2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

OS-based NUMA Optimization: Tackling the Case of Truly Multi-Thread Applications with Non-Partitioned Virtual Page Accesses

Ilaria Di Gennaro, Alessandro Pellegrini, Francesco Quaglia
DIAG–Sapienza Università di Roma, Italy

***Abstract*—A common approach to improve memory access in NUMA machines exploits operating system (OS) page protection mechanisms to induce faults to determine which pages are accessed by what thread, so as to move the thread and its working-set of pages to the same NUMA node. However, existing proposals do not fully fit the requirements of truly multi-thread applications with non-partitioned accesses to virtual pages. In fact, these proposals exploit (induced) faults on a same page-table for all the threads of a same process to determine the access pattern. Hence, the fault by one thread (and the consequent re-opening of the access to the corresponding page) would mask those by other threads on the same page. This may lead to inaccuracy in the estimation of the working-set of individual threads. We overcome this drawback by presenting a lightweight operating system support for Linux, referred to as multi-view address space, explicitly targeting accuracy of per-thread working-set estimation in truly multi-thread applications with non-partitioned accesses, and an associated thread/data migration policy. Our solution is fully transparent to user-space code. It is embedded in a Linux/x86_64 module that installs any required modification to the original kernel image by solely relying on dynamic patching. A motivated case study in the context of HPC is also presented for an assessment of our proposal.**

I. Introduction

Nowadays, computing systems are continuously increasing their number of processors and the overall size of RAM storage, which makes increasingly difficult and economically non-convenient to support uniform memory access latency. Non Uniform Memory Access (NUMA) represents therefore the reference architectural organization.

NUMA platforms are configured to have different RAM banks *close* to specific CPU-cores, hence providing low latency and high throughput of memory access, while other banks are *far* from these same CPU-cores, thus inducing higher access latency and reduced throughput. This architectural paradigm has a clear reflection on the efficiency according to which both application and system-level software is executed, given that accessing data from a far NUMA node does not favor performance and leads to reduced energy efficiency (because of the need for more clock cycles).

The baseline mechanism for optimizing memory access in NUMA platforms consists in (dynamically) migrating a thread and its working-set of virtual pages to a same NUMA node. However, one major barrier to the adoption of this technique is represented by the lack of a-priori guarantees (or assumptions) on how slices of logical memory are accessed at runtime, as typical of non-deterministic code/execution. This is an aspect that cannot be fully tackled or resolved when solely relying on off-line code analysis techniques [2].

A few solutions targeting the *runtime determination* of the access pattern can be roughly classified in *user-level* and *system-level* ones. User-level proposals are based on nesting (e.g. via instrumentation schemes or third party library profiling) monitoring code within the application software, which is used to track the memory (page) references issued by a thread. System-level solutions rely on operating-system support for memory management, which is used to induce and trap segmentation faults so as to capture page accesses via signaling mechanisms. This is achieved by temporarily denying the access to valid memory, and then re-granting the possibility to access as soon as a segmentation fault is trapped and the accessed virtual page is identified as one belonging to the current working-set of the faulting thread.

Generally speaking, user-level approaches look more intrusive for a set of reasons. They may induce overhead (e.g. due to the execution of user-space monitoring routines statically nested within the application code) along execution phases where the current working-set of a thread has already been determined. Also, they may lead to sub-optimal runtime behavior (even for highly optimized application code) due to secondary effects such as cache-efficiency reduction [11].

On the other hand, current system-level approaches (see [8], [27]) do not fully cope with *truly multi-thread* applications, namely those based on threads living in the same process (i.e. within a same address space), where threads exhibit *non-partitioned virtual page accesses*. This is a noticeable lack, given that truly multi-threading is nowadays recognized as a core technology in a lot of contexts, ranging from multi-thread servers' implementations (e.g. [20]) to HPC platforms to be run on top of multi-core shared memory systems [16], [28]. Also, shared data access across multiple threads along specific wall-clock time windows has been shown to provide a base for the design of (distributed) protocols exalting the benefits from locality [14].

The point is that existing system-level approaches rely on conventional address space management schemes, where a single page-table is used as the core memory management data structure for all the threads living within a same process. As a consequence, once the access to a virtual page is temporarily closed within the page-table, a single fault by whichever of the concurrent threads leads to re-granting the access to that same virtual page to all the other threads. In fact, the unique instance of the page-table gets modified upon fault handling by implicitly making the

978-1-5090-2453-7/16 $31.00 © 2016 IEEE
DOI 10.1109/CCGrid.2016.91

page re-accessible to all those threads. Consequently, any thread other than the faulting one can access the same page without being identified as one actually using that page (hence having it within its current locality of references).

Some proposals (e.g., [27]) suggest to tackle this problem by increasing the frequency according to which memory access is denied. However, this may lead to the adverse situation where a same thread faults multiple times on a same page, which has already been identified as belonging to its current working-set. Also, the approach of frequently denying memory access in the hope to track the accesses to the same page by the different threads running within the truly multi-thread application can only alleviate the problem, since misses of accesses' tracing can iterate over time, just depending on application external factors such as OS thread scheduling.

In this article we overcome this problem by proposing an innovative memory management facility, specifically tailored to Linux/x86_64 systems, based on the concept of *multi-view address space* (MVAS). With our proposal, any individual thread is (temporarily) associated with a thread-specific memory view that is set up via a so-called *sibling page-table*, which stands aside of the one originally instantiated for supporting the logical-to-physical memory mapping of the overall (truly multi-thread) process. Each thread-specific memory view has the capability to detect, via minor faults, the access to virtual pages by the corresponding thread. Minor faults are fully resolved via an ad-hoc version of the operating system page-fault handler, which we provide in our architecture, so that the memory access tracking scheme does not rely on the usage—and does not impose the overhead—of the full chain of supports for the `SIGSEGV` signal. The ad-hoc version of the page-fault handler we provide is in charge of maintaining consistency between the original page-table and the sibling one(s) via an elegant iterative fault-resolution scheme (limited to 2 steps), which does not require to perform system-wide synchronized updates of multiple sibling page-tables. Further, such a scheme allows treating any kind of fault that may be experienced in a conventional (single-view) execution of the multi-thread application, such as native minor faults on empty-zero memory.

