Enabling High-Performance Memory Migration for Multithreaded Applications on Linux

Brice Goglin, Nathalie Furmento

Enabling High-Performance Memory Migration for Multithreaded Applications on Linux

Brice Goglin[1], Nathalie Furmento[2]

[1] INRIA, [2] CNRS

LaBRI – 351 cours de la Libération

F-33405 TALENCE – FRANCE

[1] Brice.Goglin@inria.fr — [2] Nathalie.Furmento@labri.fr

Abstract

As the number of cores per machine increases, memory architectures are being redesigned to avoid bus contention and sustain higher throughput needs. The emergence of Non-Uniform Memory Access (NUMA) constraints has caused affinities between threads and buffers to become an important decision criteria for schedulers.

Memory migration enables the dynamically joined distribution of work and data across the machine but requires high-performance data transfers as well as a convenient programming interface. We present the improvement of the LINUX migration primitives and the implementation of a *Next-touch* policy in the kernel to provide multithreaded applications with an easy way to dynamically maintain thread-data affinity.

Microbenchmarks show that our work enables a high-performance, synchronous and lazy memory migration within multithreaded applications. A threaded LU factorization then reveals the large improvement that our *Next-touch* policy model may bring in applications with complex access patterns.

Keywords: *Memory Migration, Next-Touch, Lazy Migration, Multithreaded Applications, Affinity, NUMA, Linux.*

1 Introduction

While the number of cores within machines keeps increasing, the way to exploit the computing power has to be redesigned. The architecture is indeed becoming more and more complex, with multicore and/or multithread processors and multiple level of shared caches. Threads are a convenient way to program these highly-parallel hosts and parallel programming languages such as OPENMP try to ease the mapping of parallel algorithms onto the architecture.

It is well-known that the quality of the thread scheduling has a strong impact on the overall application performance [4] because of thread and data affinities. However, this issue is now becoming critical due to the variable memory access latencies that more and more architectures exhibit. NUMA (*Non-Uniform Memory Access*) architectures indeed appear as the solution to ease the scalability of modern memory architectures, by interconnecting distributed memory banks. With local data access being significantly faster than remote access, data locality emerges as a critical criteria for scheduling threads, and it becomes important to be able to migrate data buffers together with their accessing tasks.

Modern operating systems provide the ability to manage these NUMA architectures by allocating data buffers on a given node or migrating at runtime. Such features for instance enable the adaptation of the data distribution to the current thread scheduling in dynamic and irregular applications such as adaptive mesh refinement. While being the most widely used operating system in high-performance computing, LINUX however only gained NUMA awareness recently and still has limited support for migrating buffers within multithreaded architectures. We thus propose in this article an in-depth study and optimization of the LINUX migration primitives to provide application with a convenient way to react to dynamic thread scheduling.

The remaining of the paper is organized as follows. Background about NUMA architectures, affinities between threads and memory, and memory migration in LINUX are introduced in Section 2 as well as our motivations. Section 3 describes the implementation of a high-performance memory migration primitive for threaded applications, its usage within a user-space *Next-touch* policy, and the design of an optimized *Next-touch* strategy in the LINUX kernel. These ideas are evaluated in Section 4 which presents microbenchmarks and application performance. Before concluding, related works are summarized in Section 5.

2 Background and Motivations

In this section, we introduce NUMA architectures and their impact on application programming through affinities between threads and memory. We then present how LINUX deals with memory migration before describing our motivations to improve the provided features.

2.1 NUMA Architectures

The emergence of highly parallel architectures with many multicore processors raised the need to rethink the hardware memory subsystem. While the number of cores per machine quickly increases, memory performance remains several orders of magnitude slower. Concurrent accesses to central memory buses imply contention, causing the overall performance to decrease. It led hardware designers to drop the centralized memory model in favor of distributed and hierarchical architectures, where memory nodes and caches are attached to some cores and far away from the others (*Non-Uniform Memory Access*, NUMA). This design has for instance been widely used in high-end servers based on the ITANIUM processor. It now becomes mainstream since AMD HYPERTRANSPORT [5] (see Figure 3) and the upcoming INTEL QUICKPATH memory interconnects will dominate the server market soon. Indeed, these new memory architectures assemble multiple memory nodes into a single distributed cache-coherent system. It has the advantage of being as convenient to program as regular shared-memory SMP processors, while bringing a much higher memory bandwidth and much less contention.

However, while being cache-coherent, these distributed architectures have non-constant physical distance between hardware components, causing their communication time to vary. Indeed, a core accesses local memory faster than other memory nodes. The ratio is often referred to as the *NUMA factor*. It generally varies from 1.2 up to 3 depending on the architecture. It therefore has a strong impact on application performance. Not only the application will execute slower if accessing remote data, but also contention may appear on memory links if two processors access each others' memory nodes.

