Introducing Kernel-Level Page Reuse for High Performance Computing

Sébastien VALAT [†]
CEA, DAM, DIF
Bruyères-le-Châtel
F-91297 Arpajon, France
sebastien.valat@cea.fr

Marc PÉRACHE [†]
CEA, DAM, DIF
Bruyères-le-Châtel
F-91297 Arpajon, France
marc.perache@cea.fr

William JALBY
[†] Université de Versailles
Saint-Quentin
45 Avenue des États-Unis
Versailles, France
william.jalby@uvsq.fr

Abstract

Due to computer architecture evolution, more and more HPC applications have to include thread-based parallelism and take care of memory consumption. Such evolutions require more attention to the full memory management chain, particularly stressed in multi-threaded context. Several memory allocators provide better scalability on the user-space side. But, with the steadily increasing number of cores, the impact of the operating system cannot be neglected anymore. We measured performance impact of the OS memory subsystem for up to one third of the total execution time of a real application on 128 cores. On modern architectures, we measured that up to 40% of the page fault time is spent in page zeroing. In this paper, we detail a proposal to improve paging performance by removing the needs of this unproductive page zeroing through an extension of the mmap semantic. To this end, we added a kernel-level memory page pool per process to locally reuse free pages without content reset. Our experiments show significant performance improvements especially for huge pages.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Allocation/deallocation strategies – Virtual memory

General Terms Design, Performance, Measurement

Keywords Memory pool, Memory allocator, Kernel, Page fault, Zero page, Parallel, Process, Linux, NUMA, Many-Core

1. Introduction

Computing architectures evolutions have deeply impacted the *High Performance Computing* landscape. Since several years, the great majority of supercomputers are clusters. However the number of computing units within each node tends to increase. From multi-core, which become common even in our laptops, the HPC systems evolved towards many-core architectures. Consequently, software developers have to expose sufficient parallelism at all levels, from the operating system to applications. Such a task requires important efforts to transform existing sequential codes.

HPC are commonly operated through distributed memory programming models, generally thanks to the MPI (Message Passing Interface) standard. Most of the implementations are process-based ones, so the increasing number of cores per node starts to show some limitations of this approach. Limitations which are mainly related to inter-process communication efficiency, increasing number of communication buffers and data replication (ghost cells, constants...). More and more simulation codes try to solve the issues by combining the Message Passing model with other ones that have proved their advantages and performance results over shared memory architectures. Thus we can especially quote the thread-based models, such as OpenMP, TBB, Cilk...

Production grade parallel memory allocators are now available, improving the user-space part of the memory management chain. But, the Linux kernel still has some scalability issues in its memory management sub-system, making page faults a major limitation on large compute nodes. We observed kernel overhead up to 30% of the execution time by running applications on 128 cores.

In this article, we looked down to the operating system itself and observed that the `clear_page` function (responsible of page zeroing) accounts for 40% of the page fault cost. As a consequence, we suggest to limit the use of this function by doing page reuse at kernel level thanks to the introduction of a new per process memory pool.

We implemented this solution and observed significant improvements on tested HPC applications.

The first section of this paper introduces some basics about memory management in parallel context and describes the page zeroing problem. The second one provides some related results motivating the needs to improve the kernel itself. It is followed by the description of some related work in memory management field. Then, we propose an improvement in kernel-space to limit this performance loss. Finally, the last section shows and analyzes its performance impact on micro-benchmarks and applications.

2. Reminder on Memory Management

This first section provides some basics on memory management, detailing its trade-offs and performance issues. In particular, we focus on page fault and page zeroing which are partly responsible of the overhead.

2.1 Usage of Memory

Although processors are able to gather an increasing number of cores while keeping the same size, this is not the case for memory banks. As a result, the memory available per core tends to stabilize or even in some cases to become smaller[7]. In the best case, current supercomputers offer a ratio of 2GB (Tera100) or 4GB (Curie) per core. Other architectures such as BlueGene machines with higher number of cores (more than a million) have lower ratios (1GB). Similarly, new architectures like the Intel Xeon Phi will provide a total memory of 8GB for 60 cores or 240 threads. It lowers the ratio to 130MB per core or 34MB per thread.

In such context, developers have to limit the memory footprint of their applications. Hence, keeping unused memory buffers cannot be acceptable anymore. In some cases, it might imply more calls to the allocator. Similarly, the allocator cannot keep too much memory in user-space and have to hand it back to the underlying system. Consequently the whole memory chain is impacted, with a growing impact of the operating system, leading to new performance issues.

There is necessarily a trade-off between keeping and returning unused memory. Trade-off which tends to become more complex as it impacts multiple levels: application, allocator, operating system. Each of these levels comes with its own policy which combined might lead to undesirable performance effects.

In addition, current architectures are now multi-core and distributed programming models start to show their limits due to memory duplications. It implies that the system is now stressed with threads instead of processes. Hence, the kernel has to manage the contention on shared resources between the threads of a given process. This is particularly true for the memory management part which needs to maintain a coherent view of a common address space between every core. With multiple sequential processes, this problem is tempered by the use of separated address spaces.