Our MVAS facility can be switched on/off dynamically while the target process runs. Hence, per-thread working-set variations can be detected at runtime to apply dynamic thread/page migration schemes optimizing memory access on NUMA platforms. The design and implementation of one of these schemes are also provided in this article.

Our proposal is fully based on a Linux module[1], which avoids the need for any kernel recompilation/reinstallation in order to use our architecture. Further, the complexity of using our system is minimal, since it requires managing a few simple shell commands that are provided within the software package.

[1]Source code available at `https://github.com/HPDCS/MVAS`

Experimental data are provided for an assessment of our solution, which have been collected by running a motivated case study from HPC, namely an open source parallel simulation platform [21], on top of a 32-core machine equipped with 64 GB of RAM, partitioned into 8 NUMA nodes.

The remainder of this article is structured as follows. In Section II we discuss related work. The multi-view address space facility and the associated memory access tracing support are presented in Section III. How to exploit them for actual migration of threads/pages is discussed in Section IV. In Section V we present the experimental evaluation of our system.

II. Related Work

NUMA allocators. NUMA-aware allocators constitute a well known technology for memory access optimization in modern computing systems. This is the case of kernel-level allocators, used for ultimately serving memory requests by either the kernel or the applications. As an example, so-called *mem-policies* are adopted by Linux in order to allocate pages according to NUMA oriented rules [13]. The default rule is the so-called first-touch one, according to which a logical page is materialized in a physical frame on the NUMA node where the CPU-core performing the first access to the page resides. Our approach has a different target, given that we aim at (dynamically) determining which threads are touching—possibly in shared mode—some logical page (likely already materialized in RAM) over time in order to, e.g., migrate it towards different NUMA nodes while the application's execution is in progress.

Beyond kernel-level software, NUMA-aware allocators have also been considered in HPC, e.g., in parallel scientific applications based on explicit data partitioning [16], [22]. In such a scenario, the application-level allocator resorts to NUMA-specific system calls to support the (dynamic) binding of logical pages to the NUMA nodes where the threads touching a specific partition of the overall application state are running (or are dynamically migrated to). Our approach is different from these proposals in that we target truly multi-thread applications not necessarily based on data partitioning across different threads. Rather, we cope with scenarios where a single page can be hot (being included in the working-set) of more than one thread, which ultimately allows for clustering these threads and their hot page(s) on the same NUMA node.

Library and instrumentation based approaches. A few solutions are based on determining the data-sharing pattern across different threads and/or processes by relying on runtime profiling of actual memory operations at the level of the employed data-exchange libraries. In this category we can find approaches suited for parallel applications relying on MPI/OpenMP libraries (see, e.g., [4], [17], [26]). Variants of this category of proposals are based on either a-priori knowledge of the source/destination threads for specific data-exchange operations [10] or on a knowledge base of the communication pattern built by tracing previous executions

of a given application software [2], [9]. In both cases the data-exchange library is able to select well-suited buffers for access latency reduction on NUMA machines. Our approach is completely different since we do not operate at library (user-space) level, rather at kernel-level. Consequently, it looks more general since it can be employed in the context of multi-thread applications relying on generic (e.g. NUMA unaware) libraries for exchanging/sharing data across concurrent treads. On the other hand, our focus is exactly on truly multi-thread applications, not on multi-process ones.

Alternative approaches are based on instrumenting application level code so as to capture memory access dynamics independently of the reliance on, e.g., data exchange libraries. Along this path we can find solutions that are aimed at making instrumentation lightweight although attempting to maximize its capability in tracking memory access patters [12], [18], [24]. Our proposal is fully orthogonal to all these techniques given that it operates at kernel level. Further, it can be switched on/off periodically thus allowing for controlling the relative overhead for memory access pattern determination, which is not allowed (or at least is not allowed at zero cost) by pure user-space schemes [11]. The only exception is when relying on multi-coding within the same executable (with static coexistence of both instrumented and native application modules), which has been shown to be feasible limited to specific application domains [23].

Hardware based approaches. A few proposals try to capture data affinity towards a CPU-core (i.e. a thread running on that core) by using specific hardware-level facilities. These include statistics exposed by memory controllers [1], [3], software-managed TLBs [19], [25] (programmed ad-hoc to gain information on the memory access pattern), or even instruction-based sampling [7]. These solutions target the determination of what thread is interested in working on a specific page, while we aim at determining threads vs data affinity in contexts where a specific page can be hot for multiple threads, so as to ultimately migrate these threads and the page on the same NUMA node.

Operating system based approaches. Recent proposals like KMAF [8] and AutoNUMA [27] are both based on inducing page faults that are then traced to determine the pages accessed by threads. When a page fault occurs by a specific thread, the access is then re-granted (by updating the corresponding page-table entry), so that other threads living in the same address space are also allowed to access the same page without being traced. As we pointed out, this makes these approaches not fully suited for truly multi-thread applications with non-partitioned thread accesses to virtual pages. Rather, they are mostly suited for parallel applications based on either (A) multiple processes living in different page-tables (so that page faults by different threads do not mask those by other threads) or (B) multi-thread applications where the access pattern is such that specific sets of logical pages are exclusively accessed by different threads (which avoids false negatives when a page opened for the access by a faulting thread is then automatically accessible by any other thread of the application). Differently from these proposals, our approach provides an innovative system-level support for page access tracing in truly multi-thread applications which avoids false negatives, given that each thread is allowed to run in a specific (sibling) memory view. As a consequence, a change of the access rule on a view upon the occurrence of a faulting access does not impact the access rules on other views, hence the possibility to trace faulting accesses to the same page by other threads.

