Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects

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

Abstract

A recent work has presented the design and implementation of a software library, named DyMeLoR, supporting transparent log/restore facilities for optimistic simulation objects with generic memory layout. This library offers the possibility to allocate/deallocate memory chunks via standard API, and performs log/restore of the object state via pack/unpack techniques, exploiting ad-hoc meta-data concisely identifying the object state layout at each point in simulation time. In this paper we complement such a library with a software architecture offering the following additional advantages: (i) run-time identification of chunk updates within the dynamic memory map, (ii) reduced checkpoint latency and increased effectiveness in memory usage thanks to log/restore facilities based on (periodic) snapshots of the whole simulation object state, taken via the incremental copy of the modified (dirty) chunks only. Our approach is based on software instrumentation techniques (suited for LINUX and the ELF format), targeting memory update references performed by the application level software, and on a lightweight run-time monitoring mechanism providing minimal overhead while tracking the exact memory addresses and the size of memory areas dirtied by the execution of each event. Also, our design has been oriented to portability across 32-bit and 64-bit Intel compliant architectures, thus covering a wide spectrum of off-the-shelf machines.

1 Introduction

A recent work [20] has presented a memory management library named DyMeLoR, implemented in `C` technology, which offers completely transparent log/restore facilities while supporting, at the same time, a general programming model where the state of a simulation object can be scattered on dynamically allocated memory chunks. In particular, DyMeLoR has been designed for integration within traditional-style optimistic simulation kernels (see, e.g., [3, 7]), and allows the application programmer to employ standard malloc services within event handler modules (e.g. for changing the current layout of the object state). In this work we complement DyMeLoR via a software architecture which adds new capabilities within the memory management subsystem, while maintaining the same transparency level. Specifically, we present the design and implementation of software modules that are able to track state updates (with arbitrary granularity) occurring during event processing. These modules also offer log/restore facilities based on (periodic) snapshots of the whole state of the simulation object, which are built by incrementally copying only dirty chunks within the dynamic memory map associated with the state layout. Hence the name Di-DyMeLoR (Dirty-DyMeLoR) for the out-coming software.

On the side of state logging, the present proposal can provide direct performance advantages over the original software due to the reduced amount of memory copy operations. In fact, a complete snapshot of the simulation object state involves logging a subset of the chunks currently belonging to the memory map (those that have been dirtied since the last checkpoint was taken). It can also provide indirect advantages (e.g. via increased locality) thanks to the reduction of the memory usage for storing the snapshots.

On the other hand, the new architecture involves run-time monitoring activities to determine the exact addresses that are referenced in memory update operations by the application level software. To efficiently support this task, we have designed and implemented a lightweight memory-write tracking mechanism. It is based on transparent application software instrumentation, and is tailored for LINUX and standard ELF objects/executables on both 32-bits and 64-bits Intel compliant hardware architectures (namely IA-32 and x86-64 architectures).

The remainder of this paper is structured as follows. In Section 2 we provide an overview of DyMeLoR as the basis for the comprehension of the whole software system. In Section 3 the innovative capabilities and the design of the associated modules are presented. Related work is discussed in Section 4. Experimental data for the assessment of the effectiveness of Di-DyMeLoR are presented in Section 5.

2 Overview of DyMeLoR

From an architectural point of view, DyMeLoR can be seen as a wrapper of ANSI-C `malloc`/`free` services, which is transparently interposed between the application level code and the traditional malloc library (a schematization of this approach is shown in Figure 1.A). DyMeLoR also offers an API for integration with the simulation kernel, which consists of a set of services supporting memory management operations specifically oriented to log/restore activities proper of the optimistic synchronization scheme.

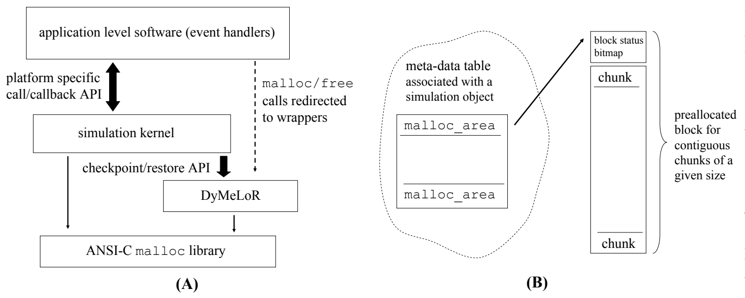
application level software (event handlers)
platform specific call/callback API
malloc/free calls redirected to wrappers
simulation kernel
checkpoint/restore API
DyMeLoR
ANSI-C malloc library
(A)
meta-data table associated with a simulation object
malloc_area
malloc_area
block status bitmap
chunk
chunk
preallocated block for contiguous chunks of a given size
(B)

Figure 1. Software Architecture and (A) main DyMeLoR's data structures (B).