2.2 Threads and Memory Affinity

The increasing number of processing cores in machines raises the need to parallelize applications within single nodes. Many approaches still rely on MPI to enable communication between all processes running on the same node. This model brings a uniform interface to communicate with both local and distant processes. However, while being convenient for programmers, this approach has performance issues and may only be interesting for clusters. Using threads, for instance with OPENMP, is now considered as a promising approach to exploiting shared-memory hosts, even when mixed with MPI for inter-node communications [3].

Running multiple threads on a NUMA machine raises the problem of NUMA penalties as we explained in the former section. Achieving optimal performance requires to carefully place each thread as close as possible to the data it accesses [2, 1]. It may be achieved either by migrating threads or by migrating memory buffers. However, such migrations have to be decided after looking at the overall load-balancing among all cores and memory banks in the machine in order to maintain a good utilization of the hardware resources. Multithreaded applications may thus distribute many threads on distant cores of a single host but they should place them while also distributing the process buffers among all memory nodes.

The commonly-used strategy to get a proper memory placement is called *First-touch*. It relies on the operating system laziness: each virtual page is allocated in physical memory only when touched for the first time. A NUMA-aware operating system is thus able to allocate the page on a memory bank near the core that caused the page-fault by touching it. However, many dynamic applications cannot rely on such a policy since there is often few guarantees that the first touching thread will be the one accessing the data intensively in the future. The application may thus only manually place the page on the right memory node at startup, assuming it knows where the associated thread will be running in the future. Another strategy would be to rely on page migration later.

Highly-dynamic applications such as adaptive mesh refinement have their thread/data affinities actually varying during the execution since the amount of computation in each buffer depends on earlier results. When the optimal thread/memory distribution evolves, the load-balancing has to be maintained between all the cores and the memory banks. If the application is aware of where the buffers have been allocated and where its threads are running, it may migrate buffers accordingly synchronously. Thus, another solution called *Next-touch* is a generalization of the *First-touch* approach: memory buffers are marked as needing to be migrated near the thread that will first touch them in the future [8, 9, 12]. This feature is very convenient for application developers since they do not have to always know where all threads and buffers are placed. It is unfortunately rarely implemented, and as all memory migration operations, it has to be efficient so as to maintain the application performance.

2.3 Memory Migration in LINUX

Some proprietary operating systems such as Solaris or IRIX have had advanced NUMA support for a long time [15, 8]. These systems were widely used on NUMA architectures which at the time were rare and needed specific software support. LINUX has been the most commonly used operating system in high-performance computing for a decade but it is not yet the obvious choice for NUMA machines due to its limited NUMA support up to recently. Although it has been able to gather NUMA information from the hardware for a long time, the LINUX kernel only acquired some actual NUMA abilities in the last years with the addition of some page binding system calls (`set_mempolicy`, `mbind`, *etc.*) and recently some memory migration primitives (`move_pages`, *etc.*) [7]. These features are made available to user-space applications thanks for instance to the `libnuma` interface [6].

Thanks to these additions and now that NUMA architectures such as AMD HYPERTRANSPORT and INTEL QUICKPATH are spreading into the mainstream server market, it is expected that an increasing number of NUMA machines will run LINUX. It is therefore important to actually have LINUX providing all features that NUMA-aware applications may require, and to implement them efficiently.

There are two methods for migrating memory between NUMA nodes in LINUX, both are synchronous system calls:

`migrate_pages` moves *all* pages of a process from a set of NUMA nodes to another set. It is usually combined with the migration of threads on different cores, thus enabling the migration of entire processes to a different part of the machine. This is mostly a load-balancing feature that administrators use to split a large single machine into pieces (*cpusets*) and share it between multiple users.

`move_pages` moves an array of virtual addresses within a process address space to an array of specified NUMA nodes. It enables fine-grain placement of different buffers within a single application. This is therefore the interesting migration primitive for affinity-based scheduling of threads and buffer placement.

2.4 Motivations

Given the possible impact of memory migration on application performance, it is important to offer flexible programming interface to user-space. While thread schedulers are already able to place threads according to their memory affinities [11], load-balancing also requires to spread threads across all cores, and thus to redistribute data dynamically to match their needs. It is therefore important for memory migration primitives to be available and efficiently implemented. We hence propose in this article a study and improvement of the LINUX migration system call implementation.

The LINUX kernel only provides the aforementioned synchronous migration primitives. Pages are therefore either allocated on the correct NUMA node in the first place, or have to be migrated synchronously later. When migrating a thread onto another core, the scheduler has to know which buffers are associated to this thread, and decide immediately whether they should be migrated as well. However, there is no guarantee that all of the buffers will actually be accessed in the near future and furthermore the same thread could migrate again soon. Moreover, maintaining a knowledge database of all thread/memory affinities is difficult since each buffer may have different reading and/or writing access patterns, by one or several threads at the same time.

An asynchronous *Lazy Migration* primitive based on the *Next-touch* policy thus appears as a good solution to these dynamic behaviors since pages will only actually be migrated when needed. Such a feature may be implemented in user-space using a segmentation-fault signal handler and the aforementioned synchronous migration primitives [12]. But this solution has an important overhead and requires the synchronous migration to be efficient. Fortunately, a kernel-based implementation in Solaris [8] has proven that the *Next-touch* approach may bring interesting results. We thus also envision in this article the implementation of such a lazy *Next-touch* migration scheme in the LINUX kernel and the comparison of its performance with a user-level model. An efficient solution is expected to provide a convenient and flexible way to support dynamic and irregular applications since the scheduler would take care of load-balancing across cores while the *Next-touch* policy would make sure the data locality is maintained.