The attempt to exploit non-conventional address space management policies (as we do) in order to capture the access pattern and to optimize the runtime in NUMA platforms has been devised in [5]. Here the authors provide the concept of a locality-aware page-table, able to track per-thread accesses to pages by directly registering the identifier of the accessing thread within the page-table upon TLB misses. Clearly, this approach requires hardware level facilities that are not currently available in off-the-shelf processors. In fact it has been evaluated limited to simulated/hardware-emulated scenarios. Rather, our proposal does not require any special hardware support, and has been integrated within Linux via a real implementation available as free software.

III. Multi-View Address Space—MVAS

A. Basic Architectural Organization

Our software architecture puts in place a kernel-level differentiation, in terms of memory views, for all the threads belonging to a specific process. To achieve this goals, we have developed a Linux module offering support for a special device file called `dev_multi_view` such that a process ID can be registered by simply issuing an `ioctl` call, with the command `REGISTER_PID`, towards the device file. In our overall software suite, this can be done via a proper shell command, which runs in a program that is fully separated from the one whose threads need to be monitored. Registering a process ID on the special device file allows the OS kernel to know that all the threads running within this process must be managed via thread-specific memory views, i.e. via sibling page-tables, so as to gain information on their working-set of logical pages along their subsequent execution phase.

The part of the OS kernel that needs to know whether some thread belongs to a process that is registered within `dev_multi_view` is the `schedule()` function. In our design, modifications of the scheduler to make it aware that per-thread memory views need to be setup, handled or switched off, is based on a *dynamic patching* approach where parts of the executable image of the kernel are rewritten at startup of the external module.

Our patching approach is based on retrieving the memory position of the block of machine instructions implementing `schedule()` from the system map and on injecting into this routine an execution flow variation such that control goes to a `schedule_hook()` routine offered by the

external module right before `schedule()` would execute its finalization part. Also, the patch works for (and has been tested on) Linux kernels from 2.6 to 3.2. The `schedule_hook()` function simply executes the same return actions originally planned by the kernel `schedule()` function. However, patching the original scheduler this way allows the hook to take control when the decision about what thread needs to take control of the CPU-core is already finalized—actually the thread has already taken control of the CPU-core, since we are just returning from the CPU-scheduling process.

The `schedule_hook()` routine implements a state machine that allows the scheduled thread to switch to its own memory view, or to switch back to the original memory view in case the working-set tracing phase has been switched off. This is again supported via a proper `ioctl` call, with the `DEREGISTER_PID` command, leading to deregister the process ID from `dev_multi_view`. A thread switches to its private view right upon being CPU-rescheduled if the process it belongs to is registered within `dev_multi_view`. This is checked by the scheduled thread by assessing the value of `current->tgid`, which is the Linux process control block field used to keep the actual process ID for all the threads living in the same process. The switch operation to the private view entails three steps:

(A) The thread registers its ID (namely `current->pid`) into a control table, still supported via our Linux module, used to keep track of what threads need to be executed by relying on private memory views.
(B) The private memory view is instantiated for the registering thread, which is done by setting up a sibling page-table.
(C) Right before returning, `schedule_hook()` updates the content of the page-table pointer register CR3 so as to point to the sibling page-table.

If the rescheduled thread finds itself registered in the control table but the process in which it lives is no longer registered into `dev_multi_view`, then `schedule_hook()` deregisters the current thread from the control table, switches it back to the original memory view by resetting the CR3 register to point to the original process page-table (whose address is kept by `current->mm->pgd`), and then tears down the sibling page-table. However, if no change is observed in relation to register/deregister operations of the process ID on `dev_multi_view` since the last reschedule, the current thread persists in the last selected memory view—the original one or the private one—just depending on the conditions observed in the last passage within the `schedule_hook()` code block. The execution flow of `schedule_hook()` is schematized in Figure 1.

B. Sibling Page-Tables

In x86_64 processors, the paging scheme is based on (up to) 4 indirection levels, as shown in Figure 2. The top-level page-table is called PML4 (or PGD—Page General Directory) and keeps 512 entries. All the other page-tables,

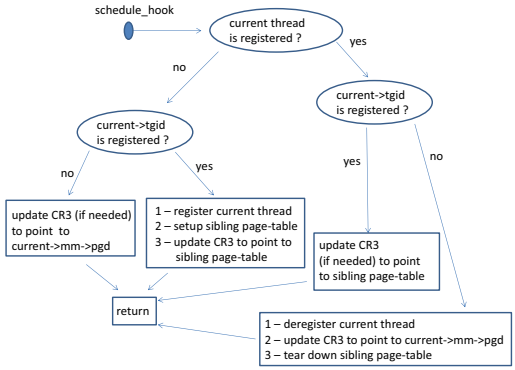

Figure 1. Execution diagram of `schedule_hook`.

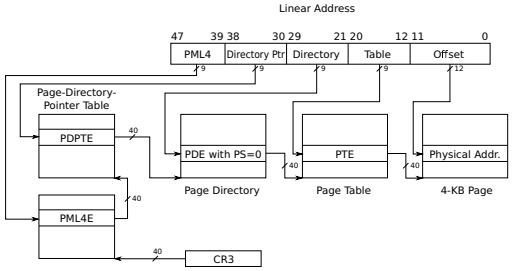

Figure 2. The paging scheme in x86_64 processors.

operating at lower levels, still have 512 entries each. In our proposal, per-thread memory views are based on setting up sibling page-tables that, in their initial state, are essentially *partial copies* of the original PML4 associated with the process within which the threads are running. This is the PML4 table pointed by `current->mm->pgd`.

As shown in Figure 3, the sibling PML4 has all the entries associated with kernel-level addresses set to be a copy of the corresponding ones from the original PML4. Rather, all the other entries are set to the value `NULL`. This way, as soon as the thread living in the memory view associated with the sibling page-table returns to user mode, any memory access to user space logical addresses will give rise to page faults[2].

