NUMA Time Warp

Alessandro Pellegrini and Francesco Quaglia
DIAG – Sapienza, University of Rome
Via Ariosto 25, 00185 Rome, Italy
{pellegrini, quaglia}@dis.uniroma1.it

ABSTRACT

It is well known that Time Warp may suffer from large usage of memory, which may hamper the efficiency of the memory hierarchy. To cope with this issue, several approaches have been devised, mostly based on the reduction of the amount of used virtual memory, e.g., by the avoidance of checkpointing and the exploitation of reverse computing. In this article we present an orthogonal solution aimed at optimizing the latency for memory access operations when running Time Warp systems on Non-Uniform Memory Access (NUMA) multi-processor/multi-core computing systems. More in detail, we provide an innovative Linux-based architecture allowing per-simulation-object management of memory segments made up by disjoint sets of pages, and supporting both static and dynamic binding of the memory pages reserved for an individual object to the different NUMA nodes, depending on what worker thread is in charge of running that simulation object along a given wall-clock-time window. Our proposal not only manages the virtual pages used for the live state image of the simulation object, rather, it also copes with memory pages destined to keep the simulation object's event buffers and any recoverability data. Further, the architecture allows memory access optimization for data (messages) exchanged across the different simulation objects running on the NUMA machine. Our proposal is fully transparent to the application code, thus operating in a seamless manner. Also, a free software release of our NUMA memory manager for Time Warp has been made available within the open source ROOT-Sim simulation platform. Experimental data for an assessment of our innovative proposal are also provided in this article.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Virtual Memory*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Discrete Event, Parallel*

SIGSIM-PADS'15, June 10–12, 2015, London, United Kingdom.
Copyright © 2015 ACM ISBN 978-1-4503-3583-6/15/06...$15.00.
DOI: http://dx.doi.org/10.1145/2769458.2769479.

General Terms

Algorithms, Performance

Keywords

PDES; Non-uniform Memory Access; Speculative Processing

1. INTRODUCTION

Speculative computing techniques are widely recognized as a means to achieve scalability of parallel/distributed applications thanks to the (partial) removal of the cost for coordinating concurrent processes and/or threads from the critical path of task processing [16, 30]. In the context of Parallel Discrete Event Simulation (PDES), the speculative paradigm is incarnated by the well-known Time Warp synchronization protocol [18], which has been recently shown to provide scalability up to thousands or millions of CPU-cores [20].

On the other hand, it is also known that a core pitfall of speculative processing schemes lies in their large usage of memory, given that they typically require maintaining (and accessing) histories of speculatively-produced data records and/or recoverability data to cope with misspeculation scenarios. This leads to reduced application locality and consequent suboptimal behavior of the memory hierarchy, given that the working set of the application will more likely exceed cache-storage.

As for Time Warp, its large usage of virtual memory has been shown to represent a potentially insurmountable obstacle to the delivery of adequate performance, as for cases where uncommitted data records saturate RAM (or even virtual memory) thus requiring secondary storage support for the parallel run. On the one hand, such extreme scenarios have been shown to be faceable by proper memory management protocols explicitly aimed at limiting the level of speculation within the Time Warp run (see, e.g., [11]). On the other hand, the classical approach that is employed for reducing the memory footprint of Time Warp applications, and hence for improving their locality, consists in reducing the amount of memory that is kept for allowing recoverability of the state of the concurrent simulation objects (also known as Logical Processes—LPs). Along this path we can find solutions based on infrequent and/or incremental checkpointing of the snapshot of the simulation object's state (or a combination of the two approaches) [25, 27, 28] or on the usage of reverse computing techniques [9]. The latter may (at least in principle) fully avoid keeping state recoverability data, at the expense of an increase in CPU requirements for rebuilding a past state snapshot.

However, despite the wide literature in the area of reducing the memory usage for state recoverability data, the large usage of virtual memory and the reduced locality in Time Warp still stand as intrinsic to this synchronization approach. As an example, event records keeping information about events that have already been processed, but which cannot yet be discarded because their commitment is still undetermined, contribute to enlarging the amount of virtual memory to be managed. Also, this memory is requested to be scanned upon *fossil collection* for buffer release to the underlying allocator, which gives rise to a change of the locality from the live event buffers' portion to the obsolete one. Hence, the efficiency according to which the cache/RAM hierarchy is exploited still stands as a core issue to be addressed while building high performance Time Warp systems.

In this article we tackle the above issue by presenting a memory management architecture that is aimed at reducing the actual latency for the access to memory locations when running Time Warp systems on top of multi-core Non-Uniform-Memory-Access (NUMA) platforms. The relevance of the NUMA architectural paradigm lies in that systems continuously increase their number of processors and the overall size of RAM storage. Hence it is increasingly difficult to build large platforms with uniform memory access latency. Nowadays, medium-end parallel machines equipped with non-minimal amounts of CPU-cores and/or RAM are de-facto configured to have different RAM zones *close* to specific CPU-cores, hence providing low latency and high throughput of memory access, while other zones are *far* from these same CPU-cores and induce higher latency and a reduced memory access throughput. This de-facto standard configuration has led system-software developers to recently include facilities for optimizing memory management in NUMA contexts, such as for the case of Linux, which starting from version 2.6.18 supports NUMA specific services that are exposed to the application-level software via system calls.

We exploit such services offered by Linux, and reflect them into a NUMA-aware memory management architecture operating at the level of the Time Warp platform, particularly in the context of multi-threaded Time Warp systems. Specifically, we provide an architectural organization where any information record logically bound to a specific simulation object (such as its event records, keeping either already processed or unprocessed events, and its state recoverability data, as well as its live state) is guaranteed to be kept on private memory pages. Hence, the simulation objects (including their event buffers) are actually allocated on disjoint sets of memory pages. Also, the memory pages associated with data for an individual simulation object are guaranteed to be located on the NUMA node where the worker thread currently in charge of running the simulation object resides. This leads the worker thread to benefit from minimum RAM-access latency and maximal throughput when performing read operations (e.g. while scanning data associated with the simulation object).

The association of the simulation object's private pages with the correct NUMA node is operated in our architecture according to both static and dynamic policies. The latter approach is actuated according to an non-intrusive daemon-based solution, where a thread (similar in spirit to the Linux `kswapd`) is in charge of periodically checking whether a migration from a source to a destination NUMA node is requested for the memory pages of any simulation object, and in the positive case executes the migration non-intrusively. Hence, our proposal guarantees NUMA-efficient memory access even in contexts where a specific simulation object is migrated from one worker thread to another one (possibly operating on a CPU-core close to a different NUMA node), such as for the case of load-sharing with dynamic migration of the objects within a multi-thread Time Warp platform [36] or when the number of worker threads is dynamically changed with consequent re-distribution of the simulation objects across the still active worker threads [38]. Further, our proposal entails optimized NUMA management also in relation to the memory buffers used for exchanging events across the concurrent simulation objects.

We finally stress that our proposal is fully application-transparent, hence allowing the application-level programmer to still rely on classical (dynamic) memory management services, such as the malloc library, for memory allocation/deallocation within the simulation model. Also, the presented NUMA-oriented architecture has been made freely available by having it been integrated within the open source ROOT-Sim (ROme OpTimistic Simulator) package [24][1]. Experimental data for an assessment of our proposal are also presented in this article.