For each simulation object hosted by the simulation kernel, DyMeLoR maintains a meta-data table of `malloc_area` entries. Each entry keeps information about a block of contiguous memory chunks (e.g. the memory location of the block), possibly allocated for serving memory requests for that object, and different entries are used for managing chunks of different sizes. As soon as a `malloc` request occurs for a given chunk size, the corresponding block is allocated by DyMeLoR (via real malloc services) so that a contiguous number of those chunks is in practice pre-allocated for serving future requests with that same size (if any). This allows exalting memory contiguity for the state layout of each simulation object, which can favor performance during both event processing and log/restore operations. In addition, pre-allocation of contiguous chunks allows DyMeLoR to use very concise meta-data for the identification of the status of each chunk (busy or free) within a block. In particular, a simple bitmap of so called "status bits" is used for the identification of those pre-allocated chunks that have also been delivered (and are currently in use) to the application software. To further optimize memory usage, a layout has been adopted where the bitmap of status bits is placed at the head of the pre-allocated block of chunks, and gets allocated only in case of real allocation of the corresponding block (see Figure 1.B for a schematization of the relation between the main data structures describing the simulation object memory map). Actually, the table of `malloc_area` entries can be expanded in case its entries have been saturated, and the simulation object goes on requesting more chunks during event processing activities.

Log/restore operations are supported by DyMeLoR via pack/unpack techniques. For a log operation, the currently in-use memory chunks are packed into a contiguous log buffer (which is dynamically allocated via the underlying malloc services), together with all the active `malloc_area` entries and the status bitmaps at the head of active blocks. For a restore operation, these data structures are extracted from the contiguous log buffer and put back in place. To make deallocation operations recoverable, each `malloc_area` entry also keeps information about the logical time (if any) at which the chunks inside a given block were all released. A block with no chunks currently allocated, and whose last chunk release occurred before the Global-Virtual-Time (GVT), can be deallocated via a `free` call towards the underlying malloc library. In such a case, the corresponding `malloc_area` entry is set to non-active.

Log operations exploit threshold-based optimizations, and an ad-hoc chunk allocation algorithm within each block, aimed at optimizing the trade-off between meta-data management and memory copies of active chunks. In particular, each `malloc_area` keeps a counter identifying the percentage of in-use chunks within the corresponding memory block. When the occupancy exceeds a given threshold, the memory block is entirely logged, without explicitly scanning the status bitmap in order to identify busy chunks. Also, the chunk allocation logic within each memory block is similar to the LINUX algorithm for the selection of the next file descriptor to be assigned while opening an I/O channel. This logic keeps block fragmentation low and tends to have busy chunks clustered at the head of the memory block. Therefore, in case the bitmap needs to be scanned upon the log operation (i.e. when the block occupancy is under the threshold), the latency of the operation can benefit by the early stopping of the bitmap scan whenever all the in-use chunks are really found clustered at the beginning of the memory block.

3 Di-DyMeLoR: Design and Implementation

As mentioned, Di-DyMeLoR is based on new software modules able to track at run-time memory update references performed by the application software. It also uses data structures and modules which augment the capabilities of the original DyMeLoR with the aim at tracking not only the current memory map of the simulation object (and its dynamic changes), but also the dirtying activities. This is done to support log/restore operations via the incremental approach.

In this section we first describe the techniques we have used for a lightweight instrumentation of the application level software, in order to support the memory update tracking process. Then we enter details of data structures' and modules' integration.

