Memory Migration on Next-Touch

Brice Goglin, Nathalie Furmento

▶ **To cite this version:**

Memory Migration on Next-Touch

Brice Goglin
INRIA, LaBRI, University of Bordeaux
`Brice.Goglin@inria.fr`

Nathalie Furmento
CNRS, LaBRI, University of Bordeaux
`nathalie.furmento@labri.fr`

Abstract

NUMA abilities such as explicit migration of memory buffers enable flexible placement of data buffers at runtime near the tasks that actually access them. The `move_pages` system call may be invoked manually but it achieves limited throughput and implies a strong collaboration of the application. Indeed, the location of threads and their memory access patterns must be carefully known so as to decide when migrating the right memory buffer on time.

We present the implementation of a *Next-Touch* memory placement policy so as to enable automatic dynamic migration of pages when they are actually accessed by a task. We introduce a new PTE flag setup by `madvise`, and the corresponding *Copy-on-Touch* codepath in the page-fault handler which allocates the new page near the accessing task. We then look at the performance and overheads of this model and compare it to using the `move_pages` system call.

1 Introduction

The democratization of *Non-Uniform Memory Access* (NUMA), from ITANIUM based platforms, to AMD HYPERTRANSPORT architecture and INTEL's new QUICKPATH interconnect, raises the need to carefully place data buffers near the tasks that access them [1]. Indeed, when local data access is significantly faster than remote access, data locality becomes a critical criterion for scheduling tasks. And the idea of migrating data buffers together with their accessing tasks has to be considered.

In the last decade, LINUX slowly learnt how to manage NUMA requirements. It first gained NUMA-aware allocation facilities in the early 2.6 kernels, either automatically on first touch, or thanks to the `mbind` and `set_mempolicy` system calls. It then acquired memory migration abilities a couple years ago with the addition of `migrate_pages` and `move_pages`. These features enable the manual adaptation of the data distribution across memory nodes to the current task locations. However, dynamic applications with migrating threads may require the corresponding data buffers to migrate automatically as well.

Indeed, threads are a convenient way to program modern highly-parallel hosts. Parallel programming languages such as OPENMP [6] try to ease the mapping of parallel algorithms onto the architecture. The quality of the thread scheduling has a strong impact on the overall application performance because of affinities. This issue now becomes critical due to the variable memory access latencies and bandwidths that NUMA architectures exhibit. We thus propose in this article an implementation of the *Next-Touch* policy which provides applications with a convenient way to have memory buffers dynamically follow their accessing tasks.

The rest of this paper is organized as follows. In Section 2, we provide background information about multithreaded applications requirements and LINUX abilities over NUMA architectures. Section 3 explains our implementation of the *Next-Touch* policy in the LINUX kernel. Experiments shown in Section 4 emphasize the performance advantages of our approach. Before concluding, both our design and implementation are discussed in Section 5.

2 Multithreading on NUMA Architectures

We briefly introduce in this section the requirements in term of memory affinity in multithreaded applications, the available migration strategies in LINUX, and our motivations to implement a *Next-Touch* strategy.

2.1 Dynamic Multithreading Requirements

Memory access requirements of multithreaded applications obviously depend a lot of on thread access patterns, but also on the way the application parallelism is mapped onto threads. Indeed, a application binding one thread per core and allocating its dataset nearby will get satisfying performance. If some threads need to exchange some data, the application has to either migrate the thread near their new target buffers, or migrate these buffers. As long as the application manipulates threads directly (for instance through the pthread interface) and knows their memory access patterns, manipulating memory buffers manually is possible.

The situation becomes far more complex when parallel programming languages are involved. OPENMP appears nowadays as a very easy way to exploit multicore architectures. Indeed, it enables easy thread-based parallelization of sequential applications by adding pragmas in the source code. The democratization of this approach in high-performance computing raises two problem. First, OPENMP does not provide the compiler or runtime-system with any information about memory access patterns. Second, parallel sections may be nested and thus cause dramatic load imbalance in case of irregular applications such as adaptive mesh refinement. Indeed, one of the OPENMP threads may open a new parallel section if it has to much work to do compared to its teammates.