The remainder of the article is structured as follows. In Section 2 we discuss related work. The NUMA-oriented memory management architecture is presented in Section 3, together with the support for dynamic migration of the memory pages destined to an individual simulation object across different NUMA nodes. Experimental data are provided in Section 4.

2. RELATED WORK

Optimized approaches and architectures supporting state recoverability in optimistic PDES systems [25, 27, 28, 29, 31, 40] can be also considered as general solutions aimed at reducing the memory demand by the optimistic simulation environment, thanks to the infrequent and/or incremental nature of the employed checkpointing techniques. Such a reduction, which can lead to improving the memory locality and the cache/RAM hierarchy efficiency, can be considered as a reflection of the whole optimization process leading to well-suited tradeoffs between log and restore overheads. However, these literature solutions do not directly tackle the issue of memory-access efficiency in NUMA platforms. In fact, they do not integrate policies for controlling the delivering of memory buffers for logging state recoverability data so as to make memory accesses NUMA optimized, which is instead the target of this article. Overall, our proposal is fully complementary to (and can be integrated with) any of the above mentioned solutions. The same is true when considering the alternative recoverability method based on reverse computation (rather than checkpointing) [9].

As hinted, the memory demand by optimistic platforms does not only involve log operations, but the need for temporary maintaining event buffers that are not yet detected as already committed (since Global Virtual Time—GVT—is

[1]Please refer to `https://github.com/HPDCS/ROOT-Sim` for the actual free software package.

typically re-evaluated periodically), and the need for supporting speculative scheduling of future events. The latter aspect may entail high frequency of buffer allocation requests just due to the fact that purely optimistic approaches can allow the simulation objects to run far ahead of the currently committed horizon in case of the absence of blocking or throttling strategies. As for memory requirements related to the already committed portion of the computation, some advanced fossil collection mechanisms have been proposed [10] that, by means of dissemination of information about causality relations among events, are aimed at the identification of the fossils (hence of memory to be recovered) in a complementary manner compared to the classical ones based on GVT computation. Still, these approaches do not provide NUMA optimized memory management approaches, rather general solutions for prompt release (thus re-usage) of memory buffers as a general approach for reducing memory demand and increasing locality.

The effects of the cache/RAM hierarchy and of the underlying virtual memory system on the performance of specific tasks (such as state saving) and/or of the overall simulation run have also been (empirically) studied by several works [2, 3, 8, 14], some of which deal with the context of NUMA platforms. Outcomes by these studies show how both caching and virtual memory may have a relevant impact on performance, thus posing the need for optimizing platform level configuration and/or design in order to limit the performance degradation phenomenon. This is exactly what we do with our memory management proposal oriented to NUMA platforms.

Interesting proposals aimed at the integration of advanced memory management schemes specifically tailored to optimistic PDES platforms can be found in [13, 19, 21, 26]. Here the authors propose techniques, such as cancelback, pruneback or artificial rollback, which are aimed at achieving efficient executions of the optimistic paradigm when considering limited available memory. This is achieved by, e.g., artificially squashing portions of speculated computation in order to avoid maintaining the related information (such as the buffers for speculatively scheduled events) when memory demand becomes critical (e.g. when page-swapping phenomena in the virtual memory system would tent to appear). With this type of integrations, the optimistic approach has been shown to be able to complete the run at reasonable performance by using an amount of memory similar (or slightly larger than) the one requested by a sequential, non-speculative run of the same simulation application. Further, the performance tradeoffs associated with these schemes have been thoroughly investigated both analytically and empirically (see, e.g., [12]). Again, these solutions are complementary to the one we provide since none of them is specifically oriented to optimizing the efficiency of the cache/RAM hierarchy in NUMA systems.

Different approaches, still tailored to the tradeoff between memory management and performance, relate to the reduction of the number of memory copies for supporting event exchanges within the optimistic platform. Particularly, the proposal in [32] provides a so called zero-copy message passing approach, suited for both conservative and optimistic simulation, which allows reducing the whole memory demand due to message buffering on shared memory architectures, thanks to the reduction of the amount of virtual memory buffers used for keeping the messages. However, this approach is not directly oriented to improving the efficiency of memory accesses in NUMA platforms.

The work in [15] faces the cache hierarchy misuse in optimistic simulators by pointing out the relevance of buffer delivery mechanisms that are cache-friendly in shared memory contexts. The work exclusively accounts for buffers reserved for exchanged messages. It presents a new approach that partitions the memory destined to messages so that the pages are accessed only by the two processes that participate in the communication, providing a reduction of the cache-coherence overhead and the cache invalidation. Our proposal intrinsically offers the same advantage, but additionally copes with memory access efficiency to live states of the simulation objects and recoverability data, also in contexts where the simulation objects can be dynamically migrated across worker threads operating in different NUMA nodes.

The work in [34] presents a partitioning of the data structures typically employed in optimistic PDES systems in order to determine the so called access-intensive vs access-mild portions, the latter being the portions of data that unlikely will be accessed in the future. Access-mild data are mapped onto virtual addresses that collide on the same cache portion, so as to avoid that write operations when generating these data will erase access-intensive data from the cache. This approach does not aim at improving data movement in the cache/RAM hierarchy, e.g., in NUMA machines, rather it is aimed at increasing cache hits.

Finally, the work in [39] presents a performance study of multi-thread optimistic PDES systems (particularly a multi-thread variant of ROSS [7]) where various optimizations are considered, one of which is related to memory management in NUMA machines. Specifically, this work studies the effects of (re-)using memory buffers belonging to the destination memory pool when exchanging events across the simulation objects, so as to increase the likelihood that the buffer is actually located on the NUMA node where the thread running the destination simulation object resides. This solution does not directly cope with empty-zero memory, thus the buffer is guaranteed to be hosted on the correct NUMA node only in case it resides on non-empty-zero memory previously touched by the destination thread. This is due to the fact that the actual memory allocation policy is intrinsically based on default local-policies adopted by NUMA-oriented operating system kernels. We overcome this problem in our approach; further, we optimize NUMA access by explicitly considering the mapping of both the live state of the simulation object and its recoverability data, and by also offering support for dynamic migration of the object (and of all its associated data) across threads running on different NUMA nodes, aspects that are not considered by the proposal in [39].

3. THE MEMORY MANAGEMENT ARCHITECTURE

3.1 Architectural Context

We target optimistic PDES systems based on the multi-thread paradigm (rather than multi-process ones), which have been recently shown to be highly suited for shared memory platforms thanks to the possibility of optimizing aspects such as data exchange and balanced usage of the

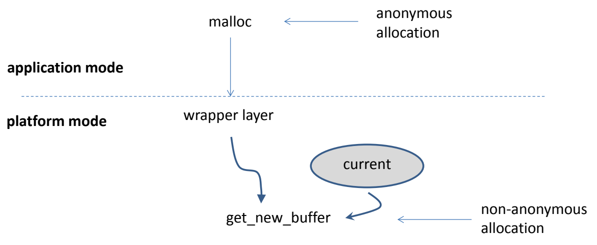

Figure 1: The dual-mode execution model.

computing power [36, 37] in application transparent manner. Also, we consider a convenient execution model where a simulation object, which we denote as $SOBJ_i$, is temporarily bound to a specific worker thread WT_j along a given wall-clock-time window. Hence the simulation objects can be periodically migrated across the worker threads, e.g., for load-sharing purposes.

