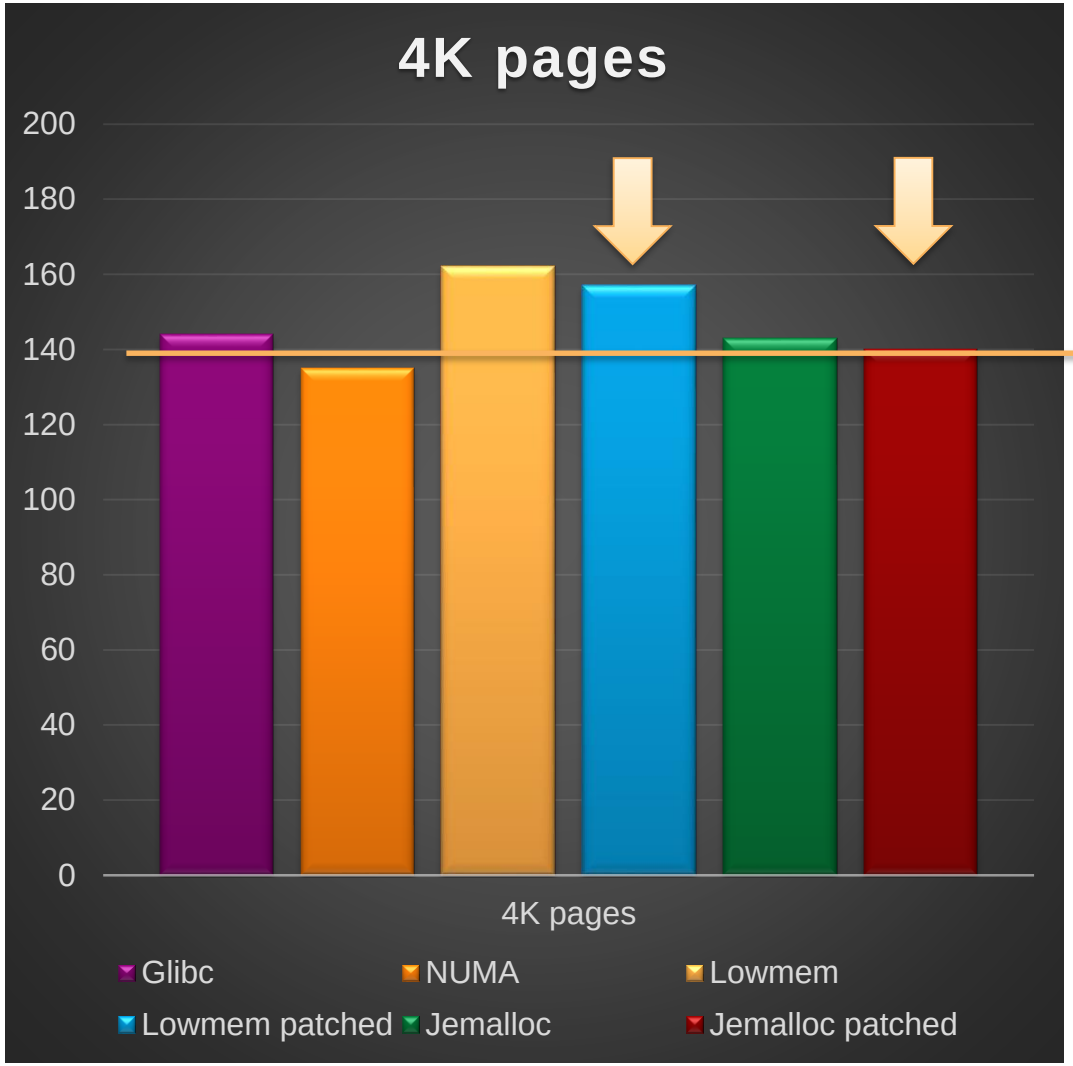
Global structure

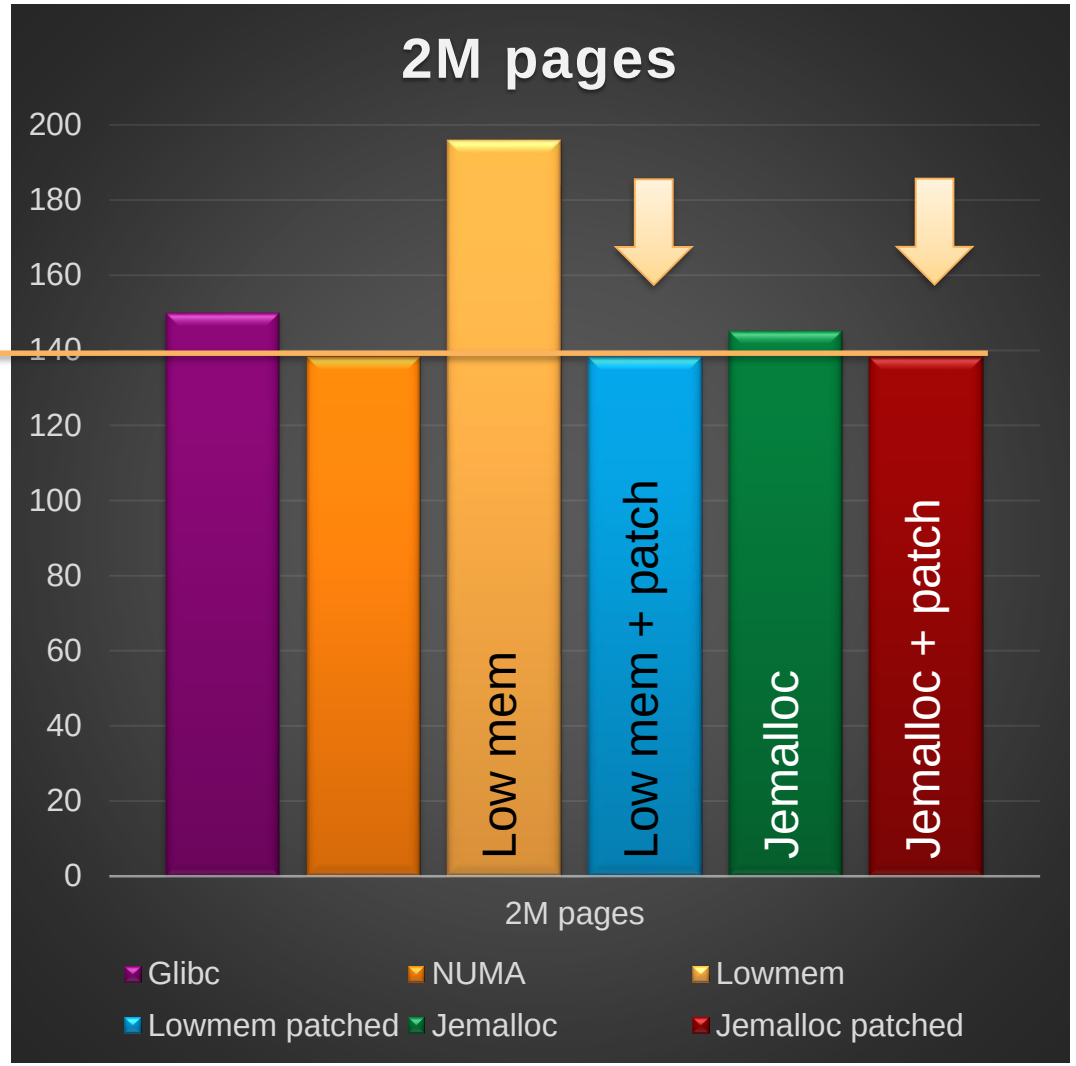
- **Memory source** :
  - Manages **requests to the OS**
  - Exchanges per **macro-blocs** larger than **2 MB**
  - Acts as a **cache** by keeping macro-blocks
  - Manages balance **performance / consumption**

