Benchmarking Memory Management Capabilities within ROOT-Sim

Roberto Vitali, Alessandro Pellegrini, Francesco Quaglia
DIS, Sapienza Università di Roma

Abstract

In parallel discrete event simulation techniques, the simulation model is partitioned into objects, concurrently executing events on different CPUs and/or multiple CPU-Cores. In such a context, run-time supports for logical time synchronization across the different simulation objects play a central role in determining the effectiveness of the specific parallel simulation environment. In this paper we present an experimental evaluation of the memory management capabilities offered by the ROme OpTimistic Simulator (ROOT-Sim). This is an open source parallel simulation environment transparently supporting optimistic synchronization via recoverability (based on incremental log/restore techniques) of any type of memory operation affecting the state of simulation objects, i.e., memory allocation, deallocation and update operations. The experimental study is based on a synthetic benchmark which mimics different read/write patterns inside the dynamic memory map associated with the state of simulation objects. This allows sensibility analysis of time and space effects due to the memory management subsystem while varying the type and the locality of the accesses associated with event processing.

1 Introduction

A traditional way to achieve high performance simulations is the employment of parallelization techniques [3]. They are based on the partitioning of the simulation model into objects that can execute events in parallel on multiple CPUs and/or CPU-Cores. For these techniques, the main obstacle to speedup and scalability is the overhead due to synchronization, which is required to ensure causally consistent execution of simulation events at each object.

The optimistic synchronization approach is likely to favor speedup in general application/architectural contexts. In particular, it has been shown to exhibit performance relatively independent of the lookahead of the specific simulation model, and has also been shown not to suffer (in terms of amount of rollback in the parallel execution) from non-minimal message delivery latency. The latter feature makes it suited for a wide variety of computing platforms, including large scale traditional GRID systems and even desktop GRID environments. On the other hand it is clearly understood that the aforementioned potential advantages become real only in scenarios where the core issue of managing log/restore operations of the simulation objects' states has been addressed for both (a) transparency to the application programmer and (b) run-time efficiency.

In [8, 17] we have presented the design and development of Di-DyMeLoR (Dirty-Dynamic-Memory-Logger-and-Restorer), namely a fully featured memory management subsystem specifically oriented to optimistic simulation. It provides recoverability, via incremental log/restore techniques, of any type of memory operation occurring inside the state of the simulation object, namely chunk allocation/deallocation and update operations. Also, the application programmer is not requested to explicitly interact with this subsystem since those operations are allowed to be triggered inside the application level software according to standard technology and programming approaches. This memory management subsystem has been integrated inside the open source ROOT-Sim platform, which relies on `C` technology and on event-handlers' based programming style for simulation software development at the application level (see, e.g., [1, 2, 6]).

At the design level, Di-DyMeLoR relies on wrapping techniques used to intercept allocaton/deallocation requests from the overlying application in order to identify any changes dynamically occurring in the layout of the object memory map. It also adopts (lightweight) instrumentation to transparently link a memory-update tracker monitor to the application software, thus allowing run-time identification of memory-update operations occurring inside the memory map. The effectiveness of the whole design/implementation has been evaluated in [8, 17] with a mobile communication system simulation application. Although such an application has been parameterized to take into account different settings (e.g. in terms of memory requirements for the state of each simulation object), it does not mimic a sufficiently wide spectrum of memory usage/access patterns. Thus, a general-settings study of the effects of Di-DyMeLoR on (i) event processing overhead (caused by the run-time memory update tracker), and (ii) log latency reduction and memory locality improvements (thanks to incremental logging) is mandatory.

In this work we focus exactly on such a type of study. In particular, we have implemented a synthetic application derived from the well known PHOLD benchmark [4], explicitly embedding parameterizable memory operations' patterns. Then we have performed a sensibility study on the behavior of the memory management capabilities inside ROOT-Sim vs diversified application level memory operations' patterns. We note that the relevance of the present benchmarking study is twofold. (A) It can assess the goodness of the design approach underlying Di-DyMeLoR under generic application software execution patterns, thus quali-

fying such a design as a potential reference for other implementations. (B) It can allow establishing the viability and the effectiveness of such a memory management operating mode for applications characterized by specific memory operations' patterns (e.g. in terms of percentage of memory-write machine instructions per event).

The remainder of this work is structured as follows. In Section 2 we discuss related work. Section 3 provides an overview of ROOT-Sim. In Section 4 the benchmark application is described. Experimental data are presented in Section 5.