On the basis of recent results in the design and development of optimistic PDES platforms, such as [23], we assume that the PDES system runs according to a dual-mode scheme where we distinguish between *application* vs *platform* modes. When a worker thread dispatches some simulation object for execution, then the application mode is entered. When the dispatched application callback function returns control to the underlying PDES environment along the running thread we (re)enter platform mode. With no loss of generality we assume that any worker thread WT_j has a notion of *current* simulation object, indicating the identity of the simulation object that has been dispatched for application execution along the thread.

Still in compliance with recent results in the area of multi-thread optimistic PDES platforms, we consider the scenario where the live state of a simulation object, its recoverability data, and its input and output queues are manipulated by a single worker thread at a time, namely the worker thread to which the object is currently bound. As a result, any event record (or anti-event record) exchanged across different simulation objects is never directly inserted into the destination queue. Rather, it is posted on a bottom-half queue, and later extracted and manipulated by the correct thread, namely the one in charge of running the destination simulation object. Also, we consider a software architecture where memory allocation/deallocation services by the application code are not directly issued towards the standard `malloc` library. Instead, they are transparently intercepted by the underlying PDES environment and redirected to proper allocators. As shown in [25], this can be achieved by simple compile/link time directives, where dynamic memory APIs, as well as third party libraries linked to the application code, can be wrapped so as to bounce control back to the underlying environment upon any call to these API functions.

According to the above premise, we target the scenario depicted in Figure 1 where:

- whenever the application code calls a standard-library memory management API (such the `malloc` service), the underlying environment can detect which is the invoking simulation object;

- the platform layer can perform non-anonymous memory allocation/deallocation operations where the platform internal API for these operations is aware of which simulation object is associated with the operation. Particularly, the function `get_new_buffer(int sobj_id, size_t size)` can be invoked so that buffer allocation will be executed selectively, on the basis of the identity of the simulation object for which data need to be stored. A `malloc` call by the application, ultimately intercepted by the platform layer, is served via a non-anonymous allocation operation in a transparent way to the application. Also, a platform level allocation, e.g. of some event buffer destined to keep data for a specific simulation object, is also executed non-anonymously.

3.2 Memory Allocator

As pointed out before, we operate in a context where the platform layer allocates memory (either for platform usage or for application usage) by relying on some allocation service `get_new_buffer` that is non-anonymous. The actual implementation of this service can be disparate. For convenience we resort on the open-source DyMeLoR allocator [33] for actual allocation of memory destined to the application, which is based on pre-allocation of large memory segments (originally via the actual `malloc` library), which are then logically partitioned into chunks. This allocator, besides delivering memory for usage by the caller, has also facilities for keeping the memory map of non-anonymously allocated chunks for a specific simulation object (via compact bitmaps) and making it recoverable. Hence it is well suited for serving memory requests originally issued by the application code, which give rise to memory layouts of the simulation objects that need to be made recoverable at past logical time values. We also developed a variant of DyMeLoR where recoverability data structures are fully removed. This variant is used in our final architecture to serve memory allocations/deallocations for platform level usage (which do not need to be made recoverable). However, we note that the actual NUMA memory management architecture we are presenting can be integrated with other kinds of user-level allocators. In fact the NUMA manager acts as the final (back-end) memory allocation service for the adopted user-level allocator.

Essentially, our NUMA manager can pre-reserve memory segments to be delivered to the overlying (user-level) chunk allocator. This is done non-anonymously, in fact the NUMA manager exposes a segment allocation function `void* allocate_segment(int sobj, size_t size)`, to be used for pre-reserving memory that will ultimately be used for managing chunks (hence data) associated with a specific simulation object. The overlying user-level allocator can install whatever information onto the segments, such as meta-data for managing the free room within the same segment (just like the `malloc` library does after pre-reserving memory from the operating system).

Pre-reserving is supported via the POSIX `mmap` system-call, which allows for validating in the process memory map a set of contiguous virtual pages, whose global size complies with the size of the segment allocation request. The NUMA allocator keeps, for each simulation object, a meta-data record `mem_map` which records the following fields:

```
void*  base;
size_t size;
int    active;
```

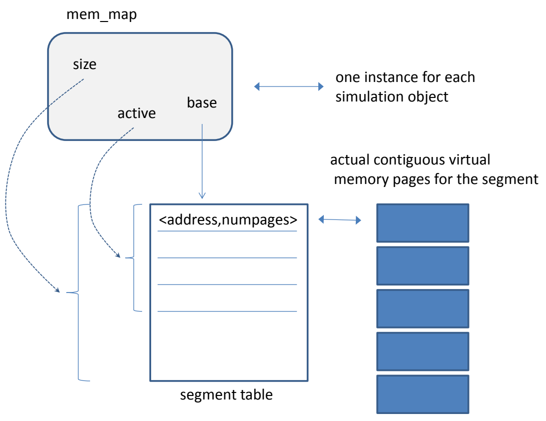
mem_map
size
active
base
one instance for each simulation object
actual contiguous virtual memory pages for the segment
<address,numpages>
segment table

Figure 2: `mem_map` data structures.

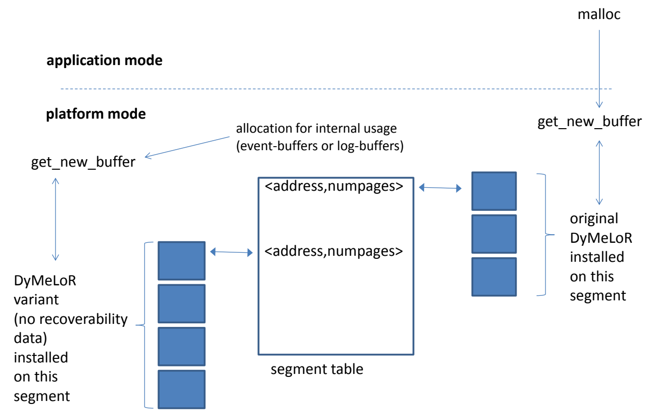
malloc
application mode
platform mode
get_new_buffer
allocation for internal usage (event-buffers or log-buffers)
get_new_buffer
<address,numpages>
<address,numpages>
original DyMeLoR installed on this segment
DyMeLoR variant (no recoverability data) installed on this segment
segment table

Figure 3: Memory allocation for platform and application usage.

where `base` is a pointer to a non-anonymous segment table, `size` indicates how many entries are currently present in the table, and `active` indicates how many of these entries are currently valid. The scheme is shown in Figure 2. Each time a new segment allocation is requested, the first free entry in the segment table associated with a specific simulation object is reserved, and after the `mmap` service is called the entry is used to keep the tuple $\langle address, numpages\rangle$ indicating the actual address of the sequence of pages that have been validated by the underlying operating system, and the total number of these pages. In case the segment table gets exhausted it is simply reallocated (again via `mmap`) with doubled size. On the other hand, in case a segment previously allocated is released, then the busy entries in the segment table are re-compacted at the top of the table by copying the last valid entry into the one of the released segment. This allows for $O(1)$ time complexity for segment allocation/deallocation (except for the cases where the table needs to be resized).