3.1 The Software Instrumentation Tool

Software instrumentation has been realized via a software Parser/Modifier (PM) specifically designed for analyzing and rewriting ELF (Executable and Linkable Format) objects generated by standard `gcc` compilers (versions 3 and 4) for IA-32 and x86-64 architectures. At the very base, PM works by parsing the object generated after linking together all the application level modules (third party libraries being excluded), and by identifying every memory-write instruction inside this object, namely `mov` instructions with a memory location as the destination. The instrumentation process is then supported by PM via the insertion of a `call` instruction to an `update_tracker` module, edited in assembly language, which performs the identification of the exact memory address and the size (amount of bytes) involved in the memory update operation. Although this is a typical way for tracking memory update references (e.g. in the context of program debugging techniques [21]), the usage of this approach in optimistic simulation systems poses (more) stringent performance issues. In particular, the monitor should likely perform its job via very few machine instructions, in order not to significantly impact event execution latency.

To cope with such a performance target we have decided not to employ run-time disassembling of the memory reference instruction, which could be onerous (compared to the event execution latency of non-instrumented software) especially due to the complexity and variable format/length of the Intel instruction set. Instead, we have adopted an orthogonal technique where a software table associated with the `update_tracker` is built and populated during the compile-time instrumentation process. This table acts as a cache of disassembling results for memory-write instructions.

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 (hence the parameters 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 which 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 disassemblig process, PM builds a table where each entry is 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 inserted as a field of 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. (Recall that access to the `update_tracker_entry` associated with the memory-write instruction occurs during the execution of the `update_tracker`. Hence, it is a performance critical operation directly impacting the event execution cost for the instrumented version of the application software.) 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. Given that this computation can unpredictably change the value of the EFLAGS register on board of the CPU, this register value is saved by `update_tracker` upon its activation together with general purpose ones, and is put back in place right before returning control to the memory write instruction for which the tracking process has been activated. Actually, PM can be parameterized in order to optimize the trade-off between the size of the hash-with-bucket table, and the access cost. Specifically, the instrumentation process can check whether the level of collision inside the hash table exceeds a prespecified value. In such a case, PM can resize the hash-with-bucket table in order to reduce the actual bucket size. The potential drawback is the increase of unused table entries, while the benefit is the reduction of the `update_tracker` overhead when accessing the table (with $O(1)$ time complexity as the best case).

`update_tracker` uses absolute addresses as keys for the hash table. In fact, as mentioned above, this module identifies the address of the memory-write operation at runtime by accessing its own stack frame. In order for PM to be able to build the table of `update_tracker_entry` records via correct insertion of the absolute addresses of memory-writing instructions inside the `ret_addr` field, we have exploited incremental linking facilities offered by standard linkers (e.g. `ld` on UNIX systems). In particular, the instrumentation process interacts with the linker for the definition of the exact (absolute) position of the object associated with application level software inside the executable layout.

In the memory model offered by 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 actually instrumented by PM. Anyway, 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,

[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`.

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 whose structure is presented in the next section.

The insertion of the call to update_tracker prior to the execution of a memory-write instruction leads to a resize of the sections associated with the object file, and to the shift of instructions and other memory locations inside the object layout. Hence, PM also has to rewrite the headers associated with the ELF object, the relocation tables, and the offsets used for the identification of memory addresses referenced by the software, e.g. the destination addresses for jmp instructions.

However, in IA-32/x86-64 processors not all destination addresses for jmp instructions can be corrected at compile-time by rewriting relocation tables. This is the case of so called register jumps (also frequently referred to as indirect branches), where the destination address is dynamically identified via the content of CPU registers. To cope with this issue, we have implemented a second run-time monitoring mechanism for supporting on-the-fly correction of destination addresses in register jumps. Like in the aforementioned approach, this mechanism is based on the insertion of a call instruction to a second assembly level monitoring module, referred to as branch_corrector, prior to each register jump in the original software. This monitoring module relies on a hash table similar to the previously described one, where each entry is associated with a single register jump instruction, and keeps the information regarding which are the registers whose values determine the destination address for the jump operation. This table is built and populated at compile-time during the instrumentation process (again to avoid costly run-time disassembling techniques). By exploiting the information inside this table, branch_corrector evaluates the original destination address for the jump instruction (by reading the CPU registers that specify the destination value). Then it corrects this address on the basis of the amount of bytes by which the original destination was shifted inside the instrumented object layout. To provide a lightweight mechanism for address correction, PM generates a third table at compile-time, which is visible to branch_corrector. Each entry inside this table identifies an interval of addresses for which the instrumentation process gave rise to the same amount of shift inside the final (instrumented) memory layout. Such an offset is also maintained in the table entry. The table is ordered by interval extremes, and branch_corrector performs a logarithmic-cost binary search to retrieve the interval containing the original destination for the register jump, and the offset to be applied for the correction. Such a correction cannot however be applied by modifying the values of the CPU registers involved in the jump instruction. This would otherwise result in an application inconsistent processor state. We have rather adopted a different approach where the original register-jump instructions are substituted at compile-time by PM with so called offset jumps (not relying on CPU registers), where the destination address is maintained inside one field of the instruction, and is appropriately set by the on-the-fly correction mechanism. To support the rewrite operation of the appropriate instruction field at run-time, without impacting typical settings associated with memory protection, the offset-jump operation has been moved inside a run-time re-writable ELF section (ad-hoc created by exploiting compiler/linker facilities). Also, a jump-label instruction has been inserted in place of the offset jump inside the original (non-rewritable) sections of the application code, which passes control to the offset jump right after the brach_corrector module has re-written the correct destination address (the offset) inside the ad-hoc re-writable section.