2.2 Memory Semantic

From the operating system point of view, the memory management is done at the granularity of a page which represents a small contiguous memory segment of 4KB for standard x86 architectures. At processor level, an independent virtual address space is established for each process. The mapping between virtual addresses and physical ones is the responsibility of the operating system. This mapping is stored in a page table which is used by the processor for translation purposes. Each thread of a given process shares the same mapping along with the same page table. Hence, the page table becomes a contention point in multi-threaded context due to locks.

When a program needs some memory, it requests it via `mmap` or `brk` system calls. It establishes a new authorized segment in the process's virtual address space or extends a fixed one. In Linux, those memory segments are named Virtual Memory Area (VMA)[3] and are freed by the `munmap` system call.

2.3 Lazy Memory Allocation: Page Fault

Memory allocation performance must not be limited to the `malloc` call itself. On modern operating system, large memory segments are directly allocated through `mmap`. But, such segments are not truly allocated and only depict authorized memory regions. This is called *lazy page allocation*. Physical pages are then provided upon a first touch policy via a *page fault*. Therefore the allocation cost is not limited to `malloc`, but partly delayed to the first memory access. This approach permits to save memory by physically mapping only the accessed virtual pages, at the cost of an overhead at the first access.

This paper focuses on the cost of those first accesses. We start by studying the cost of such page fault and their scalability over a large number of cores. To this end, we placed us on the application side, and measured the time needed for first access to a newly allocated segment. Measurement procedure is as follows and repeated multiple times: (1) Create N OpenMP threads or N MPI processes. (2) Each of them allocates a large segment for a total of 10GB. (3) First write accesses time to each page are measured with RDTSC counter in order to build a per page per thread time distribution.

To confirm with a more stressing and simpler test, we also did the measurement with a single write access (`memset`) on the whole segment while measuring the total access time. It allows to compute the mean time of a page fault, hence avoiding potential biases of the first method.

Figure 1 presents the results on a 128 core computer. Error bars depict the median, 50% and 80% quartiles. A perfect scalability might produce a horizontal line. We can see that the median time increases linearly with the number of threads, whereas, when dealing with processes, it scales correctly with a twofold increase when leaving a single socket (8 cores). The scalability issue is hardly enforced by the two-

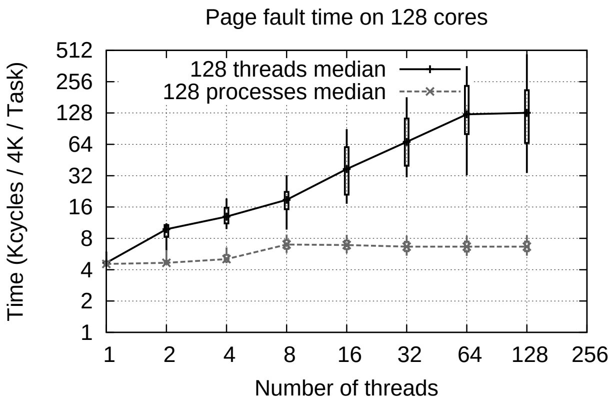

Figure 1. Page fault time measurement on a 128 core computer with 4*4 two-level NUMA with 8 core CPUs. It provides median, 50% and 80% quartiles of page fault time per thread. The measurement was done with 10GB of memory.

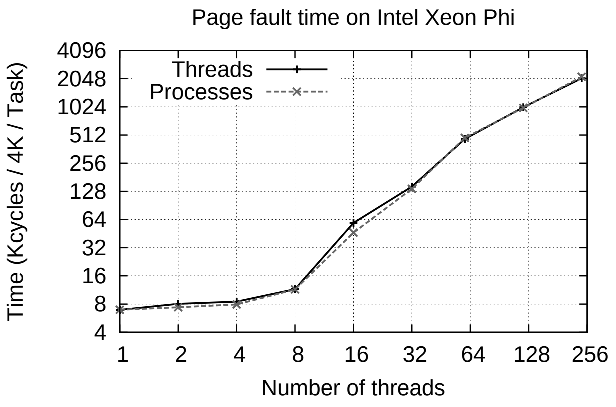

Figure 2. Page fault time measurement on the new Intel Xeon Phi architecture with up to 240 OpenMP threads with a 6GB segment.

level NUMA of the tested Bull BCS. Such thread scalability issue has already been observed by Austin T. C. and al. onto 2.6.37 version of Linux kernel with up to 80 cores[5].

Figure 2 provides the same measurement on Intel Xeon Phi with up to 240 threads. In this case, our previous measurement methodology largely underestimates the cost of page faults. It was due to larger overhead of the data aggregation code which was too slow on this architecture. Consequently, we had to fall back on the confirmation method (a simple `memset`). This explains the lack of time dispersions for this architecture. It can be noticed that the scalability issue remains starting from eight threads. At the opposite of what we expected, Xeon Phi's Linux has an issue with allocations in processes which degrades in the same way as it does with threads. We currently have no confirmed explanation for this phenomenon.

Numerical simulations based on MPI or OpenMP tend to have allocation phases, favoring those pathological cases.

2.4 Page Zeroing in Kernel

During page faults, the kernel needs to reset all the physical pages it provides to a process in order to prevent information leaks from another one or even from the kernel itself. Such behavior is performed by the `clear_page` kernel function.

