# Impact of the Operating System Memory Allocation on CPU Intensive Applications

Sébastien Valat

## ABSTRACT

In this paper we studied the impact of FreeBSD, OpenSolaris and Linux paging strategies on CPU intensive applications. We observed performance gaps up to 50% between those systems, even on computational loops. Hence, with such a large impact, we cannot ignore the Operating System while doing performance analysis. Unexpected interactions between cache associativity, paging and malloc policies can result in significant slowdown. Page coloring algorithms can be improved in order to avoid such bad interaction between system layers. As it cannot be done for huge pages, we recommend to also improve malloc implementations. Depending on the design, wrong placements tend to be the default as for FreeBSD. As Linux recently introduced a better huge page support it is important to take care to malloc implementations not fall in this issue. We also observed larger degradation on too regular paging policy in multi-core environment with shared caches with up to 80% performance loss on some parallel NAS.

## 1. INTRODUCTION

On modern architectures, memory is still a critical point for performances. Even through, a balance must be found between performance and flexibility. For memory managment most of modern operating systems rely on paging mechanisms to provide a flexible and efficient memory management support. At the opposite, processors provide cache memories fully managed at hardware level to hide global memory latencies and improve memory performances.

In the nineties, Kessler and Hill[13] discussed the impact of the Operating System (OS) paging policy on cache performances. This naturally poses the question of the impact of the OS and more particularly its memory policy on a given program. Page coloring is now available on OpenSolaris and FreeBSD but not on Linux. As the last one became heavily used in HPC context, we can ask the question of the impact of its paging strategy on current large cache memories and on parallel architectures.

In this paper, we present benchmark results on those three systems to compare their default paging policy on CPU intensive applications. On such workload we obtain gains of about 5%, but surprisingly observe large performance gap up to 50% by changing the OS, even regarding CPU intensive applications like NAS. We observe a not so well-discussed interaction between caches, paging and malloc placement strategy which could break cache efficiency and double the execution time of some functions. Such impact may be unexpected, especially on computational parts of tested applications. Moreover, we show that multi-cores with shared caches intensify the problem.

We also observe that current memory allocators are mostly focused on small bloc placement for cache optimization, but neglect large arrays and tend to align them on page limits. Measurement show that it induces undesirable side effects depending on the paging policy of the OS. We discuss this aspect in section 5.

Section 2 details the different paging policies and their relationship with associative cache memories. Then, section 3 provides measurements to compare the three selected operating systems, with a special attention to limit changes of user-space tools which could impact performance. The reported issues are discussed in more depth in section 4 to extract related parameters. It mostly focus one of the benchmarking application as it provide a good way to expose them on a real case. Finally, section 5 lists recommendations deduced from the previous points by discussing page coloring algorithms and malloc policy for large arrays.

## 2. RELATED WORK

In this section we briefly introduce the mechanisms implied in our observations. It mainly covers operating system virtual memory management, cache associativity and interaction between those two components.

### 2.1 Memory Management

The main goal of an operating system is to manage the resources shared between all active tasks and to maintain the tightness between all of them. As far as the memory management is concerned, all generalist modern systems rely on the concept of virtual memory. It offers an elegant way to solve most of the issues which arise while trying to build a multi-user and multi-process OS.

The *virtual memory*[17] is an abstract linear *address space* which represents the memory seen by a unique process. To be managed, it has been splitted into small segments (*pages*) of 4KB on *x86* processors. In collaboration with the hardware, the OS is responsible for mapping *physical pages* onto *virtual pages* used by the application. All those mappings are maintained by the OS and exposed to the processor via a *page table*.

As processes are restricted to their own *virtual memory*, the *Memory Management Unit*(MMU) of the CPU needs to translate all *virtual addresses* into *physical addresses*. To reduce the latency of this translation, most modern processors dispose of a specific cache to store a couple of translation entries : the *Translation Lookaside Buffer* (TLB). For example, on Intel Nehalem, the TLB can store 512 entries[10] addressing a maximum of 2MB for standard 4KB pages.

Because of the page-grain management at kernel level, a user-space function (*malloc*) need to be provided in order to manage allocation of non-restricted sizes. This function requests pages through *mmap* or *brk* system calls and distributes user segments over them.

## 2.2 Processor Cache Memories

Modern architectures offer another kind of memory : hardware caches. Those memories are fully managed at hardware level and transparently used to offer a fast access to data by working on a local copy closer to the processing units.

When the first access to a given data occurs, a copy is fetched from the main memory to the cache and provided to the related processing unit. This constitutes a *cache miss*. Subsequently, other operations on this memory location can be done directly on the copy, which is faster than accessing to the main memory.

As caches are smaller than the main memory, one must provide a replacement mechanism to *evict* some old copies from the cache. This eviction policy must avoid selecting memory locations which might be accessed again in order to reduce the performance loss related to cache misses. For management purpose, caches are splitted into small segments (*cache lines*) of 64 bytes on current Intel processors. Consequently, all the data movements between the cache and the main memory will occur with this defined size.

On request, the cache must find the corresponding line or manage a miss. To do so, each cache line can be *indexed* with the address of the related data. On access, a *fully associative* cache will perform a search over the whole cache to find the copy. This global activation of all lines implies a large complexity, restraining its usage to very small caches such as some TLB of Intel Pentium or Atom processors.

To be more efficient, we can force each specific memory location to be stored into a unique cache line. In *direct mapped* caches, the related line can be directly deduced from the address by applying a mask. This way, the search is reduced to a unique test. In return, this design produces higher rate of cache miss depending on data layout and access pattern.

