Cache-Aware Memory Manager for Optimistic Simulations

Roberto Vitali vitali@dis.uniroma1.it
Alessandro Pellegrini pellegrini@dis.uniroma1.it
Gionata Cerasuolo

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza, Università di Roma

ABSTRACT

Parallel Discrete Event Simulation is a well known technique for executing complex general-purpose simulations where models are described as objects (called Logical Processes) the interaction of which is expressed through the generation of impulsive events. In particular, Optimistic Simulation allows full exploitation of the available computational power, avoiding the need to compute safety properties for the events to be executed. Optimistic Simulation platforms internally rely on several data structures, which are meant to support operations aimed at ensuring correctness, inter-kernel communication and/or event scheduling. These housekeeping and management operations access them according to complex patterns, commonly suffering from misuse of memory caching architectures. In particular, operations like log/restore access data structures on a periodic basis, producing the replacement of in-cache buffers related to the actual working set of the application logic, producing a non-negligible performance drop.

In this work we propose generally-applicable design principles for a new memory management subsystem targeted at Optimistic Simulation platforms which can face this issue by wisely allocating memory buffers depending on their actual future access patterns, in order to enhance event-execution memory locality. Additionally, an application-transparent implementation within ROOT-Sim, an open-source general-purpose optimistic simulation platform, is presented along with experimental results testing our proposal on a special-purpose version of the phold synthetic benchmark.

Categories and Subject Descriptors

D.1.3 [**Program-ming Techniques**]: Concurrent Programming—*Distributed Programming, Parallel Programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance Measures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*Parallelism and Concurrency*; I.6.8 [**Simulation And Modeling**]: Types of Simulation—*Discrete Event, Parallel*

SIMUTools 2012 Sirmione – Desenzano del Garda, Italy

General Terms

Algorithms, Performance, Measurement

Keywords

Parallel Discrete Event Simulation, State Recoverability, Dynamic Memory Allocation, Cache Locality, ROOT-Sim

1. INTRODUCTION

Parallel Discrete Event Simulation (PDES) techniques are well known for being a classical means to develop simulation systems featuring high performance, which is essential in several contexts, such as symbiotic systems or simulation-based (time-critical) decision making. The core principle is to partition the simulation model into several distinct objects, also known as Logical Processes (LPs) [9], which concurrently execute simulation events on clusters, SMP/multi-core machines and/or even desktop grids [14].

The main problem in the design/development of this type of simulation platforms is synchronization, the goal of which is to ensure causally-consistent execution of simulation events at each concurrent LP [8]. In literature, several synchronization protocols have been proposed, among which the optimism-oriented ones (e.g. the Time Warp protocol [11]) are highly promising. With these protocols, block-until-safe policies for event processing are avoided, thus allowing speculative computation, and causal consistency is guaranteed through rollback/recovery techniques, which restore the system in a correct state upon the a-posteriori detection of consistency violations. This approach has been shown to exhibit a performance which is relatively independent of both the lookahead of the simulation model and the communication latency between the concurrently running LPs. It is therefore viable and effective for a wide spectrum of both application-specific and infrastructure-related settings.

Rollback/recovery techniques have been supported using two different approaches, namely reverse computation [5] and log/restore facilities [7]. The former is based on inverse events implementing undo operations, which allow simulation platforms to recompute a previous state having not to rely on a history. The latter periodically saves in a separate memory area a complete (or incremental) snapshot of each LP's simulation state so that, whenever the simulation must be restarted from a previous Logical Virtual Time (LVT) because of executed-event time inconsistencies, older correct states can be retrieved from a log chain. Memory buffers containing simulation states' snapshots are thus necessary for ensuring execution correctness, but the probability they will be actually used is directly proportional to the rollback frequency (i.e., they have a smaller locality). A standard memory allocator would assign buffers according to some

policy the purpose of which is not to control the way they will be handled by caching architectures, an aspect which we consider to be important in order to enhance the overall throughput of the system. In fact, in-cache buffers replacement can happen with a uniform probability when they are allocated according to the policies the allocators are commonly built upon, yet replacing an access-intensive buffer with a less-used one can produce a non-negligible performance drop.

At the same time, Optimistic Simulation platforms must take into account operations meant to support efficient execution of the simulation model, like releasing buffers related to older state logs which are no longer needed in any rollback operation, in order to recovery memory (namely, the fossil collection operation). Other support operations are needed as well, e.g. selecting the next event to be executed according to some scheduling policy. These housekeeping and management operations are not directly related to simulation advancement, but are nevertheless necessary in order to produce correct advancing simulations. Again, they are likely to access data structures sparsely, inducing the in-cache replacement of access-intensive data structures by memory buffers the access pattern of which can be, in the worst case, one-shot.

In addition, the current architectural trend shows us that in the upcoming future computational frameworks based on large clusters will be available in the wide. This scale up in the number of available processing units will produce more efficient executions due to a tighter match between the number of Logical Processes and the number of computational resources. At the same time, processing units are showing an always increasing size of caching subsystems. Summing up these two factors, Optimistic Simulation platforms can enhance the overall throughput by exploiting the greater amount of per-LP available cache memory, due to the growth of cache sizes and to the working set size reduction related to the increase in available computational resources, and this can be further enhanced by an aware cache usage.

In this work we address optimistic simulation throughput enhancement from a new perspective. Data structures can be divided into two major groups — access-intensive and access-mild — the former ones producing a throughput increase if they are likely to be found in the upper levels of the caching hierarchy. We therefore propose the design and the implementation of a Memory Management subsystem which is aware of the actual access patterns (i.e., if buffers will be used as an intensively or not wrt accesses) and tries to create a separation between the cache memory locations where the two groups are stored. In this way, operations involving structures which are accessed more frequently will find them in cache with a higher probability.

The remainder of this work is structured as follows. In Section 2 we discuss Related Work. Section 3 provides a preliminary discussion on problem formulation, along with generally-applicable design indications. A detailed description of our implementation is presented in Section 4. Finally, experimental data to assess the validity of our solution are presented in Section 5.

2. RELATED WORK

In optimistic simulation platforms, memory usage strongly affects the overall performance. In particular, state logs — which are one of the fundamental means used for supporting recoverability — strongly rely on memory management subsystems. Therefore, checkpoint/restore operations has been addressed in several ways and from different perspectives in literature, in order to minimize memory requirements which, in turn, can directly/indirectly affect performance.

In the context of non-incremental checkpointing, several solutions propose methodologies to define well-suited values for the checkpoint frequency, or to determine optimal positions in case of sparse checkpoints, aiming at optimizing the tradeoff between the costs of checkpoint operations and coasting forwards, the latter cost being paid when a non-checkpointed state value must be restored [7, 17, 20, 23], and in [16] a study on the effects on memory locality is provided. The work in [19] presents a software architecture which tries to reduce single checkpoint operation's latency via the exploitation of programmable DMA engines on COTS hardware for fast and (partially) non-blocking data copy between the LP state buffer and the checkpoint buffer.