2 Related Work

Memory management operations, in terms of log/restore of optimistic simulation objects' states, have been studied by several works and according to differentiated methodologies. The works in [9, 11, 15] cope with non-incremental techniques and provide analytical models for the determination of the best suited frequency for log operations in order to optimize the trade-off between log and restore costs. In these studies, there is no explicit mention about how to support/implement the log/restore protocol. Hence, compared to our work, these models act at a higher level of abstraction, and require log and restore latency values as input parameters to be provided by the specific implementation of the log/restore architecture. Similar considerations apply to the incremental-log oriented studies in [7, 16], where analytical evaluation of best suited trade-offs is performed by having exact information about incremental (vs non-incremental) log operations' costs in input. In other words, differently from the present proposal, these works do not deal with the evaluation of any real subsystem (transparently) supporting incremental log facilities.

The work in [10] is oriented to implementation aspects and to the evaluation of a specific architecture for non-blocking (asynchronous) logging. However, differently from our work, it does not deal with incremental logging, and the implementation does not allow the employment of dynamic memory for the state of simulation objects.

The work in [18] copes with the design and evaluation of an incremental logging architecture based on software instrumentation. However, as for the aforementioned work, the usage of dynamic memory is not allowed, which is instead permitted inside the architecture we are evaluating in this work. Also, the instrumentation technique presented and evaluated in [18] is tailored to RISC architectures, while we deal with IA-32/x86-64 CISC processors, which intrinsically increases the complexity of the instrumentation process independently of the integration with a dynamic memory map manager.

The work in [12] deals with implementations of supports for incremental logging in the context of object oriented simulation software, which are based on operator overloading schemes. However, differently from the architecture we cope with, a-priori knowledge of memory locations reserved for state variables is required.

The issue of supporting dynamic memory based states for optimistic simulation objects has also been addressed by the optimistic simulation frameworks in [1, 2]. Compared to our target architecture, in these frameworks ad-hoc APIs are used to explicitly notify to the simulation kernel that specific allocation/deallocation operations, and, more in general, operations on data structures based on dynamic memory (e.g. lists), need to be rollbackable. As a consequence, supports for application level full transparency vs optimistic state management are not included (and hence not evaluated).

In terms of capabilities of the memory management subsystem the software architectures closest to the one we are evaluating in this study are probably those described in [13, 14]. However, they rely on a completely different design approach since there is an explicit use of Operating System memory protection mechanisms to detect memory updates and to incrementally log dirty pages belonging to the state of simulation objects. Compared to our case, the overhead for tracking updates and incremental log operations is likely higher (e.g. since it exhibits page size granularity). Hence, evaluation studies of these architectures are not representative of what we can expect with the different design approach adopted for developing the memory management capabilities offered by ROOT-Sim.

Similar comments apply to several other works aimed at implementing and evaluating transparent software architectures for incremental log/restore in the context of fault tolerance (see, e.g., [5]). In these studies, the main focus is on the optimization and evaluation of the interaction with stable storage devices, in order to persist volatile memory losses. Instead, we deal with an architecture for main memory logs, which are used to rebuild correct snapshots of the simulation object state in case of causality errors (not in case of faults causing volatile memory losses).

3 Overview of ROOT-Sim

ROOT-Sim is an open source, general purpose platform developed using `C` technology, which is 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 1 we schematize the architecture of ROOT-Sim, with focus on the memory management subsystem. The simulation kernel interacts with the overlying application software via the following call/callback functions:

`ProcessEvent()`: this callback has a set of parameters identifying the event to be processed (in the form of an application-defined data structure) and the base pointer for identification of the address where the top data structure associated with the simulation object state resides in memory. All the other memory chunks dynamically allocated inside the object state will be reachable via pointer based linking according to the logic defined by the application programmer.

`ScheduleNewEvent()`: this service exposed by the kernel can be invoked during event processing for injecting

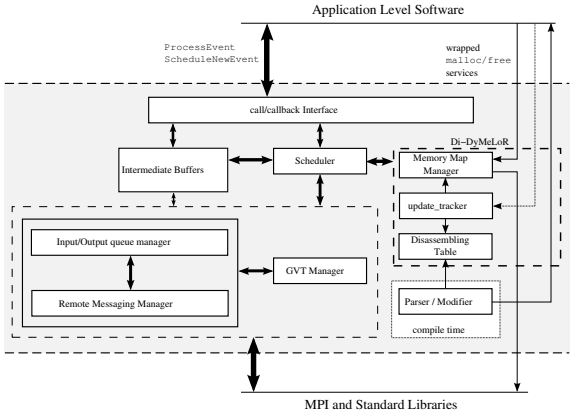

Figure 1. ROOT-Sim Architecture.

new events within the system, targeting whichever active simulation object.