3.2 Management of the Memory Map

The original data structures and modules managing the simulation object memory map have been extended/modified in Di-DyMeLoR in order to explicitly cope with the possibility to build complete state logs by incrementally logging only data that have been dirtied since the last log operation. To guarantee recoverability of each type of operation permitted on the memory map, namely chunk allocation/deallocation and chunk update, we need to deal with incremental log of both dirty data, namely dirty chunks, and dirty meta-data, namely dirty malloc_area entries associated with the memory map.

To track dirty chunks, a second bitmap, of so called dirty bits, has been associated with each block of pre-allocated chunks destined to a specific simulation object. This bitmap is placed inside the same contiguous memory segment pointed by the corresponding malloc_area and containing the original status bitmap and the chunks destined for use by the overlying application in case of malloc requests. In terms of real storage, the dirty bitmap inherits the same features of the original status bitmap since its allocation occurs only in case the corresponding chunks gets really pre-allocated. Hence, the extra storage occupancy for detecting chunks that have been dirtied since the last log operation scales well with the size of application destined storage. The bits inside the dirty bitmap are treated as sticky flags vs the memory-write monitoring mechanism described in the previous section. Hence, a memory-write operation performed by the application software can only result in a set operation of the dirty bit associated with the chunk being dirtied.

To track dirty meta-data we have added the following two integer fields inside the malloc_area data structure:

- dirty_area, which is used as a flag indicating whether any type of operation (allocation, deallocation or chunk dirtying) has occurred in the malloc_area since the last log.

- dirty_chunks, which explicitly counts the current number of in use chunks that have been dirtied in the malloc_area since the last log operation.

Once the memory map manager receives the address and the size of the memory area being dirtied from the `memory_tracker`, it identifies all the chunks that will be dirtied inside the memory map, and the associated `malloc_area` entry. Then the dirty bitmap and the `dirty_chunks` field are updated. Again in compliance with DyMeLoR's memory model, in case the address and the memory area being dirtied refer to locations outside the memory map of the currently executing simulation object (e.g. they refer to global variables outside the heap, for which recoverability is not provided), the memory map manager simply returns control to the `memory_tracker` module. The `dirty_area` field inside the `malloc_area` is anyway set to 1 each time a `malloc`/`free` call insisting on that area is performed by the application software.

3.2.1 State Log Operations

Via the exploitation of the additional fields inside each `malloc_area`, and of the dirty bitmaps, logging activities performed by Di-DyMeLoR have been differentiated in full and incremental logs. Both types of logs still result in packing the information to be logged inside a contiguous buffer allocated via the underlying malloc services. However, they pack different things (with consequently different costs). A full-log operation coincides with the original log supported by DyMeLoR. Hence, the active `malloc_area` entries are packed inside the log buffer together with the in use chunks in the corresponding memory blocks, while the dirty bitmaps are not logged. On the other hand, an incremental log performs differentiated pack operations depending on the current value of data structures explicitly used for tracking dirty data/meta-data. Specifically, for each active `malloc_area` entry we have the following cases:

A: `dirty_area` is set and `dirty_chunks` is zero. In this case the `malloc_area` is packed into the log buffer together with the status bitmap indicating the current allocation of chunks inside a given block. But the dirty bitmap and the currently in-use chunks are not logged.

B: `dirty_area` is set and `dirty_chunks` is greater than zero. In this case the `malloc_area` is packed into the log buffer together with the status bitmap, the dirty bitmap and the chunks that are currently in use, which have been dirtied.