In our solution we address an orthogonal issue. In fact, the aforementioned works propose methodologies and techniques aimed at reducing logical memory usage — which in turn can produce a benefit on caching architectures — while we directly address memory management trying to enhance physical memory locality in the overall execution, indirectly focusing at the same time both on log/restore facilities and on housekeeping operations.

Other proposals like the ones in [15, 25] aim at supporting dynamic memory management through preallocation of memory buffers. Our solution is architecturally similar to the one there proposed, in the sense that we support the definition of simulation states at runtime via the invocation of standard `malloc`/`free` services as well, but we additionally serve memory requests from different memory regions depending on the future buffers' access patterns, so that access-mild data buffers will not collide with access-intensive ones in the caching architecture, therefore enhancing data locality of the actual working sets of simulation platforms/application-level software.

A completely orthogonal technique for supporting consistency in case of a time-causality violation is reverse computation [5, 13], which is based on the concept of reverse event codes, that are able to associate undo operations with the ones which produced changes in the simulation states with no (or little) history requirements. Therefore, whenever a previous simulation state must be restored, inverse events are executed, until the previous logical virtual time associated with the target state is reached. Although this technique directly enhance data locality, as long as no cache invalidation is generated from allocation/usage of log buffers, it suffers from state-restore latency — which is directly proportional to the rollback length — and is not generally applicable, as events are not invertible in general. Our solution is therefore viable as well in context where log/restore facilities cannot be left out, due to the intrinsic nature of events.

Standard allocator implementations [2, 28] usually have minimizing-space-by-minimizing-wastage (generally due to fragmentation) as a primary goal, but explicitly address fragmentation in order to increase locality, meaning that memory chunks are kept as contiguous as possible, in order to reduce page faults and cache misses. We have designed our proposal starting from a different and orthogonal perspective, explicitly addressing physical memory usage in terms of caching-architecture exploitation efficiency, focusing on the in-cache permanence of working-set-related data structures.

Kernel memory allocators [1, 4] have a similar goal, unless using different approaches. In fact, they allocate stripe-aligned memory buffers, and data structures tend to avoid false-cache sharing explicitly. This can be effectively done given the special-purposness of kernel implementations, while we address this problem in general-purpose simulation envi-

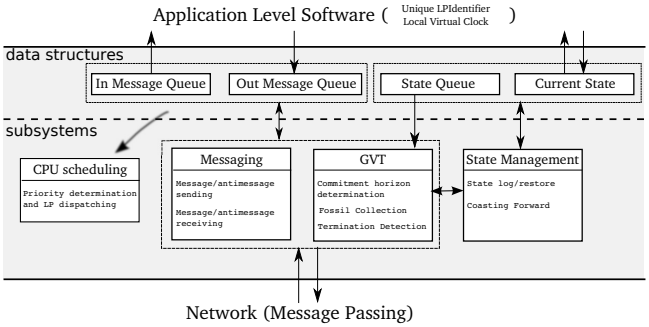

Figure 1: Reference Optimistic Simulation Architecture

ronments, where false-cache sharing is addressed through an implicit forced logical-addresses collision on the same cache stripes depending on buffers' access patterns.

3. PROBLEM FORMULATION

3.1 Optimistic Simulation Systems' Fundamentals

The optimistic PDES paradigm, as presented in [11], is based on the partitioning of the simulation model into N LPs, uniquely identified by a numerical code in the range $[0, N-1]$, which are mapped onto K simulation kernel instances. LPs handle disjoint individual states — the global simulation state just results as the union of such individual states (i.e., no shared portions are allowed) — according to the simulation model implemented at application level.

Beyond discussing basic principles underlying the optimistic paradigm, the seminal paper in [11] also provides a reference architectural organization for optimistic simulation systems, which we schematize in Figure 1. Specifically, we detail the suited set of data structures and functionalities/subsystems which should be provided in order to implement a platform relying on the optimistic paradigm[1].

Input and output message queues are used to keep track of simulation events exchanged across LPs, or scheduled by an LP for itself. They are typically separated for different LPs, so to afford management costs. For the input queues, these costs are related to both even insertions and, e.g., event move from the past (already processed) part to the future (not yet processed) in case of rollback of a specific LP. The input queue is sorted by message (event) timestamps, while the output queue is sorted by virtual send-time, which corresponds to the local virtual clock of the LP upon the corresponding event-schedule operation. As discussed by several works (see, e.g., [21]), the actual implementation of input queues can be differentiated (e.g. heaps vs calendar queues), and possibly tailored to and/or optimized for specific application contexts, characterized by proper event-timestamp patterns (affecting the insertion cost depending on the algorithm used to manage the queue). On the other hand, output queues are typically implemented as doubly-linked lists since insertions occur only at the tail (i.e. according to non-decreasing values of the local virtual clock). Also, deletions from the output queues only occur either at the tail or at the head, the former occuring upon a rollback operation which undoes the latest computed portion of the simulation at each LP. In particular, all the output messages (i.e. the scheduled events) at the tail of the output-queue with send-time greater than the logical time associated with the causality violation are marked, sent out towards the original destination in the form of anti-messages — used to annihilate previously sent message and inform the original sender of the occurred rollback[2] — and then removed from the output-queue. The latter are related to memory recovery procedures, which we shall detail later on in this section.

A *messaging subsystem* receives incoming messages from other simulation kernel instances, the content of which will be then reflected within the input queue of the destination LP. Also, it notifies output messages (i.e. newly scheduled events) to LPs hosted by other kernel instances, or the aforementioned anti-messages.

The *state queue* is the fundamental means for allowing a correct restore of the LP state to a previous snapshot whenever a causal inconsistency is detected (i.e. the LP receives a message with timestamp lower than its current simulation clock, or an anti-message that annihilates an already processed event)[3]. The state queue is handled by the *state management subsystem*, the role of which is to save/restore state images. Additional tasks by this subsystem are related to (i) performing rollback operations (i.e., determining what is the most recent suited state which has to be restored from the log), (ii) performing coasting forward operations (i.e., fictitious reprocessing of intermediate events in between the restored log and the point of the causality violation) and (iii) performing fossil-collection operations (i.e., memory recovery, by getting rid of all the events and states logs which belong to an already committed portion of the simulation).