The object state management subsystem, namely Di-DyMeLoR [8, 17], is a wrapper of `malloc` and `free` ANSI-C standard services, interposed at linking time between the application-level software and the standard malloc library. This approach allows the application programmer to use dynamic memory in a transparent way vs lower level memory management tasks (such as log/restore operations) supported by the ROOT-Sim kernel.

Di-DyMeLoR maintains per-simulation-object metadata tables, composed of so called `malloc_area` entries. Each entry keeps information about a block of contiguous pre-allocated chunks of a given power-of-2 size (i.e. the head memory location of the block of chunks and the metadata describing their status). Each time a memory allocation request is issued by the application-level software, Di-DyMeLoR checks whether a `malloc_area` of chunks of a suitable size has been pre-allocated. In the positive case, it finds an unused chunk inside it for serving the request. On the other hand, if the `malloc_area` was still unallocated, the underlying standard `malloc` service is invoked to allocate a whole block of unused chunks. Then the request is served.

A memory segment associated with a `malloc_area` keeps two bitmaps, named *use bitmap* and *dirty bitmap*, both reserving one bit for each chunk. The former bitmap keeps information about which pre-allocated chunks are currently in-use. The latter keeps information about which pre-allocated chunks have been involved in a memory write since the last log operation (those chunks will be referred to as dirty-chunks). Exploiting these data structures, a log operation is simply performed by analyzing the in-use/dirty bitmaps. In case of full logs, all the in-use chunks are packed into a contiguous log buffer appropriately sized. Instead, for an incremental log, only the dirty-chunks are packed. The log buffer, whether incremental or not, is linked to a log chain. A restore operation is performed by iteratively backward traversing the log chain, searching for logged chunks that have not been restored yet. This procedure stops when all the active `malloc_area` entries and all the in-use chunks have been restored. Although in principle this could entail an indefinite number of iterative backward steps, in practice the restore operation can be immediately finalized once a full log is found while backward retraversing the log chain. In fact, all the in-use chunks that have not yet been restored are immediately available inside the full log for copy-back operations.

To update the dirty bitmap, the memory management subsystem needs to intercept all the memory write operations occurring inside the memory map. Since dynamically allocated memory could be scattered through all the virtual addressing space, the design had to discard approaches such as memory-regions monitoring. Instead, a compile-time instrumentation approach has been adopted which analyzes the object generated after linking together all the application level modules. The analysis is carried out by a Parser/Modifier (PM) software component that has been specifically designed for instrumenting ELF objects generated by standard `gcc` compilers (versions 3 and 4) for IA-32 and x86-64 architectures. While scanning the object file, PM identifies every memory-write instruction inside it, namely `mov` instructions with a memory location as the destination. The instrumentation process is then supported via the insertion of a `call` instruction to an `update_tracker` module, written in assembly language, which performs the identification of the exact memory address and the size (amount of bytes) involved in the memory update operation.

For efficiency purposes `update_tracker` has been designed in a way to avoid disassembling of the memory reference instruction fully operated at run-time. Instead, an orthogonal technique has been adopted where a software table is built and populated during the compile-time instrumentation process. This table acts as a cache of disassembling results for memory-write instructions. Actually, in IA-32/x86-64 architectures, the address of each memory-write operation depends on a set of up to four parameters, namely `base`, `index`, `scale` and `displacement`. The former two parameters correspond to register values (i.e. they identify the registers containing the values), while the latter two correspond to specific values of fields inside the memory-writing instruction. The instruction opcode tells what of those parameters are relevant. Also, the opcode, together with its prefixes, establish the real size of the memory area touched by the write operation. Hence, to cache the results of the disassembling process, PM builds a table with entries structured as follows:

```
struct update_tracker_entry {
    unsigned long ret_addr;
    unsigned int size;
    char flags;
    char base;
    char index;
    char scale;
    long displacement;
}
```