A better balance between *fully associative* and *direct map* caches could be reached by allowing storage of specific memory location into N cache lines, forming a *cache set*. *N-way associative* caches could be simply seen as an aggregation of N *direct map* caches, each referred as a *cache way*. To give some magnitudes, on Intel Core Nehalem, depending on the model, we have a 8-way 32KB L1 and 8-way 256KB L2 and 16-ways 8MB L3.

## 2.3 Paging And Cache Associativity

As reminded in section 2.1, there is two address spaces. Hence, cache memories can be addressed physically or virtually. In the virtually addressed case, cache locality of the data only relies on placement into the virtual address space. This is the case for current L1 cache on Intel processors. Notice that virtual address space placement mainly depends on the user and *malloc* function.

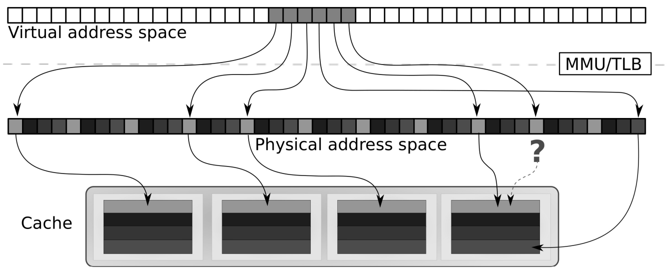

**Figure 1: Example of conflict due to non-optimal paging policy on top of physically addressed 4-ways associative cache.**

On the contrary, when dealing with physically addressed caches, it is page mapping which defines cache location of data. Consequently, the OS paging subsystem becomes a new parameter between applications and caches.

The *color* of a page can be defined as its position in the physically addressed cache. Into N-way associative cache memories, the usage of N+1 pages of the same color will lead to cache conflicts as they will use the same location in the cache. Figure 1 illustrates this case on top of a 4-way associative cache. Due to the paging mechanism, a virtually contiguous data set can become physically non-contiguous causing *conflict misses* as in our example.

On Linux, the page demand is satisfied by taking the first available free page into its buddy allocator[5]. Consequently, the OS does not take care of the color distribution of the physical pages mapped into the virtual address space. We will use the term of *random paging policy* to describe this approach.

As it was studied in [13], [3] and [9], this kind of paging policy leads to a larger number of cache miss due to a statistical side effect of the random distribution of pages. To confirm this assumption on current large caches, we compared the Linux paging policy to a fully random one. We allocate a buffer of desired size, read the Linux page table and virtually built another fully random one. Finally, we evaluated the number of unnecessary page conflicts for the two solutions, considering a linear memory access.

Figure 2 clearly shows that Linux does not perform really better than the fully random page table in term of cache conflicts. We also confirm the observation of [3] as the fully random page table gives more variable results than the Linux page table. While trying to perform the same measurement in parallel of a kernel compilation, we observed that Linux provides exactly the same results than the fully random page table, confirming their observation with intensive memory workload.

This measurement was done on a Linux station with 24GB of memory. As the system uses only on a few hundred of MB, we can expect a less random page selection due the huge free memory available. In practice, it wasn't. The issue stays largely present even just after rebooting.

With Fig. 3, we can confirm the impact in term of time on a cyclic sequential access to an array by executing the

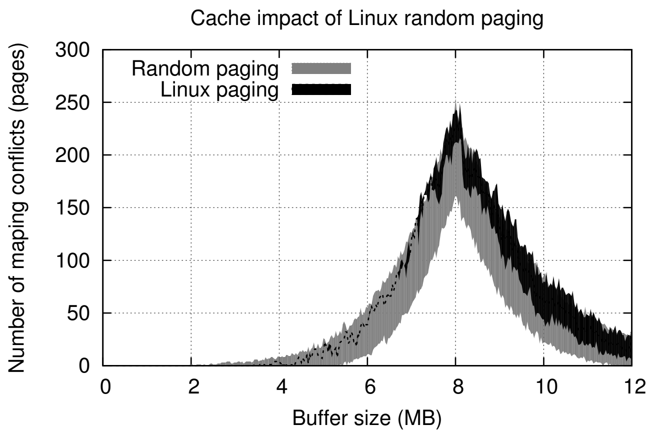

**Figure 2: Comparing the simulated conflicts of Linux page table and fully random one on top of an Intel Nehalem 16-ways 8MB L3 cache with 500 runs for each memory size. It represents the minimum and maximum number of unnecessary pages in conflict for each size.**

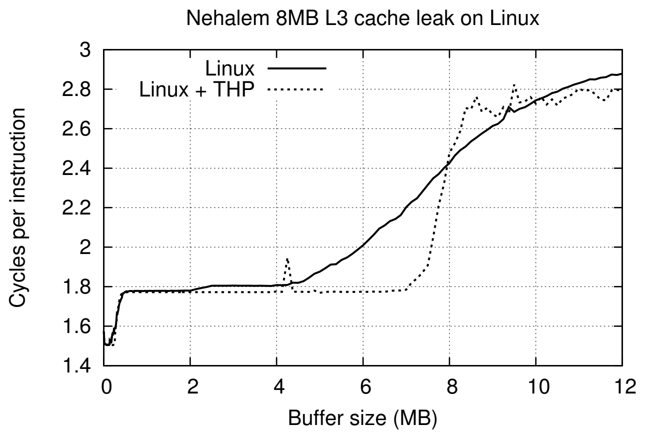

**Figure 3: Experimental time of a repeated stride 1 access to an array on top of an Intel Nehalem processor with or without Linux Transparent Huge Pages (THP).**

loop on an Intel Nehalem. We observed a leak of the L3 cache after 5MB (62%) caused by the random paging policy of Linux. The figure also gives the results of the same test on Linux huge pages which was equivalent to FreeBSD and OpenSolaris. It can be observed that Linux standard pages do not permit to fully exploit the cache.