Tools like *perf* (from Linux kernel sources) can report this reset function in the top list for intensive allocation tests. On a 12 core computer (Bi-Westmere), it costs around 1400 cycles. As a single page fault costs 3400 cycles, it implies that 40% of the cost comes from page zeroing. Furthermore, in multi-thread context, this function is used between locks, possibly degrading scalability.

2.5 Huge Pages

Current architectures rely on small caches to speedup address transitions (Translation Lookaside Buffer: TLB). 4KB pages can be too small to exploit efficiently those TLB

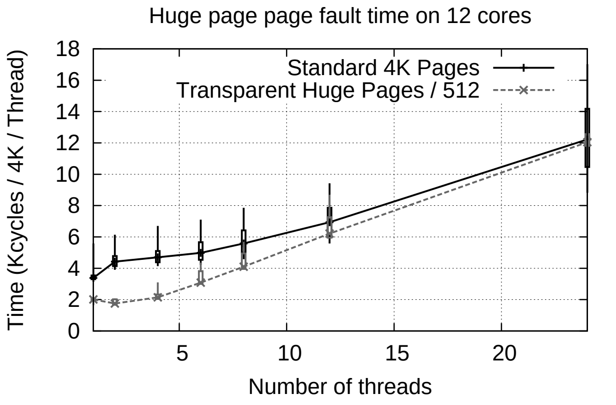

Figure 3. Page fault time measurement with Transparent Huge Pages on a 2*6 hyper-threaded core computer. The measurement was done by touching each 2MB page once. For comparison we report the time per 4K dataset by dividing the huge page time by 512.

with large datasets; therefore, current processors also provide support for larger pages (*huge pages* on Linux). Huge pages (2MB) are larger than standard ones (4KB), thus reducing the number of faults by a factor 512 for a given amount of memory. Nonetheless, an important part of the allocation cost persists as it consists in zeroing the newly allocated page, a cost which increases with page size.

Figure 3 depicts page fault timings for Linux Huge Pages on twelve cores. Fault time is measured for each 2MB page and scaled down by a factor 512 to compare them with standard pages on the bases of a fixed size. It can be seen that Huge Pages cost 2000 cycles per 4K compared to 3400 for the standard ones, yielding a 40% improvement. But with a larger number of threads (24 threads), costs are now comparable for the two methods. Hence, huge pages provide performance improvements for serialized faults, but suffer from the same scalability issue.

2.6 Insights

As discussed, the whole allocation chain has to take into account parallelism constraints. Because of the evolution of the memory per core ratio, applications might release their memory more often, issuing more calls to the memory management chain. On the kernel side we reminded a scalability problem with page fault (see figures 1, 2 and 3). In this paper, we discuss an improvement of page fault performance by limiting zero paging cost (40% of a page fault), with a possible gain for scalability by reducing the work done in a critical section.

3. Motivating Results

This section provides some related results which motivates our kernel-level proposal. It is based on tests of different memory allocators on large NUMA nodes with a complex real application. Here, we were mostly interested in keeping the memory at user-space level through the memory allocator itself in order to immune the application from the previous issue. However, even if it can solve the performance issue we will show that it increases the memory consumption, which as discussed might not be acceptable anymore for some kind of workload.

3.1 Tested Platforms

Experimentations were done on three different platforms with different number of cores and NUMA hierarchies. The first one (A) is a 12 core Bi-Westmere NUMA computer with hyper-threading support. Most of our tests were done on this node thanks to a root access. The two others provide (B) 32 or (C) 128 cores with respectively 4 and 4*4 NUMA nodes of octo-core Nehalem-EP CPUs. As those two platforms were in production, we weren't able to test our kernel patch or huge pages impact on them.

3.2 Tested memory allocators

Although user-space memory allocators were originally mostly, sequential there are now several parallel implementations such as Jemalloc[8] from FreeBSD, TCMalloc[17] from Google, Hoard[2], Streamflow[13] or MAMA[11]. Here, we focus on Jemalloc and TCMalloc which are production grade, widely used and sufficient to demonstrate our points of interest.

Naturally, allocators are all different, providing their own policies for varying purposes. For our goal of interest, Jemalloc reduces the memory footprint by continuously returning unused segments to the system. As a counter-part, it delegates the pressure to the underlying operating system. Whereas it is a good trade-off on common workstations, we will show that such a model finds its limitation with larger number of cores due to growing system time, and can become a real issue on 128 cores.

In comparison, TCMalloc aims at keeping unused memory for faster reuse. This approach tends to use more memory and might be problematic in HPC context. In addition, special care must be taken when reusing memory at user-space level, particularly in NUMA context. TCMalloc does not provide explicit NUMA support so we can observe some side effects on large compute nodes. An experimental patch has been developped by AMD to made TCMalloc NUMA aware[12], nevertheless this work haven't been tested for this paper.

These two allocators cover the policies spectrum by either carefully freeing memory to limit the footprint or keeping memory to improve performance (limiting the number of interactions with the OS).