- Per thread **local heap :**
  - **Lock free**
  - Manages **small chunks**
  - **Split** macro-blocs



Hera results on bi-westmere (2*6 cores)



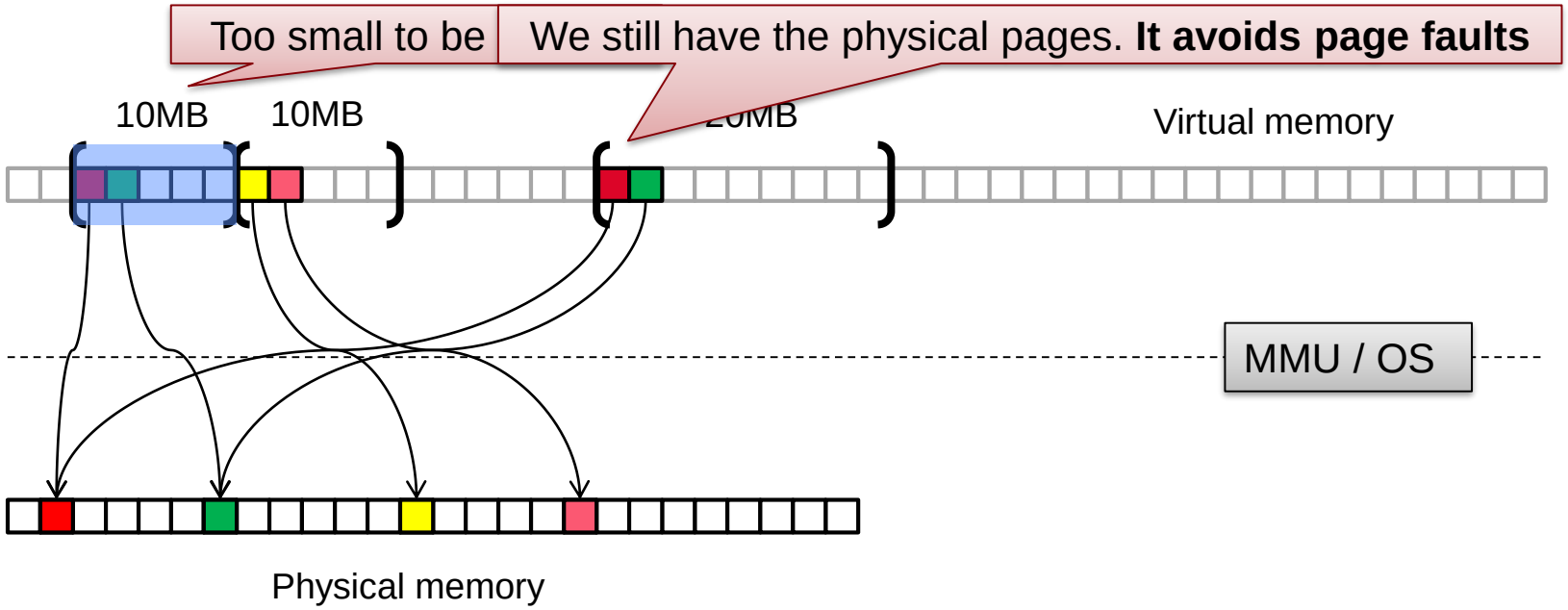


No fragmentation for large segments

- **Reuse of large segments** can induce **fragmentation**
- Example :

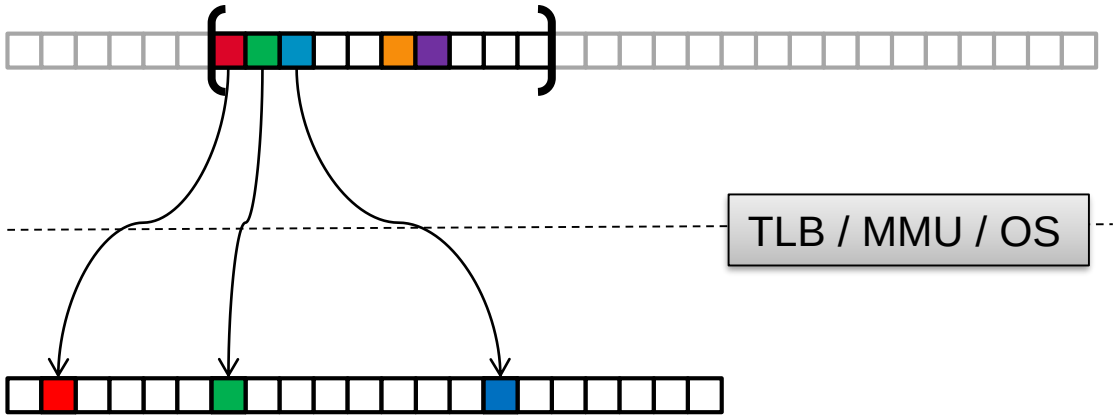
```
a = malloc(10MB);
b = malloc(10MB);
free(a);
a = malloc(20MB);
```

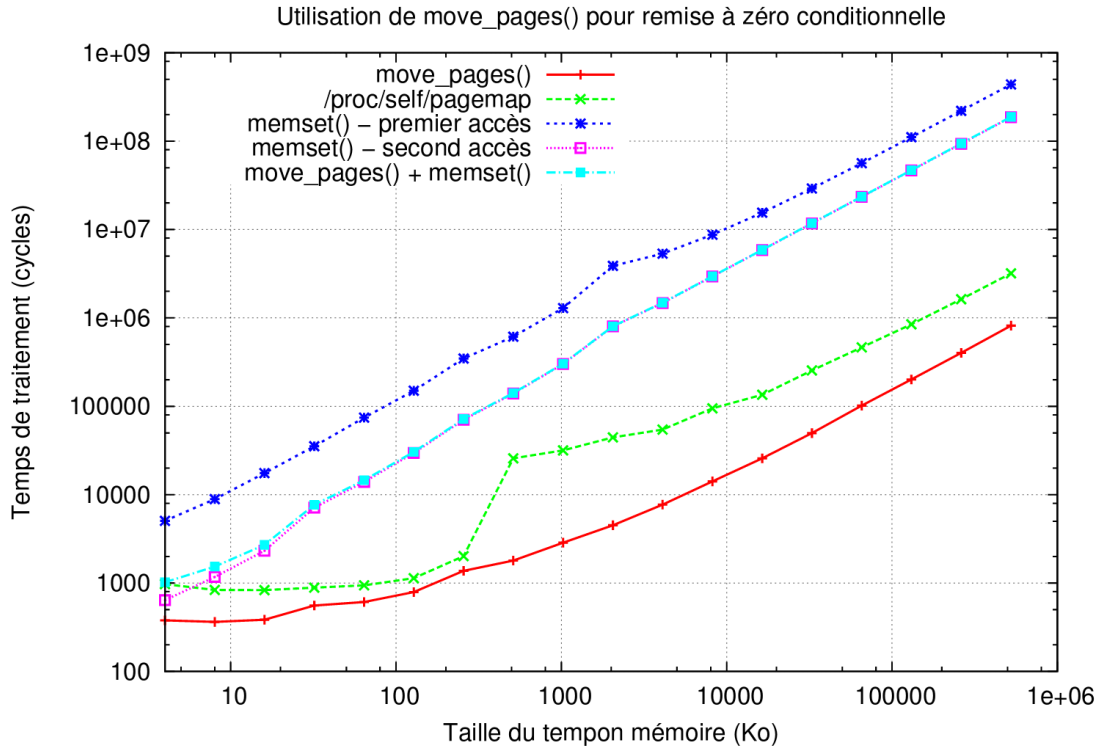
- **Can be avoided** by use of **mremap**