Hence, on Linux, when trying to apply some optimizations such as *cache blocking*; one must not forget that Linux does not provide a full access to the cache because of the paging policy. Remark that the developer cannot bypass the issue at user-space level and must eventually consider a smaller cache than physically available to reach best performances.

Even if it is a known problem, we find papers working on cache blocking without taking account of this effect or discussing its impact on their models. It is also common to present Fig. 3 on logscale axis. If measures also use a power of two for sampling we may tend to take the last point in cache as a background noise overlooking that we partially loss the last 40% of the cache.

## 2.4 Page Coloring to Solve the Issue

As it was studied in [13][3], a solution could be to select physical pages in order to keep an equal distribution of the pages into the cache. This is referred as *page coloring* and only relies on OS internal implementation.

A better paging placement could be reached with full knowledge of the data access pattern. But in the general case, the OS cannot get access to such information and must rely on some heuristics :

**Hashing the virtual address:** one can simply use modulo of the color count. This approach provides a direct mapping between virtual addresses and physical addresses in term of cache associativity. Therefore, the choice of the virtual address implies the choice of the cache location. Some also add the process ID (PID) to get different pattern for each process[13]. This is the default for the *x86* version of OpenSolaris[14]. Due to internal design, FreeBSD[16] provides a similar pattern for standard pages.

**Round robin or bin hopping:** the system only remembers the color of the last page provided to the process and increments it before each request ensuring each color appears the same number of time. It provides a small adaptation to the access pattern by capturing the first access ordering. This is available in a parameter of OpenSolaris on *Sparc* only.

**Best bin:** the system counts the usage of each color and tries to allocate the less used one. This is implemented in *Sparc* version of *OpenSolaris*. It is also possible to take in account the number of free pages of each color as discussed in [13]. This approach was mostly designed to provide a better adaptability in small free memory context.

## 2.5 Huge Pages

When using virtual memory, the processor needs to translate all virtual addresses into physical ones. To reduce the latency of this translation, most of the modern processors provide a specific cache to store a couple of translations: the *Translation Lookaside Buffer* (TLB). For example, on Intel Nehalem, the TLB can store 512 entries[10] addressing a maximum of 2MB for standard 4KB pages. One can remark that this is less than the last level cache size. It can be observed on Fig. 3 on Linux with a tiny performance lost for standard pages at 2MB.

When using large amount of memory, we will create many TLB misses getting a penalty. To solve that issue, many processors offer a support of larger pages (*huge pages*). This way, a larger amount of memory can be addressed with the same TLB size, hence reducing the number of TLB misses.

On *x86_64*, we can use huge pages of 2MB or 1GB[10], which are bigger than the cache way size of current processors (512K for Intel Nehalem L3). Using such page size provides a direct mapping between virtual and physical addresses in term of cache associativity. Consequently, as a side effect, it provides a page coloring policy. The cache aware aspect of huge pages only relies on their hardware definition, not on their software support. Hence we cannot change the mapping rules as for standard pages.

For now, huge pages are supported into FreeBSD with a transparent implementation[16]. On OpenSolaris, huge pages can be requested with environment parameters which is less flexible than FreeBSD.

There is some work done to provide a better support of Linux huge pages, we can cite the work of Ian Wienand[18] in 2008, or the work of K. Yoshii and al. in 2009 for their "big memory" on Blue Gene Linux[21]. Moreover, the current work of A. Arcangeli[1] on his *Transparent Huge Pages* is now included into the new RedHat 6 and added to the mainline since revision 2.6.38, providing Linux with a usable support of huge pages.

### 2.6 Conclusion

We shown that OS paging policy interacts with application performances while running on top of associative cache memories. Some solutions have been proposed in the past and are currently implemented into FreeBSD and OpenSolaris. On the contrary, Linux still uses a random paging policy. In the next section, we experimentally compare those distinctive approaches on current processors and discuss some of their limits.

## 3. CONTRIBUTION : EVALUATE IMPACT OF OPERATING SYSTEMS

As discussed, each operating systems provide different paging policies. So the question is, how do they perform with CPU intensive applications ? In this paper, we compare three of them with native and production-class paging policies. We describe our experimental protocol followed by some of the results we obtained. Those results might permit to point out a limitation of the available cache aware paging strategies.

### 3.1 Experimental Protocol

We targeted three production-class kernels: OpenSolaris, FreeBSD and Linux, fixing as much as possible all other parameters. As seen each of them provide a different paging policy. For the hardware, we used a unique Dell T5500 equipped by two Intel Nehalem E5520 (2.27Ghz) with a 16-ways L3 cache of 8MB and 24GB of memory. As OpenSolaris didn't boot with NUMA memory enabled and FreeBSD didn't explicitly manage it, we disabled NUMA in BIOS to run in interleaved mode for all our measurements even for Linux to keep comparable results.

For the software part, we fixed as much components as possible in order to keep the kernel as unique variable. The compiler was forced to gcc-4.4.1 and manually recompiled for each system. Similarly, we fixed some libraries such as binutils-2.20, gmp-5.0.1, mpfr-2.4.2, libatlas-3.8.3 and mpich2-1.2.1p1. Each of them was compiled when possible with the same options. All the benchmarks were compiled with this tool chain by using flags : *-m64 -fomit-frame-pointer -O3 -march=core2*.