The `flags` field is used to identify which of the aforementioned four parameters are actually relevant and should be considered by `update_tracker` for computing the

exact address for the memory-write operation. Also, the `size` field immediately indicates to `update_tracker` the (compile-time defined) size of the memory area to be dirtied by the current memory-write instruction ([1]). Finally, the `ret_addr` field indicates to `update_tracker` where control will be returned after its execution. This field corresponds to the memory address of the write instruction which immediately follows the current instance of the call to `update_tracker`. It has been included in the `update_tracker_entry` data structure for allowing the disassembling results' table produced during the instrumentation process to be organized as a fast search hash-with-buckets table. In particular, upon its activation, `update_tracker` checks inside its own stack frame the return address value, which is used as the key for accessing the hash table maintaining `update_tracker_entry` records, and is compared to the `ret_addr` field inside these records for selecting the correct entry within the bucket. Once this is done, the memory address for the write operation and the size of the memory being dirtied are easily computed by the monitor via a few machine instructions.

In the memory model offered by Di-DyMeLoR, locations associated with automatic variables (allocated inside the stack) do not belong to the object memory map, since they do not survive across different invocations of the event handler. Hence, all those memory-write instructions that can be detected at compile-time to access the stack (e.g. `mov` instructions addressing memory via base pointer or stack pointer displacement) are not instrumented by PM. However, in some cases write access into the stack cannot be recognized at compile time. For this reason, after having computed the address for the memory-write operation, `update_tracker` compares it with the current value of the stack pointer. In case the access is an actual stack update, `update_tracker` simply returns. Otherwise, the information about the identified memory address and the size of the area being dirtied is passed to the Memory Map Manager via a call to an internal function. The latter module identifies the `malloc_area` containing the chunks being dirtied via a software address-cache, used to speed up the retrieval ([2]). Thus, the monitoring operation ends with the Memory Map Manager setting to 1 all the bits in the dirty bitmap associated to those chunks that have been involved in the memory-write operation.

4 Benchmark Description

The benchmark we have used in this experimental study is derived from PHOLD [4]. 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., [10, 18]), 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.

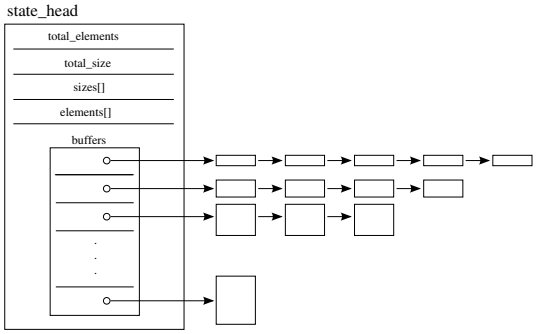

Figure 2. Object State Layout.

We have adapted this benchmark in order to cope with dynamic memory employment at the application layer level. In particular, we have re-implemented the benchmark by having each simulation object state formed by a main data structure called `state_head`, containing a set of N pointers for accessing N distinct linked lists of buffers relying on dynamic memory allocation (see Figure 2). The different lists keep track of buffers with different sizes, linearly distributed in between a min and a max value (this may give rise to chunk internal-fragmentation in Di-DyMeLoR since, as pointed out, it manages power of 2 chunk sizes). 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 ([3]).

Two types of events have been included in the benchmark logic:

`BUFFER_ALLOCATE[size]` - Upon its execution at logical time t by some simulation object, a new buffer is al-

[1] The only exception is for `movs` and `stos` instructions, used for moving arbitrary size memory blocks. These instructions keep the information for identifying the destination address and the current size of the memory block being written into predefined registers, namely `EDI` and `ECX`, which are directly accessible by `update_tracker`.

[2] Per-chunk headers have been explicitly avoided in Di-DyMeLoR, so a chunk has no information identifying the `malloc_area` it belongs to.

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

located and linked to the i-th local list, associated with $size(i) = size$.

`BUFFER_DEALLOCATE[null]` - Upon it's execution at logical time t by some simulation object, a $size$ value is randomly selected from the pool of $size(i)$ possibilities (with $0 \leq i \leq (N-1)$). Then a random buffer in the list associated with $size(i) = size$, if any, gets released and a `BUFFER_ALLOCATION[size]` event is scheduled at the same simulation time t for whatever simulation object. Also, a new `BUFFER_DEALLOCATE[null]` event is scheduled for the local simulation object, with logical time $t + incr$, where $incr$ follows some exponential distribution. In other words, the `BUFFER_DEALLOCATE[null]` event occurring at some simulation object reschedules itself for that same simulation object, so to generate a periodic deallocation operation on that object while simulation time proceeds (with consequent allocation on whatever object).