Calloc case

- **Calloc** need to clear all the memory to **ensure zeroing**
- One can remark that **untouched memory** will **already be cleared** by OS
- Can we **avoid** to **clear untouched pages** ?
  - **Yes**
  - We can detect with **move_pages()**
  - Or **/proc/PID/pagemap** [not anymore]





About OpenSolaris page coloring

- There is a parameter to **choose coloring policy**

- **From documentation** we can **choose within** :

ID	Nom	Description
0	Hashed VA	La couleur de la page est choisie à l'aide d'un hash de l'adresse virtuelle pour assurer une distribution des adresses virtuelles sur le cache. Est aussi ajouté un hash de l'adresse du processus afin d'éviter d'utiliser les mêmes lignes de chaches entre plusieurs instances d'un même programme.
1	P. Addr = V. Addr	La couleur de la page est choisie de sorte qu'elle soit égale à l'adresse virtuelle à un modulo près.
2	Tourniquet	Les couleurs des pages sont choisies à l'aide d'un tourniquet.

TAB. 3.1: *Table extraite des manuels OpenSolaris[3]*

- But from source code (*onnv/onnv-gate/usr/src/uts/**i86pc**/vm/vm_dep.h*), x86_64 is only Hash VA:

```
#define AS_2_BIN(as, seg, vp, addr, bin, szc)     \
      bin = (((((uintptr_t)(addr) >> PAGESHIFT) + ((uintptr_t)(as) >> 4)) \
             & page_colors_mask) >>    \
             (hw_page_array[szc].hp_shift - hw_page_array[0].hp_shift))
```

[3] Jim MAURO et Richard MC DOUGALL: Solaris Internals, core kernel Architecture

Example NUMA allocation issue

- Allocate A, B and C such as (SIZE = 128M)

```
double * A = malloc(SIZE);
double * B = malloc(SIZE);
double * C = malloc(SIZE);
```

- Init with 0:

```
memset(A,0,SIZE);
memset(B,0,SIZE);
memset(C,0,SIZE);
```

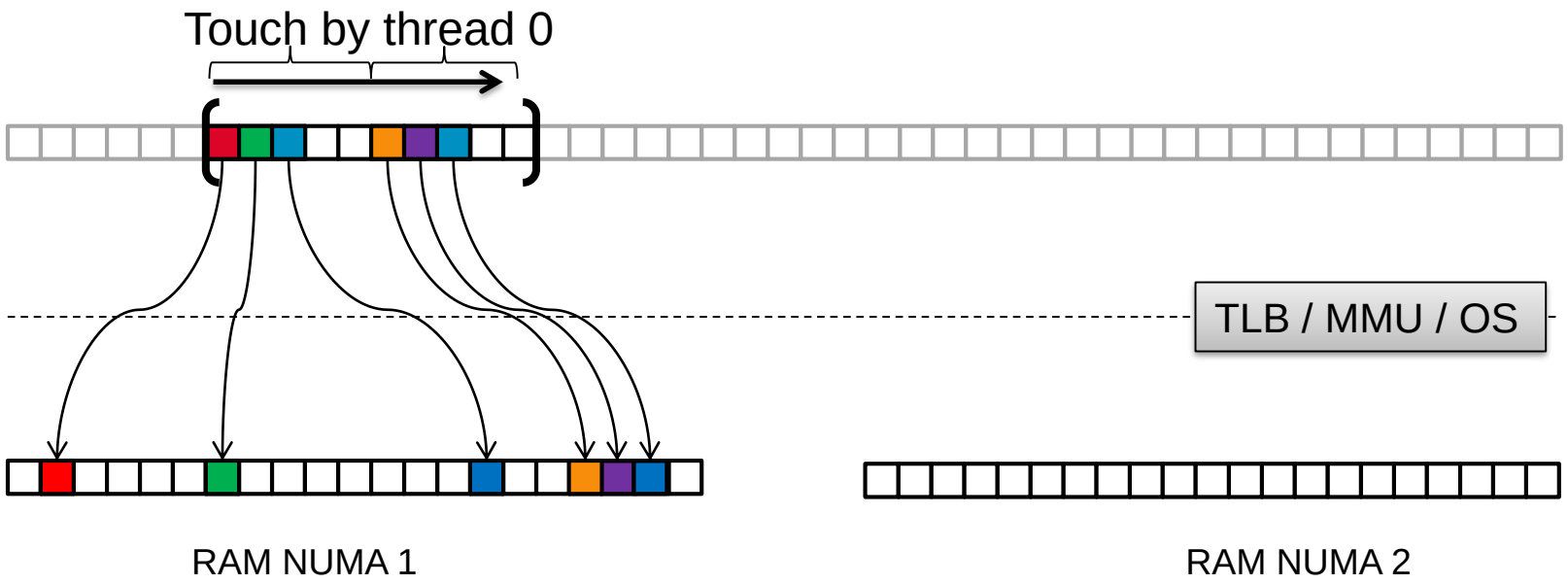
- Measure execution time on **8 threads**, **2 NUMA nodes**:

```
for (rep = 0 ; rep < 500 ; rep++)
        #pragma omp parallel for
        for (i = 0 ; i < SIZE / sizeof(double) ; i++)
                A[i] = B[i] + C[i];
```

- **What is the performance mistake ?**

Example of NUMA allocation issue

- Thread 0 call malloc
- Then is call memset and touch all the memory
- Then we access with multiple threads……
- But all the memory have been mapped on the NUMA node 0 !



What you need to do…

- Use same access for initialization than usage

```
#pragma omp parallel for
for (i = 0 ; i < SIZE / sizeof(double) ; i++)
{
		A[i] = 0;
		B[i] = 0;
		C[i] = 0;
}
```

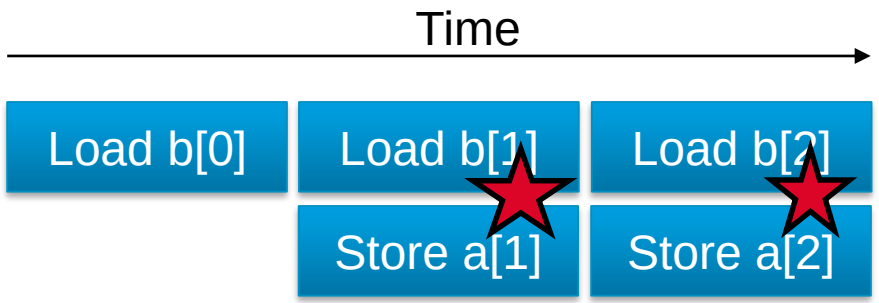
- Example on a small setup :

Init method	Elaped time (seconds)
Memset	35
#pragma omp for	15

4K aliasing (old issue but fun !)