The *Global Virtual Time (GVT) subsystem* accesses the message queues and the messaging subsystem in order to periodically perform a global reduction aimed at computing the new value for the commit horizon of the simulation, namely the time barrier currently separating the set of committed events from the ones which can still be subject to a rollback. This barrier corresponds to the minimum timestamp of not yet processed or in-transit events. In addition, this subsystem cares about termination detection, by either checking whether the new GVT oversteps a given predetermined value, or by verifying some (global) predicate (evaluated over committed state snapshots) which tells whether the conditions for termination model execution are met. Finally, this subsystem is also in charge of starting the aforementioned fossil collection procedure.

Additionally, a CPU-scheduling approach is used to determine which among the LPs hosted on by a given kernel instance must take control for actual event processing activities. Among several proposals [18, 24], the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [12], which selects the LP whose pending next-event has the minimum timestamp, compared to pending next-events of the other LPs hosted by the same kernel. Variants for LTF exist, among which a basic (stateless) approach relies on traversing the pending next-events across the input queues of all the LPs., and a stateful approach [22], is based on reflecting variations of the priority of the LPs into the CPU-scheduler state, so that the LP with then highest priority can be determined via a query on the current CPU-scheduler state.

3.2 Design Indications for a Cache-Aware Memory Manager

[1]By *subsystem* we just mean a logical differentiation, not an execution by a separate thread/process.

[2]Upon their arrival, chained rollback can be generated in case the the events carried by the originally sent messages have already been processed by the destination LPs.

[3]We note that in case the aforementioned reverse computation technique [5] is used, state restore is not needed.

Cache misses have a profound impact on performance: A miss requires accessing data directly from main memory, an operation which is at least one order of magnitude more expensive even than the last level of the cache. Misses happen due to limited cache size, with respect to the application working set size, the latter being dependent on application purpose, algorithm and implementation.

Optimistic simulation platforms usually suffer from a non-local behaviour. Many operations, although necessary, unload from cache the most accessed data structures, producing a non-negligible performance drop. In particular, log/restore operations usually allocate and use buffers for saving the current simulation state, producing a large invalidation of in-cache buffers, but the probability that these buffers will be used in the near future is near-zero. In order to increase the portion of the working set which is resident in cache, we propose to allocate these buffers in a way such that they will collide between each other, leaving untouched cache areas which are meant to contain data structures which are likely to be used more frequently.

To this purpose, we have to discriminate between memory buffers which are access-intensive and access-mild. In order to perform this classification correctly, we have to follow through our analysis in a separate way for the application level and the simulation kernel. By the discussion in 3.1, we clearly see that the only access-intensive data structure in the simulation kernel is the *input message queue*. In fact, upon receiving a new message from whichever LP in the system (an operation which is inherently frequent) the queue must be scanned for insertion or rebalanced, according to its actual implementation. Any other data structure is commonly accessed at particular positions (e.g., the output queue, which is accessed only at head or tail) or is scanned completely yet infrequently (e.g., structures needed for GVT reduction or for state management), therefore giving them the possibility to invalidate cache buffers which can be accessed in the near future can make the system suffer from secondary effects.

As far as the application-level software is concerned, an a-priori decision is hard, given that we want to provide the user with complete transparency, and considering that an Optimistic Simulation platform should be general-purpose. Therefore, the actual access pattern of the simulation model cannot be inferred with no additional knowledge. Nevertheless, common simulation scenarios show models where simulation states mostly coincide with models' working sets. Considering that event execution relies only on the simulation state to produce advancements in the simulation, we can assume that the whole simulation state is composed by access-intensive buffers.

This might seem a strong assumption, but in fact it provides no performance decrease even when the application shows a completely non-local behaviour. In fact, if we suppose that a simulation model has no locality at all in its memory accesses, then cache usage will become similar to the one shown by an application which makes no assumptions at all, therefore exhibiting a performance which we expect to be similar to the one of a simulation not using our Memory Manager, provided that most operations in the simulation kernel have a large (sparsely/infrequently-accessed) working set as well, and are therefore cache-miss prone.

In order to increase the in-cache resident set, we propose a Memory Management subsystem (oriented at supporting log/restore facilities) which partition the cache between access-intensive and access-mild memory buffers, trying to embank the cache-miss phenomenon proper of Optimistic Simulation platforms. Basically, our proposal preallocates a cache-aligned portion of the available address space and serves memory requests in a differentiated way depending on their expected access rate. In particular, a cache-aligned preallocated memory region (which we call a `stock`) is divided into portions (which we call `blocks`) the size of which corresponds to the one of the lowest-level cache available in the system. Memory buffer requests are served from fixed-size chunks within memory blocks, and depending on their access patterns, they are clustered in a way such that they will be mapped to separate cache regions. Following this policy, whenever the execution undergoes a housekeeping or management operation, memory accesses will not invalidate application-level data related to the actual simulation working set in the caching architecture, thus providing an enhancement in event execution data locality.

In order to reduce internal fragmentation, chunk size contained into a memory block is determined at runtime upon receiving a request. In this way, a memory block can contain chunks of different sizes for the application and the kernel levels. This is important because usually kernel- and application-level memory requirements are different, simulation models requiring smaller buffers and simulation kernels requiring larger ones. If a memory block were to contain same-sized chunks for both layers, internal fragmentation could arise, since blocks containing application-level chunks of a certain size would not likely contain any kernel-level chunk of the same size, and vice versa.

Of course, in order to enhance even more cache locality, memory chunks should be allocated in a cache-stripe aligned way. This choice allows the underlying hardware architecture to reduce the number of cache misses and, whenever one is encountered, the number of memory chunks replaced in cache is minimum, as a chunk is overlapped to more than one stripe only if its size is greater than the stripe's. We note that if the whole preallocated memory is cache-aligned, then this behaviour can be easily obtained by fine tuning chunks' sizes as powers of 2. Given that the cache is partitioned between access-intensive and -mild buffers, we note that to avoid interfering with the alignment, the separation between the two regions must be stripe-aligned as well.