Except for the libc, we let mostly the kernel as unique variable and compared Linux Fedora 16 (kernel 2.6.38), OpenSolaris 2009.06 and FreeBSD 8.2 each of them in 64bit mode. We also tried to reduce the impact of schedulers by closing all unnecessary applications during the benchmarks.

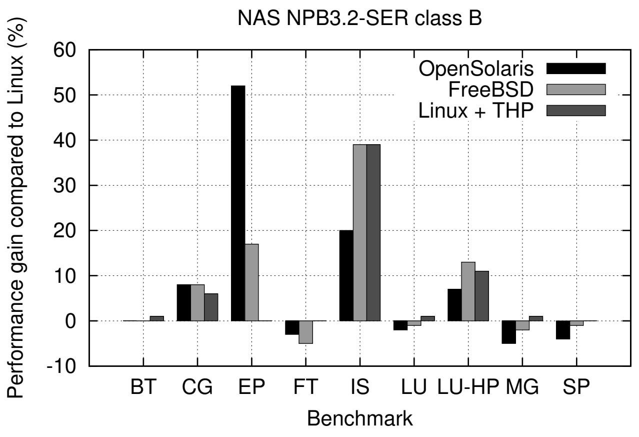

**Figure 4: Performance gain of NAS-SER benchmark compared to standard Linux. Higher is better.**

For benchmarking, we selected NAS 3.2[4][11], and Euler-MHD which is a magnetohydrodynamic 2D Cartesian grid explicit solver[20].

### 3.2 NAS benchmarks

The NAS benchmark was run in sequential, MPI and OpenMP modes. Results are presented in Fig. 4, 5. On sequential mode, we observe a performance improvement in favor of FreeBSD and OpenSolaris compared to Linux. We obtaine performance gain ranging from 7% to 51%. The EP benchmark have a particular improvement of up to 51% on OpenSolaris and 19% on FreeBSD. Slower benchmarks are penalized by up to 5% on OpenSolaris and FreeBSD. We excluded DC as it benchmarks the file system which has no links with this work.

We also provide the results obtained while using the Transparent Huge Pages (THP)[1] of Linux. To use it, we enabled THP in *always* mode. Figure 4 clearly shows the correlation with the FreeBSD pattern. Notice that we didn't get the exact same policy as THP will use huge pages only for segments larger than huge page size (2MB) and aligned to those limits. For others segments it uses the default random paging policy which differs for FreeBSD and OpenSolaris standard one. This is why we didn't get exactly the same impact on EP benchmark.

For sequential mode, we have a mean reproductible improvement of 7% compared to standard Linux. Hence, demonstrating that Linux default mode can be improved for such workload. As the NAS test suite is focused on CPU usage, we can argue that the change came from the scheduler or the memory management, not from IO. But, as we limited the scheduler activity, we can reasonably assume that the memory management is actually the major source of performance issue. This is conforted by the correlation with Linux huge pages results.

By default, OpenSolaris didn't provide huge pages but only page coloring. As we also seen improvement on this OS, one can assume that TLB reduction is not the major source of improvement. But cache aware aspect might be.

Fig. 5 provides NAS results in MPI modes with 8 processes. In this configuration we obtained a huge slowdown on OpenSolaris and FreeBSD. On class S, W and A, we have

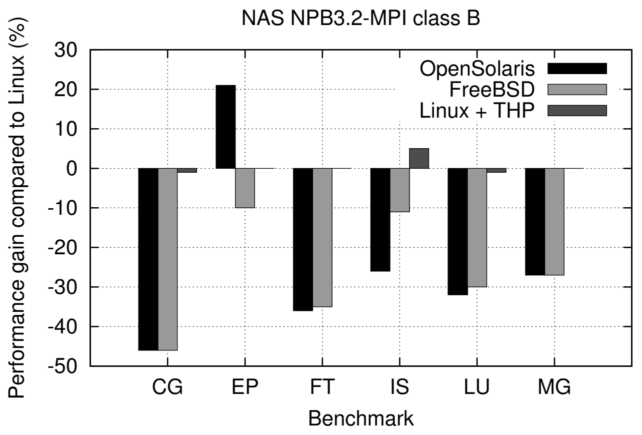

**Figure 5: Performance gain of 8 processes NAS-MPI benchmarks compared to standard Linux. Higher is better.**

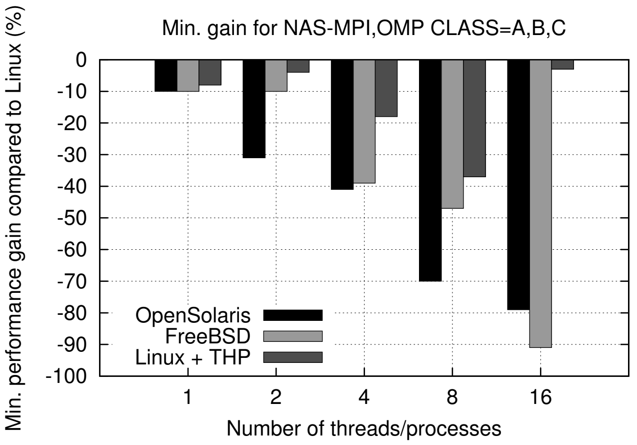

**Figure 6: Minimum performance gain of all NAS-SER, NAS-MPI and NAS-OMP benchmarks depending on the number of threads/processes in use. We use results from class A,B and C.**