Overall, as shown in Figure 3, anytime some worker thread needs to allocate a buffer for platform level usage, which is destined to keep input or output events of a specific simulation object, or which is needed to record recoverability data for the simulation object state (namely a checkpoint), the `get_new_buffer` interface of the DyMeLoR user-level allocator (particularly the variant not keeping allocation/deallocation recoverability data) is called, and an actual buffer is delivered by this allocator which will necessarily reside on a non-anonymous segment pre-reserved by DyMeLoR. The same happens when the platform level software issues a call to the `get_new_buffer` service after having intercepted a `malloc` call issued from the application, particularly from the current simulation object dispatched along the worker thread (with the only difference that the instance of DyMeLoR is this time the one keeping allocation/deallocation recoverability data). Hence also the chunks belonging to the live state of the simulation object will be actually located on a non-anonymous memory segment.

By the above architectural organization, the NUMA memory manager guarantees that the set of virtual memory pages destined to keep event buffers, live state and recoverability data for any individual simulation object is actually disjoint from the set of virtual memory pages used for storing data associated with other simulation objects hosted by the multi-thread PDES system.

Let us now discuss the aspect of how these virtual memory pages are allocated on the different NUMA nodes. As well known, once a memory segment is allocated by the NUMA manager via the `mmap` system-call, its pages do not actually exist in physical memory (they are referred to as empty-zero) and will be allocated only upon a real access by the software (these are the so called minor-faults in operating system terminology). Modern operating system kernels, such as Linux 2.6.18 (or later versions) expose system-calls driving the actual allocation of pages onto NUMA nodes, e.g., upon the occurrence of minor faults. Particularly, the `set_mempolicy` system-call allows to specify the rules according to which the allocation of virtual to physical memory needs to be carried out. In our design, this system-call is employed upon starting up the NUMA manager in order to post a strict binding policy so that empty-zero memory is materialized on the NUMA node associated with the CPU-core where the memory access is performed. This policy, together with the memory access rules we consider, such that the memory pages destined for keeping data (chunks) associated with a specific simulation object are only accessed by the worker thread to which the object is bound, leads the physical memory allocation to occur on the NUMA node associated with the CPU-core where the thread is running.

The above management rules well fit scenarios where the worker thread is statically bound to a specific CPU-core (such as when running with the `sched_setaffinity` service posted) and the simulation objects are statically bound to a specific worker thread. How to cope with dynamic scenarios where these constraints are not met will be discussed in Section 3.4.

3.3 Management of Data Exchange

Clearly, page-disjoint access by the worker threads to the chunks hosting data for the different simulation objects can be guaranteed for input/output event queues, for the live state of the object and for state recoverability data. However it cannot be guaranteed when we need to exchange events (or anti-events) across the different simulation object. In order to provide effective NUMA-oriented support for the exchange operation of events across different simulation objects, we provide a NUMA-oriented implementation of the scheme originally described in [37], based on the bottom-half paradigm.

In particular, when a worker thread needs to post a new event/antievent having with destination $SOBJ_k$, no direct access to the event queue of the target object is performed. In fact, in case the memory pages keeping the records of the event queue of $SOBJ_k$ were located on a different NUMA node, the access (for scanning records and correctly installing the new one) might be costly. Rather, the event is posted on an intermediate queue (the bottom-half queue of $SOBJ_k$) and the actual incorporation into the recipient event queue will be carried out by the worker thread in charge of running $SOBJ_k$ (which will access close memory while manipulating the queue). The NUMA-oriented optimization lies in that the bottom-half queue is guaranteed to be hosted by the NUMA node where the destination object (hence thread) operates.

More in detail, the `mem_map` data structure (recall that an individual instance of this data structure is associated with any simulation object) previously presented has been augmented with the additional fields:

```
void* live_bh;
int   live_msgs;
int   live_boundary;
```

where `live_bh` points to a page aligned buffer that is used as the buffer for hosting the exchanged events, `live_msgs` is the counter of currently available messages into the bottom-half buffer, and `live_boundary` is the offset (into the buffer) where free room is available for additional events to be sent towards the destination ([2]). The pages forming the bottom-half buffer are allocated again via the `mmap` system-call, hence we guarantee that the different bottom-half queues are located on disjoint sets of memory pages for the different simulation objects. Overall, all the worker threads sending an event (or anti-event) towards a specific simulation object will deliver the information on specific memory pages reserved for the bottom-half queue of the destination. On the other hand, the bottom-half pages are materialized (namely they switch off the empty-zero state) upon their first access, which can take place by whichever worker thread (running on whichever NUMA node). Hence on the basis of the employed operating system memory policies, depicted in the previous section, the bottom-half queue pages will not necessarily reside on the NUMA node where the thread hosting the destination simulation object is running. The realignment (if requested) of these pages on the correct NUMA nodes will be dealt with in Section 3.4, where we cope with the mechanisms for (dynamically) migrating pages across the different NUMA nodes.

Overall, with our architecture we guarantee that the pages forming the bottom-half buffer associated with a specific simulation object will reside (or will be migrated to) the NUMA node where the worker thread, say WT_j, managing this object is running. Hence the read operations by WT_j for reading the messages and incorporating the content into the destination object event queues will occur from close memory. This will not penalize the event-send operations by the other threads (possibly operating in different NUMA nodes) given that the NUMA architecture is not adverse to memory writes on remote NUMA nodes (thanks to the fact that writes occur into the cache) rather, they are adverse to reads (in case these are not served via cached data).

[2]The reason why we use the term *live* in the names of the data structures will be clarified later in this same section.

An additional optimization has been introduced in order not to hamper concurrency in the management of the bottom-half queue. In fact, given that multiple worker threads can concurrently send data towards the same destination simulation object, any insertion of an element in the bottom-half queue must occur in a critical section (protected by spin-locks in our implementation). The same would be true for the reads from the bottom half, in case the bottom-half buffer would be handled as a circular buffer. To avoid explicitly synchronizing read and write operations onto the bottom-half queue, we extended the bottom-half queue support by having two different bottom-half buffers. One is the live buffer (as illustrated above) which is managed via the aforementioned fields kept by the `mem_map` record. A second one is the *expired* buffer, which is a clone of the live one, in terms of implementation and data structures. The read operations occur from the expired buffer (and given that they are carried out by a single worker thread, namely the one managing the destination simulation object, they are not subject to concurrency problems), while the write operations occur into the live one. Each time a read operation is issued on an empty expired buffer, live and expired buffers are exchanged (so as to give rise to a new era of their usage), which occurs in a fast critical section, e.g., by switching the respective pointers to the actual buffers. In this way, we do not need to synchronize reads with writes (except for the start of a new era), rather only write operations need to occur in critical sections. Also, a dynamic resize mechanism is employed so that when a new era is started, and the occupancy in the bottom-half buffers exceeds a threshold percentage, then these buffers enlarged by reallocating them (still via `mmap`), which allows resizing the memory used for in transit messages across different worker threads depending on the frequency of messages arrival towards specific destinations.