The last general consideration we want to point out is concerned about the transient behaviour at the beginning of the simulation execution, which can in turn affect the entire simulation as well (in the context of multicore cache sharing). In fact, considering that simulation kernels are handling memory addresses separately (i.e., there is no runtime agreement between different kernel instances on how the memory should be allocated), at the beginning of the execution, different simulation kernel instances will start allocating buffers which will be mapped to the same cache locations, as long as the memory manager's behaviour is deterministic. If the application-level execution pattern is likely to allocate memory during the whole execution (i.e., the simulation state can grow indefinitely) and operate uniformly at random on it, this transient behaviour will produce an increase in cache invalidations in the initial part of the simulation. On the other hand, if the application level is likely to allocate the whole simulation state at simulation startup (i.e., an operational behaviour where the simulation state is non-growing), this cache conflict related to allocation determinism will produce a bias in cache exploitation which will produce a performance much smaller than the one generated by a common allocator which tries not to optimize with respect to the caching architecture, due to different kernel instances' (i.e., processes') buffers being conflicting during the whole simulation, and therefore producing a large

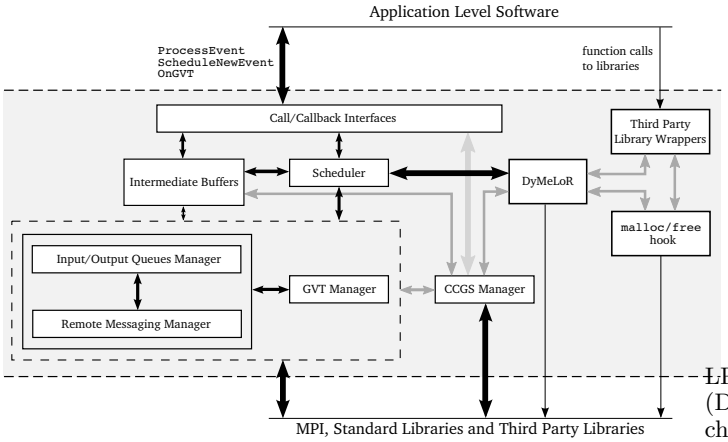

Figure 2: ROOT-Sim Architecture

number of cache misses. This problem can be faced by forcing the memory manager to start serving memory requests starting from different memory addresses according to the actual simulation kernel instance it is running within, following some circular allocation policy. Therefore, if memory buffers start to conflict, the application has allocated a larger amount of memory, and thus the probability that the working set is greater increases, producing a reduced number of cache-miss anomalies, under the assumption of uniformly-at-random accesses.

4. IMPLEMENTATION WITHIN ROOT-SIM

We have implemented the cache-aware memory management subsystem within the ROme OpTimistic Simulator (ROOT-Sim), which is an open-source, general-purpose simulation platform developed using C/POSIX technology, based on a simulation kernel layer that ultimately relies on MPI for data exchange across different kernel instances. The platform transparently supports all the mechanisms associated with parallelization (e.g. mapping of simulation objects on different kernel instances) and optimistic processing. In Figure 2, the internal architecture of ROOT-Sim is sketched.

The platform API exposed to the application programmer is quite simple, consisting of one service — namely `ScheduleNewEvent()` — and two callback services to be implemented by the application layer — `ProcessEvent()` and `OnGVT()`. Execution semantics associated with this API are as follows:

- `ScheduleNewEvent()` allows injecting a new simulation event within the system, to be destined to whichever simulation object, either locally hosted by the same kernel instance, or by a different kernel instance. This service allows specifying the destination object, simply identified via a numerical code, the timestamp for the event to be scheduled, and the event payload (as a flat sequence of bytes).

- `ProcessEvent()` supports the actual processing of simulation events. This callback is used by the kernel to give control to the application layer, in particular to a specific simulation object, in order to execute a single simulation event. The identity of the dispatched object is specified by the numerical code uniquely identifying the object within the system, which is passed as input parameter to the callback together with the payload of the event, and the event timestamp.

- `OnGVT()` allows passing control to the application layer by also providing a committed snapshot of the simulation object. This facility can be used to support (periodic) audit of the object state trajectory while the simulation run goes on, giving support for distributed termination detection [6]. Each simulation object, once dispatched via this callback, can to indicate to the run-time environment that the predicate is locally verified on a consistent committed snapshot. The environment, in its turn, can halt the simulation once all the objects have provided positive indications on the predicate.

As for management and recoverability of the state of the LPs, ROOT-Sim offer a memory management subsystem (DyMeLoR [15, 25]) which takes into account two main architectural approaches. First, dynamic memory allocation and release by the application, performed via the standard `malloc` library, are hooked by the kernel and redirected to a wrapper. Second, the simulation platform is "context-aware", i.e., it has an internal state which allows distinguishing whether the current execution flow belongs to the application-level code or the platform's internals.

Our new memory management subsystem has been specifically implemented within DyMeLoR. Nevertheless, the integration into other optimistic simulation platforms can be straightforward: In fact, our memory manager relies on standard linking facilities for wrapping standard library `malloc/free` services, which allow users to leave the simulation-platform and application-level implementation mostly untouched. In fact, as described before, our memory manager relies on a `malloc` hooking facility, which allows to determine whether an invocation is issued by the application-level software or the simulation kernel. By the considerations in section 3.2, the application-level software must be associated with buffers which must be regarded as access-intensive, while simulation kernel ones must be treated as access-mild. The intrinsic context-awareness allows to perform this differentiation in a straightforward way. The only additional requirement is to mark the input message queue (which is allocated by the simulation kernel) as access-intensive. To this purpose, we provide a new API, namely `intensive_buffer(int true))` which can be used to override the invocation-context-based decision, in order to mark a particular buffer returned by a `malloc` call depending on its parameter, allowing the simulation kernel to make the memory management subsystem able to specify more information on the future access patterns of its buffers. The solution hereby proposed provides several overall benefits: i) the application-level user can rely on our subsystem in a completely transparent way, and ii) given the flexibility of the new API, specially-designed kernel-level data structures which are recognized by the developers as being access-intensive, can be marked accordingly, therefore providing a performance enhancement at kernel level as well.

The core data structure which allows the memory manager to organize data buffers and serve application/kernel requests is the `memory_block`, the structure of which is shown in Figure 3. A `memory_block` is a portion of the `stock` buffer currently being used for serving memory requests. Its size is the same as the cache one, and it is logically divided into two different regions, one for containing access-intensive buffer and one of containing access-mild ones. The dimensions of these two regions depends on the separation threshold, the position of which is tunable at compile time. We note that the position of the separation threshold is a factor which can impact the overall performance. In fact, constraining memory accesses within a reduced cache region might in-

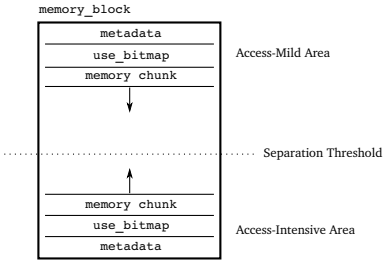

Figure 3: `memory_block` structure

crease cache-miss frequency, if an access pattern entailing reads/writes of many buffers belonging to the smaller region is followed. We will discuss the impact of this in Section 5, as specific tests have been performed in order to investigate on the impact of this choice.