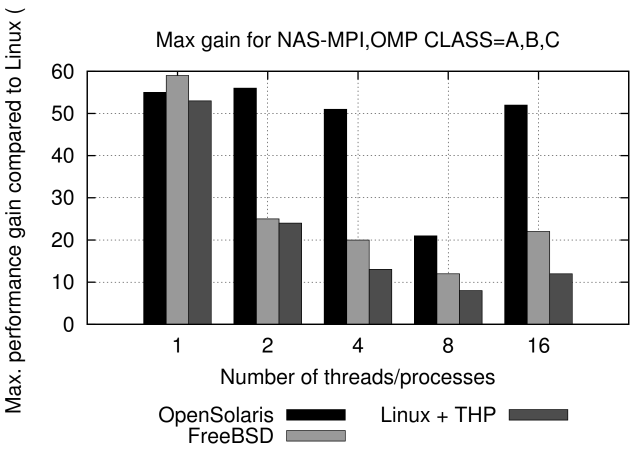

**Figure 7: Maximum performance gain of all NAS-SER, NAS-MPI and NAS-OMP benchmarks depending on the number of threads/processes in use. We use results from class A,B and C.**

similar pattern while enabling huge pages on Linux which permits to reject the hypothesis of scheduler as major parameter. We also obtained globally the same results with or without using *hwloc*[6] to bind NAS threads and processes. In NAS-MPI, EP on class C size still have same large improvement as for sequentiel on the two systems.

To get a better view we can compute the *min* and *max* improvements for all NAS benchmarks keeping number of threads and paging system as parameters. Fig. 6 and 7 show the correlation between all paging systems while increasing the number of execution streams. For large number of execution streams, page coloring and huge pages seems to favor some pathological cases in detriment of performance. This has not been observed on Linux random paging. Correlation with Linux huge pages effect conforts the hypothesis of memory management issue. We will discuss this issue in more detail in section 3.3 after showing a similar effect on EulerMHD.

Note that Linux huge pages provide smaller impact as it will not apply huge pages on all segments due to threshold effects on segment size smaller or larger than 2MB.

### 3.3 EulerMHD application

In order to test a more complex benchmark, we retained a magnetohydrodynamic 2D solver on a regular Cartesian mesh with high order numerical scheme[20]. The first results obtained are given in Fig. 8 (a). They shown an extreme case with improvement of 4% on Linux huge pages and an execution time which was doubled on FreeBSD. Surprisingly, the most affected function contains intensive computation loops with no system call. This issue will be discussed in the coming section, but we noticed a similar behavior with the benchmark 410.bwaves from SpecCPU2006 with variation up to 40% depending on the OS.

### 3.4 Conclusion

With the previous benchmarking results we observed that OpenSolaris and FreeBSD seem to perform better than Linux on some sequential benchmarks. But we obtained large degradations with others, mostly in parallel. The fact that those two systems provide mostly the same behavior may benefit to the hypothesis of paging policy interferences as they provide mostly the same one in term of cache associativity. This is confirmed by results obtained on top of Linux huge pages.

The issues observed on EulerMHD will be discussed in coming sections and may contribute to the understanding of the NAS one.

## 4. PAGING AND MALLOC STRATEGIES

This section will detail some huge slowdown observed while studying the interaction of paging and malloc strategy on top of associative cache memories. It will permit to clarify the key points which must be avoided to sustain decent performances.

### 4.1 Impact of Malloc Implementation

As studied in [7], the relative starting address of each array can impact the performance of an application due to cache behaviours. C. Lemuet studied the impact of the arrays

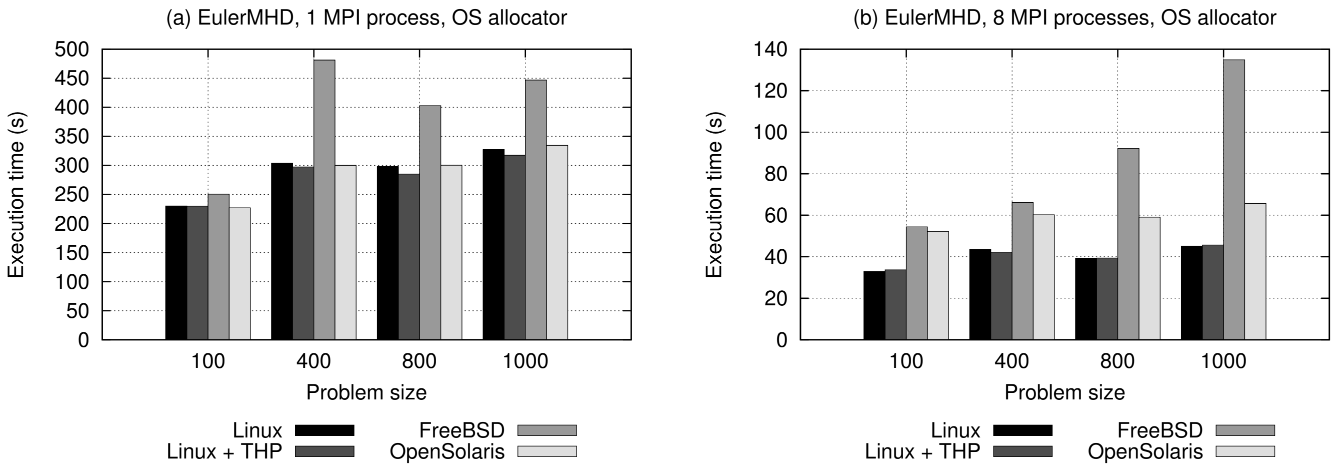

**Figure 8: Benchmarking results with the EulerMHD application on all operating systems (a) in sequential mode, (b) with 8 MPI processes.**

placement on cache memory banks of the itanium2. If two arrays are used at same time with a particular alignment, the program would get a penalty caused by a concurrent access on the same bank. Hence padding the starting address of the array fixes the issue.