- Consider the simple loop :

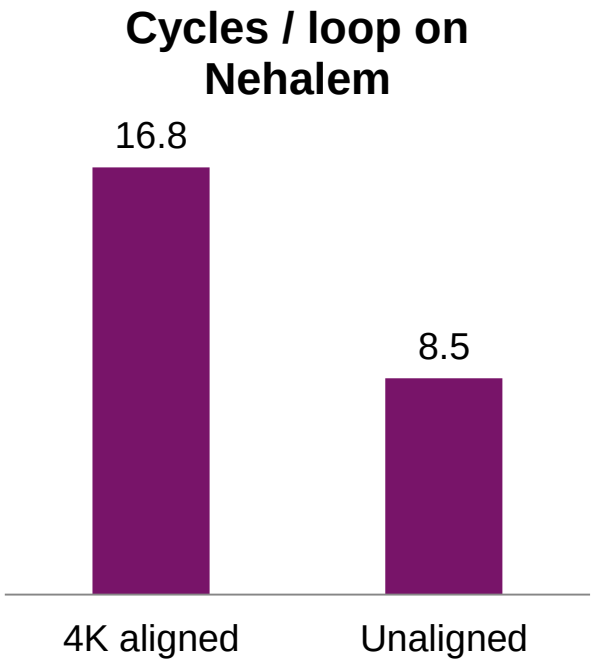
```
for (i = 1 ; i < SIZE ; i++)
        a[i] = b[i-1]
```



- If addresses verify :

```
a % 4Ko = b % 4Ko
```

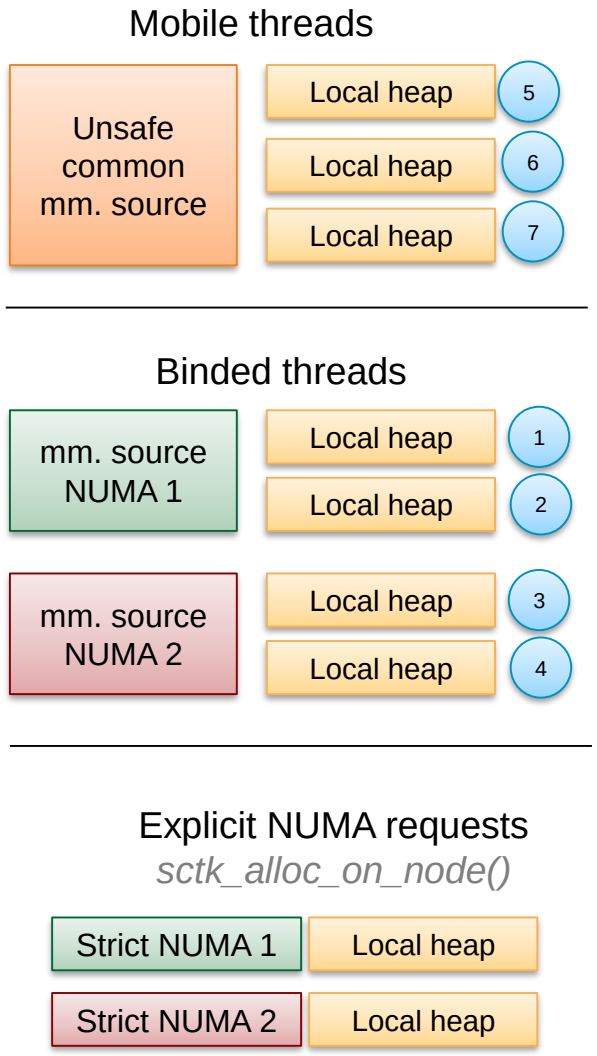
- **Processor thinks** (fast check with 12 lower bits) **addresses are equals** (alias)
- Processor do **not execute** them in **parallel** (**out of order**)



- In malloc, **direct call to mmap** generate **4K alignment by default** !
- Mainly **fixed since sandy bridge**

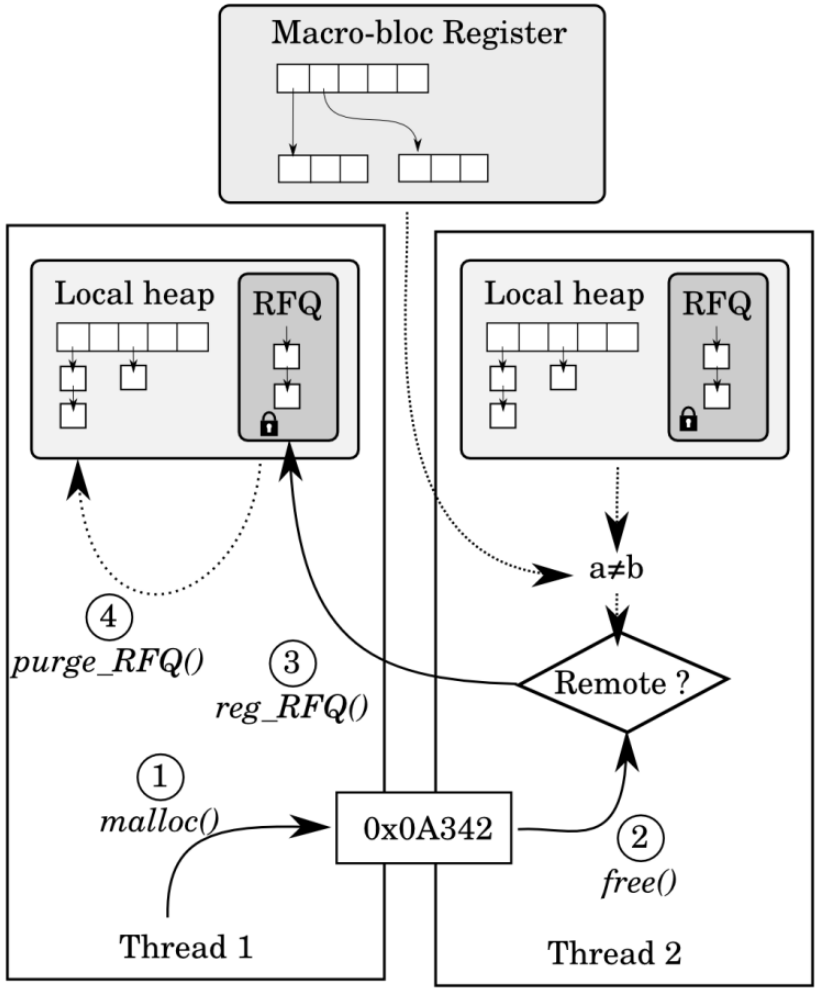
NUMA strategy

- With **standard API**, we can only **suppose local use**
- **Local heap** guarantees **NUMA isolation**
- **No exchanges** between **NUMA sources**
- **MM. sources** are **selected** with **hwloc** at **thread init.**
- **Threads** are **not binded by default**, so they **move** !
- Create memory sources with **confidence levels** :
  - A **common one** for **mobile threads**
  - **Per NUMA** for **binded threads**
  - **Per NUMA** for **explicit requests** (binded with hwloc)