These page faults are handled in our architecture via a modified version of the original page-fault handler, which is setup while mounting our external module via simply rewriting the associated entry of the Interrupt-Descriptor-Table (IDT). Upon gaining control, this modified version needs to discriminate whether the fault is a real one (minor or not), or is simply due to the fact that the sibling page-table is not aligned to the original one, hence not yet granting

[2]In our design, sibling page-tables can be also configured to have the PML4 entries associated with the application `text` sections already filled with the corresponding ones kept by the original PML4. This can help in all the scenarios where the threads running within the truly multi-thread application rely on the same modules within the application code, such as for categories of HPC platforms [28], [29]. In these scenarios, all the concurrent threads are highly likely correlated in the access to text-reserved virtual pages, so that tracing the accesses and moving these pages across different NUMA nodes might produce pure overhead with no advantage in terms of clustering of threads/data on a same NUMA node.

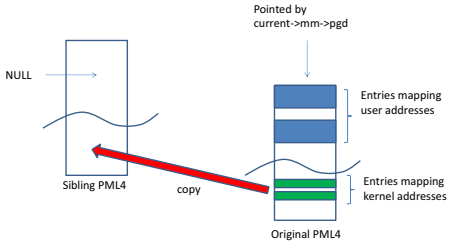

Figure 3. Initial setup of the sibling PML4.

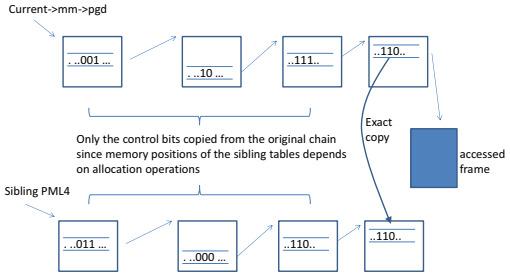

Figure 4. Setup of the per-thread sibling chain of page-tables.

access to the target page. To determine how to handle the fault, the modified handler tries to traverse the original chain of PML4/PDP/PDE/PTE tables so as to assess whether the target page is in memory. In the positive case, the sibling view is simply realigned to the original one by allocating and setting up the proper tables (and entries) along the sibling PML4/PDP/PDE/PTE chain corresponding to that page. An example of this scheme is shown in Figure 4.

If the page fault was a real one (e.g. a minor one on empty-zero memory), meaning that the page could not be located along the original chain of page-tables, the original page-fault handler is invoked, which will resolve the fault along the original chain. In fact, it uses `current->mm->pgd` to locate in memory the chain to be consulted/updated. Upon the return of the original handler, our modified version passes control back—along the same thread—to the application code, which lives within the memory view associated with the sibling page-tables. This will regenerate a fault that can anyhow be treated as explained before (by simply realigning the sibling view to the original one, which now allows pointing to the correct page in memory). This leads to an iterative procedure with no more than 2 steps.

By inducing the faults on the sibling page-table, thanks to its initial setup with `NULL` PML4 entries for user space addresses, we can detect the exact pages that are accessed by any individual concurrent thread living in a process that is registered in the `dev_multi_view` device file. The addresses of the faulted pages are then passed as input to an audit subsystem used to record the actual per-thread working-set of pages, still supported via our external module, whose details will be provided in the next section.

We also note that our scheme is able to handle both 4KB page size (which exactly relies on all the 4 levels of paging we described above) and large pages, namely 2MB pages. In the latter case, the sibling chain that maps a 2MB page will only entail 3 levels of page-tables, namely PML4/PDP/PDE. In fact, our custom fault handler, while traversing the original chain of page-tables, is able to determine whether the target page is a large one or not, and to setup the sibling page-tables' chain accordingly.

C. Logging the Faulting Accesses

Any fault occurring on sibling memory views is logged in our architecture via a kernel-level data structure, whose

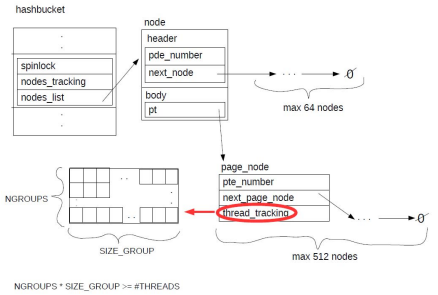

Figure 5. Data structure used to log the faulting accesses.

schematization is provided in Figure 5. Essentially, it is based on a scalable hash table, where each hash bucket allows for keeping track of faults occurring on any of 64 memory regions associated with the same number of PDE tables. Each PDE-record is kept via a node that allows pointing to a list of up to 512 `page_node` structures. Each of these structures allows for keeping, via its `thread_tracking` field, the number of accesses that have been performed on that same page by up to `NGROUPS`×`SIZE_GROUP` different threads. These parameters can be configured depending on the scale of the used multi-core machine, and on the (maximum) amount of threads that are supposed to be executed within the hosted applications. In particular, the value of `SIZE_GROUP` determines the reference cardinality of the number of threads that can be monitored within a process (64 in our suggested configuration). If the number of employed threads is expected to exceed this value, then more groups can be used. We also note that the management logic of the above data structure allows logging memory accesses according to two alternative modes: (A) **One-shot**, where we simply keep track of whether a logical page has been accessed or not since the last activation of the MVAS facility; (B) **Count**, where we count the number of faults that have occurred for a given thread across multiple activations of the MVAS facility.