Such nested parallelism causes the operating system or the runtime OPENMP system to load-balance newly created threads across all cores in the machine. Each migrated thread may thus have to migrate its own dataset so as to reduce distant memory accesses and avoid congestion in the memory interconnect [7]. However, having a precise knowledge of which buffer to migrate is often hard. And the absence of memory-affinity-related pragmas in parallel languages does not help. Moreover predicting or detecting each thread migration is also difficult (unless the scheduler is embedded in the application). Migrating memory buffers on time when threads are migrated is thus a challenge.

2.2 NUMA Management APIs in LINUX

NUMA-awareness was added to LINUX during the development of 2.6 kernel. Most features are made available to user-space applications thanks for instance to the libnuma interface [3]. It provides applications with memory placement primitives such as `set_mempolicy` and `mbind` that insure buffers are allocated on the right NUMA node(s). Such static policies are indeed useful when the application knows where the accessing threads run: if a thread is bound to a NUMA node, its dataset may be bound there as well.

When the thread location is unknown, a commonly-used approach is *First-Touch*. It relies on the operating system laziness when it comes to actually allocating physical pages. Many OPENMP applications thus add an initialization round where each thread touches all the pages of its own dataset so that they are actually allocated on its local NUMA node. However, as soon as some threads have to migrate (for instance because of load-balancing in irregular parallelism), the *First-Touch* approach cannot help anymore since pages have already been touched and allocated during the initialization round.

Memory migration is thus required as a way to have datasets migrate with their accessing tasks. LINUX earned migration primitives such as `move_pages` and `migrate_pages` a couple of years ago [4]. The latter migrates an entire process address space onto some NUMA node(s). It was designed together with *Cpusets* as a way for administrators to partition machines. And therefore it is not relevant for multithreaded applications where only part of the address space may migrate.

The `move_pages` system call is the only way to explicitly migrate a buffer within an application[1]. However, it is a synchronous function that must be invoked by user-space, for instance when a thread is migrated or when it starts working on a new dataset. It requires a strong cooperation between the application and the runtime system when some threads decide to migrate (assuming they even properly know their thread access patterns).

2.3 The Need for Dynamic Migration Primitives

The democratization of dynamic parallelism such as adaptive mesh refinement, especially thanks to nested parallel sections in OPENMP, goes beyond what LINUX memory management interface targets nowadays. Both

---

[1]The `mbind` system call with the `MPOL_MF_MOVE` flag is actually somehow equivalent to `move_pages`.

the operating system and user-space applications or run-time systems may have to migrate threads for load-balancing reasons. It thus becomes important to have an easy way to migrate the corresponding buffers at the same time. However, the *First-Touch* approach is not applicable except during the initialization phase. And explicit migration requires the precise knowledge of when each thread is migrated and of their memory access patterns. It is difficult to achieve because parallel languages such as OPENMP do not provide the required annotations, and also because user-space has no way to specify task-memory affinity to the LINUX kernel.

Actually, it is not clear that such affinities belong in the kernel. People have been working on user-level thread scheduling as a way to get highly-configurable scheduling algorithms as well as reduced management costs. This model enables the addition of affinities between threads and/or data and the design of custom schedulers dedicated to some classes of applications [9]. One may think that migrating data buffers may thus have to be managed in user-space as well. However, it brings back the issue of parallel languages not providing the required annotations to explicit task/data affinities. And still, requiring a precise knowledge of these affinities is a very hard work for the developer, for instance because some buffers may be accessed by multiple threads with different read/write patterns. From the user-level developer point of view, it is much more work than just trying to load-balance threads across all cores of the machine.

We thus envision the addition in the LINUX kernel of a new mechanism for managing the requirements of such multithreaded applications. The idea behind the *Next-Touch* policy is to have data buffers automatically migrate near their accessing tasks when touched. As many other LINUX features such as page allocation or *Copy-on-Write*, this policy relies on the operating system laziness since migration only occurs when it is actually needed. The application thus just has to mark buffers as *Migrate-on-Next-Touch* when it knows that thread migration is expected in the near future: for instance when the application begins a new phase with different memory access patterns, or when it enters a new OPENMP parallel section. Such events are indeed very common in multithreaded applications, and may be easily located by their developer.