Compared to the original PHOLD, the differentiation in the previous two types of events actually implies that, while simulation time goes on, we are migrating buffers across the different simulation objects, with exponentially distributed migration rate (according to the rate for the reschedule of deallocation events). In other words, while simulation times goes on, dynamic memory allocation/deallocation operations take place, causing migration of a given buffer (with a randomly selected size) from one simulation object to another. In any case, at each simulation time, the global memory occupancy across all the objects is constant (thus reflecting, at any point in simulation, the specific space complexity of the simulation model for which the benchmark configuration is the current mimic).

We have also augmented the event execution logic with read/write accesses inside the buffer lists currently belonging to the simulation object memory layout. This has been done in order to emulate read vs write intensive applications. The more write intensive the event, the larger the number of chunks for which memory write tracking is expected to occur during event processing (namely, the dirty chunks), and which are expected to be incrementally logged upon taking the object state snapshot. This will allow us to observe how the costs of memory write tracking and log/restore operations scale vs ROOT-Sim implementation choices and vs conventional technological trends (e.g. the interaction between compiler choices and the specific instruction set of IA-32/x86-64 processors).

To determine the span of read/write operations across the buffers inside the object lists, we have adopted a breadth-first visit on those lists. Specifically, we have introduced an additional parameter x (with $x \leq S$) indicating the total amount of bytes that need to be read and/or written during the event execution across the dynamically allocated buffers currently linked inside the object state. Initially x represents the residual amount of bytes to be touched. Upon starting the execution of an event, we randomly select the list of buffers from which starting the visit, and we touch in read and/or write mode 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, starting from that list, 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 read/write operation. This goes on until the residual amount of bytes to be read/written becomes zero. Actually, the lower the value of x, the higher the memory locality of the application.

In any case, we note that the breadth-first visit mimics a worst case scenario for incremental log/restore facilities offered by ROOT-Sim since write operations are not localized inside one or a few `malloc_area` regions. This implies that the amount of meta-data that are dirtied due to event processing is non-minimal (recall that when a memory chunk delivered to the overlying application gets updated, the associated meta-data also gets updated), thus ultimately imposing significant meta-data management costs for supporting incremental log/restore operations.

Finally, we have implemented the read/write operating modes by exploiting `C` services offered by stdlib. Specifically, the read operating mode has been supported via `memcmp` calls, while the write operating mode has been supported via an instrumented version of `memset`, transparently linked to the `update_tracker` module (and hence to the lower level memory management subsystem) at compile time. Among services in the standard library, the aforementioned ones seem to better emulate read-only vs write-only activities. In particular, `memcmp` only involves data movement from memory towards CPU registers, while `memset` only entails data movement from CPU registers to main memory. Other stdlib services, such as `memcpy`, entail both types of data movement, thus not fitting the aim of our study, where a specific tuning of the mixture of read vs write memory operations needs to be explicitly selected for each configuration point.

5 Experimental Data

The hardware platform used in this experimental study is a QuadCore machine, equipped with an Intel Core 2 Quad Q6600 (64bit execution support, 2.4GHz, 4MB L2 Cache per couple of cores, 32KB L1 Cache per core, 1GHz Front Side Bus speed) and 4GB of RAM memory. The running Operating System is GNU/Linux (kernel 2.6.22-31 64bit, distribution OpenSUSE 9.2), the used `gcc` version is 4.2.1, the used `binutils` version (`ld` and `gas`) is 2.17.50 and the used MPI version is OpenMPI 1.2.4. Regarding ROOT-Sim run-time parameters, the GVT period (namely, the interval for memory recovery of obsolete logs) has been set to 1 sec. With this value, RAM usage never exceeds 60/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures. The benchmark configuration is formed up by four simulation objects (each one hosted by one instance of the simulation kernel running on the QuadCore machine). The performed tests require each simulation object to execute at least 10.000 buffer allocations, scattered over 8 different buffer chains with sizes ranging from 32 bytes to 4KB.

We have measured Event Latency, Checkpoint Latency, Restore Latency and Memory Usage (per checkpoint) with different interleaving steps between full and incremental logs, namely incremental logs taken at each event and full logs taken every 5 or every 20 log operations.