3 High-Performance Memory Migration in LINUX

We introduce in this section the design of our high-performance memory migration features for threaded applications in LINUX. The performance of the synchronous `move_pages` system call is first discussed, before being used as a user-level-based *Next-touch* model. We then describe an optimized kernel-assisted *Next-touch* implementation.

3.1 Improving `move_pages` Performance

The `move_pages` system call has been introduced in LINUX kernel 2.6.18 as a way to migrate individual pages of a process address space. It is the preferred way to manipulate migration in threaded applications since the other primitive (`migrate_pages`) migrates an entire process. `move_pages` performs migration by reading an array of process virtual addresses and an array of destination nodes from user-space. It then migrates the corresponding pages accordingly before returning an array of status to user-space.

We diagnosed a dramatic performance problem when moving large buffers (see Section 4.2) and discovered that it was caused by the quadratic complexity of the implementation. Indeed, the processing of each array slot caused a linear lookup in the entire destination node array. By reworking the implementation, we were able to restore a linear behavior by removing this potentially very expensive lookup. This work will be integrated in the upcoming LINUX kernel 2.6.29. It enables buffer-size independent migration throughput, making dynamic migration of large memory areas in multithreaded applications acceptably fast (see Section 4.2).

3.2 Naive *Next-touch* Migration

Implementing a *Next-touch* policy in user-space on top of the LINUX kernel requires to be notified when the next touch occurs. Since memory accesses from the application are transparently processed by the processor and its *Memory Management Unit* (MMU), the only way to get an event is to generate a page-fault. The operating system page-fault handler usually either kills the process because of an invalid access (segmentation fault) or allocates a new page and fills it with the corresponding data (swap-in, copy-on-write, ...). A first way to implement the *Next-touch* policy in user-space would be to force pages to be swapped-out to the disk so that the next application access moves them back to the host memory, possibly on a different NUMA node. However, LINUX does not offer any reliable way to force such a swap-out[1] and its performance will be strongly limited by the storage subsystem.

The other solution consists in having the operating system generate a segmentation fault on next-touch and letting a user-space library catch the corresponding signal in user-space. We implemented this model thanks to the `mprotect` primitive enabling *fake* segmentation faults on valid areas. The signal handler then migrates pages and restores the initial protection as described on Figure 1.

The handler is actually responsible for deciding which buffer should be migrated. Since the workset structure in memory is known to the user-space library, the handler knows which buffer is actually accessed by the current thread. It is thus possible to benefit from `move_pages` optimized performance by migrating large buffers at once instead of migrating single pages when they are actually touched.

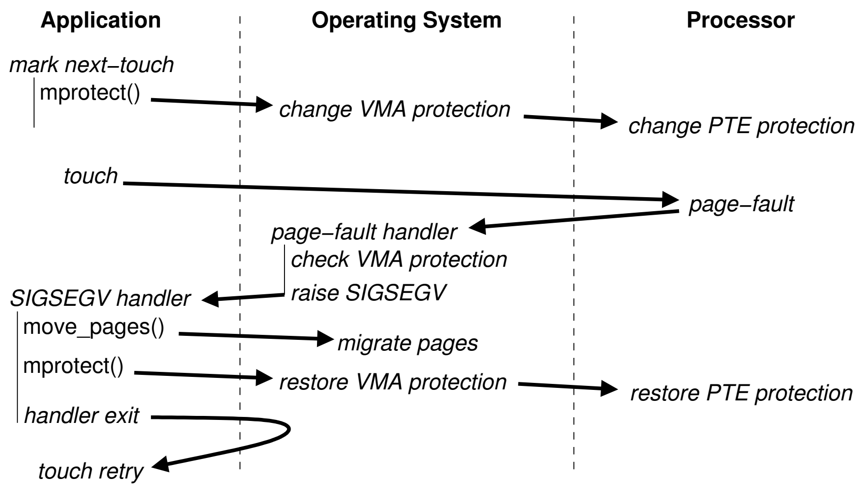


Figure 1: Implementation of the *Next-touch* policy in user-space through `mprotect` and a signal handler for segmentation fault.

Thanks to the `move_pages` performance improvement that we describe in the above section,

[1]The existing behaviors of the `madvise` system call in LINUX supports (for instance `REMOVE` or `DONTNEED`) do not implement the exact proper behavior.

this user-space implementation enables *Next-touch* migration with satisfying throughput. Once memory areas have been marked as *Next-touch*, each thread will automatically migrate the areas it accesses to its local memory bank. This approach also enables variable granularity since the user library may migrate larger or more complex areas (for instance a matrix column) since it knows the data structure in memory and the thread access patterns.