As a result, as soon as a thread touches a marked buffer that is not allocated on its local memory node, it is automatically migrated. The scheduler may then freely migrate threads to accommodate load-balancing to dynamic/nested parallelism without having to care about data affinities. This model dramatically reduces the work for the developer since there is no need to know where buffers are allocated, when each thread is actually migrated, and which buffers are manipulated by each thread.

Some proprietary operating systems such as SOLARIS actually implement such a policy and it has been proven to significantly help high-performance computing [5]. We detail in the next section our design and implementation of a similar solution in the LINUX kernel.

3 Implementation of the Next-Touch Policy

We now explain why a *Next-Touch* policy requires kernel support and how we implemented it.

3.1 Why a User-Space Implementation is a Bad Idea

Implementing a *Next-Touch* policy is possible in user-space thanks to user-directed memory protection and segmentation fault management. This strategy has been used to implement distributed shared memory across machines and may also be used to detect next touches. Indeed, the mprotect system call may be used to prevent application accesses to any memory zone and cause segmentation faults that a custom signal handler will catch. This handler then just needs to migrate the corresponding buffer and set the default protection back. This strategy is however hard to implement safely in multi-threaded environment and obviously exhibits an important overhead. For instance, it has been shown to increase the performance of a Jacobi Solver on NUMA machine much less than a native *Next-Touch* approach under SOLARIS [8].

One unexpected drawback of this approach is actually the limited performance of the move_pages system call. Indeed, aside from having to call mprotect twice (hence flushing the TLBs) and handle the segmentation fault signal, migrating pages has a very large initialization overhead. One could think that this *Next-Touch* approach could then be used only for large buffers, but the asymptotic throughput of move_pages is actually low as well[2].

---

[2]Even after move_pages was fixed in 2.6.29 to have a linear complexity as shown in http://lkml.org/lkml/2008/10/11/117.

Still, one advantage of a user-space implementation is that migrating at the user level lets the user application manage buffer granularity. The signal handler may thus migrate a single page or a very large buffer depending on the application datasets and wishes. However, again, it requires the application to know the memory access patterns of its threads. Also not that many applications actually rely on very large granularity. And it is not clear that migrating a large buffer at once will always be faster than migrating multiple pages independently in the kernel.

One way to observe the relative slowness of a user-space implementation is to compare it with a *Copy-on-Write* across different NUMA nodes. The pseudocode below is indeed able to copy-on-write pages from NUMA node #0 to #1 at more than 800 MB/s on a quad-socket Barcelona machine. However, as of 2.6.29, `move_pages` cannot migrate the same pages at more than 600 MB/s. Actually, *Next-Touch* may be seen as *Copy-and-Free-on-Read-or-Write*. We therefore feel that LINUX should be able to provide a *Next-Touch* policy with a similar implementation and performance as *Copy-on-Write*.

```
buffer = mmap(NULL, LENGTH, ...,
              MAP_PRIVATE, ...);
mbind(buffer, LENGTH, <node #0>);
/* prefault on node #0 */
memset(buffer, 0, LENGTH);
if (!fork()) {
  mbind(buffer, LENGTH, <node #1>);
  sched_set_affinity(<node #1>);
  /* copy-on-write on node #1 */
  for(i=0; i<LENGTH; i+= PAGE_SIZE)
    buffer[i] = 0;
}
```

3.2   Page-Faulting on Next-Touch

LINUX implements *Copy-on-write* by removing the write-access bit from the PTEs (*Page Table Entry*) so that any write generates a page-fault. The page-fault handler verifies in the VMA flags (*Virtual Memory Area*) that this write-access is actually valid from the application point of view. If so, it copies the page instead of killing the process with a segmentation fault. The strategy thus relies on the difference between the high-level VMA flags (defined/visible at the application
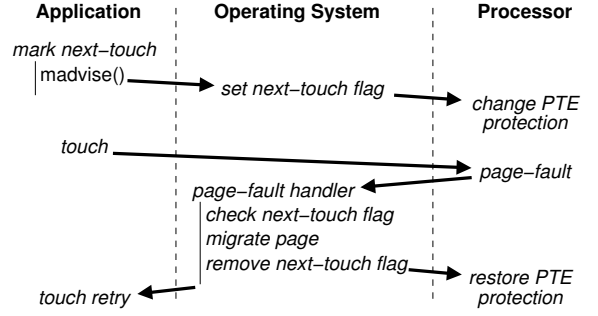