We have performed similar measurements by also excluding software instrumentation and related incremental log capabilities. This has been done by linking the environment to DyMeLoR [17], whose structure is similar to Di-DyMeLoR, except for that no write-memory tracking is supported, thus log and restore operations are always *full* (i.e. all the currently in-use chunks are packed/unpacked to/from the log buffer). We will refer to the simulation software running with incremental log/restore capabilities as *instrumented* software, while we will refer to the simulation software running with non-incremental log/restore capabilities as *non-instrumented*. The latter acts as a reference for the evaluation of the fully featured instrumented version. The measures have been taken with different read/write memory access patterns. Specifically, the value of the parameter x in the benchmark has been varied in order to generate read/write operations spanning their locality from the 20% to the 80% of the whole size of the simulation object state. In addition, we have varied the objects' state startup size S, specifying it to be 10KB, 100KB and 1MB.

In Figures 3, 4 and 5 we present the observed parameter values while varying read/write locality. As hinted in Section 3, Di-DyMeLoR traces memory updates caused by event processing via the insertion of a call to a monitoring routine which inevitably adds some overhead. However, such an overhead is relatively limited up to 40% spanning of write operations inside the simulation object state (see Figures 3-A, 4-A and 5-A). Also, when the state increases in size (i.e. when the amount of memory buffers to be read or written is higher), the overhead gets relatively reduced.

The limited overhead up to relatively large spanning of the locality of write operations can be notably justified on the basis of classical technological trends in compilers for IA-32/x86-64 architectures, according to which (large) data structures are updated by modules relying on optimized `movs` and/or `stos` instructions. These allow copy/set operations of arbitrary size memory chunks via a single machine instruction, for which Di-DyMeLoR tracks the memory write access via a single call to the `update_tracker` module. In other words, the cost of memory tracking well scales due to synergy between the tracking architecture and the compiler/instruction-set. We recall that in standard libraries, such optimizations are present in the compilation of many modules, including the `memset` module we have instrumented and, more importantly, the traditional `memcpy` facility. The event processing overhead of the instrumented software is moreover counterbalanced by reduced checkpoint latency (see Figures 3-C, 4-C and 5-C). Further, this latency is not significantly affected by the variation of the interleaving step between full and incremental logs.

The capability for such a checkpoint overhead reduction would acquire a highly increased relevance in scenarios where the application software is not Piece-Wise-Deterministic, which require a log to be taken at each event since un-logged state values cannot be guaranteed to be correctly reconstructed starting from a previous log and replaying intermediate events. In these scenarios, the latency of a log operation becomes as critical as the event latency in determining the final perceivable performance.

Memory requirements for each log operation in the instrumented case are definitely lower than those observed for non-instrumented software (see Figures 3-B, 4-B and 5-B). This further strengths the capabilities of the fully featured incremental version of the software in case of applications with very large memory requirements for the objects' states.

The non-instrumented configuration typically provides gains in state restore operations. However, as pointed out in Section 3, the state restore latency in Di-DyMeLoR directly depends on the interleaving between full logs and incremental logs along the log chain. In fact, by the plots (specifically, those in Figures 3-D, 4-D and 5-D) we see that the performance decrease in the state restore for the case of instrumented software can be controlled (while maintaining the aforementioned advantages on the side of logging) via proper selection of a non-oversized interleaving step between full and incremental logs. Also, for 1MB state size, state restore operations in the non-instrumented version become more expensive that those of the instrumented version. Given that our tests have been performed with logs taken at each simulation event (which mimics the settings required by application software not entailing Piece-Wise-Determinism), the larger restore latency for the non-incremental case is supposed to occur due to largely reduced locality of state management operations. In particular, given the large size of the simulation objects' states (and hence of the log buffers used to pack state information by the non-instrumented version) a relevant amount of virtual addresses are referenced for building the log chain in the non-instrumented case before memory recovery operations associated with GVT are executed. This provides reduced locality that reveals unfavorable to restore operations (i.e. the likelihood of cache availability of logged information to be restored gets reduced). Such a phenomenon is especially evident for finer grain events, namely when the memory access pattern mostly entails read accesses.