D. Safe Execution with Sibling Page-Tables

An important aspect to consider when dealing with sibling memory views is the fact that sibling page-tables need to be kept consistent with the memory mapping kept by the original page-tables' chain, even in scenarios where pages that were valid at some point in time are then invalidated,

or undergo a change of their access rules or of their actual physical mapping. One example of invalidation is when a call to the `munmap` system call for a virtual memory region that is currently valid is issued by the application, which was already allocated in RAM (e.g. because of minor faults when that region was still in the empty-zero state). As for these scenarios, the Linux kernel already entails mechanisms such that system calls of this kind are executed atomically in terms of (A) changes performed on the original page-tables' chain (e.g. invalidation of the entries at some level of the chain) and (B) TLB invalidation on one or multiple CPU-cores. The latter target is achieved via so called TLB-invalidate Inter-Processor Interrupt (IPI) notifications across the CPU-cores.

In our architecture, we need to reach a similar level of atomicity, so that the sibling page-tables need to be atomically updated in case of the execution of a system call that leads to shrinking the set of valid pages in the process address space, or to changes of the access permissions and/or of the position of logical pages in physical memory. To achieve this objective, we wrapped all the system calls of this kind by hacking the system call table at module startup, so that the wrappers temporarily lock the access to the sibling page-tables, invalidate their entries and then call the original system call code. Clearly, all the operations depicted in Section III-B, namely those that lead to updating the chain of sibling page-tables upon a faulting access, need to grab the lock, so as not to incur in inconsistency in case of concurrent operations by the above wrappers. However, each sibling page-tables' chain has its own spin-lock, so that two different threads that concurrently fault on their own memory views do not block each other in the update of the corresponding chains of sibling page-tables. Overall, global synchronization is only requested in case any of these threads calls one of the above-cited system calls, given that the corresponding wrappers need to lock all the sibling memory views.

As a final note, we exploited `swapoff`/`swapon` services natively offered by Linux in order to temporarily avoid asynchronous modifications of the original page-tables' chain due to page swapping by the `kswapd` daemon while the working set tracing phase is active.

IV. Migration Rules and Support

In this section we propose a greedy policy whose objective is to cluster as much as possible a thread and its working set of pages (some of which possibly shared with other threads) on a same NUMA node. Let us suppose that we are dealing with a truly multi-thread application with T concurrent treads, each one associated with an ID in the interval $[0, T-1]$. Let us suppose, additionally, that the overall estimated working-set of the application is composed of P pages. For each page, we generate an *access-count tuple*:

$$p_i = \langle n_0, n_1, \ldots, n_{T-1} \rangle \tag{1}$$

where each n_j is the number of accesses by each thread j on the i-th page. For each tuple p_i we compute the maximum value $M_i = \max_j\{n_j\}$, which keeps the highest access count for the i-th page. The ID associated with the corresponding thread is stored in the *maximum access count* vector $\mathbf{m}$, which will be later exploited. We then convert the access-count tuple p_i in a *relative access-frequency tuple*:

$$\varphi_i = \left\langle \frac{n_0}{M_i}, \frac{n_1}{M_i}, \ldots, \frac{n_{T-1}}{M_i} \right\rangle \tag{2}$$

The tuple φ_i allows to estimate which threads have a higher fraction of accesses on a given page p_i. This tuple is then combined in a *per-page access-frequency matrix*. Specifically, given two indices l and m in $[0, T-1]$, the per-page access-frequency matrix (related to the i-th page) is a symmetric matrix where $\forall l \neq m$ the corresponding elements are defined as:

$$\Phi^i_{l,m} = \Phi^i_{m,l} = \begin{cases} (\varphi_{i,l} + \varphi_{i,m})/2 & \text{if greater than } \alpha \\ 0 & \text{if } \varphi_{i,l} \vee \varphi_{i,m} = 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $\alpha \in [0, 1]$ is a *controlling factor*. The higher the value of α, the higher must be the access frequencies to a given page to consider two different threads as correlated on that page. From each Φ^i, we then build the final (symmetric) *access matrix*, telling whether two threads share a certain amount of the memory pages in their working-set, which is defined as:

$$A_{l,m} = A_{m,l} = \sum_{i=0}^{P-1} \Phi^i_{l,m} \tag{4}$$

We then extract from A all the elements such that $l \neq m$, in order to generate a vector $\mathbf{v}$ of tuples $\langle t_l, t_m, a_{l,m} \rangle$, where $a_{l,m}$ is the component of A at position (l, m). This vector is then ordered according to the descending order of $a_{l,m}$ values, allowing us to identify which are the pairs of threads having higher affinity in terms of accessed memory pages. We also note that, when employing the **one-shot** fault-logging approach, $A_{l,m}$ elements will either acquire value 0 or 1, with value 1 indicating that two threads are correlated given that they touched (at least one time) the same virtual page, as tracked via the multi-view facility.

The next step of our greedy approach maps threads to cores taking into account the corresponding NUMA node that is close to each core. In particular, we assume that the machine on which we are running is composed of Z NUMA nodes and that the CPU-cores belong to some cluster z_k, which has direct access (say is close) to the memory of node $k \in [0, Z-1]$.

We iterate over the elements of $\mathbf{v}$, starting from v_0 by picking the thread IDs t^0_l and t^0_m, and we add these threads to the cluster z_k of CPU-cores, with k initially set to 0. Then, we check the next element $v_1 \in \mathbf{v}$, and again extract t^1_l and t^1_m. If either t^1_l or t^1_m corresponds to either t^0_l or t^0_m, we check whether the previous cluster z_k has still enough space (namely, not all CPU-cores are already

assigned to threads). In the positive case, we add t_l^1 or t_m^1 to z_k, otherwise we switch to z_{k+1}. If neither t_l^1 nor t_m^1 are equal to t_l^0 or t_m^0, we directly add them to the next z_{k+1} which has enough space. This step is repeated until all threads are assigned to some cluster z_k. If T is greater than the total number of available CPU-cores, we necessarily need to operate a mapping possibly leading to time-sharing concurrency across (some of) the T threads. In this situation, we simply leave the decision to the underlying operating system kernel—by not forcing any binding at all—so as to exploit the (already optimized) CPU-sharing policies offered by the operating system.