In addition, we used the allocator from MPC (Multi-Processor Computing)[4, 15] framework. This framework aims at providing a unified parallel runtime to mix different parallel programming models (currently MPI, pthreads and OpenMP) for multi-processors/multi-cores NUMA nodes. From the memory management point of view, the MPC allocator provides a configurable reuse policy of large segments. As we have more control (especially NUMA support) we can use several profiles to observe the effects which impact the performance with a unique allocation algorithm. Hence, it provides a coherent way to extract parameters' impact. We used three different profiles in this paper. The *low memory* (*lowmem*) profile return all the unused memory to the kernel similarly to Jemalloc. The *NUMA* and *UMA* ones try to keep up to 500MB of unused memory per NUMA node. The two last policies differ by their explicit or non-support of NUMA hierarchy. The UMA profile is quite similar to TCMalloc behavior, except for the amount of unused memory it maintains in the allocator and some implementation details.

3.3 User-Space Pools for Large Segments

We validated both our approach (user and kernel-space) with a large MPI application. Hera[10] is a 2D/3D multi-material, multi-physic simulation code with Adaptive Mesh Refinement (AMR). Due to the AMR mechanisms, this application consumes a lot of memory and stresses the allocator with a huge number of allocation cycles of various sizes.

Tests were executed in multi-threaded context by using the MPC framework which provides a thread-based MPI implementation. It executes MPI tasks in threads instead of processes. Table 1 provides some measurements obtained while running Hera in hydrodynamic 3D/AMR mode on the three different architectures with four allocators: the default one from glibc, Jemalloc, TCMalloc and the MPC one.

Those results show the large impact of the allocator choice for this application. Of course, such large nodes imply NUMA effects, but we can also notice the effect of consumption policy on system time. On each test, jemalloc provides gains around 2GB of memory compared to default glibc, but it multiplies system time by 7 on the 128 core node (C4,C5). TCMalloc provides better performance on the 12

A: Bi-Westmere 12 cores (2 * 6)					
	Allocator	Total (s)	User (s)	Sys. (s)	Mem (GB)
1	MPC-NUMA	135.14	132.63	1.79	4.3
2	MPC-UMA	146.11	143.50	1.86	4.3
3	MPC-lowmem	162.96	130.98	16.20	2.0
4	Standard glibc	143.89	130.10	8.53	3.3
5	Jemalloc	143.05	128.07	14.53	1.9
6	tcmalloc	141.14	139.98	0.65	6.9

B: Nehalem-EP 32 cores (4 * 8)					
	Allocator	Total (s)	User (s)	Sys. (s)	Mem (GB)
1	MPC-NUMA	89.33	64.34	2.39	15
2	MPC-UMA	94.82	71.41	2.58	15
3	MPC-lowmem	248.17	74.19	87.21	6.7
4	Standard glibc	101.11	67.43	9.41	8.1
5	Jemalloc	145.73	70.49	57.32	6.7
6	TCMalloc	106.28	82.97	1.96	8.6

C: Nehalem-EP 128 cores (4 * 4 * 8)					
	Allocator	Total (s)	User (s)	Sys. (s)	Mem. (GB)
1	MPC-NUMA	120.07	100.44	5.64	16.9
2	MPC-UMA	229.38	207.25	5.88	16.5
3	MPC-lowmem	762.47	460.53	56.13	14.1
4	Standard glibc	284.06	170.94	15.9	14.1
5	Jemalloc	351.49	214.54	123.99	12.2
6	TCMalloc	438.42	396.59	27.57	14.4

Table 1. Performance measurement of Hera on various NUMA nodes by using different memory allocators with a single process and one thread per core. To be comparable, the user and system times are given per thread.

core computer (A6) but shows some limits on larger ones (B6,C6).

The three policies provided by MPC allocator help us to decouple the origin of such large performance gap (NUMA or system time). Comparing the NUMA to UMA mode extracts the cost related to non-explicit NUMA support in allocators. On the 128 core computer it improves the performance by 48% (C2,C1). Comparing NUMA with the low memory profile extracts the cost of freeing intensively the memory. This case shows a huge degradation on all architectures, increasing with the number of cores up to a factor 6 on 128 cores (C1,C3).

3.4 Conclusion

Those results show that we can obtain significant improvements (up to a factor 2) by using memory pools at allocator level. This technic requires an explicit NUMA support to be efficient on large NUMA nodes. But it encounters two major limitations while controlling memory consumption. Firstly, the reuse of large segments can increase the memory consumption if the application rely on lazy paging and do not use all the memory it requests. Secondly, the kernel has no ways to request the ununsed memory from user-space pools in case of memory starvation. This work motivates our proposal to improve page fault performance.

4. Related Work

Previous part of this paper refers to and completes the scalability issue observed on Linux by Austin T. C. and al.[5] with up to 80 cores. In their paper, they proposed a new scalable algorithm to update the page table. In contrast, our article focus on reducing the other part of page fault time: the `clear_page` function, approach which can complement their own one.

Similar work on page zeroing strategy has already been done by Microsoft developers and currently used in Windows kernel[16]. Their implementation moves the call to page reset into an independent low priority system thread. Experimentally, a single page fault of windows 7 64bit on Intel Core i7 Sandy Bridge cost 1900 cycles. The Linux 3.6.8 one cost 3700 cycles on same hardware. We propose to be stricter and avoid as much as possible the unproductive work done by `clear_page`. It can be achieved by introducing page reuse at kernel level. As shown in section 6, we achieve similar performance improvements without an extra system thread. In HPC context it might be preferable to avoid the extra noise of a system thread or one per NUMA node (16 on our 128 core node).