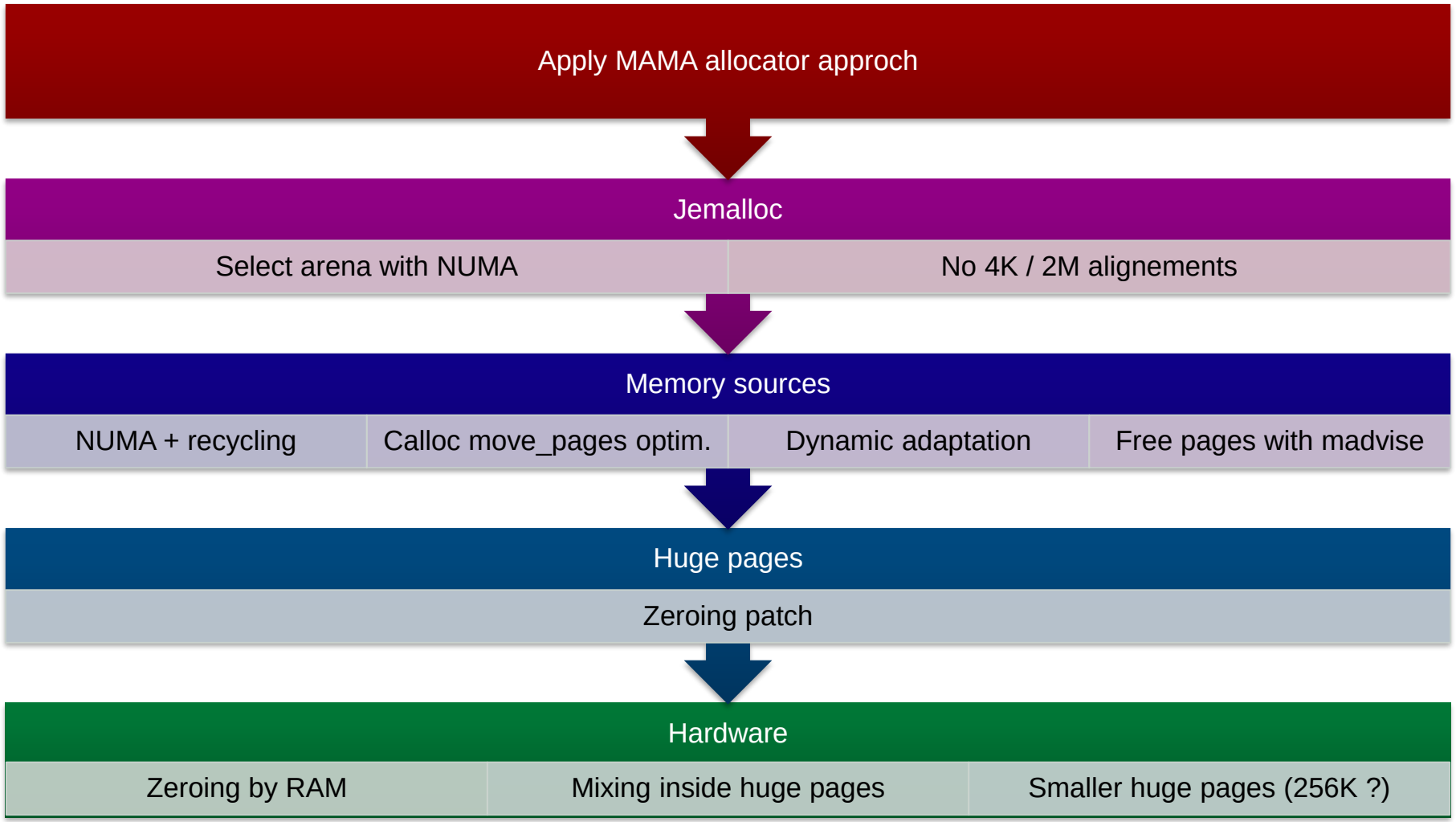


Remote free without locks

- **Remote Free** :
  - Chunk allocated by a thread.
  - Freed by another thread.
- Commonly **implies locks** on **all local heaps**
- We use a **dedicated atomic queue (RFQ)**
- **RFQ flush** on **next** memory **operation**
- Tracking **ownership** with a **lockfree register**



Ideal view of HPC memory management stack



Conclusion

Paging / alignment policies :

- **Avoid large alignments** in malloc.
- Need to avoid **regular coloring**.
- **Random paging** is **more robust** !
- **Huge pages** are regular by **hardware definition**.
- Need to **co-design malloc** and **OS** paging **policies**.

Malloc :

- Interest of **large allocation recycling**.
- **NUMA** support is required on large nodes.
- **Speed-up** of **2x** on Hera 128 cores.

Page faults (OS) :

- Observe a **scalability issue**.
- **40%** of fault time : **zeroing memory** !
- Proposal for a **semantic extension**.
- **New interest** for **huge pages** : **47x** !

Published articles :

[1] A Decremental Analysis Tool for Fine-Grained Bottleneck Detection (Partool 2010)
Souad Koliaï, Sébastien Valat, Tipp Moseley, Jean-Thomas Acquaviva, William Jalby

[2] Introducing Kernel-Level Page Reuse for High Performance Computing (MSPC 2013)
Sébastien Valat, Marc Pérache, William Jalby

Future work

Paging / coloring / alignments

- Implement **controlled non regular coloring**
- **Hardware mixing** inside **huge pages** ?
- **Linux huge pages**: be aware of **alignments** (**allocator / mmap**)
- Smaller huge page size ?