After having determined the mapping of threads to CPU-cores, we start scanning the maximum access count vector $\mathbf{m}$, so as to identify the NUMA node each page should be migrated to. In particular, as mentioned, each entry $m_i \in \mathbf{m}$ holds the ID of the thread which has the highest access ratio for the i-th page. We therefore identify the cluster z_k the thread belongs to, and we flag the memory page to be mapped to the corresponding NUMA node k.

The thread/page migration support based on the above greedy algorithm has been implemented via a (low-priority) user-space separate daemon that is activated via a shell command (fully external to the multi-thread application whose runtime needs to be monitored and optimized). In order to let this daemon retrieve from the kernel-level data structures the information used to build the access-count tuples p_i, we have added a system call for flushing data related to logged accesses into the `syslog` buffer via `printk`.

V. Experimental Data

A. Reference Hardware/OS Platform

We executed experiments on a 64-bit NUMA HP ProLiant server, equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 12MB L3 cache (6 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is OpenSuse 13.2 (Harlequin) (x86_64), with Linux kernel 3.16.7. The computing platform entails 8 different NUMA nodes, each one close (distance 10) to 4 CPU-cores and far (distance 20) from all the others.

B. Test-bed Software Platform

To test our proposal we selected the ROOT-Sim [15] open source truly multi-thread HPC platform, which supports multi-core parallel execution of discrete event simulation models[3]. On top of this platform we have run a high fidelity discrete event simulation model of a cellular system (we refer the reader to [23] for details). The motivations for which we believe the selected ROOT-Sim platform (coupled with the overlying test-bed application) represents a good case study are the following ones:

(A) It relies on speculative processing techniques hence leading—as is well known [6]—to large usage of memory for (temporarily) keeping both speculatively produced data records and in-memory logs for recoverability. Overall, it is a memory intensive case study that, in the selected configuration, has stably reached RAM occupancy on the order of 30 GB.

(B) The running threads are definitely CPU-bound. Hence their execution speed can be significantly affected by the efficiency according to which data are accessed within the NUMA platform (i.e. performance is not interfered by I/O or other blocking services). We have run the platform by relying on 32 concurrent worker threads, thus fully exploiting the underlying 32 CPU-core platform.

(C) It implements a load-sharing policy (see [28]) that allows different worker threads to operate on (say to take care of the execution of) different sets of simulation objects (either in forward mode or in rollback mode in case of causal inconsistencies) along different wall-clock-time windows. This leads to the scenario where the virtual pages used for the objects' representation and for their recoverability data are accessed in shared mode by the threads (depending on runtime decisions of the load-sharing policy and on how it clusters the objects across the worker threads).

(D) It already embeds a NUMA oriented memory management support (see [22]) that operates in user-space, which can be exploited for a comparative analysis of our current proposal.

As for the latter point above, the user-space NUMA oriented subsystem is based on ad-hoc allocation services that keep track of what virtual pages are destined for which simulation object, and for its recoverability data. Also, when an object is periodically (re-)bound to specific worker threads for load-sharing purposes, the associated virtual pages are all migrated to the NUMA node hosting these threads. This scheme can be classified as a library based approach where the allocation/migration sub-system only keeps a map of the overall page allocations, and does not know what pages are—and likely will be—of interest for the object execution during the next wall-clock-time window. On the other hand, it does not require runtime tracing of memory accesses. Overall, this scheme can be seen as based on the concept of *resident-set*, given that all the pages associated with data structures used to support the execution of an object are migrated. Rather, our new proposal is based on migrating pages belonging to the working-set since we are able to accurately–and periodically—trace memory accesses per-thread, and the threads' data-share profile (thus migrating only the pages that are currently hot for sets of threads). Orthogonality of the two approaches allows us to compare different philosophies targeting NUMA optimization.

C. Results

Multi-view Address Space Overhead: We initially focused on evaluating the overhead induced by our solution. To this end, we run the selected test-bed platform by activating the MVAS facility and logging the faulting accesses, comparing the execution time with the one achieved with

[3]Code is available at `https://github.com/HPDCS/ROOT-Sim`

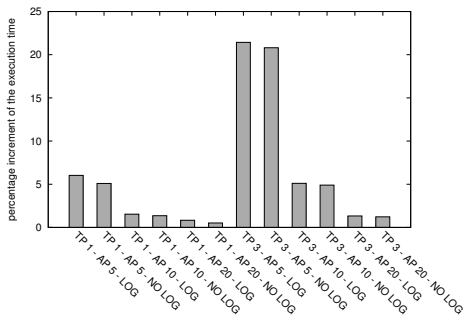

Figure 6. Overhead evaluation.

a baseline configuration with no activation of MVAS. Each execution-time sample has been computed as the average over 5 runs of the same application (under the same settings). Further, we considered different settings in relation to the usage of our multi-view based memory access pattern tracking system. In particular, we varied both the period according to which the MVAS facility is switched on while the application is in progress (referred to as AP - Activation Period - in the plots), and the interval of time during which the multi-view and the associated memory access pattern determination is kept in place (this parameter is referred to as TP - Trace Period - in the plots). As a last variant, we have run the multi-view configuration by excluding the logging of the faulting accesses (referred to as NO LOG in the plots), so as to be able to assess the relative overhead for managing the logging data structure presented in Section III-C, as evaluated by comparing these data with the ones achieved when the log facility is turned on.

We report in Figure 6 the percentage increment in the execution time induced by the different configurations compared to the baseline case. From the results we see very low or negligible overhead when running with MVAS except for configurations leading to extremely intensive memory access tracing. This occurs either for very reduced values of AP (say when the MVAS facility is frequently activated every 5 secs) or when the threads live within sibling views for longer time (say when TP is set to the value 3 secs). We also see that the relative overhead by the logging facility of the faulting accesses supported via the data structure presented in Section III-C is quite negligible.