Since version 2.6.38, thanks to the work of A. Arcangeli (Transparent Huge Pages[1]) Linux huge pages can be used without being "booked" at boot time, making them more usable. This patch was also back-ported to the version 2.6.32 provided by Redhat and Centos, hence, all Huge Pages tests from this paper were done with this implementation. Most of the time, huge pages are studied to reduce TLB misses [9, 14, 18]. But, in this article, we studied them for their potential improvement of allocation performance.

5. Kernel-Level Memory Pools

Section 3.3 is interested in limiting interactions with the OS thanks to user-space memory pools. Of course it implies higher memory consumption which can be problematic for upcoming architectures. In contrast, this section focuses on improving the cost of memory page faults in order to help finding a more favorable balance between memory consumption and performance. It targets the `clear_page` function with the goal of reducing its usage to the minimum.

5.1 General Design

In order to optimize page faults, Windows uses an independent thread to reset the memory on unused CPU cycles. In contrast, we decided to relax the zeroing constraints. After allocating memory, most users do a first write access to initialize their data segments making zeroing unproductive. Moreover page zeroing accounts for an important part of page faults' costs. Hence, it might be better to avoid them as most as possible, saving CPU cycles, bandwidth and energy. Nonetheless, page zeroing is required for security reason. Thus, constraints can be expressed as follows:

1. Page zeroing is required to prevent information leaks between processes or from the kernel itself.
2. Some user-space tools suppose lazy page allocation and rely on page zeroing to initialize their segments on first use.

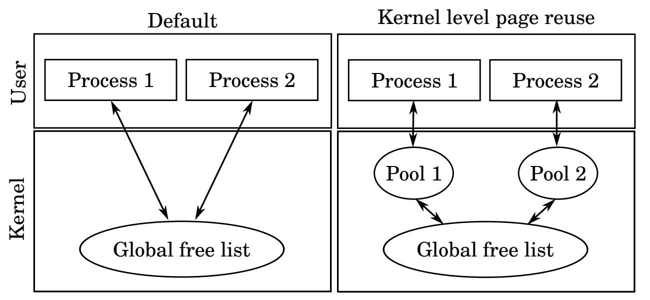

Figure 4. Introduce per process memory pools at kernel level to avoid page zeroing for local pages. The local page pools are used only for segments with reuse flag enabled.

The `malloc` function commonly recycles small segments to allow quicker allocations. So, we propose the same at kernel level. We considered that pages coming from one process can be reused in the same process without being zeroed. Hence, we implemented a per process kernel level memory pool with a page granularity (figure 4).

It completely removes page zeroing costs but alter `mmap`'s semantic by possibly returning non zeroed pages. To maintain compatibility this behavior is triggered only explicitly via new flags for `mmap` and `madvise`. This approach has several benefits compared to the user-space memory pools:

1. At kernel level, it does memory reuse at page granularity where allocators use virtual segment granularity, so have no control for the inner part of large segments.
2. It supports transparently kernel's NUMA policies with the advantage of manipulating pages instead of virtual segments.
3. Kernel can easily reclaim those pages to reuse them into other processes or for its own usage.

The `malloc` and `realloc` definitions do not impose to reset the requested chunks, allowing those functions to safely exploit this new feature.

5.2 Memory Consumption and Swap

With such a memory pool per process, one can question the limits on memory consumption. We answer this question here even if this part is currently not fully implemented.

In order to limit the memory consumption, we capture only the pages from freed segments with page reuse flag enabled. It avoids the pathological case which consists in capturing all the pages but not reusing all of them if part of the new allocated segment does no enable the page reuse flag. This approach naturally limits the memory consumption of the pool to the maximal fraction of virtual space with page reuse flag.

This default behavior is the best case for performance without needing a shared counter (which may add inter-thread contention) to enforce this limit. The non-capture of standard segments also maintains the security of potential cryptographic libraries which might not want their keys to be reused elsewere in the process after freeing the memory.

Another and more important question is about swap and page reclaim. In case of memory needs, the page reclaim algorithm must loop over those kinds of pages as a first priority before checking the current inactive page list. In term of implementation, it may be done by adding an additional list near to the current LRU (Least Recently Used) active/inactive page lists in each memory zone (NUMA). This new list has to point per process pools and not pages to ensure synchronizations. We propose to use round robin or random selection to reclaim pages in a different process between each pass. Round robin or random selection algorithm may provide smoother impact on running applications. Searching the larger pool may be another solution, but seems costly compared to the others available solutions. Of course in case of high memory pressure, it could be a good idea to disable or force a lower limit on local memory pools size.

Swap being a very critical area of memory management; we decided to postpone our integration into the page reclaim algorithm. Nonetheless we made sure that there are no major difficulties in its implementation. This choice can be justified in HPC context where we deal with a reduced number of processes which can monopolize resources, so current support is sufficient for a first evaluation of interest.

5.3 Support of Huge Pages

Our current patch was tested on both kernel version 2.6.36 and Redhat 2.6.32, so, with basic support of huge pages. Such a support was obtained by applying the same approach as for standard pages. The cache structure was modified to support two page sizes in two distinctive lists. Page capture and reuse was done in the same way as for standard 4K pages.

Supporting dual page size may require a more complex counter to ensure a global limitation on the two sub memory pools. If an application uses successively both page sizes we may keep a large unused pool. Page reclaim support can solve this issue which might be considerd uncommon in practice.