3.4 The Page Migration Daemon

A core additional component complementing the above presented memory management architecture is the page migration daemon, which we refer to as `pagemigd`. This daemon, which runs as a set of CPU non-intrusive threads, is in charge of periodically moving the memory pages associated with the memory map of a specific simulation object to a target NUMA node. The work by this daemon is based on an additional data structure, which we name `map_move`, instantiated on a per-simulation object basis. This data structure has the following fields:

```
pthread_spinlock_t spinlock;
unsigned           target_node;
int                need_move;
```

where `need_move` indicates whether a request to migrate the whole memory map associated with a given simulation object has been posted, and (if posted) `target_node` indicates the target NUMA node for the move operation. The `spinlock` field is used to make the access to this data structure atomic, and to make atomic also the actual move of the memory map towards the target node (in case a move has been requested). The move request can be posted to the daemon via the function `void move_sobj(int sobj_id, unsigned target_node)`, which can be called by the worker

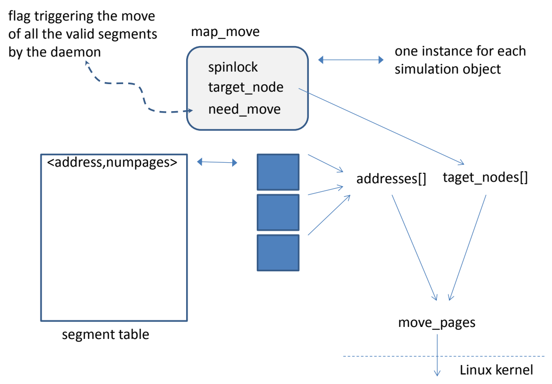

Figure 4: Segment migration operation.

threads operating with the PDES platform. In case a move request is found to be posted (i.e. the flag `need_move` is found to be raised), `pagemigd` accesses the `mem_map` data structure associated with the object to move, and migrates (one after another) all the currently active segments, namely the ones registered within the segment table of that simulation object. As already pointed out, a valid entry provides the initial address of the segment of contiguous pages to be moved, and the number of these pages. Hence, being the page size fixed on the target architecture and operating system (say 4KB), the address of any page belonging to a given segment is also known (in fact it is determined on the basis of a simple offset rule from the segment start address).

The actual move of the pages belonging to an individual segment is carried out by invoking the `move_pages` system call which is supported in Linux starting from kernel 2.6.18. Among other parameters, this system call takes as input two arrays, one is the array of virtual addresses of pages to be moved, the second one is the array telling to which NUMA nodes the pages need to be moved. Hence, the actual activities for moving the segment towards the target NUMA node are the ones schematized in Figure 4.

We underline that the support for the `move_pages` system-call in the Linux kernel logically treats the move of an individual page from a NUMA node to another one as a couple of swap-out/swap-in operations (but with no mandatory interaction with stable storage). Particularly, the page logically disappears from memory (namely from the origin node) and is then faulted-in towards the target node. In case some worker thread issues a memory access to a page that is currently being moved, then the access is treated similarly to a minor fault, hence at low cost. On the other hand, in case the access occurs after the page move has already been finalized, then no fault occurs along the execution of the worker thread. Also, empty-zero pages are ignored by the kernel, hence using the above approach leads to pay no cost for virtual memory that is actually not yet allocated in RAM.

Other aspects deserving attention in the above organization is related to migrating the pages reserved for the bottom-half queues. Recall that these pages are not registered within the segment table associated with the simulation object, rather they are directly accessible via the meta data kept by the `mem_map` record. In particular, `pagemigd` always attempts to move these pages towards the last value registered in `target_node` for a specific simulation object, independently of whether the `need_move` flag is raised. This is because, as pointed out before, these pages can be materialized as non-empty-zero on generic NUMA nodes, depending on which worker thread issued the first access to the bottom-half buffer (i.e. to any generic page of that buffer) while sending events/anti-events to the destination simulation object. Issuing the move request periodically via `pagemigd` allows the pages hosting the bottom-half queue to be promptly migrated towards the correct NUMA node (even if originally allocated on a different node). On the other hand, in case these pages are already located on the correct node, the `move_pages` system-call will simply return by ignoring the move operation, thus inducing minimal overhead for useless calls.

In our design, the operation of moving the pages hosting the bottom-half queues does not lock the `spinlock` in the `map_move` data structure, and is executed in full concurrency with the actual access to the bottom-half buffers by the worker threads. To allow safe access to the addresses of the pages forming these buffers, the `pagemigd` daemon does not use the `live_bh` and the `expired_bh` pointers in the `mem_map` data structure. In fact, these are switched (although infrequently, namely at the start of a new era for the bottom-half buffers usage) and would need to be accessed atomically for correct determination of the page address. To overcome this issue, as shown in Figure 5 these addresses are duplicated at startup and recorded in two stable pointers in the array `bh_addresses[]`. The only situations in which these pointers are changed is when dynamic resize of the bottom-half buffers is carried out (as hinted, this occurs when a threshold occupancy is reached). The resize implies that the buffers are reallocated via `mmap` so that the `bh_addresses[]` are updated. Given that the `pagemigd` daemon uses the pointers stored in `bh_addresses[]` to issue the page move request to the kernel for migrating the pages forming the bottom-half buffers towards the target NUMA node, a fast critical section (implemented via CAS instructions) is used in order to lock the content of `bh_addresses[]` so that any resize operation leads to temporary block the page move action up to the finalization of the `bh_addresses[]` content.

Overall, the `spinlock` in the `map_move` data structure is only used for (i) avoiding races when posting new move requests towards `pagemigd`, (ii) avoiding that the segment table is changed while a move is in progress. In fact, each time the upper level allocator, say DyMeLoR, pre-allocates memory (or releases) memory segments, this operation blocks the `spinlock` associated with the memory map so as to prevent the `pagemigd` daemon to scan the segment table while it is being modified.

Summarizing, the presence of `pagemigd` allows the whole NUMA-oriented architecture to:

- dynamically rebind the pages keeping the data (event buffers, live state, and recoverability information) of any simulation object to the NUMA node that is closest to the CPU-core where the worker thread managing the simulation object runs, which is useful both when the migration of an object from a worker thread to another one is performed (e.g. for load-sharing purposes [36]) and when the worker threads are switched across CPU-cores operating in different NUMA nodes;

- dynamically move the pages used for data exchange towards the NUMA node where the destination simulation object of the data is hosted (given that the

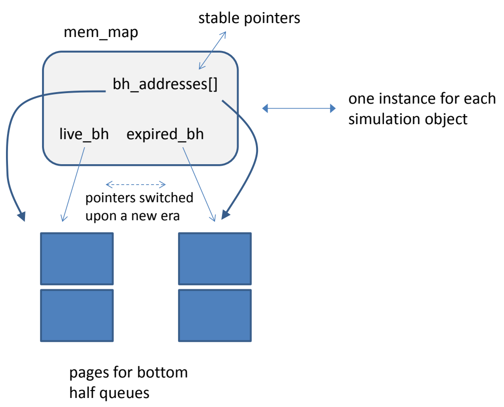
stable pointers
mem_map
bh_addresses[]
one instance for each simulation object
live_bh
expired_bh
pointers switched upon a new era
pages for bottom half queues

Figure 5: Data structures for bottom-half queues maintenance and migration.

worker thread managing it runs on a CPU-core close to that NUMA node).

As a final aspect, our pagemoved is not forced to be executed as a single non-CPU intrusive thread. Rather the number of threads `Num_pagemoved` forming the daemon (as well as their wake-up period) can be selected at startup of the NUMA memory management architecture. Each of these threads is in charge of checking the request for migration, and of actually migrating, a subset of the simulation objects (selected according to a balanced hashing scheme based on thread identifiers). The reason for this kind of load-sharing across multiple page-migration threads is twofold. First, if a move request is standing, it will more likely be observed promptly after the thread wake-up from the sleep period. Second, avoiding to perform the whole migration work along a single thread leads to avoiding the pagemigd daemon (although being CPU non-intensive) to interfere in a highly unbalanced manner with the worker threads operating in the PDES environment, rather, upon their periodic wake-up, all the pagemigd threads will lead to (more) proportionally slow down the worker threads, which allows not to increase the skew in the advancement of the simulation objects along the logical time axis [39]. Consequently, this approach likely not favors the generation of additional rollbacks due to the pagemigd daemon interfering work-load.