Both areas in a `memory_block` maintains some header information, namely some metadata and a `use_bitmap`, the former being structured as follows:

```
struct memory_block {
    void *init_address;
    int num_chunks;
    int busy_chunks;
    size_t chunk_size;
    int next_chunk;
};
```

The `init_address` field is a pointer which is initially set to `NULL`, meaning that the current `block` has not yet been initialized for serving memory requests. In this case, we say that the `block` is currently not valid. As depicted in Figure 3, chunks are allocated stacked upon each other, with opposite growing direction depending on whether they are regarded as access-intensive or not. In particular, whenever a `memory_block` is validated because of a first `malloc` invocation belonging to that `memory_block`, the memory manager computes the number of chunks which will be available to serve requests within that `block` — this information is stored in the `num_chunks` field — and makes `init_address` pointing to the first allocable chunk.

The `use_bitmap` shown in Figure 3 is a compact data structure which allows the memory manager to fast check whether a memory chunk has already been used for serving a request. When a memory allocation is executed, the field `next_chunk` of the involved `memory_block` is used to identify the best convenient position for starting the bitmap search in order to identify a free memory chunk. At this point, `busy_chunks` counter is updated upon reservation of the identified free chunk and the corresponding bit in the bitmap is set. The manipulation of `next_chunk` is based on the classical algorithm used by the LINUX kernel for managing the bitmap of file descriptors associated with a process. Specifically, `next_chunk` is always increased upon chunk allocation within the block. Instead, it gets decreased in case a chunk is released having its index being associated with the first chunk currently available within that block (the bit associated with the released chunk is reset within the bitmap). This is a first-fit oriented policy (although acting on chunks of the same size) aimed at reducing both free-chunks and bitmap fragmentation by having allocated chunks mostly aggregated in the initial part of the block.

Additionally, we emphasize that the seeming redundancy coming from the duplicated metadata both in the access-intensive and in the access-mild area of a `memory_block` is, on the contrary, a solution to address internal fragmentation. As stated in section 3.2, application-level software is likely to issue requests for buffers the size of which is expected to be smaller that simulation kernel ones, in the general case. To cope with this possible scenario, considering the fact that most of the data structure used by the simulation kernel are access-mild, there might arise a strong internal fragmentation due to `memory_block`s holding many access-intensive buffers and other `memory_blocks` holding many access-mild buffers of different sizes. Duplicating metadata allows a single `memory_block` to host memory chunks of different sizes depending on their typology, a solution which is suitable for facing the aforementioned scenario as well as the one where buffers' sizes are more uniform.

We have explicitly chosen not to associate a header with each memory chunk in order to save on space and enhance even more locality in the access pattern, considering the scenario where logically connected data structures are not allocated sparsely wrt time execution. If this logical connection holds, then a single cache miss will make the caching architecture to load a cache stripe maybe containing more than one local data structure with a higher probability. In addition, we have decided to place metadata within the `memory_block`, as this decision allows a faster access to data structures in case a `free` operation is invoked, as the metadata are at a fixed position which can be derived by simple arithmetics from the address passed to the `free` itself, given that a `memory_stock` is cache-aligned and evenly divided into fixed-size `memory_block`s. This is another point of differentiation with the original `malloc` library, where a much more complex header is associated with each managed chunk in order to maximize flexibility in memory usage (e.g. by dynamically partitioning or aggregating chunks according to the so called "boundary tagging" scheme [2]).

Upon startup of the simulation platform, our memory management subsystems enters an auto-tuning phase. In particular, as stated above, memory stocks must be cache-aligned, and memory blocks must be cache-sized. To this purpose, the `/proc` file system is accessed, in order to obtain the actual lowest-level cache size. This information is later used to determine which is the most suitable `memory_block`'s size, so that an accurate mapping between memory buffers and cache regions can be created. In addition, this information is also used to fine tune the separation threshold accordingly, depending on the compile-time requirements specified. Cache-stripe size is determined as well, in order to insert some padding within the `memory_block`'s metadata, which is to reduce the possibility of cache conflicts between this metadata and the boundary chunks. In fact, we consider that metadata are more likely to be accessed frequently, and their access pattern is in no way related to the boundary chunks' one. This choice will further reduce the data conflict, avoiding conflicts between chunks (be they belonging to the same metadata's `memory_chunk` or not) and metadata themselves. The last part of the auto-tuning phase involves the computation of a per-kernel threshold which will force memory requests to be served starting from a particular chunk, depending on the simulation kernel the request has been issued by (we recall that, as mentioned in section 3.2, this is a solution to a possible biasing problem in shared-cache misses). In particular, the threshold is computed as $\frac{cache\_size}{num\ cores} \cdot core$, $core$ being the numerical id of the core hosting the kernel instance for execution.

Before returning control to the simulation kernel, the first `memory_stock` is allocated[4]. As mentioned before, in order

[4]A `memory_stock`'s size can be tuned at compile time.

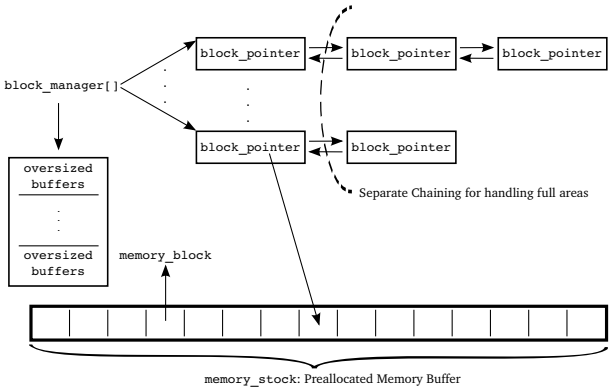

Figure 4: Memory Manager Architecture

to make our access optimizations possible, the stock must be cache-aligned. We therefore rely on `posix_memalign()`, a memory allocator which allows to specify an alignment factor, which we select as being the cache stripe size derived at startup time.

In order to maintain and manage `memory_block`s, our memory manager relies on a couple of `block_manager` data structure — which is organized as depicted in Figure 4 — one for access-intensive buffers and one for access-mild ones. Each entry in these data structures is used for managing a `memory_block`. The chunks within a block (associated with one of the two `block_managers`) have the same size, while different blocks host chunks with size corresponding to K different powers of 2. As soon as a `malloc` call is issued, the chunk size that best fits the request is identified. If no previous request for that particular size was received, the first free available `memory_block` is selected for serving requests for that particular size, and its metadata are set up. In particular, a bitmap accordingly sized to the number of available chunks is setup[5], and `init_address` is pointed to the initial memory region used for serving requests. If the `memory_block` associated to the requested size is available, but no free chunks are left over, a new `memory_block` is selected as before, and it is connected in the list associated with that particular size, kept in the `block_manager`.

In case a `memory_block` area (be it access-intensive or -mild) gets exhausted, a separate-chaining approach is used, connecting the `memory_block` entry to a not-yet-used `memory_block` which is setup accordingly. If no `memory_block` is available, a new `memory_stock` is allocated, relying again on `posix_memalign()`.