3.3 Kernel-based *Next-touch* Migration

The aforementioned user-space *Next-touch* model involves a return to user-space to run the signal handler before re-entering the kernel again for migration. Moreover, the *Translation Lookaside Buffer* (TLB) has to be flushed on all processors for each `mprotect`, while another flush is already needed for migration. These possibly expensive overheads justify the idea of optimizing the *Next-touch* implementation by modifying the LINUX kernel.

We propose a kernel-based *Next-touch* strategy that migrates pages within the page-fault handler as described on Figure 2. Our implementation was inspired by the *Copy-on-write* implementation in LINUX. The application marks pages as *Migrate-on-next-touch* using a new `madvise` parameter. The LINUX kernel removes read/write flags from the page-table entries (PTEs) so that the next access causes a fault. The page-fault handler checks whether the page has been marked as *Migrate-on-next-touch*. If so, it allocates a new page, copy the data and frees the old one. This implementation enables *Next-touch* migration since the new page is allocated on the current NUMA node by default.

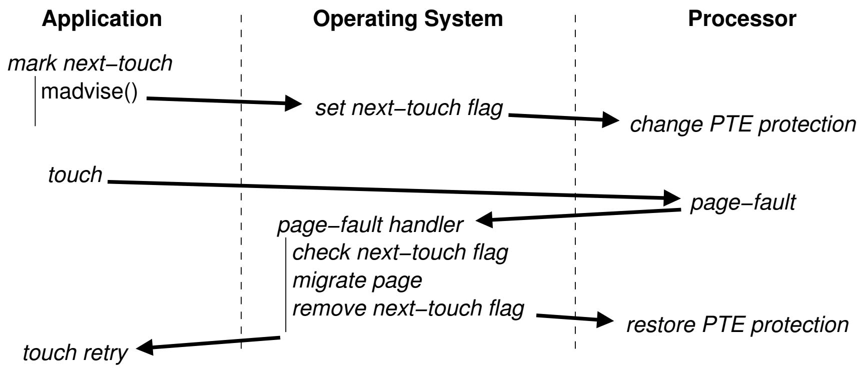


Figure 2: Implementation of the *Next-touch* policy in the LINUX kernel using `madvise` and a dedicated flag in the page-table entry (PTE).

3.4 Taking Migration Decisions

Both user-space and kernel *Next-touch* implementations actually have different semantics. The kernel one is page-based: even if the application touches many pages successively, each of them is migrated individually. The user-space implementation manipulates larger or more complex areas: the library offering the method can obtain from the application the description of the whole memory area (for instance a matrix column) and migrate it entirely as soon as a single page is touched. These different semantics are expected to make the kernel implementation to be used for small granularities while the user-space high overhead makes it more suitable for very large granularities. Moreover the user-space migration library knows the location of each page after the *Next-touch* has occurred, it does not have to query the kernel again for page location. This additional knowledge could enable some optimization for complex migration patterns where multiple successive migration are involved.

The migration decision should be taken when the scheduler actually moves a thread to a core on a different NUMA node. A basic model would synchronously migrate all buffers attached to the thread. The *Next-touch* policy provides a very convenient way to perform this joined migration of threads and buffers: When a thread migration is expected, the whole workset may be marked as *Migrate-on-next-touch*. Each migrated thread would then automatically migrate some pages in the background as soon as it touches them. There is thus no useless migration (unaccessed buffers are not touched and therefore not migrated) and the thread scheduler does not have to know which buffers are attached to which thread. We envision that marking as *Migrate-on-next-touch* could be driven by the OPENMP runtime system. Indeed, entering a new parallel section is usually a natural event that would cause the thread distribution to evolve. Some dedicated OPENMP pragma could also be used as a way to tell the system that the data layout in memory should be redistributed.

This work also leads to the idea of implementing a *Lazy Migration* model. Indeed, instead of migrating synchronously with `move_pages` when a thread starts accessing a distant buffer, the thread could mark the buffer as *Next-touch* and have it migrate in the background when it touches its pages. This is actually a very different usage of the *Next-touch* policy since the destination node is known in advance (*Next-touch* usually serves as a way to scatter a single buffer across multiple NUMA nodes when multiple threads start accessing it in a unpredictable manner). Additionally, if the thread actually touches only part of the buffer, only the corresponding pages will be migrated for real, reducing the overall overhead. This *Lazy Migration* strategy is especially expected to improve performance when the future access patterns of the threads are unknown, causing a synchronous migration of the entire buffer to be hard to decide.

4 Performance Evaluation

We now present in this section a performance evaluation of our migration techniques. We first look at microbenchmarks to understand each strategy performance and then look at real applications with a LU matrix factorization.

4.1 Experimentation Platform

The experimention platform is composed of a single host with four quad-core 1.9 GHz OPTERON 8347HE processors as depicted on Figure 3. Each processor contains a 2MB shared L3 cache and has 8 GB memory attached. This NUMA machine thus consists in four memory nodes (one per processor). Accessing a buffer on a non-local node costs from 1.2 to 1.4 more than accessing local memory.