5.4 Integration in User-space Memory Allocator

The user-space memory allocator (`malloc`) is the main entry point to support this new feature in applications. Most allocators base their implementation on `mmap` for large memory segments, in this case, the new `mmap` flag can be activated for `malloc` and `realloc` while making sure that `calloc` provides zeroed memory in order to preserve the semantic.

Such support in allocators can be trivial if current allocator doesn't rely on zero pages to setup their headers. Otherwise integration might require more work to make sure that all segments are correctly initialized. For instance this support was implemented in MPC parallel memory allocator without too much change except propagating a boolean over the call paths leading to `mmap`, possibly activating the

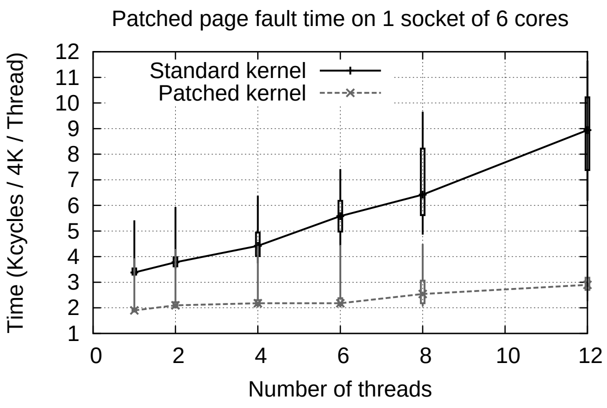

Figure 5. Micro-benchmarking of the patched kernel while binding the measurement process on a single 6 core Westmere socket to avoid NUMA effects.

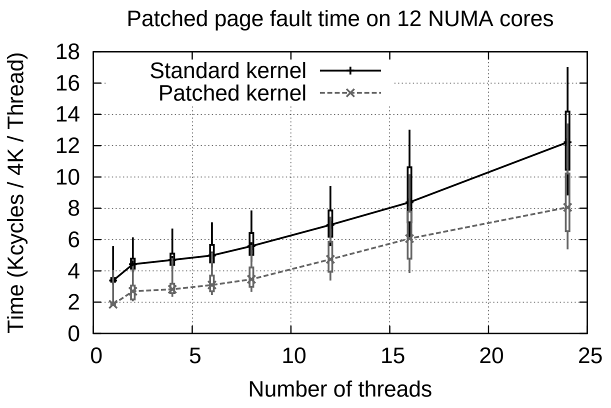

Figure 6. Micro-benchmarking of the patched kernel while running on 12 core Bi-Westmere NUMA node.

reuse feature. We also patched Jemalloc in the same way as it already propagates such a boolean in its call paths. This one is a good candidate to validate this approach due to its low memory profile.

5.5 Conclusion

This section developed an approach to reduce Linux page fault time for applications doing frequent large memory allocations. We went further than Windows by removing as most as possible the needs of page zeroing instead of moving it elsewhere. We extended current `mmap` and page fault semantic such that the caller can express his constrain about zeroing. In order to avoid page zeroing while maintaining tightness between processes, we added a per process memory pool at kernel level for local memory reuse. Such improvement may benefit to user-space allocators by allowing them to free memory more regularly with lower system overhead.

6. Experimental Results

The previous section describes a kernel-level solution to improve allocation performance by reducing the need of page zeroing. This section provides some measurement showing benefits of our approach on a micro-benchmark and two HPC applications. Remark that the given experiments have been repeated with reproducible results.

6.1 Micro-benchmark

The first test consists in running the micro-benchmark from section 2.3 bound on one socket of architecture A (Bi-Westmere) to avoid NUMA conflicts. Thanks to hyper-threading, measurements used up to twelve threads. Figure 5 shows significant gains on page fault performance. The sequential time decreases from 3400 to 1900 cycles (a 45% improvement). In this configuration, scalability is also slightly improved falling from 8950 cycles for twelve threads to 2900 cycles thanks to the patch, so a performance improvement of 66%.

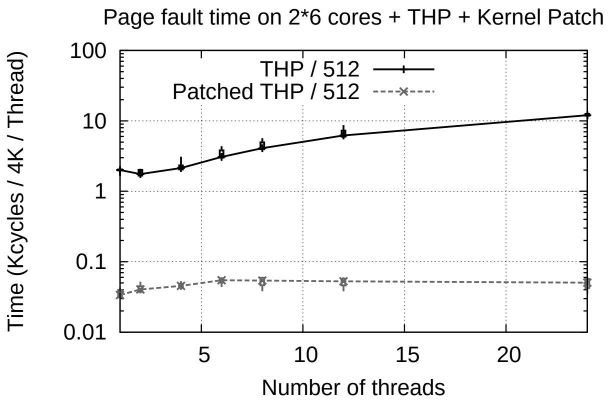

Figure 7. Micro-benchmarking of the patched kernel while running on a 12 core Bi-Westmere NUMA node with huge pages.