The main issue with the so-far-explained approach is that no memory request the size of which is larger than the whole cache size can be served using `memory_block`. Although we believe that common simulation kernels/applications do not rely on such large data structures, we provide two solutions to this issue. The `block_manager` has a special entry which, if a large memory request is issued, is made pointing to a special buffer which holds memory chunks according to the standard `malloc` policy. Since we note that this behaviour can deeply affect the overall performance of our new memory management subsystem — as long as these oversized buffers can produce conflicts both in access-mild and access-intensive cache areas — the memory manager can be configured at compile time in order to deny these oversized requests, thus returning `NULL` if this condition is met. The decision whether to rely on the conservative (more performing) or on the reliable (less performing) behaviour, is left to the user at compile time.

[5]We recall that a `memory_block` is fixed-size, so the number of available chunks depends on their actual size.

5. EXPERIMENTAL DATA

5.1 Test-bed Platform and Application Benchmarks

We have implemented the proposed cache-aware memory management subsystem within ROOT-Sim. The hardware architecture used for testing our proposal is a 64-bit NUMA machine, namely an HP Proliant server, equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 10MB L3 cache (5 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.

The benchmark we have used in this experimental study is derived from phold [10]. In this benchmark, each simulation object executes fictitious events which only involve the advancement of the local simulation clock to the event timestamp. Each time an event is executed, a new fictitious event is scheduled, destined to whatever object inside the system, with a timestamp increment following some exponential distribution. In implementations of this benchmark (see, e.g., [19, 27]), the execution of an event has included a busy loop (which emulates a specific CPU delay for event processing, and hence a specific event granularity) and/or read/write mode access to a fictitious, memory contiguous state buffer of a given size S. Large values for S would mimic applications with large memory requirements. On the other hand, the spanning of read/write operations across the state buffer determines the specific locality inside the object state, associated with the event execution.

We have adapted this benchmark in order to mimic different working sets resident in dynamically allocated memory. In particular, in our implementation each LP's state is formed by a main data structure (`state_head`), containing a set of N pointers for accessing N distinct linked lists of varied-size buffers relying on dynamic memory allocation. Sizes are linearly distributed in between a min and a max value We denote as $size(i)$ the exact size of the buffers inside the i-th list (with $0 \leq i \leq (N-1)$).

At setup time, the S bytes forming the fictitious state are allocated according to the following rule:

- $\frac{S}{N}$ bytes are destined for buffer allocation inside each of the N lists.
- $\lceil \frac{S}{N}/size(i) \rceil$ buffers are allocated for the i-th list, and linked together.

In other words, there is a bias towards buffers associated with smaller sizes. This is reasonable when thinking that the logic used in general software contexts tends to rely on allocation of large amounts of relatively small memory chunks, and smaller amounts of relatively large chunks[6].

In order to specifically account for access patterns, the simulation is based on an event which entails access operations inside the buffer lists currently belonging to the simulation object memory layout. This will allow us to observe whether and how the access-intensive buffers are indeed found in the caching architecture, therefore letting us

[6]As an example, this is the typical layout for dynamic memory based data structures inside the LINUX kernel.

measure the actual cache-miss frequency and how it is related with overall performance.

To determine the span of access operations across the buffers inside the object lists, we have adopted a breadth-first visit on the lists. Specifically, we have introduced an additional parameter $x \leq S$ indicating the total amount of bytes that must be accessed during the event execution across the dynamically allocated buffers within the object state. Initially x represents the residual amount of bytes to be touched. Upon event execution, we randomly select the list of buffers from which to start the visit, and we touch all the content inside the buffer at the head of this list, thus also decreasing the residual amount of bytes to be touched by subsequent access operations. Then, every other list is accessed the same way according to a circular policy. At each access on a given list, we move on the subsequent (untouched) buffer in that list (if any) and then perform the access operation. This goes on until the residual amount of bytes to be accessed becomes zero. Actually, the lower the value of x, the higher the memory locality of the application. Access operations are implemented via stdlib services, specifically `memcmp` calls, involving data movement from memory to CPU registers, and therefore forcing the caching architecture to load data if not already present.

After the startup phase, every LP schedules for itself the first event at timestamp t, derived according to an exponential distribution. Upon event execution, the access operation is performed. Then, an LP in the system is selected uniformly at random amongst all the LPs in the system, and a new event is destined to it at a simulation time $t + inc$, where inc is again derived according to the same exponential distribution. The simulation halts when all the LPs in the system have executed at least 500000 memory-access events.

5.2 Results

We have performed three main tests, which can be described as follows:

A. The simulation-state size has been tuned in order to completely fit the lower-level cache size (i.e. it is 2/3 of the whole cache size), and read/write accesses span across 75% of each LP's state. This configuration mimics a scenario where simulation models rely on a reduced state, where most of its data are necessary to follow through the execution. We consider this to be the best case for our memory management architecture, as we expect to find a reduced number of conflicts.

B. The second test case represents the most general one, where simulation states are 3 times the cache size, and accesses span 25% of each LP's state. We expect that in-cache data buffers are more frequently replaced, yet allowing some data locality to be captured.

C. The third test case mimics a worst-case scenario, related to simulation objects which rely on growing states which are sparsely accessed. In particular, we define simulation states to be much larger than the cache size (in the order of 10), and the access operations to span 10% of the whole simulation state. We expect to find an increased number of cache misses, nevertheless showing a performance which should resemble the one provided by a generic memory allocator.

The above considerations only take into account application-level buffers, not considering the kernel read-intensive buffers (e.g. input message queue). The presence of such buffers in the access-intensive cache area may add some entropy to the actual behaviour of our memory management subsystems, because data structures like the input message queue can arbitrarily grow depending on the actual application-level events' generation pattern, i.e. LPs interaction.