We note that, the longer TP, the higher is the likelihood that the finally determined working-set contains both very hot pages (with high current frequency of accesses by a specific thread) and pages that are used less frequently along the current execution phase. Hence, maintaining the MVAS facility active for shorter periods (say when setting TP to reduced values, like 1 sec) would anyhow allow capturing the accesses to the hot pages within the working-set, whose placement optimization within the NUMA platform would expectedly lead to the highest benefits. At the same time, shorter TP values would expectedly lead to minimal overhead by MVAS as pointed out by the experimental data.

Benefits Assessment: We then focused on evaluating the effects of the NUMA oriented architecture made up by combining MVAS with the thread/data placement policy provided in Section IV. In particular, we compared the execution time provided by our solution with the one achieved with the baseline configuration of the test-bed platform, and with the one achieved when resorting to the resident-set NUMA oriented migration facility already supported by the platform (referred to as *RS migration* in the plots).

We included in the comparison an operating system based variant of NUMA optimizer where all the threads living within the same truly multi-thread application (hence within the same address space) are traced, in terms of page accesses, by relying on a single page-table supporting the traditional single-view address space. This is the philosophy characterizing solutions like KMAF [8] and AutoNUMA [27]. Overall, although we did not test directly against these systems, our experiments are somehow representative of testing against the core memory access tracing philosophy they rely on, with the advantage of not inducing biases due to the different logic for page-fault handling (since for both multi-view and single-view we exploit the same implementation of the kernel level tracer/logger of accessed pages, while KMAF and AutoNUMA have different implementations for carrying out this task).

On the other hand, the employed migration policy for both multi-view and single-view contexts (and its actual implementation support) is exactly the one we provided in Section IV. This policy is fully orthogonal to the method used to detect actual memory accesses, which further favors fairness while comparing multi-view and traditional single-view approaches for the (periodic) determination of the working-set of the concurrent threads, and their final impact on performance optimization in NUMA machines when migrating threads/data.

For both multi-view and single-view scenarios we decided to rely on TP set to one (thus targeting the determination of the hot pages within the current working-set), and on cumulating the faults by a same thread on a given page, via the **Count** mode of the logging data structure, across two subsequent activations of the multi-view facility. This has been done according to a kind of second chance approach, leading a page to be considered as belonging to the current set of hot pages because of a faulting access in either the first or the second of the two subsequent activations of the MVAS facility (or of the single-view based memory access tracing scheme). If a page is tracked as being accessed by a thread in both the activations (which would lead to the value 2 for its counter of references by a given thread), it will represent a very hot page. Another parameter that we have varied in the experiments (for both multi-view and single-view configurations) is AP, by setting it to either 10 or 20 secs. Finally, we experimented with three different values of the parameter α used by the thread/data migration policy provided in Section IV in order to discriminate whether different threads can be considered as correlated in the

access to a given virtual page. As a last note, any time the policy solver and the thread/data migration support are activated (this occurs at each AP), the corresponding shell command of our software suite is launched with `nice` factor 19 thus running in non-intrusive mode. This is done to avoid alterations of performance data (potentially) caused by resource demand from policy solving tasks and thread/data migration commands.

The gathered performance data are reported in Figure 7. We can observe how the baseline configuration not exploiting thread/data affinity and move facilities shows, as expected, the worst performance. The configuration based on RS migration allows for reducing the execution time by about 19%. The execution time is further reduced by relying on both single-view and multi-view memory access tracking schemes coupled with the thread/data migration policy and support of our own. However, while the maximal reduction in the execution time by the single-view scheme is of the order of 35%, the reduction provided by the multi-view approach further improves to 40% (thus providing more than 10% further gain compared to the single-view approach). This looks a significative result when considering that fault induction in the single-view approach represents the foundation of last generation state of the art methods for operating system based memory access pattern determination. Another important point to note is that the performance offered by the single-view scheme is poorly influenced by variations of the parameter α. This is still expected, given that single-view based access pattern determination suffers from false negatives (missing access-tracking) when multiple threads share pages within their working-sets at a given point in time. Hence, it mostly allows for detecting whether some page can be considered hot for at least one thread, while it may result less effective in detecting the actual correlation in the accesses. On the other hand, our innovative multi-view based approach avoids false negatives and traces the working-set (e.g. the current hot pages) of individual threads within a same process accurately. When filling the tracing outcome as input to the thread-migration policy, we get the possibility to increase the level of correlation while decreasing the value of α which allows for better detecting the utility of clustering specific threads and pages on a same NUMA node. In fact, the configuration with α set to 0.9 leads the multi-view scheme to correlate two different threads on a same page only if, after the MVAS facility is shut down during the current tracking and optimization period, they both have their counters of references to that page set to the value 2 (meaning that the page is very hot for both). On the other hand, lower values of α allow for detecting useful correlations even when the page is likely becoming hot (e.g. it exhibits access count set to the value 1), which might further help optimizing thread/data placement.

The very last note is related to the difference in the execution time between the RS migration approach, and the ones based on working-set determination via runtime tracking of the accessed pages. The latter ones outperform

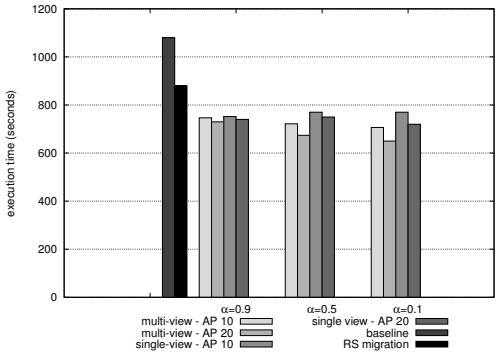

Figure 7. Performance data.

the former not because the former is unable to place the right pages close to the right threads (given that it knows what pages represent the overall memory segment of interest for a given thread along a given execution phase). Rather, it happens because it blindly moves all the pages of potential interest for some thread, independently of actual accesses. Although the move is carried out via lower priority demons (as we also do in our proposal), the oversized memory amount that is actually moved generates additional overhead which is instead avoided via solutions that discriminate at runtime the actual access pattern and the associated hot pages. This is allowed by our proposal, with the new feature of enabling the accurate tracking of the access pattern per-thread within the same address space.