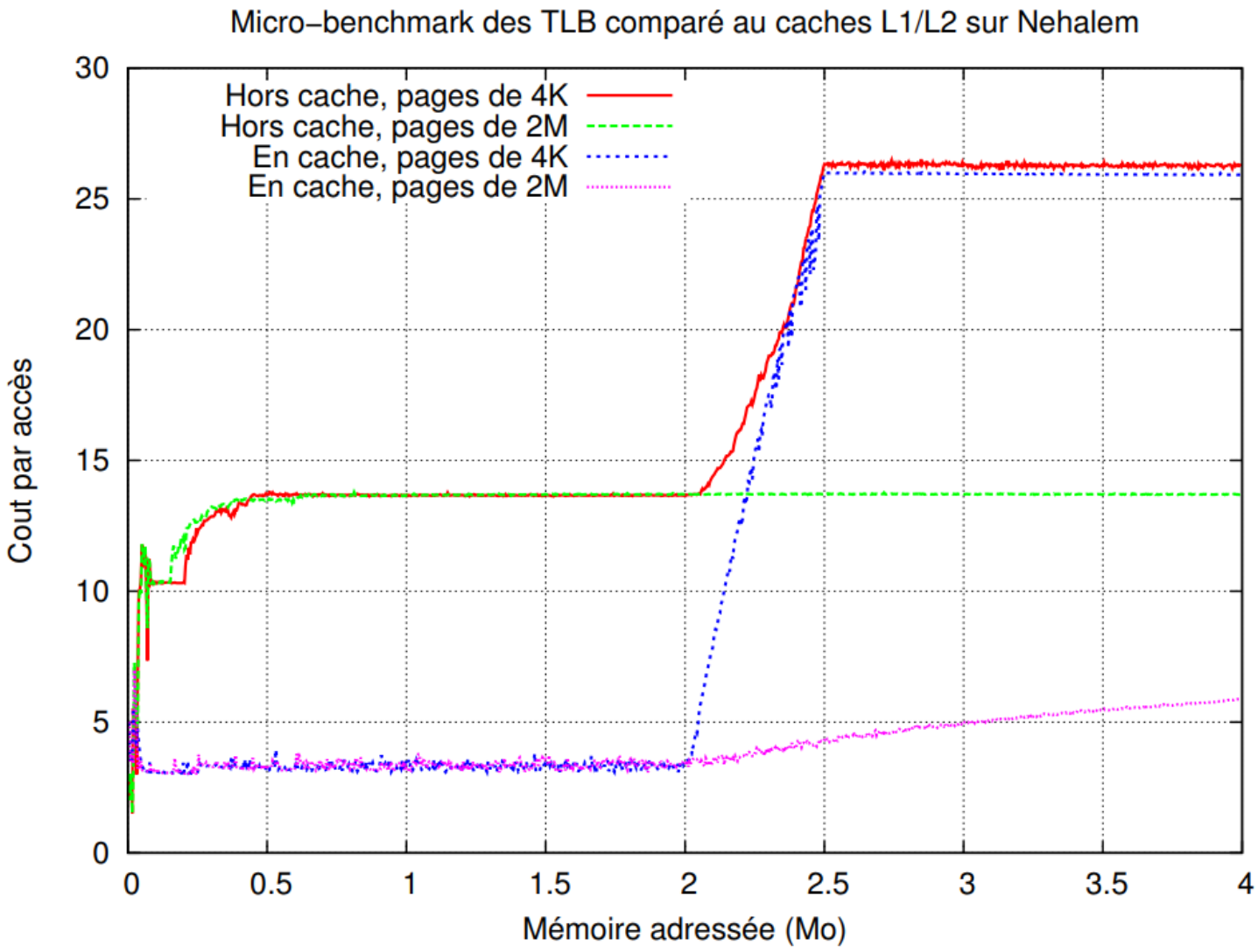
Page zeroing :

- **Cleanup** the patch (swap) and **discuss with community**
- **Hardware zeroing** done by **RAM** ?

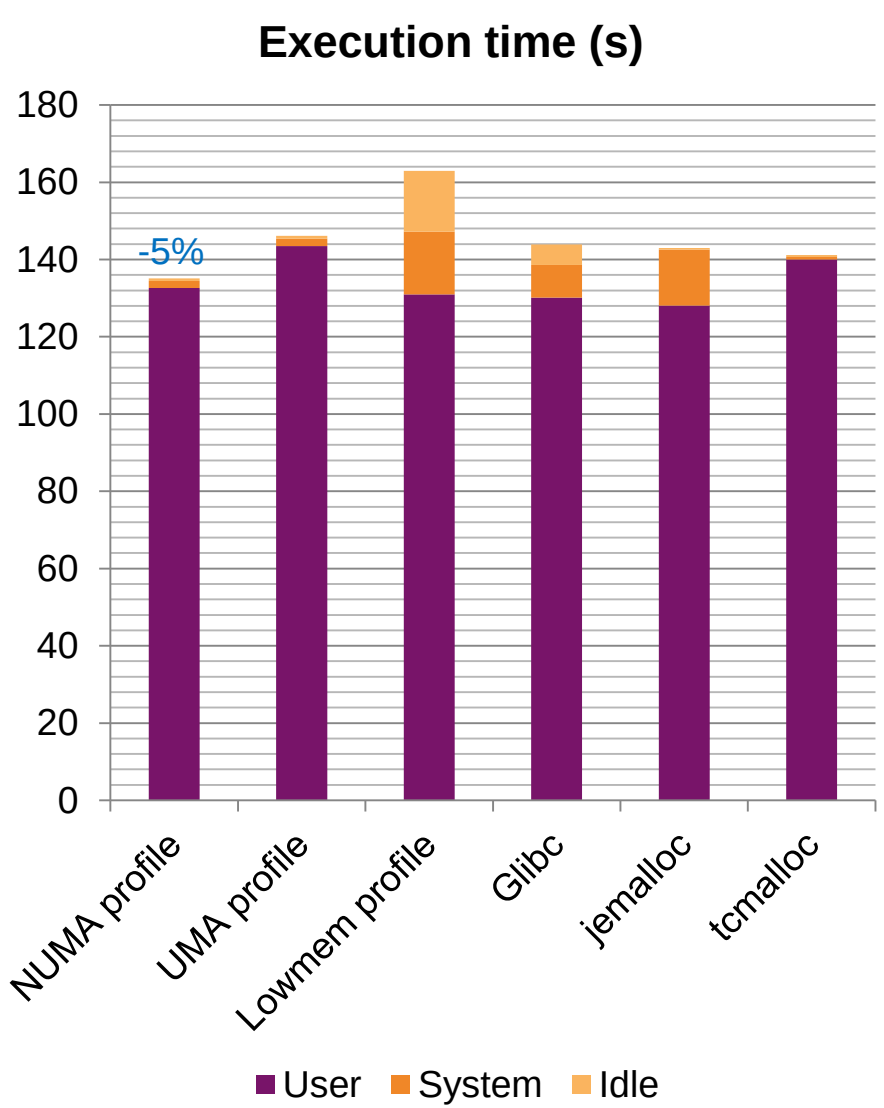
Malloc :