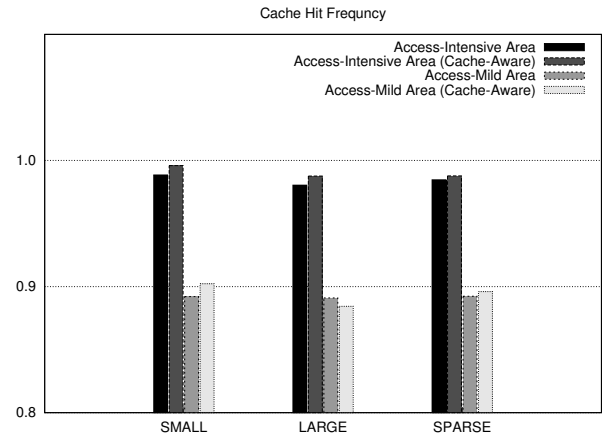

Figure 6: Cache Hit Ratios in CSS Execution

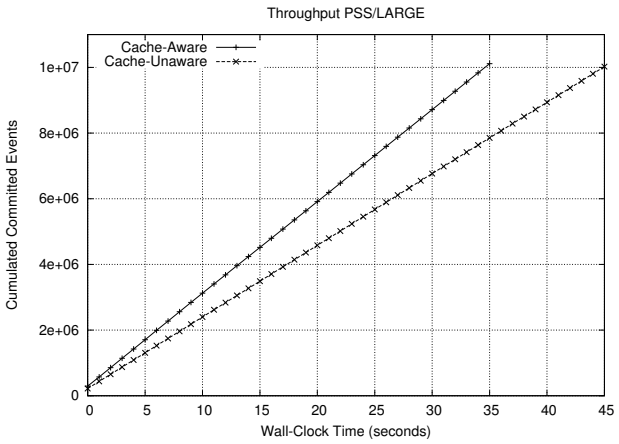

Figure 7: Throughput with less Interaction

Our simulations have been executed running 32 instances of ROOT-Sim's simulation kernel (one process on each CPU core) and 32 LPs (one for each kernel instance). We believe that this scenario can significantly describe the benefits of our memory managements architecture when there is a tighter match between processing resources and simulation objects, a scenario which will be always more present in the future, according to the current technological trends described in Section 1. We note that the aforementioned executions exhibit a large degree of parallelism, causing a strong divergence in the logical virtual time associated with the LPs hosted on different processors, inducing a high number of rollback operations because of out-of-order events' execution. This is a worst-case scenario for our memory management subsystem, as the large number of rollback operations make access-mild data buffer be scanned frequently, in order to restore a correct simulation state.

For each of the specified test cases, two different executions are provided, one taking a snapshot of the whole simulation state at each event execution (namely, a Copy State Saving behaviour, CSS), and one which saves the state on a periodic basis (namely, a Periodic State Saving behaviour, PSS) with the log period is autonomically set according to

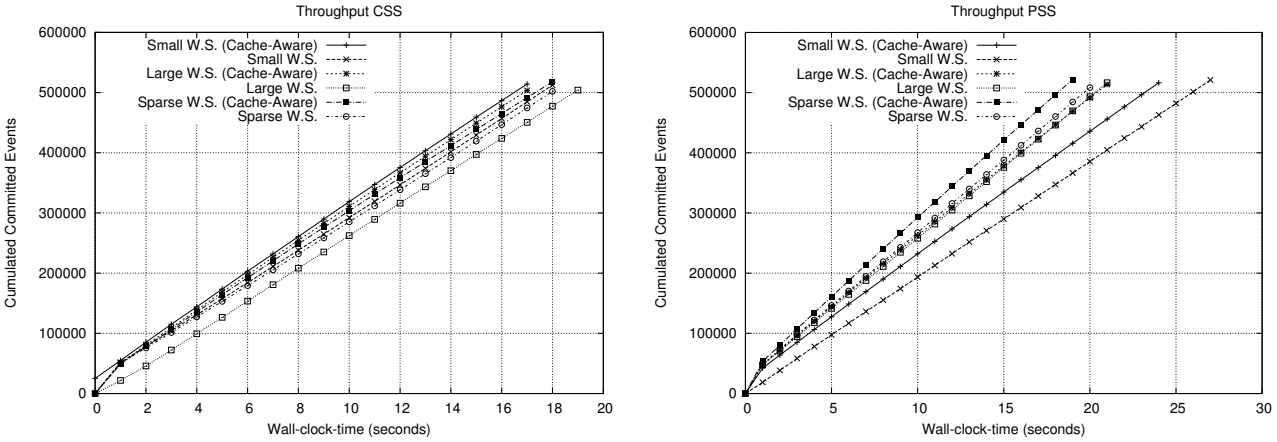

Figure 5: Throughput: High Degree of Parallelism

the proposal in [26], ensuring that the best checkpointing interval is selected according to actual execution dynamics. We consider the CSS an important test case, because it is significant for non Piece-Wise-Deterministic scenarios [3], entailing simulation events the re-execution of which (e.g. after a rollback operation) is likely not to regenerate the same events' pattern. This makes silent execution unexploitable for rebuilding a previous state starting from a certain state log, requiring the exact simulation state to be restored. Additionally, the separation threshold has been set in order to show a 1:4 ratio, meaning that the access-intensive area is four times the access-mild one.

In order to better interpret the experimental results, in Table 1 we present the per-LP memory occupancy (i.e. the amount of memory requested to the system during the whole execution) wrt access-intensive and access-mild buffer requests, in the PSS execution case. In particular, we can clearly see that the access-mild area is much larger than the access-intensive one (on the order of 6-10 times). By the definition of the separation threshold according to a 1:4 ratio, access-mild buffers are made colliding into the smaller cache area, exactly because due to their sparse access pattern, cache utilization would be sub-optimal, and therefore more importance must be given to the access-intensive ones, which represent the actual working set.

In figure 6 a histogram showing the cache-hit ratio is provided for the CSS case. By the plot it can be seen that the differentiation in the access pattern which the cache-aware subsystem is based on is actually real. In fact, the hit rate associated with the non-cache-aware execution clearly shows that there is a difference in the order of 10% between the operations' frequency on access-intensive and -mild buffers. Moveover, we recall that in this test case the rollback length is less than two (i.e. a configuration that accesses with a high probability a log buffer shortly after it has been saved) and the rollback frequency is extremely high due to an increased degree of parallelism. We note that a more general configuration would provide a smaller cache hit for access-mild buffers. To produce a better performance a higher global hit rate is required. The access ratio between access-mild (mostly kernel-related) and access-intensive (application and some kernel structures) strongly depends on the simulation model's logic, but since the access-mild buffers are typically large and sparsely accessed, a cache-hit increase on them could be infeasible, while an access-intensive cache-hit increase is more easily achievable due to the fact that application-level software is more likely to be local.

	Access-Mild	**Access-Intensive**
SMALL	62,79 MB	6,10 MB
LARGE	62,86 MB	6,95 MB
SPARSE	73,84 MB	11,20 MB

Table 1: Per-LP Memory Occupancy (PSS)

In Figure 5 we present the results of the aforementioned test executions, showing the throughput related to test A, B and C. CSS execution exhibits a performance increase that ranges in between 5% and 10%. This is because of the higher availability of in-cache buffers related to the actual working set.

In the PSS case, a different behaviour comes from the differences in working sets. In particular, we note that because of the high number of rollbacks related to the degree of parallelism in this benchmark, the coasting forward operation is more relevant to the overall performance. In particular, an increased number of events must be silently reprocessed (on average) in order to reconstruct correct states. This again entails accessing buffers within the access-intensive area, which have been already proven to exhibit a better performance due to in-cache residence of the actual working set. By the plots, we can see that when dealing with a small/sparse working set, a performance increase in the order of 10% is provided. This is again related to the in-cache residence of memory buffer and the effects of the coasting forward operations. In the case of a large working set, both the cache-aware and -unaware executions mostly show the same throughput. This is because housekeeping operations become more relevant, and (at the same time) they have a smaller cache area to rely on for accessing data.