C: `dirty_area` is not set. In this case, no information associated with the area is logged at all.

Full and incremental logs both involve the re-set of all the data structures tracking dirty data/meta-data. For incremental logs, this occurs independently of the actual case among the aforementioned ones.

We finally underline that incremental state log operations no way require to be forced at each simulation event, but can be taken periodically. In fact they are based on recognizing memory portions that have been dirtied since the last log, independently of the amount of events actually performing the dirtying operations. Hence, state reconstruction at whichever simulation time can be supported via a mix of state restore from the log (see next section), and classical coasting forward.

3.2.2 State Restore Operations

Each log is stamped with the current simulation time, and all the logs (full and incremental) are linked together as a chain. When a restore operation needs to be executed at simulation time T, the log chain is searched to determine the more recent log with time less than or equal to T (logs with time greater than T are simply discarded since they refer to causally inconsistent memory maps). In case the log found is a full one, then a restore operation is executed by simply unpacking all the logged data and putting them back in place. A different restore algorithm is executed in case the log found is an incremental one. Specifically, the following steps are iterated by backward traversing the chain of logs:

1. A `malloc_area` found inside the log buffer, which has not been restored, is put back in place inside the meta-data table. The associated status bitmap is also copied back from the log buffer (recall that independently of the type of log and of the specific case for incremental logging, a logged `malloc_area` is always associated with the corresponding status bitmap inside the log buffer to guarantee recoverability of chunk allocation/deallocation operations).

2. Each dirty chunk found inside the log and associated with the `malloc_area`, which has not yet been restored in a previous iteration while backward traversing the log, is copied back in its correct position inside the corresponding memory block.

The iterative restore procedure stops when all the active `malloc_area` entries have been restored and all the in-use chunks that have been dirtied are also restored. Although in principles this could entail an indefinite number of iterative backward steps along the log chain, in practice the restore operation can be immediately finalized once we find a full log while backward re-traversing 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. Actually, to optimize the detection of already restored chunks, which must therefore not be copied-back again from the log, the iterative restore procedure has been based on temporary bitmaps (each associated with an active `malloc_area`) on which a couple of fast bitwise OR-XOR operations are executed each time a dirty bitmap (associated with that same `malloc_area`) is extracted from the incremental log.

3.2.3 Caching Write References for Latency Reduction while Managing the Memory Map

Our implementation is based on the avoidance of per-chunk headers. This design choice is aimed at minimizing the amount of meta-data to be logged/restored ([2]). Hence, when a chunk gets released, no header information can be exploited for fast access to the `malloc_area` involved in the deallocation operation. To speed up deallocation, via the avoidance of scan operations over all the active `malloc_area` entries, DyMeLoR originally provided a software-level direct-map caching subsystem, implemented as a hash table, with cache line formed by the tuple $<chunk\_addr, m\_area\_index>$.

The issue of identifying the correct `malloc_area` starting from the memory address associated with a chunk becomes even more critical in Di-DyMeLoR. Specifically, the memory map manager needs to retrieve the `malloc_area` for updating the information about dirty data/meta-data each time an instrumented memory-write operation dirtying whichever chunk inside the memory map occurs. Also, in Di-DyMeLoR we need to retrieve the correct `malloc_area` starting from a memory address which does not necessarily coincide with the chunk boundary address (as instead occurs for `free` operations).

To cope with such an issue, the original cache has been extended by having the cache line augmented with the chunk-end-address and represented by the tuple $<chunk\_start\_addr, chunk\_end\_addr, m\_area\_index>$. The start address for a memory write operation intercepted by `update_tracker` is stripped of `n` less significant bits by the memory map manager and is then used as the key for accessing the hash table. The value of `n` is chosen with the aim at making the whole range of addresses belonging to each single chunk collide into a single cache line. Actually, given that the size of the chunks delivered to the application software can be different, `n` has been set as the mean value between the number of bits needed to make the smallest and the greatest chunks collide, biased to the smaller sizes.

3.3 Interaction with Third Party Libraries

With the original DyMeLoR, any memory write operation on allocated chunks was allowed to occur inside functions in third party libraries, provided that these functions did not allocate any further memory buffer (as is the case for most functions inside the `C` standard library stdlib). This is no longer automatically the case when using Di-DyMeLoR and its incremental log/restore facilities. In fact, libraries are not instrumented hence it would not be possible for `update_tracker` to catch memory changes made inside those libraries.