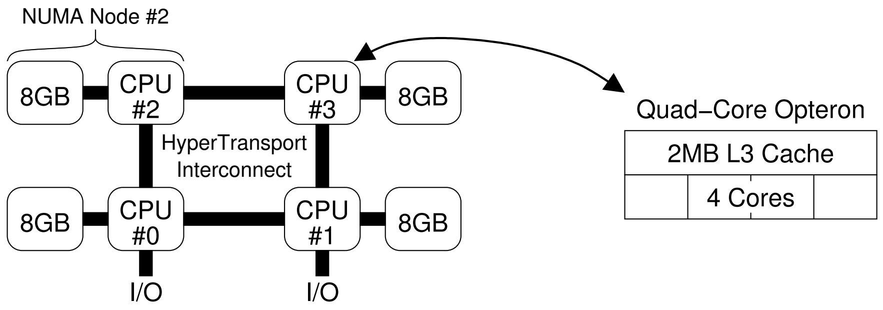


Figure 3: The experimentation host is composed of 4 quad-core OPTERON.

The machine runs LINUX kernel 2.6.27 with our `move_pages` performance improvement patch as well as our *Next-touch* page-fault-based implementation.

4.2 Synchronous Migration Performance

Figure 4 presents the throughput of memory copy and migration on our experimentation platform. It first shows that our improvement of the `move_pages` system call behaves as expected. When thousands of pages are manipulated at once (several megabytes), the throughput remains near 600 MB/s while the original implementation drops dramatically. However, the base overhead remains high (near 160 μs). According to profiling results, this is caused by intensive locking and page-table manipulations that cannot be optimized easily.

Migrating an entire process with `migrate_pages` (instead of only part of the address space with `move_pages`) has a higher overhead (near 400 μs) due to the whole process virtual address space having to be traversed. However, once the process address space is large, the throughput of `migrate_pages` reaches 780 MB/s. We think that this better throughput is related to a better locality and less locking being involved since the virtual addresses are processed in order while `move_pages` has to support random sets of pages.

Both migration primitives are however much slower than a regular memory copy between NUMA nodes. First, the kernel does not benefit from optimized copy instructions as user-space does (MMX/SSE). We observed that pages are copied during `move_pages` at only 1 GB/s. Secondly, expensive virtual memory management operations are involved during migration, for instance TLB flushing and page-table updates. We take a deeper look at these overheads in the next section.

4.3 *Next-touch* and *Lazy* Migration Microbenchmarks

Figure 5 presents *Next-touch* migration performance. It shows that the user-space strategy basically maps the `move_pages` performance, reaching 600 MB/s for large buffers. Indeed, as shown in Figure 6(a), the additional *Next-touch* operations (signal handler and changing the protection) are almost negligible. The base overhead of `move_pages` appears very high. Even for large buffer

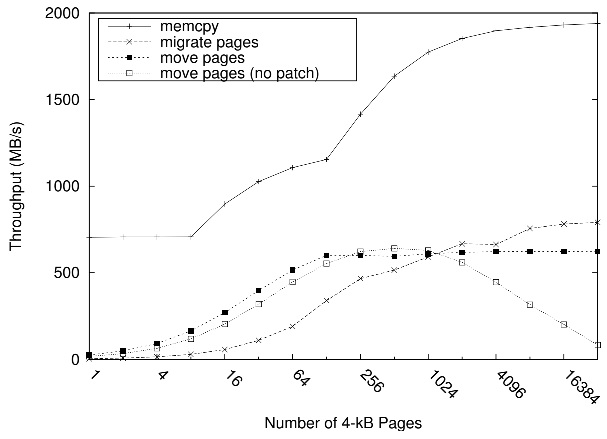


Figure 4: Migration and memory copy throughput comparison between NUMA nodes #0 and #1.