In addition, we provide the results for an execution where 10M events are executed using the B test case configuration and the PSS checkpointing scheme, as described above. The main difference in this test lies in the fact that with a 95% probability a new event is scheduled to the same LP. This configuration induces a lower amount of rollback operations, due to a smaller interaction between simulation objects. While the aforementioned experimental results were referred to a worst-case scenario, in this opposite situation the real advantages produced by a reduced non-necessary in-cache invalidation can be shown more clearly. By the results in Figure 7, we can see that the actual system's speedup is in the order of 25%. This is related to the smaller relevance

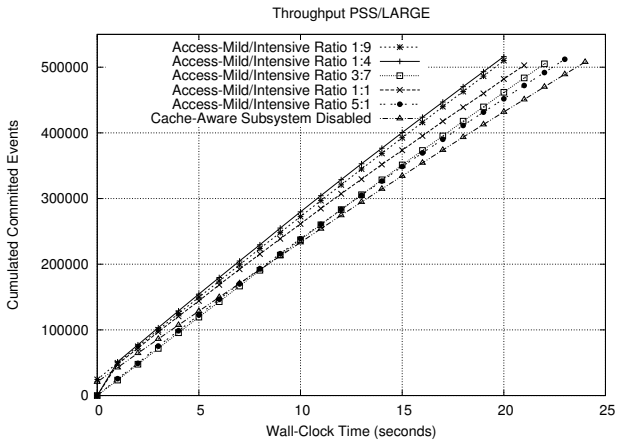

Figure 8: Test with Varied Separation Threshold

of accesses on access-mild areas, therefore assessing the performance increase related to the permanence of working-set-related buffers in the caching architecture.

Finally, we provide a test case for assessing the impact of the separation threshold on the overall performance. The importance of the separation threshold is related with the assumption that if we give access-intensive buffers more in-cache space, the total amount of cache misses is expected to be reduced because non-necessary invalidations against the actual working set are avoided. On the other hand, access-mild buffers are anyway accessed at some point during the simulation. If too little in-cache space is given to these buffers, we are unavoidably forcing an increased number of cache misses, although on a periodic basis. Festering this behaviour can, in turn, produce more secondary effects rather than avoiding them. Moreover, if little space is given to the access-intensive buffers, then induced conflicts on the actual working set will likely produce a performance decrease. We have selected the B test-case using PSS state saving as a litmus test for this possible drawback. We consider this as a revealing test because the reduced number of state saving operations invoked during the simulation allows properly focusing on the effects without exacerbating the possibile cause behind.

By the plots in Figure 8, we can see that the best performance is anyway obtained when a larger in-cache space is assigned to the access-intensive buffers. In the opposite situation (i.e. where far more space is destined to the access-mild buffers) the total amount of cache misses induced by internal conflicts on the actual working set (i.e. the access-intensive buffers) produces a performance drop in the order of 20%. Any other configuration lies in between.

6. CONCLUSIONS AND FUTURE WORK

In this work we have presented an innovative Memory Management subsystem for Optimistic Simulation platforms, the purpose of which is to enhance memory locality by differentiating memory allocation between access-intensive and access-mild buffers, in order to increase probability of finding them in the caching architecture. Generally-applicable design indications and an implementation relying on a simple API allow the extension/integration of this proposal within any optimistic simulation platform build according to the specifications in [11].

We have provided a set of experimental results on a modified version of the phold benchmark which show the viability of our proposal, using different working sets' sizes and on different checkpointing intervals. An analysis of the impact of the separation threshold between the access-intensive and access-mild buffers in cache has been provided.

Future work entails the design and the implementation of a heuristic able to autonomically tune the separation threshold position in cache, in order to further enhance benefits derived from secondary effects.

7. REFERENCES

[1] Linux kernel. http://www.kernel.org/.

[2] A memory allocator. *http://g.oswego.edu/dl/html/malloc.html*, 1996.

[3] L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel. An analysis of communication induced checkpointing. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 242–249, 1999.

[4] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

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

[6] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 227–234, 2007.

[7] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in Time Warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Computer Society, June 1995.

[8] R. M. Fujimoto. Parallel discrete event simulation. In *WSC '89: Proceedings of the 21st conference on Winter simulation*, pages 19–28. ACM Press, 1989.

[9] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[10] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Proceedings of the Multiconf. on Distributed Simulation*, pages 23–28. Society for Computer Simulation, Jan. 1990.

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

[12] Y. B. Lin and E. D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Advances in Parallel and Distributed Simulation*, 1991.

[13] A. Naborskyy and R. M. Fujimoto. Using reversible computation techniques in a parallel optimistic simulation of a multi-processor computing system. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pages 179–188. IEEE Computer Society, 2007.

[14] A. Park and R. Fujimoto. Optimistic parallel simulation over public resource-computing infrastructures and desktop grids. In *Proc. of the 12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, 2008.

[15] A. Pellegrini, R. Vitali, and F. Quaglia. Di-dymelor: Logging only dirty chunks for efficient management of

dynamic memory based optimistic simulation objects. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53, Washington, DC, USA, 2009. IEEE Computer Society.

[16] B. R. Preiss, W. M. Loucks, and I. 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.

[17] 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, Feb. 2001.

[18] F. Quaglia and V. Cortellessa. On the processor scheduling problem in time warp synchronization. *ACM Trans. Model. Comput. Simul.*, 12, July 2002.

[19] 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, June 2003.

[20] R. Rönngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[21] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, 1997.

[22] T. Santoro and F. Quaglia. A low-overhead constant-time ltf scheduler for optimistic simulation systems. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 948 –953, june 2010.

[23] S. Sköld and R. Rönngren. Event sensitive state saving in time warp parallel discrete event simulations. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 653–660, Washington, DC, USA, 1996. IEEE Computer Society.

[24] T. K. Som and R. G. Sargent. A probabilistic event scheduling policy for optimistic parallel discrete event simulation. In *Proc. of the 12th Workshop on Parallel and Distributed Simulation*, pages 56–63. IEEE Computer Society, May 1998.

[25] R. Toccaceli and F. Quaglia. Dymelor: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 163–172, Washington, DC, USA, 2008. IEEE Computer Society.

[26] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic log/restore for advanced optimistic simulation systems. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:319–327, 2010.

[27] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Computer Society, May 1996.

[28] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, 1995. Springer-Verlag.