Notice that we kept the *libc* of each system and consequently, their implementation of *malloc*. But simple tests proved that each of them produced different memory layout for the large arrays used by EulerMHD.

On Linux, with the glibc, arrays larger than 128KB are allocated directly with a call to *mmap*, causing such arrays to start on a page limit plus 16 bytes of header. On FreeBSD, we tend to be aligned on chunk size and directly on the 2MB huge page limits for larger sizes[8]. OpenSolaris uses a more "random" distribution based on multiples of 16 bytes.

In order to remove this factor, we implemented a simple allocator allowing us to keep a similar memory layout on each system. This implementation directly calls *mmap* for each allocation, wasting a large amount of memory. In return, it provides a simple way to ensure a fixed layout. All arrays were aligned to the page limit which may lead to some side effects, but allows us to objectively compare results on each system. On Linux we forced alignement on huge page size to produce the issue which wasn't present with default glibc allocator.

Figure 9 gives the results obtained using this allocator. Now, FreeBSD perform as well as Linux, but, OpenSolaris doubles run time just as FreeBSD used to do. As presented in Fig. 9, we can solve the OpenSolaris issue by adding a random offset multiple of 4KB to the addresses returned by *mmap*. In theory, due to paging, moving from 4KB must not change anything.

Further analysis showed that OpenSolaris automatically aligns all *mmap* requests on 2MB limits if the requested size overpasses 1MB. This alignment may permit the usage of huge pages as they can only be mapped on virtual addresses multiple of their own size. On OpenSolaris with our allocator, all arrays became aligned to 2MB limits instead of the standard 4KB limits of FreeBSD and Linux. The random padding breaks alignments to fall back from 2MB to 4KB ones and fixes the performance issues.

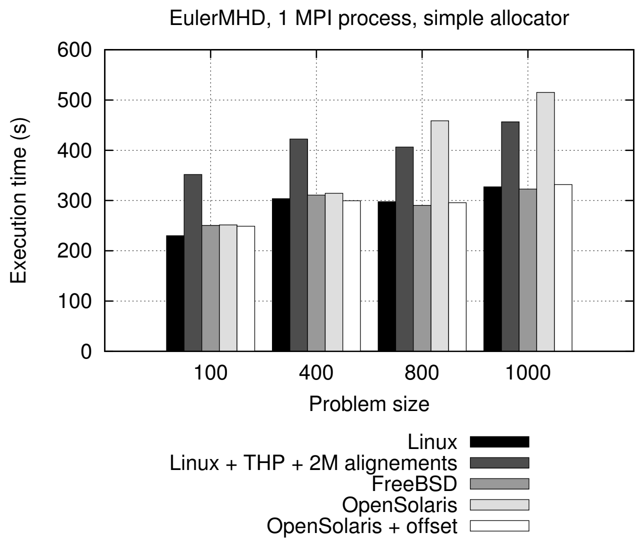

**Figure 9: Benchmarking results with the EulerMHD application in sequential mode with a simple *malloc* implementation instead of the system one. On linux huge pages, we deliberately forced alignement to 2MB. On OpenSolaris we add a mode by padding with offsets multiple of 4K.**

As NAS only uses static allocation, the same procedure did not exhibit major changes compared to the first results obtained in section 3.2.

## 4.2 The Counterpart of Too Regular Paging

We have seen that changing *malloc* avoids the FreeBSD pathological case, but creates another one on OpenSolaris which can be fixed by avoiding the 2MB alignment of *mmap*. As explained in section 2.3, the paging policy decides of the cache placement of data. But *malloc* policy defines the cache placement for the inner part of the pages. On Linux's random paging, moving an array by 4KB may not change the statistical position in the cache as we replace a random page by another one. So we do not expect impact of 4KB offsets on such policy.

In section 2.4, we explained that the *x86* version of OpenSolaris applies a hash on the virtual address in order to select the page color. As the hashing method is implemented by a modulo, it produces a *regular coloring* with a unique pattern repeated over the virtual memory.

On OpenSolaris, if our arrays are allocated aligned on 2MB limits, it implies that the first page of each array must be a page of color 0. So, all pages are mapped on the same cache location. As EulerMHD uses more than eight arrays in some functions, it exceeds the cache associativity and intensively creates cache conflicts reducing performances.

We pointed out a limit of OpenSolaris and FreeBSD paging policy: problem arise if the application uses too much arrays aligned on the cache way size. With such regular paging, the *malloc* function must be aware of cache associativity out of the standard page limits, thus, avoiding such relative alignments.

## 4.3 Shared Cache Associativity and Threads

In the previous section we dealt with EulerMHD which uses more streams than admitted by cache associativity. Such pattern is a mistake — a better solution may be to fix the code. As modern caches offer associativity of eight or sixteen this problem might be avoided.

But nowadays we had to deal with multi-threading aspects. Considering this, one can note that a 16-ways cache shared to four threads may only permit four aligned streams per threads. If accounting hyper-threading it lessens to two. With such parameters, efforts made in application developments might by nullified by the system if it provide a inadequate memory allocation strategy.

It explains the performance loss observed on parallel NAS. As they use static arrays, in MPI mode, all instances will use the same addresses. On too regular paging, it implies the usage of same cache lines for all virtually and physically addressed caches. Consequently it largely increases the pressure on shared caches when using more execution streams.