Figure 1: Description of the implementation of the *Next-touch* policy using `madvise` and a dedicated flag in the page-table entry (PTE).

level) and the low-level PTE flags (defined by the kernel and used by the processor).

We implemented the *Next-Touch* policy in a similar manner, i.e. by removing read and write-access permissions from the PTEs so that a page-fault occurs on next touch, as depicted in Figure 1. However, the implementation is harder than *Copy-on-Write* since there cannot be any high-level VMA flag for *Next-Touch*. Indeed, the migrate-on-next-touch status is only temporary. It must be cleared when the touch occurs. And a VMA might have been only partially touched, causing only some pages to have been migrated yet while some other are still marked.

For this reason, we implemented the *Next-Touch* policy by only modifying PTEs: When the *Next-Touch* flag is added, read and write access is disabled. When the page-fault occurs, the flag is removed and regular permissions are re-enabled.

The application interface to enable the *Next-Touch* policy relies on a new `madvise` behavior which is implemented in the kernel by `madvise_migratenexttouch()` and in the end by `set_migratenexttouch_pte_range()`. The whole kernel implementation is quickly summarized in Figure 2.

3.3   Migrating in the Page-Fault Handler

Once read/write access has been disabled for some pages, the page-fault handler has to be able to actually detect whether a fault was caused by the *Next-Touch* policy. Comparing VMA and PTE flags cannot help here, but our new *migrate-on-next-touch* PTE flag was

```
/***** in mm/madvise.c *****/
static void
set_migratenexttouch_pte_range(mm, vma, pmd, addr, end)
{
  ...
  if (pte_present(oldpte)) {
    pte_t ptent = ptep_modify_prot_start(mm, addr, pte);
    ptent = pte_modify(ptent, vm_get_page_prot(0)); /* no access rights granted */
    ptent = pte_mkmigratenexttouch(ptent);
    ptep_modify_prot_commit(mm, addr, pte, ptent);
  }
  ...
}
static long
madvise_vma(vma, prev, start, end, behavior)
{
  ...
  case MADV_MIGRATENEXTTOUCH:
    error = madvise_migratenexttouch(vma, prev, start, end);
    break;
  ...
}


/***** in mm/memory.c *****/
static int do_migrateontouch_page(mm, vma, address, page_table, pmd, ptl, orig_pte)
{
  ...
  /* if page already local, no need to migrate */
  if (page_to_nid(old_page) == numa_node_id())
    goto reuse;
  ...
  /* similar to do_wp_page() and clear the migrate-on-next-touch PTE flag */
  ...
}
static inline int handle_pte_fault(mm, vma, address, pte, pmd, int write_access)
{
  /* handle !pte_present */
  ...
  /* handle migrate-on-next-touch */
  if (pte_migratenexttouch(entry))
    return do_migrateontouch_page(mm, vma, address, pte, pmd, ptl, entry);
  /* handle copy-on-write */
  ...
}
```

Figure 2: Summary of the implementation of the *Next-Touch* policy through a new `madvise` behavior, a new PTE flag, and the corresponding page-fault handling code which mimics a copy-on-write.

designed specifically for this. The detection must occur after having taken care of non-present pages (since only pages that are present in physical memory may need migration). Migration on next touch should however be handled before looking at *Copy-on-Write* so that the latter does not have to check our new PTE flag which disables write-access in a similar way. The way `handle_pte_fault()` manages these cases and the invocation of our new `do_migrateontouch_page()` function is summarized in Figure 2.