6 Summary

In this paper we have presented an experimental evaluation of the memory management capabilities offered by the optimistic parallel simulation environment ROOT-Sim. In particular, our focus was on incremental log/restore aspects, and on the software instrumentation techniques used to track memory write operations with arbitrary granularity. We have developed and presented a synthetic benchmark, derived as a variation of the PHOLD benchmark, which mimics different read/write patterns inside dynamic memory based maps of the simulation objects' states, thus allowing sensibility analysis of both time and space effects due to the memory management subsystem. This has been done while varying both the type and the locality of the accesses performed by the simulation object while processing

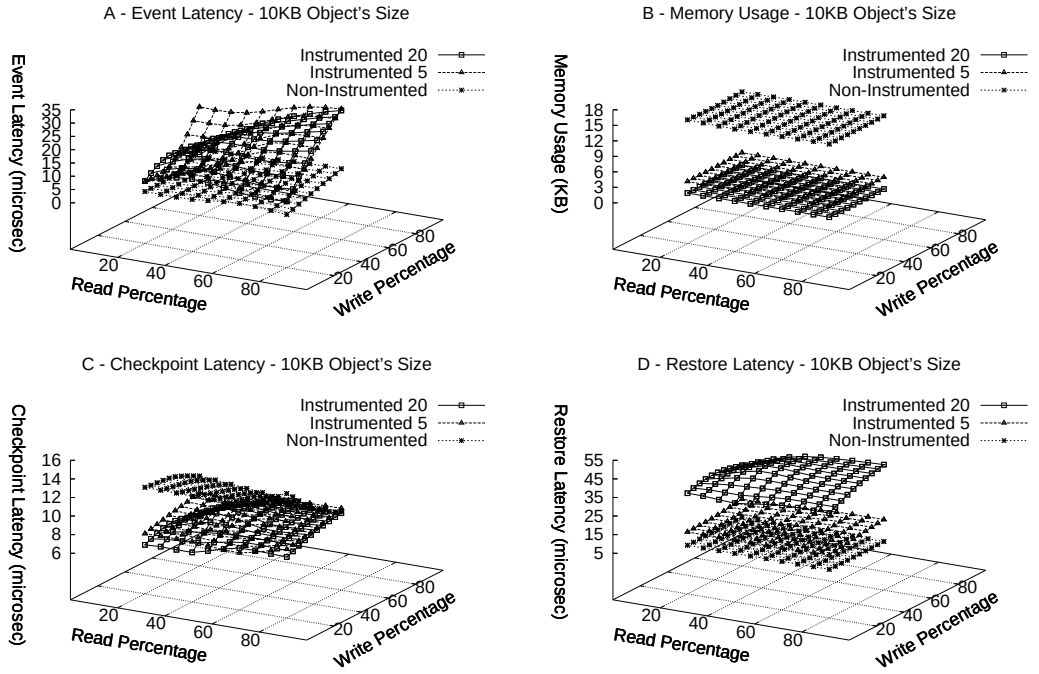

Figure 3. Benchmark with 10KB Objects' State Size.

the events. Our work is on the path of assessing the efficiency of design/implementation of supports for high performance simulation systems, which are becoming increasingly important in contexts such as simulation based decision making with temporal constraints.

References

[1] SPEEDES. *http://www.speedes.com*, 2005.

[2] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In *Proceedings of the 26th Winter Simulation Conference*, pages 1332–1339, San Diego, CA, USA, 1994. Society for Computer Simulation International.

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

[4] 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.

[5] K. Li, J. Naughton, and J. Plank. Low latency concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):474–479, 1994.

[6] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. WARPED: A Time Warp simulation kernel for analysis and application development. In *Proceedings of the 29th Hawaii International Conference on System Sciences - Volume 1: Software Technology and Architecture*, page 383, Washington, DC, USA, 1996. IEEE Computer Society.

[7] A. C. Palaniswamy and P. A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 127–134. IEEE Computer Society, 1993.

[8] A. Pellegrini, R. Vitali, and F. Quaglia. Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 23nd Workshop on Principles of Advanced and Distributed Simulation*, pages 45–53. IEEE Computer Society, 2009.

[9] 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.

[10] 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.

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

[12] R. Ronngren, 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*, pages 70–77. IEEE Computer Society, May 1996.

[13] A. Santoro and F. Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 171–180. IEEE Computer Society, 2005.

[14] A. Santoro and F. Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Proceedings of the 9th International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2005.

[15] S. Skold and R. Ronngren. Event sensitive state saving in Time Warp parallel discrete event simulation. In *Proceedings of the Winter Simulation Conference*, pages 653–660. Society for Computer Simulation, 1996.

[16] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.

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

[18] 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.