4. EXPERIMENTAL STUDY

As hinted, we have integrated all our solutions into the ROme OpTimistic Simulator (ROOT-Sim) [24, 22, 17], an optimistic simulation platform based on the Time Warp protocol [18] and tailored for UNIX-like systems. ROOT-Sim is designed to be a general-purpose platform, that can support any kind of discrete event model developed by adhering to a very simple and intuitive programming model (in fact, the platform transparently handles all the mechanisms associated with parallelization and synchronization). Particularly, the programmer is only requested to code the logic implementing the callback entry points to the application code, whose execution is triggered by the underlying ROOT-Sim platform along any worker thread right upon CPU-dispatching the execution of some event targeting a specific object, or when inspecting some newly committed state of the simulation object. Also the programmer can use a very simple API offered by ROOT-Sim for injecting new events in the system, destined to whichever simulation object.

The hardware architecture used for running the experiments is a 64-bit NUMA machine, namely an 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 64-bit Debian 6, with Linux Kernel version 2.6.32.5. The compiling and linking tools used are `gcc` 4.3.4 and binutils (`as` and `ld`) 2.20.0. Also, the architecture entails 8 different NUMA nodes, each one close (distance 10) to 4 CPU-cores and far (distance 20) from all the others.

As a test-bed application, we have used *traffic*. This application simulates a complex highway system (at a single car granularity), where the topology is a generic graph, in which nodes represent cities or junctions and edges represent the actual highways. Every node is described in terms of car inter-arrival time and car leaving probability, while edges are described in terms of their length. At startup phase, the simulation model is asked to distribute the highway's topology on a given number of simulation objects. Every object therefore handles the simulation of a node or a portion of a segment, the length of which depends on the total highway's length and the number of available simulation objects.

Cars enter the system according to an Erlang probability distribution, with a mean inter-arrival time specified (for each node) in the topology configuration file. They can join the highway starting from cities/junctions only, and are later directed towards highway segments with a uniform probability. In case a car, after having traversed part of the highway, enters a new junction, according to a certain probability (again specified in the configuration file) it decides whether to leave the highway. Whenever a car is received from any simulation object, it is enqueued into a list of traversing cars, and its speed (for the particular object it is entering in) is determined according to a Gaussian probability distribution, the mean and the variance of which are specified at startup time. Then, the model computes the time the car will need to traverse the node, adding traffic slowdowns which are again computed according to a Gaussian distribution. In particular, the probability of finding a traffic jam is a function of the number of cars which are currently passing through the node. A `LEAVE` event is scheduled towards the same simulation object at the computed time. Additionally, when a car is enqueued, the whole list of records associated with cars is scanned, in order to update their position in the queue, which reflects updates on the relative positions of the cars along the path they are traversing. Note that this does not involve time stepped execution of the simulation model.

Accidents are derived according to a probability function as well. In particular, they are more likely to occur when the amount of cars traversing the highway portion modeled by a simulation object is about half of the cars which can be hosted altogether. In fact, if few cars are in, accidents are less frequent. Similarly, if there are many, the traffic factor produces a speed slowdown, thus reducing the likelihood of an accident to occur. Therefore, the model discretizes a Normal distribution, computing the Cumulative Density Function in a contour defined as $\textit{cars in the node} \pm \frac{1}{2}$, having as the mean half of the total number of cars which are at the current moment in the system, and as the variance a factor which can be specified at startup. The total number of cars which can be hosted by a simulation object is computed according to the actual length of the simulated road, which is determined when the model is initialized. When an accident occurs, the cars are not allowed to leave the path portion modeled by the corresponding simulation object, until the road is freed. In fact, if a `LEAVE` event is received, but its execution is associated with a car involved in an accident, the record associated with the car is not removed from the queue. Rather, its leave time is updated according to the accident's durations, and a new `LEAVE` event is scheduled. The duration of an accident phase is determined according to a Gaussian distribution, the mean and the variance of which are again specified at startup.

During the scan of the queue entries, with a certain small probability (specified at startup), a car decides to stop for a certain amount of time (e.g. for fuel recharge). This is reflected by setting a special flag in the record, and a duration for the stop is drawn from a Gaussian distribution. In this case, if a `LEAVE` event is received associated with a stopped car, the behavior of the model is the same as in the case of an accident. During a queue scan, if a stopped car expires its stop time, the relevant flag is reset, so that the next `LEAVE` event will allow it to exit from the path portion modeled by the current simulation object.

In our execution, we have simulated the whole Italian highway network. We have discarded the highways segments in the islands in order to simulate an undirected connected graph, which allows to have the actual workload migrating overall the highway. The topology has been derived from [4], and the traffic parameters have been tuned according to the measurements provided in [5]. The average speed has been set to 110 Km/h, with a variance of 20 Km/h, and accident durations have been set to 1 hour, with 30 minutes variance. This model has provided results which are statistically close to the real measurements provided in [1]. Overall, the used application is a real world one, that we have produced in cooperation with colleagues from Logistic Engineering such in a way to provide support to decision making processes (such as scheduling of delivery services across the country). We consider this application benchmark to be significant for showing how our proposed NUMA memory management architecture is able to capture the differentiated memory access patterns, and react via proper migration of memory pages towards the most access-intensive NUMA nodes.

For this benchmark application, we report performance data when running according to differentiated modes. In one mode, the Italian highway has been modeled via 137 simulation objects (individually handling up to 130 Km) in a configuration fairly balanced and stable in terms of workload by the different simulation objects, which is achieved by having the traffic parameters set to simulate scenarios where jams due to, e.g., accidents are very infrequent. This configuration does not require dynamic re-mapping of the simulation objects across the 32 worker threads operating within the ROOT-Sim platform in order to achieve competitive parallel executions[3]. For this balanced case, we compare

[3]In all the experiments, the worker threads are run with CPU-affinity setup, which is the classical approach for avoiding cross CPU-core migration of highly CPU intensive applications, which may induce overhead. As a reflection, any worker thread constantly operates on a specific NUMA node.

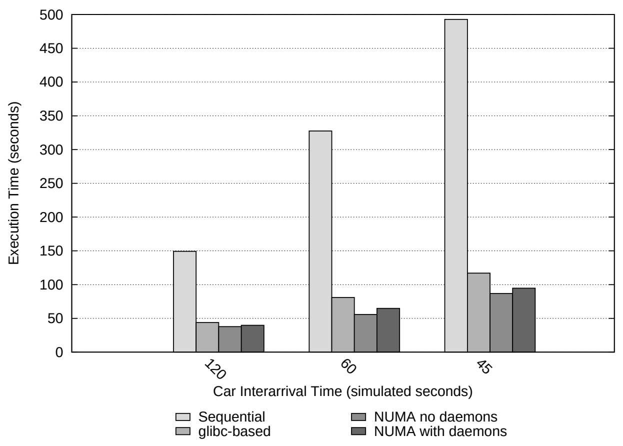

Figure 6: Balanced configuration: total execution time.

the execution time achieved with (i) a traditional sequential execution (carried out on top of a properly-optimized scheduler based on a Calendar-Queue [6]), (ii) a parallel execution relying on memory management based on the standard `glibc` library, and (iii) a parallel execution relying on our NUMA memory management architecture. The sequential execution is a baseline one, which allows us to verify the level of speedup achievable via parallel runs. On the other hand, the `glibc`-based parallel run is a reference one allowing us to assess the effectiveness of the ad-hoc NUMA-oriented memory management support for Time Warp (when compared with standard memory allocation and management facilities).