Figure 10 demonstrate this with a simple benchmark using a variable number of arrays in OpenMP threads. Depending on the alignment of all arrays, we loss a factor two when using more arrays than associativity. Avoiding bad alignments secures decent performances even with up to 64 arrays. Value which has to be compared to the 16 associativity of L3 cache.

## 4.4 Avoid 4KB Aliasing Issue

On current Intel processors, it is not possible to write and read two data distant of 4K-multiple at same time. Hence, an issue can be observed while trying to compute :

```
for (i = 1 ; i < SIZE ; ++i)
  X[i] = Y[i-1]
```

Due to superscalar model, iteration $i+1$ will try to load $Y[i]$ while write operation on $X[i]$ from iteration $i$ still be in the pipeline. Consequently if X and Y have same base addresses modulo 4KB we must wait before starting iteration i+1. This can be observed with the hardware counter LOAD_BLOCK.OVERLAP_STORE on Intel Core 2 Duo. Such example shows a slowdown of a factor larger than 2. Notice that such access pattern is common on large arrays in HPC like for mesh interpolations... Similar issue was discussed in [15] due to placement in the stack.

Intel Nehalem and Sandy Bridge solve the issue for the given exemple, but the problem is still present while using more complexe pattern like using two such streams in the loop :

```
for (i = 1 ; i < SIZE ; ++i) {
  X1[i] = Y1[i-1]
  X2[i] = Y2[i-1]
}
```

On a real case, we observed that we can reach the same performance optimization by breaking arrays alignments rather than doing expensive optimizations. Table 1 give the measured time we obtained on Core 2 Duo. The optimized version offers a better and stabler performance but required much more writing effort yielding less readable code.

The default malloc policy of glibc lead to this issue for segments larger than 128k. Padding large arrays as described in previous section can immune the application without touching the code which may cost more than doing it at malloc level. Breaking regular patterns may help to sustain better performances by reducing possible pathological cases on various micro-architectures.

## 4.5 Conclusion

We can summarize this section by saying that memory issues arise if we combine three parameters :

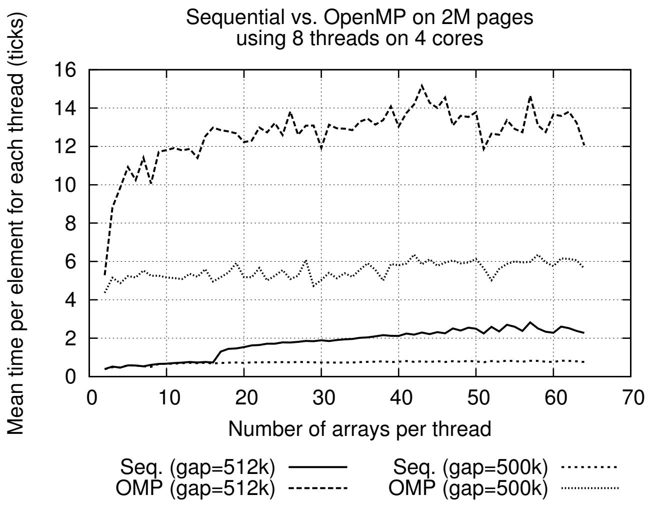

**Figure 10: Micro-benchmarking a sum between N independent arrays into each thread allocated on top of huge pages. Gap defines the distance between starting position of those arrays. The total memory for each thread fit in L1 cache.**

| | Original | Optimized |
|---|---|---|
| Default alloc | 51.0 | 21.7 |
| Padded arrays | 21.8 | 21.3 |

**Table 1: Performance evaluation of an optimized function and it's original version while using default allocation scheme or padded one. The timings are given in CPU cycles per iterations.**

- using more streams than permitted by cache associativity.
- malloc trying to maintain alignments multiple of problematic size (page or cache ways size).
- generating conflicting allocation patterns while interacting with malloc on physically addressed caches.

## 5. SOLUTIONS TO MEMORY ALLOCATION ISSUES

As a contribution, this section exposes some improvements which can be done at system level to avoid the pathological cases listed in previous sections. We found solutions which didn't imply changes at application level as it may improve performance portability and save development efforts. As a consequence we may focus our analysis on the last points two and tree exposed in section 4.5.

### 5.1 Avoid Regular Page Coloring

The issue on EulerMHD is partially caused OpenSolaris's *regular paging* policy. Linux didn't support such coloring preventing such a large slowdown. But not permitting to fully exploit cache space. As a solution, we propose a better balance between Linux's fully random paging policy and too regular one of OpenSolaris. This way, we expect to keep the full availability of the cache for sensitive applications, avoiding pathological cases which may penalize the others.

This can be achieved by using a more complex hashing function. In OpenSolaris, the virtual address is hashed with the process ID and transformed with a modulo. We could add the address of the *mmap* segment into the equation, or use a more complex hashing procedure before the modulo to randomize bits ordering. The requirement may be to maintain a roughly flat color distribution in segments of a cache way size, while randomizing the pattern between those segments.

The issue was observed due to a negative interaction of malloc and paging policy. Patching the page coloring algorithms may avoid doing to complex trick in malloc implementations. It may be harder to control alignments larger than page size at malloc level without impacting too much memory consumption and/or allocator performances.

Dealing with multi-threading, one can move to more radical solutions such as cache partitionning[2][12]. Even though, we need to take care not to apply too regular pattern in threads mapping otherwise we fall in same issue than EulerMHD in sequential mode.