Figure 6 presents the results on two sockets with up to 24 threads. There are still performance gains, but NUMA becomes the new limitation on page fault performance. In this situation our patch provides a constant gain of 1500 cycles corresponding to 33% for 24 threads. Performing the same measurement without using our pool but returning directly non zeroed pages (creating a security hole) left the results unmodified, proving that NUMA related performance impact mostly comes from page fault implementation itself (locks), excluding zeroing and our pool implementation. As shown, our approach improves Linux page fault system performance and scalability. However, page fault's scalability on large number of cores with NUMA effects still requires important work.

Transparent Huge Pages gains are larger as page fault's cost mainly imputes to page zeroing. In this case, we ob-

Huge pages with Hera.					
	Allocator	Total (s)	User (s)	Sys. (s)	Mem (GB)
1	MPC-NUMA	137.89	135.13	1.86	6.2
2	MPC-UMA	147.15	144.38	1.97	6.2
3	MPC-Lowmem	196.51	140.39	28.24	3.9
4	jemalloc	144.72	129.62	14.66	2.5
5	std	149.77	130.08	12.92	4.5
6	Tcmalloc	150.13	149.03	0.51	6.5

Table 2. Huge page benchmark with Hera application on the Bi-Westmere computer. User and system times are given per thread.

served (figure 7) improvements up to a factor of 57 in sequential with a cost of 35 cycles per 4K page.

6.2 Hera Application

We start by looking for improvements through the standard huge pages of Linux kernel for the Hera application in the same way as section 3.3. Table 2 can be compared to the one from section 3.3 (architecture A). It shows that in our specific case, the 2MB huge page size is too large and tends to increase memory consumption of all memory allocators except TCmalloc which maintains fix consumption. This growing effect can be explained by the use of virtual allocations which are fully mapped with physical huge pages where it wasn't with 4K pages. In addition, due to the page fault scalability issue and zeroing of more physical memory, it degrades slightly the performance compared to the results from section 3.3.

Table 3 depicts results from running Hera with and without our kernel patch on the Bi-Westmere computer (A). On the two tested allocators with standard pages (S2,S3 and S4,S5), it improves the global performance by 2%. But system time shows an overall improvement of 33%, matching our previous micro-benchmark. Huge pages now improve the global execution time by 30% and 5% for the two low memory consumption allocators (H2,H3 and H4,H5). System time was respectively improved by a factor 9.7 and 2.3. With MPC allocator, it made the low memory mode as fast as the NUMA one (H1,H3), in other words, we compensate the overhead induced by freeing the memory.

6.3 HydroBench Application

HydroBench[6] is a lighter numerical simulation available on github. This application is a 2D hydrodynamic benchmark with hybrid MPI + OpenMP support. It is executed in full OpenMP mode with 12 threads on the Bi-Westmere computer (A). The first version of this benchmark suffered from a large number of memory allocations. As it stresses the memory management chain, we run our tests with this slower version and also compare to the patched one. Due a simpler allocation pattern it has roughly the same memory consumption with all tested allocators.

Table 4 shows gains around 12% on the overall runtime and a reduction of 37% on system time imputes to our kernel patch (S2,S3). Of course, we can reach better performance

Kernel patch and standard 4K pages (S)						
	Allocator	Kernel	Tot.	User	Sys.	Mem.
1	*MPC-NUMA*	Std.	*135.14*	*132.63*	*1.79*	*4.3*
2	MPC-Lowmem	Std.	161.58	131.00	15.97	2.0
3	MPC-Lowmem	Patched	157.62	132.70	10.60	2.0
4	Jemalloc	Std.	143.05	128.07	14.53	1.9
5	Jemalloc	Patched	140.65	130.80	9.32	3.2
Kernel patch and Transparent Huge Pages (H)						
	Allocator	Kernel	Tot.	User	Sys.	Mem.
1	*MPC-NUMA*	Std.	*137.89*	*135.13*	*1.86*	*6.2*
2	MPC-Lowmem	Std.	196.51	140.39	28.24	3.9
3	MPC-Lowmem	Patched	138.77	131.70	2.90	3.8
4	Jemalloc	Std.	144.72	129.62	14.66	2.5
5	Jemalloc	Patched	138.47	130.44	6.40	3.2

Table 3. Benchmarking our kernel patch with the Hera application on the Bi-Westmere computer. We used 12 threads in one process. User and system times are given per thread. Times are given in seconds and memory in GB.

Standard pages. (S)							
	App.	Kernel	Allocator	Tot.	User	Sys.	MFlops
1	Std.	Std.	Glibc	1:29	543.3	30.7	1770
2	Std.	Std.	Cust.	1:28	532.9	31.5	1775
3	Std.	Patch	Cust.	1:19	534.9	19.7	1775
4	Std.	Std	Cust. KM	0:59	528.6	0.5	2649
5	Patch.	Std	Glibc	0:43	475.0	0.4	3606
Transparent Huge Pages. (H)							
	App.	Kernel	Allocator	Tot.	User	Sys.	MFlops
1	Std.	Std.	Glibc	1:13	533.3	18.8	2140
2	Std.	Std.	Cust.	1:18	550.7	17.8	2007
3	Std.	Patch	Cust.	1:11	557.0	7.0	2224
4	Std.	Std	Cust. KM	1:05	566.3	1.0	2412
5	Patch.	Std	Glibc	0:50	539.2	0.4	3554

Table 4. Measurements with HydroBench OpenMP code with 12 threads on the Bi-Westmere computer. The *KM* (Keep Memory) option of our allocator tries to keep large memory segments in user-space level for future reuse.

improvement (44%) by working at allocator level as the number of fault is drastically reduced (S2,S4). Huge pages improve by themselves performance by 12% (S1,H1) and our kernel patch improves the related system time by a factor 2.5 (H2,H3) making the user-space memory pool less efficient (H3,H4). Better gains, up to 52%, are logically obtained by fixing the problematic allocations pattern at code level (S4,S5 and H4,H5). But such a patch might not be possible in more complex applications.