In Figure 6, we provide the execution times for such balanced configuration of the *traffic* application, while varying the car inter-arrival time. Specifically, we have set all the source nodes (namely, all the junctions) in the highway topology to have a inter-arrival time set to 120 (less loaded), to 60, and to 45 (more loaded), respectively. Also, for the NUMA-oriented memory management architecture, we have considered two different settings. In one we do not activate `pagemoved` daemons. In the other one we set the number of `pagemoved` daemons to 32 (one per each CPU-core), and we set an aggressive activation period for them of one second. This produces a high interference with the actual simulation work carried out by the worker threads, and furthermore it provides no actual benefit, as the number of page migrations required is extremely low, due to the balanced nature of the workload. Overall, we consider a kind of worst case configuration for the parallel runs given the relatively reduced workload (in terms of cars to be managed per virtual time unit), which tends not to favor speedup by parallel executions, and also leads to reduced efficiency in optimistic synchronization, given the fine granularity of the events. The worst case profile for NUMA is further related to interference by the `pagemigd` daemons (in case they are activated).

By the results, we see that the parallel runs provide anyhow speedup vs the sequential execution, which increases while increasing the workload. On the other hand, the performance achieved by our NUMA-oriented memory manager is up to 27% better than the one of the `glibc` configuration in case no `pagemigd` daemons are activated, and up to 20% better in case these daemons are activated. This is a rele-

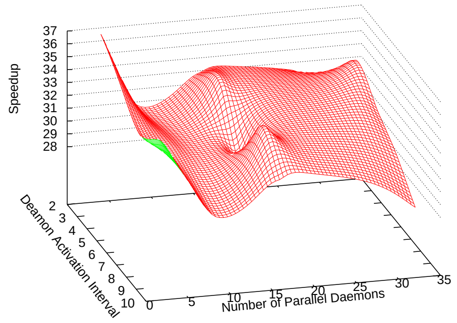

Figure 7: Unbalanced configuration: speedup of the NUMA memory manager vs sequential execution.

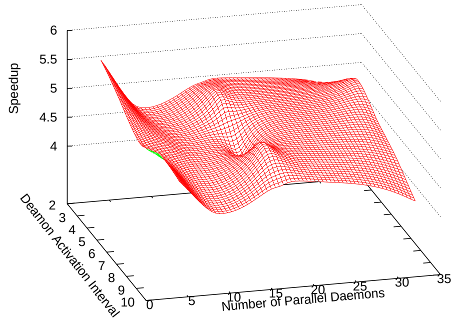

Figure 8: Unbalanced configuration: speedup of the NUMA memory manager with daemons vs with no-daemon.

vant achievement when considering that the maximum efficiency (namely, the percentage of work that is not eventually rolled back) which has been observed in the optimistic parallel runs is on the order of 65% (this is achieved when the car inter-arrival time is set to 45 simulated seconds, which gives rise to an increased simulation workload compared to the other configurations, with consequent increase of the average event granularity), which means that the ROOT-Sim Time Warp system is actually executing significant volumes of memory operations (allocation/deallocation and accesses) for either forward and rollback modes, which allows assessing the dynamics of our NUMA-oriented memory manager (vs glibc) according to a relatively wide spectrum of memory related activities. Also, the actual rollback pattern seems not to be significantly influenced by the interference caused by the pagemigd daemons, given that they lead to a reduction in performance (compared to the NUMA-oriented memory manager with no daemons) which is upper bounded by 10%.

In Figures 7-9 we report results related to a different configuration of the *traffic* application, where the volume of cars managed is greater (particularly, the car inter-arrival frequency is derived from [5]), and where we consequently get the possibility of accidents to occur according to the aforementioned statistical distribution. This gives rise to dynamic unbalance of the workload across the simulation objects, such that the dynamic re-balance of the actual workload on the computing platform is actuated by ROOT-Sim according to the simulation-object-to-worker-thread migration policy presented in [35]. Also, for this scenario we improve the level of granularity of the representation of the Italian highway by increasing the total number of simulation objects to 1024 (hence each object simulates a shorter potion of the highway, which allows for a finer grain representation of the –relative– movement of vehicles). Therefore, this configuration allows us to assess whether the page migration facility embedded in our NUMA memory manager is able to promptly respond to variations of the workload and to the rebinding of the simulation objects towards the worker threads (hence to the variations of the memory access pattern by these threads). Additionally, in order to study how the pagemigd daemons' activation frequency and interference affects the overall execution, we have varied the number of parallel daemons in the interval $[4, 32]$, and the activation frequency in the interval $[2, 10]$ seconds.

By the results we see that the speedup achieved by the parallel run based our NUMA memory management architecture vs the sequential run (see Figure 7) is up to the order of 37 (this is achieved with a few pagemoved daemons, activated relatively frequently), and is at least on the order of 28/30 (for any other suboptimal configuration of the daemons, in terms of their number of instances and activation frequency). These higher speedup values, as compared to the balanced configuration of *traffic* are clearly due to the higher granularity of simulation events, which comes from the higher workload of cars that are simulated. As for the peak speedup configuration, we might expect it to appear in the actual observed point given that, as hinted before, accidents duration is set to the average value of 1 simulated hour, which leads to the scenario where each new imbalance persists for a limited amount of wall-clock-time. Hence frequent activation of even a few page move daemons represents a configuration that is able to promptly react to the re-balance phase and to promptly put in place a new RAM optimized placement of virtual memory pages. Also, running with pagemigd daemons allows the NUMA architecture to achieve up to 5.5 speedup compared to the case where the daemons are not activated (see Figure 8), which is clearly due to the fact that the migration of simulation objects for re-balance purposes is not fully complemented by the migration of the associated virtual pages towards the correct NUMA node in case of daemons' exclusion. Finally, the NUMA management architecture achieves up to 7x performance improvements compared to the case where the glibc library is used in combination with simulation object migrations across the worker threads as supported by ROOT-Sim (see Figure 9). In fact, the glibc memory manager exhibits a twofold performance loss vs the NUMA-oriented memory manager we have presented: (1) is does not ensure optimized NUMA access (even for balanced workload, as we already observed from data in Figure 6) – (2) it does not allow to move pages across NUMA nodes in order to follow the migration of simulation objects across the different worker threads (in their turn operating in the different NUMA zones). Both the aspects are dealt with by our NUMA memory management architecture, with relevant impact on performance improvements.