### 5.2 Cache Aware Malloc on Top of Huge Pages

Section 2.5 explained that huge pages provide a page coloring as a side effect due to their large size compared to cache ways. By definition, such paging offers a direct mapping between the virtual addresses and the physical ones in term of cache associativity. Hence, it could be considered as a *regular page coloring* policy. But in this case, as it was fully constructed in hardware, the OS cannot fix the issue by using rules in page management.

On top of such paging we must take care of the memory layout returned by *malloc* implementation. As there is some active work to provide a transparent integration of huge pages in Linux, some must take care of the behavior of the glibc on such paging. As OpenSolaris and FreeBSD, it may be tempting to force *mmap* to align large virtual memory segments on the 2MB limits to fully benefits of the huge pages. But as the glibc directly calls *mmap* for large allocation, we may observe performance degradation because of the unfortunate interaction between space runtime and operating system. Fortunately this is not the case in current implementation.

Opposingly from FreeBSD, our observation shows that the default OpenSolaris *malloc* provides slightly better performances on top of this regular paging. But clearly, we must acquire a better understanding of the interactions of user-space runtime and kernel-space strategies to fully benefits of huge pages.

### 5.3 Inner Page Alignment of Large Malloc

Papers on malloc implementation[19][8] largely discussed the problem of small blocks alignment to keep cache efficiency. A common approach is to avoid using too large headers and too large alignments not to loose cache space between user data. The extreme case was to use external chunk headers.

But for larger blocks, it is common to directly use the *mmap* system call. Hence, such blocks tend to be aligned on page size. Taking a look at current L1 caches, one can remark that a way size correspond to standard page size (4KB).

With such pattern, all functions using more large arrays than associativity will reduce L1 cache performances. This case can be observed on EulerMHD (section 3.3). We can improve performances up to 7% by randomly padding large arrays by multiples of 64B in range of 4KB.

For direct *mmap* allocation, we may already have unused space at the end of segment. Most of the time, padding may not cost more physical memory. It will also reduce chances to getting 4KB aliasing or equivalent alignments issues. Randomly padding on multiple of cache line size seems to yield the better results on tested cases.

## 6. CONCLUSION AND FUTURE WORKS

In this paper, we evaluated the impact of the Operating System on CPU intensive applications by testing Linux, OpenSolaris and FreeBSD.

Our experiments gave better results on OpenSolaris and FreeBSD. We observed gains of about 7% on sequential NAS. Surprisingly, we observed impact up to 50% for some CPU intensive applications proving the non negligible impact of Operating Systems on computational loops. Hence, the system must not be ignored while doing performance analysis of such applications.

In some cases, we noticed that the regular page coloring policy of FreeBSD and OpenSolaris favors some issues causing more slowdown than potential gain. This problem might be solved by keeping a better balance between fully random and too regular paging.

Despite, huge pages can improve performances, we observed that it leads to important issues on multi-cores with shared caches. For such usage, it is better to maintain standard page coloring approach as it provides solutions to break regularities playing with page selection algorithms.

Moreover, we confirmed that the malloc placement policy can largely impact performance. Hence, when trying to push huge pages as the default paging policy, we must take care of *malloc* which must become more cache aware in order to avoid pathological cases. For large arrays which tend to be neglected, randomizing malloc alignement seems to provide a simple way to break regularities and improve performances on associative caches.

We mainly discussed the impact for the outer page placement. But our results tend to show that we can find some effects for the inner part of the pages which were not fully discussed here. Future works will be to study in more depth the *malloc* alignment policy in order to clearly understand its impact and interaction with the paging subsystem.

## 7. REFERENCES

[1] A. Arcangeli. Transparent Hugepage Support, KVM Forum 2010, Boston.

[2] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *HPCA*, 2009.

[3] S. Bahadur, V. Kalyanakrishnan, and J. Westall. An empirical study of the effects of careful page placement in Linux. In *ACM 36th Southeast Conference*, 1998.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, 1991.

[5] D. Bovet and M. Cesati. *Understanding The Linux Kernel.* Oreilly, 3 edition, 2005.

[6] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a generic framework for managing hardware affinities in HPC applications. In Ieee, editor, *PDP 2010.*

[7] W. J. C. Lemuet and S. Touati. Improving Load/Store Queues Usage in Scientific Computing. In *Proceedings ICPP 2004.*

[8] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD, 2006.

[9] M. Hocko and T. Kalibera. Reducing performance non-determinism via cache-aware page allocation strategies. In *Proceedings of WOSP/SIPEW 2010*, pages 223–234.

[10] Intel. *Intel®64 and IA-32 Architectures Software Developer's Manuals.* 2009.

[11] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. Technical report, 1999.

[12] M. Kandemir, R. Prabhakar, M. Karakoy, and Y. Zhang. Multilayer cache partitioning for multiprogram workloads. Euro-Par'11, 2011.

[13] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, volume 10:338–359, November 1992.

[14] J. Mauro and R. McDogall. *Solaris Internals (2nd Edition).* 2006.

[15] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! ASPLOS, 2009.

[16] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36:89–104, December 2002.

[17] A. S. Tanenbaum. *Structured Computer Organization (5th Edition).* 2005.

[18] I. Wienand. Transparent Large-Page Support for Itanium Linux, 2006.

[19] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995.

[20] M. Wolff, S. Jaouen, and H. Jourdren. High-order dimensionally split Lagrange-remap schemes for ideal magnetohydrodynamics. In *DCDS-S: proceedings of NMCF 2009.*

[21] K. Yoshii, K. Iskra, H. Naik, P. Beckmanm, and P. C. Broekema. Characterizing the Performance of "Big Memory" on Blue Gene Linux. In *ICPPW 2009.*