Migrating the page in the handler is the key to performance here. We chose to target the performance critical case, which is private anonymous mappings. Fortunately, the `madvise` system call is only an advise given by the application to the kernel. It thus does not definitely enforces that any kind of memory mapping should actually be migrated on next touch. We discuss this design choice further in Section 5.

Migrating private anonymous mapping is actually very simple since there is no need to handle shared pages properly. The final code is therefore very similar to the *Copy-on-Write* code (in `do_wp_page()`). The old page is copied into a new page that was allocated on the local node. Then the old page is released.

4   Performance Evaluation

We present in this section a performance evaluation of our *Next-Touch* policy implementation in the LINUX kernel. The experimentation platform is a quad-socket quad-core OPTERON Barcelona (2347HE, 1.9 GHz) machine. It runs 2.6.27 with the `move_pages` performance-fix patches and our *Next-Touch* patches.

4.1   Migration Throughput

Figure 3 presents a comparison of the throughput of various data migration strategies. It first shows that the existing migration system calls have a very large initial overhead and a limited asymptotic throughput (800 MB/s for `migrate_pages` and 600 MB/s for `move_pages`).

Our *Next-Touch* implementation shows a very small base initialization overhead. Its asymptotic throughput (800 MB/s) is reached with very small buffers. It shows
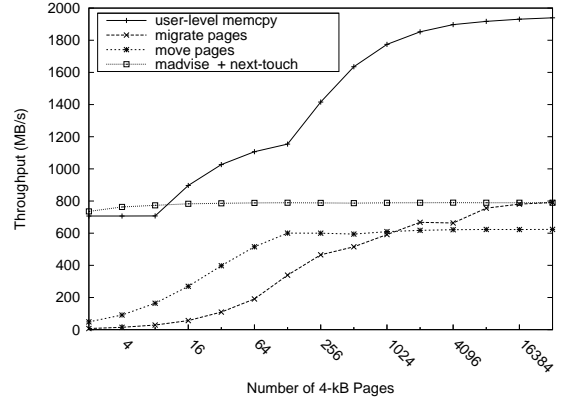

Figure 3: Migration and memory copy throughput comparison between NUMA nodes #0 and #1.

that only our *Next-Touch* implementation is able to migrate small buffers efficiently.

Overall, migrating pages appears to be much slower than copying data manually with `memcpy` both for small and large buffers. It explains why some people even consider copying data between different buffers and modifying the application pointers instead of actually using migration. We measured that the kernel copies data during migration at a 1 GB/s throughput. It is much slower than user-space copies due to less MMX/SSE-like optimizations. But the actual slowness of migration is also related to the management overheads that we detail in the next section.

4.2   Understanding the Overheads

We measured that setting a memory range as migrate-on-next-touch with madvise only costs about 600 ns plus 40 ns per page. Then the actual migration on next-touch costs about $5.2\,\mu$s per page. Since the actual memory copy throughput in the kernel is about 1 GB/s, it shows that only $1\,\mu$s ($20\,\%$) is needed to manage each page (handle the page-fault and then do the actual update of kernel structures such as the PTE).

On the other hand, `move_pages` shows a $72\,\mu$s base overhead and then requires $2.5\,\mu$s to handle each page migration (before copying at 1 GB/s rate). We found out that it is possible to reduce it down to $1.4\,\mu$s/page[3] but it remains much higher than the $1\,\mu$s management

---

[3]`move_pages` performance might be further improved in upcoming kernels, as explained at `http://lkml.org/lkml/2009/4/15/75`.

cost in the *Next-Touch* fault handler. The reason is that `move_pages` works on pages instead of PTEs. It is able to migrate many different types of pages, including shared mappings. Therefore, it has to isolate pages from concurrent process access during migration. Once the data has been copied, it also has to update the PTEs of all involved processes (not only the migrating one).

The initialization code of `move_pages` therefore drains pages out of the per-CPU *pagevec* lists so that they may actually be isolated from migration once they are in the main LRU list. This invocation of `lru_add_drain_all()` schedules a deferred work on each processor. It is actually responsible for the whole $72\,\mu s$ base overhead of `move_pages`. However we do not understand why this cost appears to be linear with the number of processors in the machine (about $6\,\mu s$ plus $4\,\mu s$ per processor on our machine) while we expected this parallel operation to scale with satisfying performance under normal load. Further optimization may be needed here.