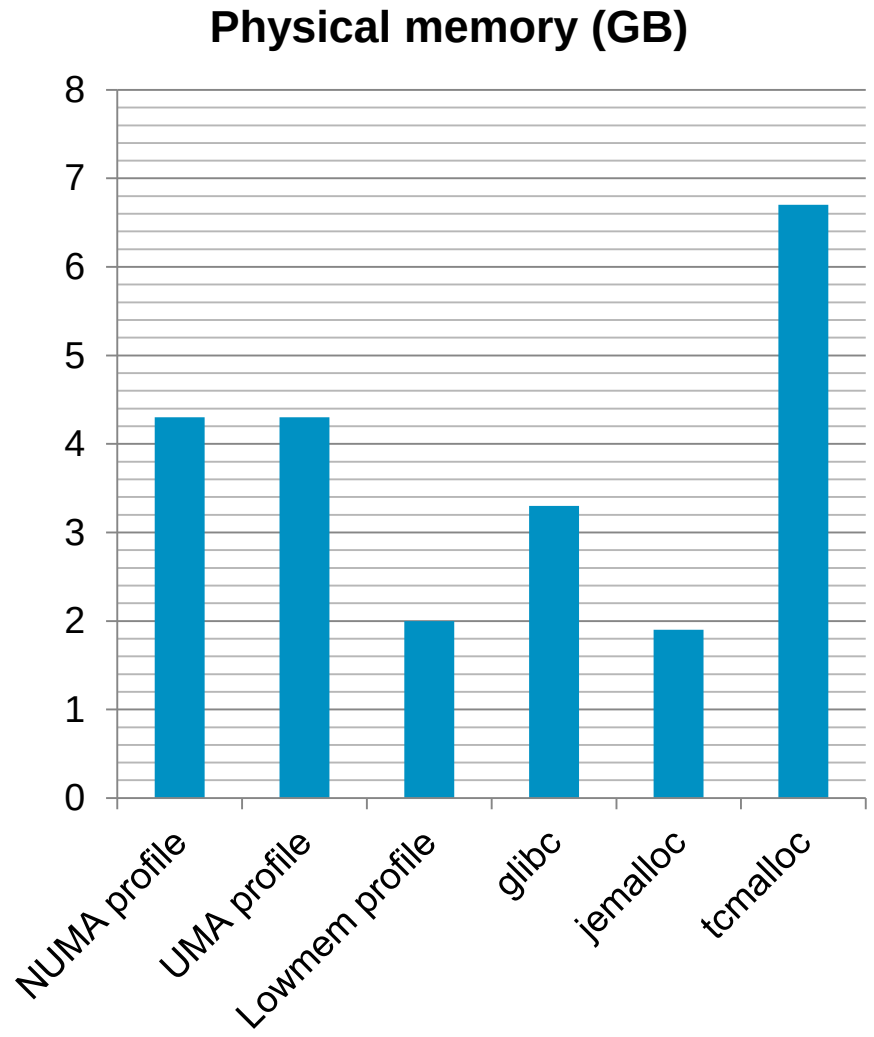
- Using our **memory sources** and **NUMA strategy** inside **Jemalloc** ?
- Mix with **TCMalloc method** (madvise(DONT_NEED)) ?
- **Dynamic control of consumption / performance ratio**

TLB effects on Nehalem



Hera on bi-Westmere (12 : 2 * 6 cores)





Hera results on bi-westmere (2*6 cores)

Standard pages (4K):

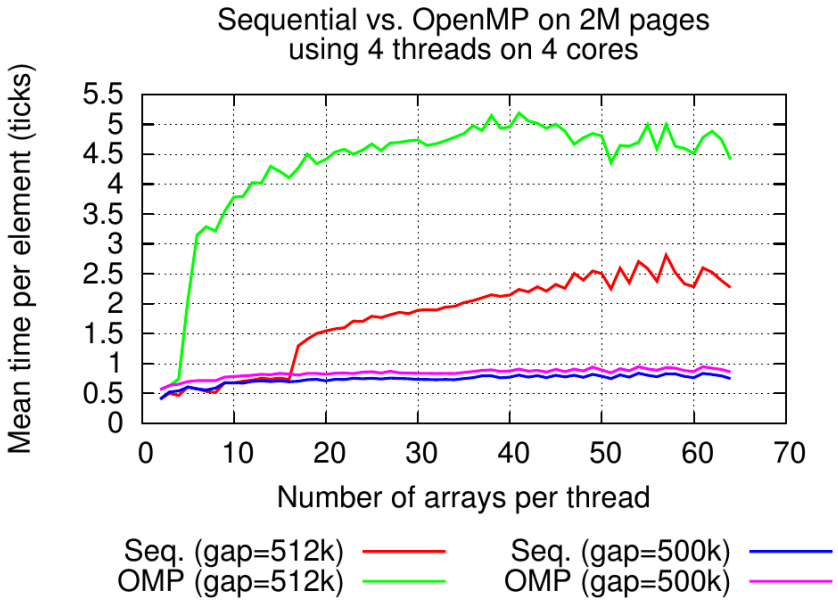
Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
Glibc	Std.	144	9	3,3
NUMA profile	Std.	**135**	**2**	**4,3**
Lowmem profile	Std.	162	**16**	**2,0**
Lowmem profile	**Patched**	157	**11**	2,0
Jemalloc	Std.	143	15	1,9
Jemalloc	Patched	140	9	3,2

Transparent Huge Pages (2M):

Allocator	Kernel	Total (s)	Sys. (s)	Mem. (GB)
Glibc	Std.	150	13	4,5
NUMA profile	Std.	**138**	**2**	**6,2**
Lowmem profile	Std.	196	28	3,9
Lowmem profile	**Patched**	138	3	3,8
Jemalloc	Std.	145	15	2,5
Jemalloc	Patched	138	6	3,2

Solution

- The **Linux random** approach **prevents pathological cases**
- Do not use r**egular patterns** for **page coloring** (eg. **single modulo**)
- **Huge pages** are **regular** by **hardware definition**
- **Malloc** must **take care** of OS **paging strategy**
- **Malloc** must avoid **too large alignments**
- Existing **similar cases** for **4K alignments** (eg. L1 caches, 4K aliasing)



Kernel-space VS. user-space memory pools

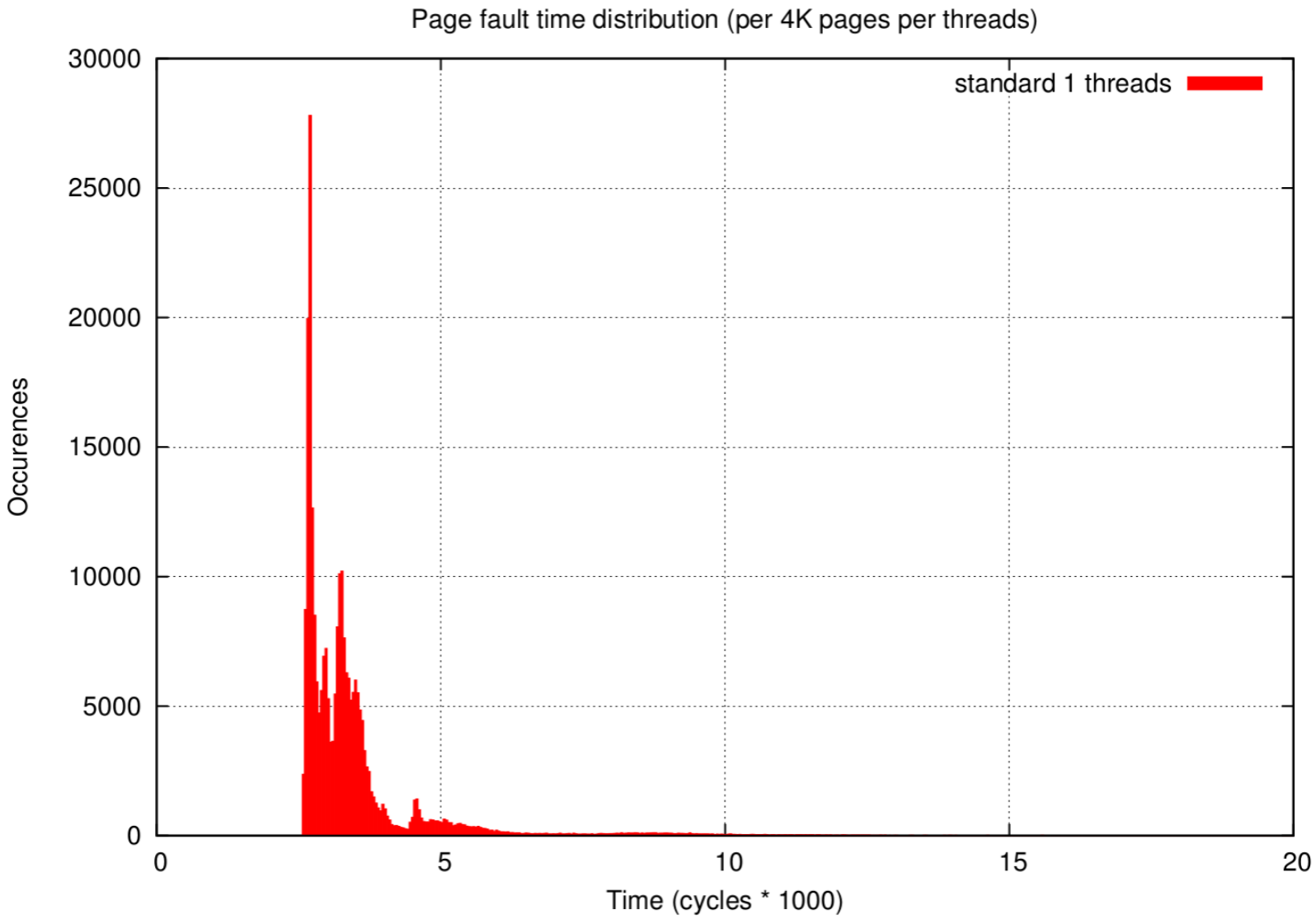
Kernel-space advantages:

- Control the **physical memory**, not virtual one
- Follow the **real access pattern**
- **NUMA support** at page level, not segment
- Buffered memory **can be reclaimed** by kernel.

Limitations:

- **More efforts** to implement.
- Do not remove the **interruption** and **locking costs**

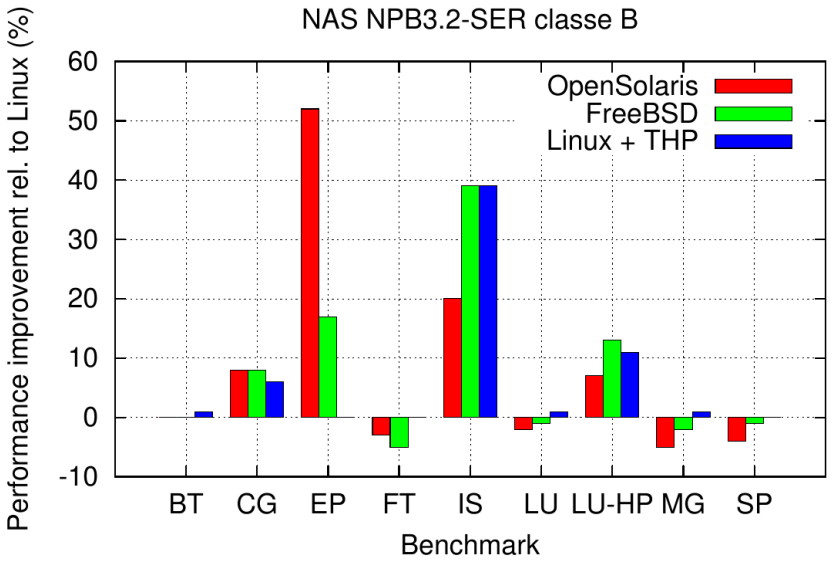
Example of distribution

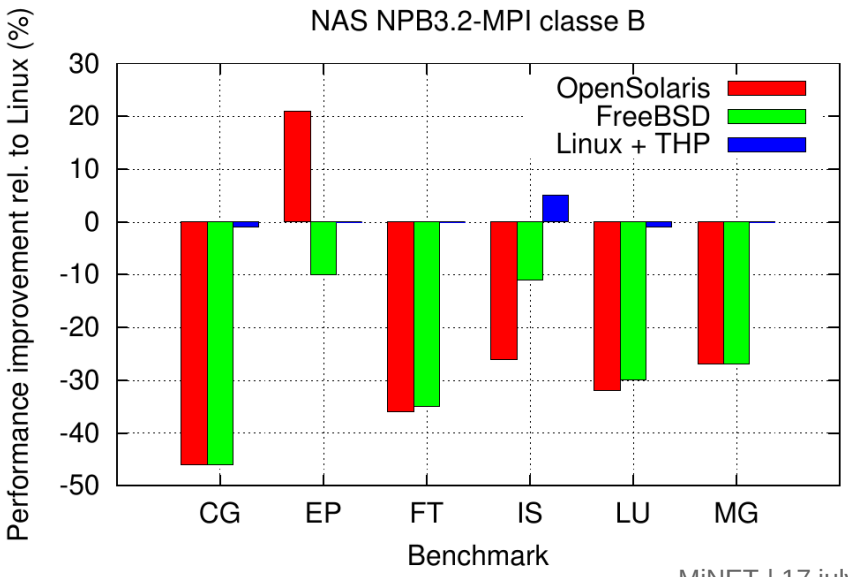


OS strategies comparison

- Each **system** has its default paging **strategy**:
- Is **Linux** slower due to **random paging** ?
- Tested architecture : **Nehalem bi-socket**
- Use a fixed compile chain : **GCC/Binutils/MPI/BLAS**

OS	Strategy
Linux	4K random
OpenSolaris	Page coloring
FreeBSD	Superpages





Report a list of similar issue

- Need to take care of **large alignments** on **regular page coloring**
- **Huge pages** are regular by **hardware definition**
- **Malloc** and **OS** politics **interact.**
- Studies **must consider the two.**
- We **reported other similar issues** (see the manuscript) :
  4K aliasing, L1 and TLB associativity

Impacte	Nom	Alignement	OMP	OS	Pages	Condition	Solutions	Probabilité
LL	Fuite dernier niveau de cache	-	-	Oui	4kB	– Utilisation de l'ensemble du dernier cache.	color, nrcolor, huge ou smcache	Élevé : Linux, Faible : SunOS
	OpenMP sur coloration régulière	$LLSS$	Oui	Oui	4 Ko	– SBA aligné relativement à $LLSS$ – $NBS > LLASSO$ – $NBTH <= CPUTH$	16bp, 4kp, nrcolor, nrsplit ou chnbs	Élevé : SunOS, Null : Linux
				Non	$>= LLSS$		16bp, 4kp, nrsplit ou chnbs	Moyen
L1 ? ,LL	Pagination régulière	$LLSS, L1SS?$	Non	Oui	4 Ko	– $NBS > LLASSO$ – SBA aligné sur $LLSS$ (ou à $L1SS?$)	16bp, 4kp, nrcolor ou chnbs	Élevé : SunOS, Null : Linux
				Non	$>= LLSS$		16bp, 4kp ou chnbs	Moyen
L1	Conflits Load/Store	4 Ko	Non	Non	???	– Utilisation d'accès de type a[i] = b[i-1]. – Tableaux alignés sur 4 Ko.	16bp ou chacc	Élevé
TLB, L1	Limite des PDE	PDEASIZE	Non	Non	4 Ko	– $NBS > TLBASSO$ – BSA aligné sur $TLBSASIZE$ – BSA distants de plus que $PDEASIZE/NBS$	16bp, 4kp ou chnbs	Faible
hline TLB	Limit d'associativité du DTLB	$TLBSASIZE$	Non	Non	4 Ko	– BSA aligné sur $TLBSASIZE$ – $NBS > TLBASSO$	16bp, 4kp ou chnbs	Moyen