We have explicitly addressed the case of update operations performed by third party software, just focusing on stdlib. Specifically, we have implemented inside Di-DyMeLoR a set of function wrappers for all those functions whose signature allows the overlying software to pass a pointer for a memory write operation to be performed by the library. Those wrappers simply throw back the call to the underlying standard-library function, and then pass control to the memory map manager with explicit indication of the address of the updated buffer, and the size of the updated memory block. In case the size cannot be retrieved by the library function signature (as for pointers to buffers used for strings), the memory map manager is provided with a special flag, which triggers the manager to update the dirty bits for all the currently allocated contiguous chunks starting from the pointed address. This is obviously a conservative way of managing the memory map which can only result in an increased log/restore overhead (due to the fact that some chunks that have not been really dirtied by the library are actually considered as dirty ones). Correctness is in no way touched given that the wrapped library functions are all stateless, thus posing no issue on the side of memory log/restore.

Anyway, we are currently working on techniques for application transparent management, and integration with Di-DyMeLoR, of all those library functions which explicitly allocate memory and/or have an internal state.

4 Related Work

In the optimistic simulation context, several solutions have been introduced for logging the whole state of a simulation object (at each event execution or after an interval of executed events) [8, 11, 13, 14], or incrementally logging modified state portions [15, 19, 22], or supporting a mix of the two approaches [9, 18]. With these solutions there is the need (i) to supply the necessary code to collect snapshots of the objects' state inside the application level software, or (ii) to employ calls to functions within the API of proper checkpointing libraries, or (iii) to statically identify (e.g. at compile-time) which portions of the address space need to be considered part of the state. Consequently, perfect transparency is not supported since the programmer must necessarily be faced with issues related to state snapshots. Also, static identification of the memory locations to be included inside the snapshot is non-compatible with dynamic memory allocation/deallocation (e.g. via standard libraries) at the simulation object level. This is the case for the work in [22], which has some technical similarities to our work on the side of automatic instrumentation, but does not allow dynamic memory to be employed, thus not supporting recoverability for each permitted operation (allocation, deallocation and update). Compared to all those approaches, our solution supports state management, based on incremental log capabilities, without the need for specific log/restore modules within the application code, or for explicit interfacing with log/restore libraries, and allows the simulation object state to be scattered on dynamically allocated memory chunks.

The issue of dynamic memory based states for optimistic simulation objects has also been addressed by the optimistic simulation frameworks in [3, 7]. However, ad-hoc

[2]Flexibility in memory management via partitioning/aggregating free memory buffers according to the so called "boundary tagging" scheme [2] is anyway inherited by Di-DyMeLoR thanks to per-chunk headers used at the level of the underlying malloc library.

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. Hence, differently from our approach, dynamic memory based layouts via ANSI-C memory allocation/deallocation services are not supported.

In terms of capabilities of the memory management subsystem the closest works to our approach are probably the ones in [16, 17], which present software layers for transparent log/restore in optimistic simulation based on the High-Level-Architecture (HLA) interoperability standard. These layers rely on Operating System memory protection mechanisms to detect memory updates and to incrementally log dirty pages belonging to a dynamically changing federates' memory layout. Compared to our proposal, the overhead for tracking updates and incremental log operations is likely higher (e.g. since it exhibits page size granularity) and affordable only when comparable with the cost of interoperability services supported by HLA middleware.

Some recent advances [5] have shown the viability and effectiveness of optimistic state management via reverse computation, where a reverse version of application level simulation code is employed for backward computation aimed at restoring the state of the simulation object. Anyway, in general simulation contexts (e.g., possibly exhibiting non-reversible execution paths), this approach still needs to be complemented via optimized log/restore techniques like the one we have presented in this work.

Our proposal is also related to a number of works in the field of program execution tracing (see, e.g., [1, 4, 12, 23]) for debugging, vulnerability assessment and repeatability. These approaches provide detailed analysis of changes in the state of the program, and of the execution flow. However, this is achieved via performance intrusive techniques relying on dynamic instrumentation and/or kernel level services, unsuited in contexts (e.g. parallel simulation) where performance cannot be sacrificed. Debugging supports showing basic operating mode comparable to our one (namely, the employment of trap mechanisms based on code insertion/replacement to detect memory write accesses) are those addressing data watch points (see, e.g., [21]). However they have performance targets different from ours since optimizations mostly cope with search techniques for verifying whether a memory reference falls inside a region that is currently subject to a watch point. In other words, aspects related to the identification of areas that have been dirtied and to incremental log/restore operations are not considered.