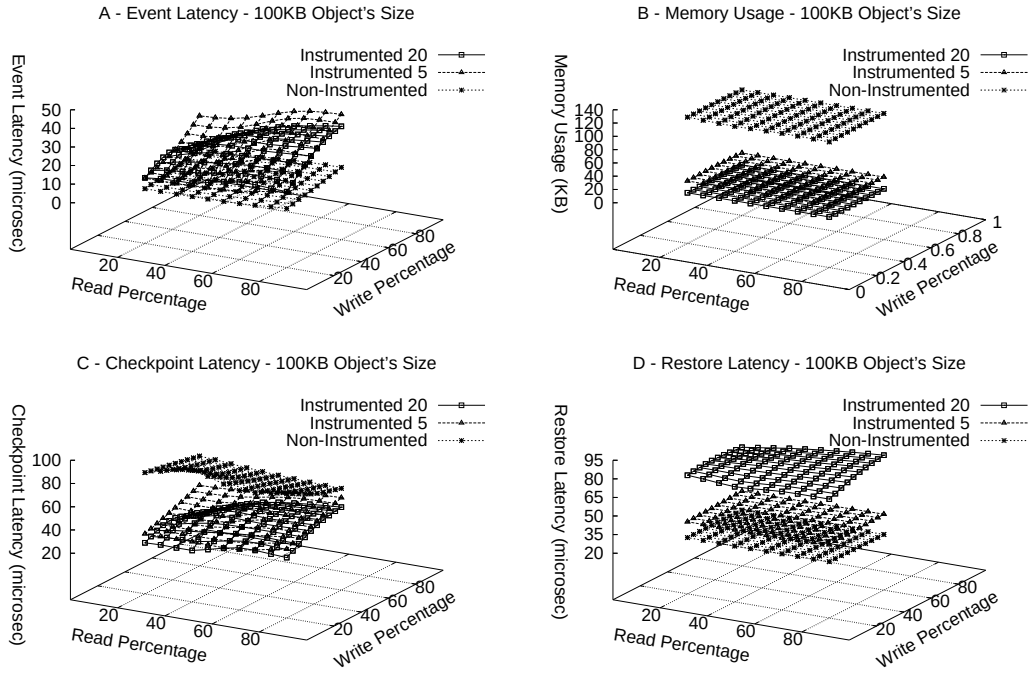

Figure 4. Benchmark with 100KB Objects' State Size.

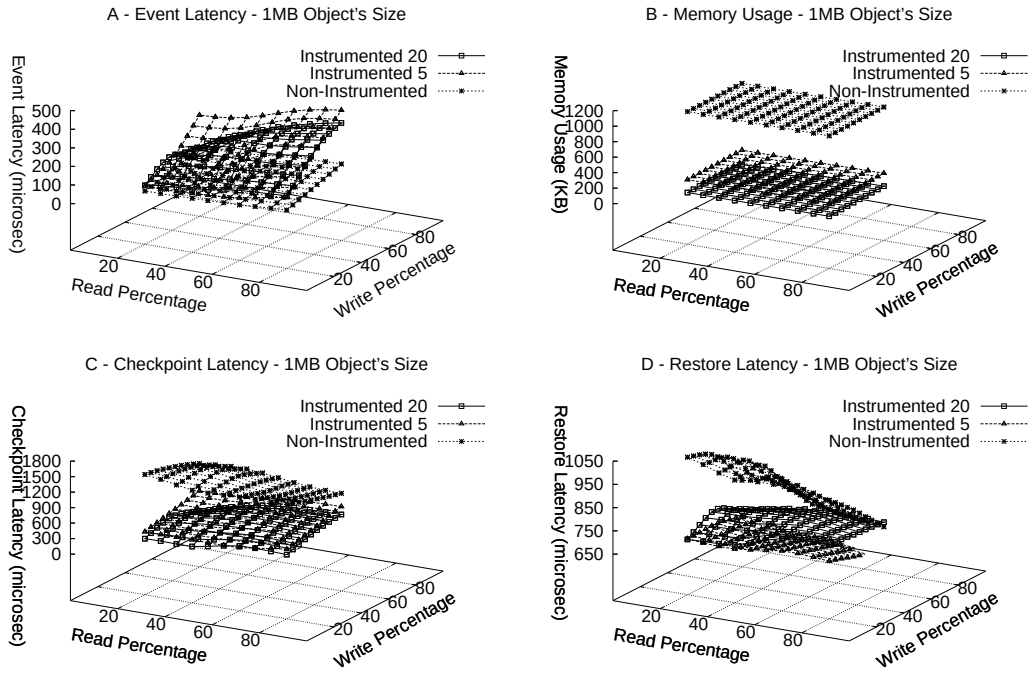

Figure 5. Benchmark with 1MB Objects' State Size.