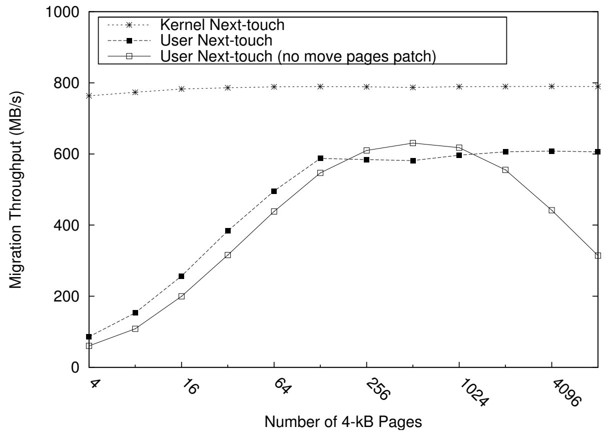
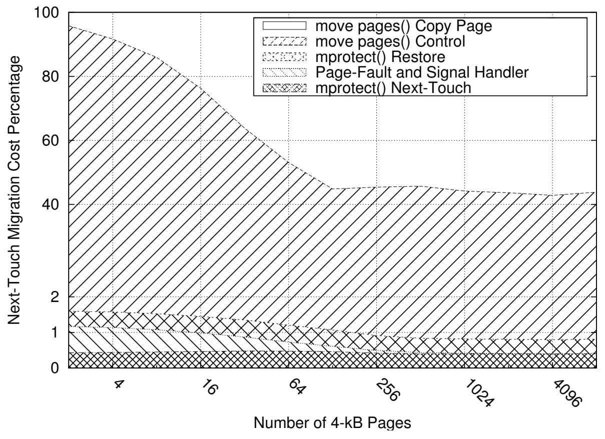
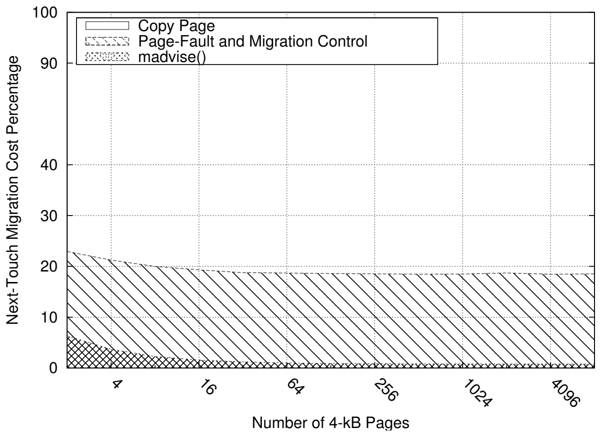


Figure 5: *Next-touch* performance comparison

migration, the *Control* (locking, page-table updates, ...) represents 38 % of the overall migration cost, while one would expect the actual copy to be the only expensive part.



(a) *Next-touch* in user-space



(b) *Next-touch* in the LINUX kernel

Figure 6: *Next-touch* implementation overhead details.

On the other hand, our kernel-based *Next-touch* implementation achieves 800 MB/s even for very small buffers. As shown on Figure 6(b), the page-fault management and migration *Control* cost is only 20 %, causing the overall performance to be much higher. The lower overhead is related to our optimized *Next-touch* implementation in the kernel which was only designed for this specific operation while `move_pages` supports a wide variety of pages, shared mappings, *etc.*

This result justifies the aforementioned idea (see Section 4.3) of implementing a *Lazy Migration*. Indeed, since our kernel-based *Next-touch* is 30 % faster than the synchronous `move_pages` strategy, the lazy migration will as well perform much faster. And obviously, if part of the buffer to migrate is never touched by the thread, the corresponding pages will not be migrated for real, improving the observed migration throughput accordingly.

4.4 Threaded Migration Scalability

Now that we explained the performance of synchronous and lazy migration strategies, we look at their scalability. Indeed, NUMA architectures have multiple cores per node. The thread scheduler is expected to group on the same node threads which are accessing the same data, for instance within an OPENMP parallel section. We thus expect to have concurrent accesses from several threads to a buffer needing to be migrated.

Figure 7 presents the migration throughput on our experimentation platform when some threads are bound to NUMA node #1 and migrating memory from node #0. It first shows that parallelizing the migration (either lazy or synchronous) does not bring any improvement for buffers smaller than 1 MB. We feel that this is related to lock contention in the kernel in both implementations.

The figure also shows that both strategies achieve between 50 and 60 % improvement when migrating large buffers with 4 threads (one per core). Lazy migration seems to scale a bit better since it still improves a bit (+6 %) when adding a fourth thread, achieving up to 1.3 GB/s. This

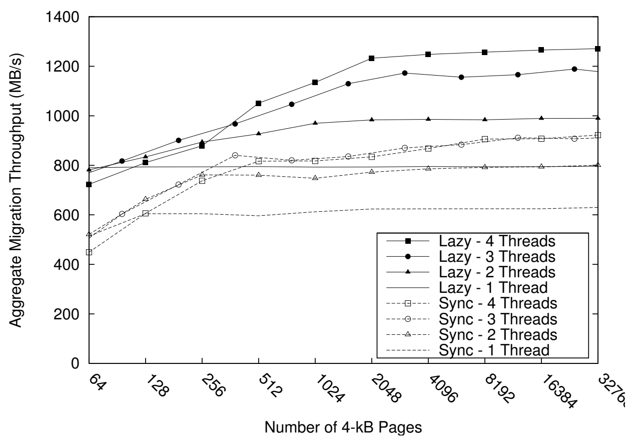


Figure 7: Throughput of a parallel *Lazy* migration (kernel *Next-touch*) and a synchronous migration (`move_pages`) using up to 4 threads on the same NUMA node.

throughput remains much lower than a regular memory copies between NUMA nodes (see Figure 4), but it has to be noted that a page-fault and intensive page-table locking are involved in each underlying page migration. Even if the overall throughput is limited, threaded migration still appears as an interesting solution when multiple threads working on the same dataset are migrated to another node at the same time.