5. CONCLUSIONS

In this article we have presented a fully featured NUMA-oriented memory management architecture to be employed in Time Warp platforms running on top of multi-core machines. The architecture is based on NUMA specific allocation facilities that allow all the virtual pages destined to store application or platform level data bound to different

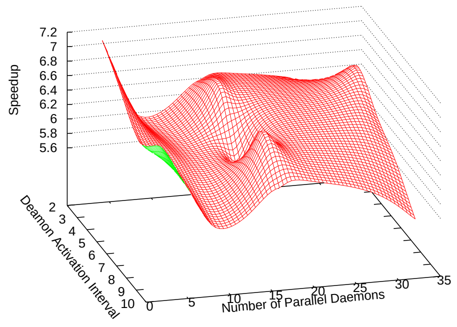

Figure 9: Unbalanced configuration: speedup of the NUMA memory manager with daemons vs glibc.

simulation objects (such as the object live state, its recoverability data and its events) to belong to disjoint sets of pages. Also, the pages associated with each simulation object are recorded onto proper memory maps that are also managed by daemons that support dynamic page migration towards the correct NUMA zone any time this might be requested for performance optimization purposes (such has when migrating the simulation objects across different worker threads for load balancing purposes). The NUMA memory management architecture has been released as free software and integrated in a reference open source Time Warp platform. Further, experimental data are reported in this article demonstrating its effectiveness for the case of a real word simulation application (particularly, a vehicular network simulation application) run in top of an off-the-shelf 32 CPU-core Linux machine equipped with 64 GB of RAM.

6. REFERENCES

[1] ACI. Dati e statistiche. *http://www.aci.it/?id=54* (last accessed: May 12, 2015).

[2] I. F. Akyildiz, L. Chen, S. R. Das, R. Fujimoto, and R. F. Serfozo. Performance analysis of "Time Warp" with limited memory. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Newport, Rhode Island, USA, June 1-5, 1992*, pages 213–224, ACM Press, 1992.

[3] I. F. Akyildiz, L. Chen, S. R. Das, R. Fujimoto, and R. F. Serfozo. The effect of memory capacity on Time Warp performance. *Journal of Parallel and Distributed Computing*, 18(4):411–422, 1993.

[4] AUTOMAP. Atlante stradale italia. *http://www.automap.it/* (last accessed: May 12, 2015).

[5] Autostrade per L'Italia S.p.A. Reportistica sul traffico. *http://www.autostrade.it/it/la-nostra-rete/dati-traffico/reportistica-sul-traffico* (last accessed: May 12, 2015).

[6] R. Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[7] C. D. Carothers, D. W. Bauer, and S. Pearce. ROSS: A high-performance, low-memory, modular Time Warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[8] C. D. Carothers, K. S. Perumalla, and R. Fujimoto. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Proceedings of the Winter Simulation Conference (WSC), Phoenix, AZ, USA, December 5-8, 1999*, pages 1624–1633, SCS, 1999.

[9] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, july 1999.

[10] M. Chetlur and P. A. Wilsey. Causality information and fossil collection in Time Warp simulations. In *Proceedings of the Winter Simulation Conference (WSC), Monterey, California, USA, December 3-6, 2006*, pages 987–994, SCS, 2006.

[11] S. R. Das and R. Fujimoto. An adaptive memory management protocol for Time Warp simulation. In *Proceesings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Vanderbilt University, Nashville, Tennessee, USA, May 16-20*, pages 201–210, ACM Press, 1994.

[12] S. R. Das and R. M. Fujimoto. A performance study of the cancelback protocol for Time Warp. *ACM SIGSIM Simulation Digest*, 23(1):135–142, ACM Press, 1993.

[13] 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.

[14] S. R. Das and R. M. Fujimoto. An empirical evaluation of performance-memory trade-offs in Time Warp. *IEEE Transactions on Parallel and Distributed Systems*, 8(2):210–224, 1997.

[15] R. M. Fujimoto and K. S. Panesar. Buffer management in shared-memory time warp systems. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation (PADS), Lake Placid, New York, USA, June 14-16, 1995*, pages 149–156, IEEE Computer Society, 1995.

[16] S. Hirve, R. Palmieri, and B. Ravindran. Archie: a speculative replicated transactional system. In *Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014*, pages 265–276, ACM Press, 2014.

[17] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. *http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/* (last accessed: May 12, 2015).

[18] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

[19] D. R. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing (PODC), Quebec City, Quebec, Canada, August 22-24*, pages 75–89. ACM Press, 1990.

[20] P. D. B. Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: executing Time Warp on 1,966,080 cores. In *Procesing of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (SIGSIM-PADS), Montreal, QC, Canada, May 19-22, 2013*, pages 327–336, ACM Press, 2013.

[21] Y.-B. Lin and B. R. Preiss. Optimal memory management for Time Warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, 1991.

[22] A. Pellegrini and F. Quaglia. The ROme OpTimistic Simulator: A tutorial (invited tutorial). In *Proceedings of the 1st Workshop on Parallel and Distributed Agent-Based Simulations (PADABS), Aachen, Germany, August 26-27*, LNCS, Springer-Verlag, pages 501–512, 2013.

[23] A. Pellegrini and F. Quaglia. Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS), Denver, CO, USA, May 18-21*, pages 105–116. ACM Press, May 2014.

[24] A. Pellegrini, R. Vitali, and F. Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools), Barcelona, Spain, March 22-24*, pages 96–98, ICST, 2011.

[25] A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems (preprint)*, May 2014, doi:10.1109/TPDS.2014.2323967.

[26] B. R. Preiss and W. M. Loucks. Memory management techniques for Time Warp on a distributed memory machine. In *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation (PADS), Lake Placid, New York, USA, June 14-16*, pages 30–39, IEEE Computer Society, 1995.

[27] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

[28] F. Quaglia. A cost model for selecting checkpoint positions in Time Warp parallel simulation. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):346–362, 2001.

[29] F. Quaglia and A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[30] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. On speculative replication of transactional systems. *Journal of Computer and System Sciences*, 80(1):257–276, 2014.

[31] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation (PADS), Philadelphia, PA, USA, May 22-24*, pages 70–77. IEEE Computer Society, 1996.

[32] B. P. Swenson and G. F. Riley. A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation (PADS), Zhangjiajie, China, July 15-19, 2012*, IEEE Computer Society, pages 44–52, 2012.

[33] R. Toccaceli and F. Quaglia. DyMeLoR: Dynamic Memory Logger and Restorer library for optimistic simulation objects with generic memory layout. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), Roma, Italy, June 3-6, 2008*, pages 163–172. IEEE Computer Society, 2008.

[34] R. Vitali, A. Pellegrini, and G. Cerasuolo. Cache-aware memory manager for optimistic simulations. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTOOLS) Sirmione-Desenzano, Italy, March 19-23, 2012*, pages 129–138, ICST, 2012.

[35] R. Vitali, A. Pellegrini, and F. Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing (HiPC), Pune, India, December 18-22, 2012*, pages 1–10. IEEE Computer Society, 2012.

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

[37] R. Vitali, A. Pellegrini, and F. Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS), Zhangjiajie, China, July 15-19, 2012*, pages 211–220, IEEE Computer Society, 2012.

[38] J. Wang, N. B. Abu-Ghazaleh, and D. Ponomarev. Interference resilient PDES on multi-core systems: towards proportional slowdown. In *Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, (SIGSIM-PADS), Montreal, QC, Canada, May 19-22, 2013*, pages 115–126, ACM Press, 2013.

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

[40] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the Workshop on Parallel and Distributed (PADS), Philadelphia, PA, USA, May 22-24, 1996*, pages 78–85, IEEE Computer Society, 1996.