This result raises the question of whether all the `move_pages` complexity is actually required in most cases. Indeed, migrating shared mapping pages may not be interesting for many applications. It may then be interesting to look at a new migration primitive that would ignore complex cases such as shared mappings. We will discuss this idea further in the next sections.

5 Discussion

We discuss in this section several questions that have to be raised when designing and implementing a *Next-Touch* policy.

5.1 User Interface

Our implementation expects applications to mark contiguous areas as migrate-on-next-touch using the `madvise` system call and the new `MADV_MIGRATENEXTTOUCH` behavior. Some other interfaces such as `set_mempolicy` or `mprotect` could be considered but they work on VMAs and rely on setting static flags in the kernel structures. However, the *Next-Touch* policy has no reason to be defined on a per-VMA basis, and it must only be set temporarily since the status disappears after the actual touch. Moreover, the `madvise` interface only gives

Hints to the kernel. It is thus possible to ignore it under some special circumstances such as non-migratable pages. For the record, SOLARIS uses `madvise` with the behavior `MADV_ACCESS_LWP` meaning that the next lightweight process will access the memory range heavily. The implications are therefore even less strict than ours since the SOLARIS kernel could even try to optimize some internal structures that are not directly related to memory migration.

Our implementation is page-based while a user-space implementation may migrate large buffers at once. Adding granularity information to a kernel implementation would require a new interface. VMAs cannot be used to do so since they may be merged/split by the kernel during many system calls such as `mprotect`. Actually, applications may enforce the migration of a whole segment on our page-based implementation by touching all pages. The overhead of this strategy is linear. And migrating pages enables more laziness and thus may reduce the actual copy overhead in some cases. Indeed, as the `madvise` overhead is small, applications may mark very large buffers as *Next-Touch* even if there is a chance that some pages are never actually touched, and thus never migrated for real.

Another question that needs to be raised is whether read and write accesses must be distinguished. Our implementation migrates pages in both cases, but some applications may actually want to migrate only in case of a write-access. Indeed, for instance, a single-producer-many-consumers model may need a privileged write access. Implementing a `MADV_MIGRATEONNEXTWRITE` might thus be interesting as well. Its implementation would be even closer to the existing *Copy-on-Write* code.

Finally, it is not clear whether applications may sometimes need to query the migrate-on-next-touch status of a segment, or even clear it. No such usage looks obvious in the context of high-performance computing. Clearing would be easy to implement using another `madvise` behavior. However, retrieving the status of pages looks harder. The `move_pages` system call is able to retrieve the location of a set of pages. We could imagine marking its return values with a special bit if a new flag `MPOL_MF_GETNEXTTOUCH` was given.

5.2 Implementation Details

The main drawback of our experimental implementation is that it only works with private anonymous mappings. Migration of file-backed mappings is not supported so far because it does not seems to be widely used in high-performance computing. However, we do not see any reason to not implement it. We do not support the migration of shared mappings either, because it is harder to implement since it requires to update all address spaces pointing to the migrated pages. It may actually be one of the reason why `move_pages` is slower than our approach.

Our implementation enforces the copy of the touched page into a new one even in case of a read touch. If a private page is still used by 2 processes because none of them modified it yet, the *Next-Touch* always causes its duplication as *Copy-on-Write* does. Both processes then keep their own private copy of the original page. This strategy does not break the semantics of memory mappings but it may slightly increase the memory consumption. Indeed, pages that are marked as migrate-on-next-touch may actually be duplicated earlier than with a regular *Copy-on-write* model. Our feeling is that this implementation has the advantage of not migrating pages that are used by other processes. It is not clear to us that some process should have a privileged access to a shared page regardless of the other processes using it. Our early duplication of pages on next-touch causes each process to keep their own pages locally as they wish. Moreover, since setting the *Next-Touch* policy is only a hint, the idea of ignoring it for shared pages has to be considered anyway. This idea goes with our proposal for a new migration primitive as explained in Section 4.2.