4.5 Threaded LU Matrix Factorization

To study the impact of our *Next-touch* implementation on multithreaded real applications, we now look at a threaded LU matrix factorization. As usual, the implementation splits the matrix into smaller data blocks that are actually manipulated by a BLAS library. During each step, a new "pivot" block is computed on the diagonal. Then, the values for the corresponding column and row are updated as well as the ones for the remaining bottom-right blocks. This is done using *for* loops.

We added OPENMP *parallel for* pragmas in these update loops so that they are computed in parallel by one thread per core. We also inserted a *Next-touch* `madvise` hook at the beginning of each iteration so that the data is redistributed among the NUMA nodes when needed, depending on OPENMP thread access patterns. The data was initially allocated among all NUMA nodes in an interleaved manner since it seems to be the best static allocation policy for this memory-bandwidth intensive problem. Indeed, when using the GCC OPENMP compiler, there is no guarantee about which thread will compute which block on which processor.

Table 1 presents the factorization time. It shows that the *Next-touch* approach benefits the overall performance as soon as large worksets are involved: at least 512 elements per block dimension, within large matrices. It is not clear why some large test-cases get small improvement (a few percent) while some other get about a factor 2. We feel that small improvements are mostly caused by the removal of remote memory accesses thanks to our *Next-touch* policy increasing data locality. Larger improvements may be related to congestion when multiple threads access each others'

Matrix size	Block size	Static	*Next-touch*	Improvement
4k × 4k	64 × 64	2.60 s	4.92 s	-47.1 %
4k × 4k	128 × 128	2.60 s	3.58 s	-27.5 %
4k × 4k	256 × 256	4.66 s	5.07 s	-8.04 %
8k × 8k	128 × 128	19.9 s	24.4 s	-18.2 %
8k × 8k	256 × 256	26.8 s	27.8 s	-3.81 %
8k × 8k	512 × 512	87.5 s	69.2 s	+26.5 %
16k × 16k	256 × 256	166 s	173 s	-4.15 %
16k × 16k	512 × 512	675 s	363 s	+85.8 %
16k × 16k	1024 × 1024	1721 s	1651 s	+4.24 %
32k × 32k	256 × 256	2553 s	1519 s	+68.2 %
32k × 32k	512 × 512	6819 s	2971 s	+129 %

Table 1: Execution time of the LU matrix factorization with 16 OPENMP threads.

NUMA memory across a single HYPERTRANSPORT link.

The 512 block size threshold is actually related to a single page not being shared between 2 threads. Indeed, when using smaller blocks, a single page contains some lines from different consecutive blocks, causing all of them to migrate at once as soon as a single one is accessed. Once each block is large enough to be page-independent, it migrates independently near its single accessing thread thanks to the *Next-touch* policy. We do not present the impact of our user-level *Next-touch* implementation because its overhead makes it unusable for such small granularities.

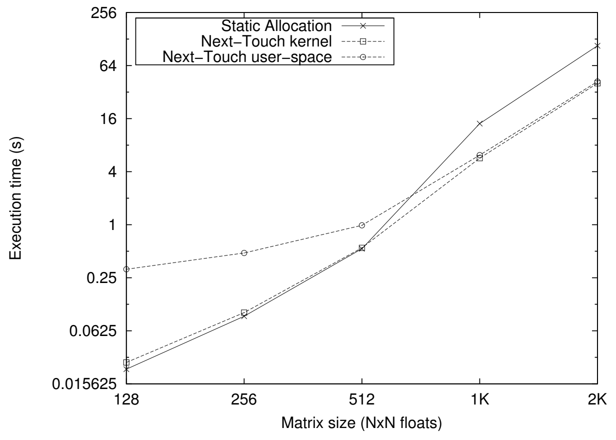


Figure 8: Execution time of a 16 concurrent BLAS3 matrix multiplications within 16 independent threads.

To confirm this result, we ran some independent BLAS3 multiplications within multiple threads to evaluate when it actually becomes important to migrate data before computation. Figure 8 confirms that 512 is the block size where data locality becomes critical since memory migration (even with the user-space implementation) becomes interesting. This result is specific to BLAS3 operations since they are memory bandwidth intensive. We observed that the performance of BLAS1 operations (vector operations) never improves thanks to memory migration, probably because the processor cache hides the remote access latency and thus makes migration almost useless.

5 Related Works

The need to place threads and memory buffers depending on their affinities has been known for a long time [2, 11]. Solaris [8], IRIX [15] and LINUX [7] all acquired some NUMA-aware capabilities within their memory management and thread schedulers. The needs for efficient memory migration emerged with the advent of large NUMA architectures [10, 1, 16].

A *Next-touch* approach has already been implemented in LINUX using with `mprotect` and a segmentation fault signal handler [12]. This implementation is similar to ours (see Section 3.2) and it helped performance on some small threaded test-cases. However, it relies on `move_pages` and thus cannot expect to be efficient for large buffers (megabytes) unless the implementation is fixed as shown in this paper. Given the increasing amount of data that real applications manipulate nowadays, large buffer migration has to be efficient, and this is one of the strong points of our work.