6.4 Conclusion

This section provides results obtained by introducing memory pools at kernel-space level for the Bi-Westmere computer. The results of our kernel prototype show improvement close to 30% on system time for the tested cases. It generates in one case gains of 12% on total run time. Coupled with huge pages, our kernel patch provides improvement of 33% on the Hera application, hence, providing the same improvement as the user-space approach but with lower memory consumption.

7. Conclusion

This paper focuses on page fault cost for memory intensive applications in multi-threaded context. As it starts becoming a bottleneck on large NUMA nodes, we explored some possible solutions. As user-space solutions tend to increase the memory consumption, we looked down to the kernel.

Hence, we studied the page fault strategy and more precisely the use of kernel `clear_page` function which is responsible of clearing the physical pages before their usage by processes. With current Linux kernel and hardware, it costs 40% of a page fault. Some OSs like Windows already uses optimizations to move the use of this function out of page fault management. Conversely, by looking to the page fault semantic we proposed a stricter approach by removing as most as possible the need of this function. Hence we can expect to save some CPU cycles, memory bandwidth and possibly energy.

We proposed to add a per-process memory pool in kernel structures. This way, free pages are reused by the same process; so, do not need anymore to be cleared. Its usage is enabled by extending the flags provided to `mmap` and can be supported by patching memory allocators for `malloc` and `realloc`. At the opposite of user-space methods, it handles reuse at page granularity which is better to fit with the real access pattern of large segments and to maintain NUMA properties. In addition, those free pages can be reclaimed for others processes or for the kernel itself.

We prototyped our kernel-space proposal in Linux 2.6.32 and 2.6.36 and ran some tests on a 2*6 core NUMA computer. By micro-benchmarking, we observed performance improvements up to 45% on sequential page faults and up to 66% for 12 threads. But on NUMA architectures, additional effects increase the cost of the other part of the fault system reducing gains to the expected constant cost of a page reset. Anyway it still significantly improves the page fault time by 32% for 24 threads.

We finally tested two multi-threaded applications (Hera and HydroBench) with our patched kernel. Depending on their compute/allocation ratio, it provided respectively an improvement of 1% and 12% on the global runtime. In the two cases we observed the expected 30% reduction of system time. With huge pages, it provides improvements of 33% comparable to the user-space memory pools without increasing the memory consumption of the application.

Acknowledgments

Acknowledgement : this work is supported by the PERF-CLOUD project. A French FSN (Fond pour la Socit Numrique) cooperative project that associates academics and industrials partners in order to design then provide building blocks for a new generation of HPC datacenters.

References

[1] A. Arcangeli. Transparent Hugepage Support, KVM Forum `http://www.linux-kvm.org/page/Kvm_Forum_2010`, 2010.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. In *Proceedings of ASPLOS IX*, 2000.

[3] D. P. Bovet and M. C. Ph. *Understanding the Linux Kernel*. Third edition edition.

[4] P. Carribault, M. Pérache, and H. Jourdren. Enabling low-overhead hybrid mpi/openmp parallelism with mpc. In *Proceedings of IWOMP'10*, 2010.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *Proceedings of ASPLOS XVII (2012)*.

[6] G. C. de Verdière. Hydrobench, `https://github.com/HydroBench/Hydro`.

[7] J. Dongarra, P. Beckman, and al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1).

[8] J. Evans. A Scalable Concurrent malloc(3) Implementation for FreeBSD `http://www.canonware.com/jemalloc/`, 2006.

[9] M. Gorman and P. Healy. Performance characteristics of explicit superpage support. In *Proceedings of ISCA'10*, 2012.

[10] H. Jourdren. HERA: A Hydrodynamic AMR Platform for Multi-Physics Simulations. In *Adaptive Mesh Refinement - Theory and Applications*, Lecture Notes in Computational Science and Engineering, 2005.

[11] S. Kahan and P. Konecny. "MAMA!": a memory allocator for multithreaded architectures. In *Proceedings of PPoPP '06*, 2006.

[12] P. Kaminski. Numa aware heap memory manager (amd).

[13] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of PLDI '04*, 2004.

[14] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of OSDI '02*, 2002.

[15] M. Pérache, P. Carribault, and H. Jourdren. Mpc-mpi: An mpi implementation reducing the overall memory consumption. In *Proceedings of European PVM/MPI '09*, 2009.

[16] M. Russinovich and D. A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. 2009.

[17] P. M. Sanjay Ghemawat. Tcmalloc : Thread-caching malloc, `http://goog-perftools.sourceforge.net/`.

[18] K. Yoshii, K. Iskra, H. Naik, P. Beckman, and P. C. Broekema. Performance and scalability evaluation of 'big memory' on blue gene linux. *Int. J. High Perform. Comput. Appl.*, 25(2), May 2011.