Another point that might need to be discussed is whether the *Next-Touch* flag should be stored in PTEs or in pages. One advantage of switching to page flags would be that they are more room for additional flags than in PTEs. However using page flags would also imply that shared pages are migrated as `move_pages` does. However, as explained above, it is not clear that it is the desired behavior. The idea of using PTE flags has the advantage of keeping the page-fault handler very similar to the *Copy-on-Write* handler (`do_wp_page()`). Merging our implementation into a more generic *Copy-on-Write* handler might even be possible.

Our implementation as well as the *Copy-on-Write* handler uses `alloc_page_vma()` to allocate the new page. The default behavior is thus to allocate a local page, except if the application sets a NUMA binding policy on the virtual region. It may result in funny situations where *Next-Touch* pages get migrated to another NUMA node than the one touching them. It is not clear whether this should be handled automatically by the kernel, since a valid application should have canceled the NUMA binding policy before enabling the *Next-Touch* policy. However, our `do_migrateontouch_page()` only checks whether the old page is already local. It might have to be changed into checking whether the old page matches the memory allocation policy.

The last question that may have to be raised is when the *Next-Touch* status of a page should be cleared. It looks obvious that calling `move_pages` should cancel pending migration on next touch since the application is trying to enforce the actual location of pages synchronously. However, it is less obvious for cases where the allocation policy is modified with `set_mempolicy` or `mbind`. Meanwhile, PTE modifications (for instance in case of `mprotect`) probably needs to maintain the *Next-Touch* flag. It raises again the question of adding a `madvise` behavior for cancelling a pending migration on next touch.

6 Conclusions

As NUMA architectures are becoming mainstream thanks to the spreading of HYPERTRANSPORT and QUICKPATH technologies, affinities between tasks and data becomes a critical criteria for scheduling decisions. Dynamic applications such as adaptive mesh refinement with OPENMP threads have complex and irregular access patterns. The ideal thread and data distribution across the machine may thus evolve during the execution. Migration of data buffers therefore becomes a convenient way to dynamically maintain locality.

The LINUX kernel has gained NUMA abilities during the 2.6 development but we explained that the existing primitives are mostly designed for static application behaviors. Dynamic parallelism requires more complex capabilities so as to take care of affinities between threads and data buffers dynamically and automatically. Requiring the application to pass the whole knowledge of memory access patterns down to the thread scheduler would lead to way too much development overhead.

The *Next-Touch* policy is a convenient way to implement the automatic migration of data buffers near their

accessing tasks. We feel that it may easily be applied to multithreaded applications by locating the application phases, where the thread-data affinities may change, or when a new OPENMP parallel section begins. We presented an implementation of this policy in LINUX and showed that it provides interesting performance improvements thanks to minimal initialization overhead, page-based granularity, and satisfying asymptotic throughput. Applying this strategy to high-performance computing application is under work and shows interesting result such as 100 % speedup on a OPENMP-threaded LU factorization.

Several key points have been discussed regarding the actual user interface that should be offered to applications and its internal implementation in the kernel. We also detailed the overheads of the existing `move_pages` system call and of our implementation. The former is still under optimization, but it still exhibits a large initialization cost due to its ability to handle complex cases. We thus raised the idea of adding a new migration primitive with improved performance thanks to relaxed guaranties and a more simple interface. Other ideas could be studied, such as offloading page copies during migration on DMA engine hardware [2]. We hope that these results will attract developers into working in this area.

References

[1] 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.

[2] Andrew Grover and Christopher Leech. Accelerating Network Receive Processing (Intel I/O Acceleration Technology). In *Proceedings of the Linux Symposium (OLS2005)*, pages 281–288, Ottawa, Canada, July 2005.

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

[4] Christoph Lameter. Local and Remote Memory: Memory in a Linux/NUMA System. In *Linux Symposium (OLS2006)*, Ottawa, Canada, July 2006.

[5] 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.

[6] OpenMP: Simple, Portable, Scalable SMP Programming. `http://openmp.org`.

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

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

[9] 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.