Solaris used to provided a *Next-touch* implementation and it has been successfully used for improving threaded application performance [8]. This kernel implementation was based on the `MADV_ACCESS_LWP` flag for `madvise` which would indicate that the next accessing thread will touch the pages intensively. This implementation may be similar to ours (see Section 3.3) but it has never been described or evaluated in a published paper to the best of our knowledge.

We propose in this work an in-depth description and performance evaluation of our *Next-touch* implementation in LINUX. This is the first such implementation as far as we know and we feel that it is important that a widely used operating system such as LINUX offers such a feature. NUMA architectures are indeed becoming mainstream (through INTEL QUICKPATH and AMD HYPERTRANSPORT technologies), raising the need to be able to exploit them efficiently.

Our kernel-based implementation appears 30 % faster than the user-space model and has a much lower base overhead when migrating small buffers. However, as explained in Section 3.4, the user-space implementation seems to be a good candidate for migrating large and complex memory buffers while keeping the knowledge of migrated page locations.

6 Conclusion and Future Works

The increasing complexity of modern architectures, with many cores and distributed memory banks, raises the need to involve affinities between threads and data in scheduling decisions. Dynamic applications such as adaptive mesh refinement with OPENMP threads have complex and irregular access patterns that cause the ideal thread and data distribution across the machine to evolve during the execution. Migrating data buffers near the threads is known to provide a convenient way to dynamically distribute work and data jointly.

We showed in this paper that although the LINUX kernel is NUMA-aware, its memory migration primitive for multithreaded application (`move_pages`) had a significant performance limitation for large buffers. We were able to restore a constant migration throughput and integrated this work in the upcoming LINUX kernel 2.6.29. We also presented two different implementations of the *Next-touch* policy which brings a convenient way to have data automatically follow its accessing threads. Our kernel-based design shows an improved throughput even for small buffers, and it enables the idea of high-performance *Lazy memory migration* that can be easily parallelized. Applying this *Next-touch* policy to a threaded LU matrix factorization shows a significant performance improvement for large worksets thanks to better data locality being maintained during the whole execution. We are now running similar experiments on larger NUMA machines where data locality is more critical to the overall performance, making the *Next-touch* policy even more interesting.

Our *Next-touch* implementation should still be improved by first supporting shared areas and file mappings instead of only private anonymous pages so that all existing applications can benefit from it. Then, we will study the idea of replicating read-only pages among NUMA nodes so as to achieve local access performance from anywhere. We are also looking at improving the LINUX migration system call interface to reduce the `move_pages` overhead further more. Huge pages are another feature that will have to be studied since they are known to help performance by reducing the TLB pressure, but LINUX does not currently support their migration.

This research is carried out in the context of designing an efficient OPENMP runtime system for hierarchical and NUMA architectures. A tight integration of our *Next-touch* support within the NUMA-aware MARCEL user-level threading library [13] is expected to lay the foundations of a combined model for dynamically scheduling threads and placing memory buffers depending on their affinities. It should enable a clever distribution of work and data within our FORESTGOMP OPENMP runtime which has been designed to run on these architectures [14].

References

[1] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *Proceedings of the International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 2006.

[2] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[3] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Dallas, TX, November 2000.

[4] Ryan E. Grant and Ahmad Afsahi. A Comprehensive Analysis of OpenMP Applications on Dual-Core Intel Xeon SMPs. In *Proceedings of the International Workshop on Multi-*

Threaded Architectures and Applications (MTAAP), held in conjunction with IPDPS'07, page 365, Long Beach, CA, March 2007.

[5] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, March 2003.

[6] Andreas Kleen. A NUMA API for LINUX, April 2005. Novell, Technical Linux Whitepaper.

[7] Christoph Lameter. Local and Remote Memory: Memory in a Linux/NUMA System. In *Linux Symposium*, Ottawa, Canada, July 2006.

[8] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 387–392, Cambridge, MA, November 2005.

[9] Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In *Second International Workshop on OpenMP (IWOMP 2006)*, Reims, France, 2006.

[10] Nathan Robertson and Alistair Rendell. OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In Springer Verlag, editor, *3rd International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 648–656, 2003.

[11] Martin Steckermeier and Frank Bellosa. Using Locality Information in Userlevel Scheduling. Technical Report TR-95-14, University of Erlangen-Nrnberg – Computer Science Department – Operating Systems – IMMD IV, Martensstrae 1, 91058 Erlangen, Germany, December 1995.

[12] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384, New York, NY, USA, 2008. ACM.

[13] Samuel Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Proceedings of the Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, MA, June 2005.

[14] Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In Barbara M. Chapman, Weimin Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, *A Practical Programming Model for the Multi-Core Era, 3rd International Workshop on OpenMP, IWOMP 2007*, volume 4935 of *Lecture Notes in Computer Science*, pages 161–172, Beijing, China, June 2007. Springer.

[15] Steve Whitney, John McCalpin, Nawaf Bitar, John L. Richardson, and Luis Stevens. The SGI Origin Software Environment and Application Performance. In *COMPCON 97*, pages 165–170, San Jose, California, 1997. IEEE.

[16] Rui Yang, Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*, pages 31–36, 2008.