VI. Conclusions

In this article we have presented an operating system innovative facility called multi-view address space (MVAS) enabling the accurate determination of the working-set of logical pages (and of its variation over time) for any individual thread running in a multi-thread process, when also considering shared page accesses by different threads. The target operating system is Linux/x86_64 and the whole architecture we provide is fully included in an external loadable module. This facility is exploitable for moving threads and pages in NUMA machines so as to allow the pages belonging to the (intersecting) working-sets of the threads to be placed on NUMA nodes that are close to the CPU-cores where the corresponding threads are executed. A policy for such a migration, and its support, are also provided. The proposed solution has been experimentally assessed by relying on a motivated case study in the context of HPC.

References

[1] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proc. of the 19th International Conference on Parallel Architecture and Compilation Techniques*, pages 319–330, 2010.

[2] N. Barrow-Williams, C. Fensch, and S. W. Moore. A communication characterisation of splash-2 and parsec. In *Proc. of the International Symposium on Workload Characterization*, pages 86–97, 2009.

[3] H. Casanova, F. Desprez, and F. Suter. On cluster resource allocation for multiple parallel task graphs. *J. Parallel Distrib. Comput.*, 70(12):1193–1203, 2010.

[4] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *Proc. of the 20th Annual International Conference on Supercomputing*, pages 353–360, 2006.

[5] E. H. M. da Cruz, M. Diener, M. A. Z. Alves, L. L. Pilla, and P. O. A. Navaux. Optimizing memory locality using a locality-aware page table. In *Proc. of the 26th International Symposium on Computer Architecture and High Performance Computing*, pages 198–205, 2014.

[6] S. R. Das and R. M. Fujimoto. Adaptive memory management and optimism control in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 7(2):239–271, 1997.

[7] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *Proc. of the 8th Internatinal Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–394, 2013.

[8] M. Diener, E. H. M. da Cruz, P. O. A. Navaux, A. Busse, and H. Heiß. KMAF: automatic kernel-level management of thread and data affinity. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation*, pages 277–288, 2014.

[9] M. Diener, F. L. Madruga, E. R. Rodrigues, M. A. Z. Alves, J. Schneider, P. O. A. Navaux, and H. Heiss. Evaluating thread placement based on memory access patterns for multi-core processors. In *Proc. of the 12th International Conference on High Performance Computing and Communications*, pages 491–496, 2010.

[10] R. M. Fujimoto, K. S. Panesar, and K. S. Panesar. Buffer management in shared-memory Time Warp systems. In *Proc. of the 9th Workshop on Parallel and Distributed Simulation*, pages 149–156, 1995.

[11] X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *Proc. of the Workload Characterization Symposium*, pages 46–55, Oct 2005.

[12] X. Gao, B. Simon, and A. Snavely. ALITER: an asynchronous lightweight instrumentation tool for event recording. *SIGARCH Computer Architecture News*, 33(5):33–38, 2005.

[13] M. Gorman. *Understanding the Linux Virtual Memory Manager*. PRENTICE HALL, 2004.

[14] D. Hendler, A. Naiman, S. Peluso, F. Quaglia, P. Romano, and A. Suissa. Exploiting locality in lease-Based replicated transactional memory via task migration. In *Proc. of the 27th International Symposium on Distributed Computing*, pages 121-133, 2013.

[15] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator. https://github.com/HPDCS/ROOT-Sim.

[16] D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev. Optimization of parallel discrete event simulator for multi-core systems. In *Proc. of the International Parallel and Distributed Processing Symposium*, pages 520–531, 2012.

[17] C. Karlsson, T. Davies, and Z. Chen. Optimizing process-to-core mappings for application level multi-dimensional MPI communications. In *Proc. of the International Conference on Cluster Computing*, pages 486–494, 2012.

[18] M. A. Laurenzano, J. Peraza, L. Carrington, A. Tiwari, W. A. W. Jr., and R. L. Campbell. PEBIL: binary instrumentation for practical data-intensive program analysis. *Cluster Computing*, 18(1):1–14, 2015.

[19] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Proc. of the Symposium on Principles and Practice of Parallel Programming*, pages 90–99, 2006.

[20] Oracle. Plug into The Cloud with Oracle Database 12C (white paper), http://www.oracle.com/technetwork/database/plug-into-cloud-wp-12c-1896100.pdf.

[21] A. Pellegrini and F. Quaglia. The ROme OpTimistic Simulator: A tutorial (invited tutorial). In *Proc. of the 1st Workshop on Parallel and Distributed Agent-Based Simulations*, pages 501–512, 2013.

[22] A. Pellegrini and F. Quaglia. NUMA Time Warp. In *Proc. of the 3rd SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 59–70, 2015.

[23] A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Trans. Parallel Distrib. Syst.*, 26(6):1560–1569, 2015.

[24] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira. Compiler support for selective page migration in numa architectures. In *Proc. of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, pages 369–380, 2014.

[25] M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, 68(9):1186–1200, 2008.

[26] F. Trahay, F. Rué, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra. Eztrace: A generic framework for performance analysis. In *Proc. of the 11th International Symposium on Cluster, Cloud and Grid Computing*, pages 618–619, 2011.

[27] R. van Riel and V. Chegu. Automatic NUMA Balancing. Technical Report 1.0, 2014.

[28] R. Vitali, A. Pellegrini, and F. Quaglia. Load sharing for optimistic parallel simulations on multi-core machines. *ACM Performance Evaluation Review*, 40(3):2–11, 2012.

[29] J. Wang, D. Jagtap, N. B. Abu-Ghazaleh, and D. Ponomarev. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1574–1584, 2014.