5 Experimental Data

We have integrated Di-DyMeLoR within the open source ROme OpTimistic Simulator (ROOT-Sim) [6]. This is a traditional-style (event-handler based) optimistic simulation platform, which transparently supports all the mechanisms associated with parallelization (e.g. the mapping of simulation objects on different kernel instances) and optimistic processing. It relies on MPI for data exchange across different simulation kernel instances.

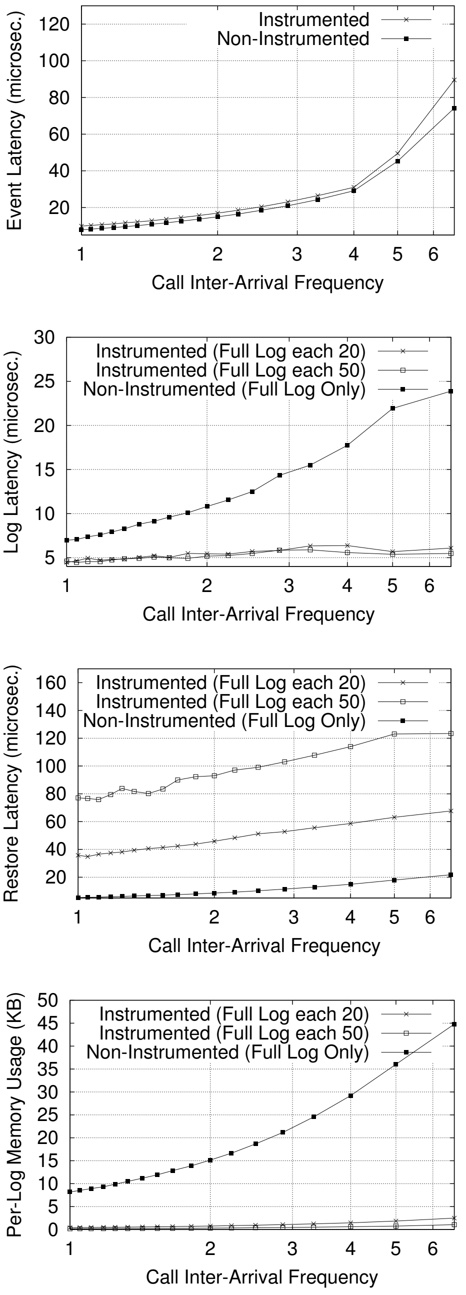

Figure 2. Basic Statistics for the Test-bed Configuration.

The hardware platform used in this experimental study is a Quad-Core machine equipped with four 2.4-GHz/4MB-cache 64-bits Intel processors and 4 GB of RAM memory, running LINUX (kernel version 2.6.22). Each core hosts one instance of the optimistic simulation kernel.

The test-bed application software is a parameterizable cellular system simulator, explicitly modeling fading and channel interference phenomena [10]. Each simulation object instance models a single cell, by tracking, via dynamically allocated data structures, channel allocation and power management information for ongoing calls. Specifically, upon the start of a call destined to a mobile device currently hosted by the cell, the simulation object allocates a new call-setup record via a couple of dynamically allocated data structures, and links it to a list of already active records. Each record gets released when the corresponding call ends or is handed-off towards a different cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call-setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the currently setup call to achieve the threshold-level SIR value, according to GSM technology. Data structures keeping track of fading coefficients are also updated while scanning the list. We have simulated macro-cells, each one managing up to 1000 wireless channels, using classical settings such as exponential distribution of the call inter-arrival time, and average call duration of 2 minutes. Also, the call inter-arrival frequency to each cell has been varied in the interval between 1 and 6.25 calls per simulation time unit, thus providing increasing values of the channel utilization factor (in between 12% and 75%), and hence increasing values of the expected length of the aforementioned list of in-use records. This has a twofold effect: (1) The storage requirement for the state of each simulation object varies in between 4KB and 32KB (meta-data for the maintenance of the memory map being excluded). (2) The event granularity grows from finer to coarser values. These variations allow us to evaluate the effects of the innovative capabilities provided by Di-DyMeLoR in differentiated configurations.

For instrumented and non-instrumented software we comparatively report in Figure 2 the measured values for the below parameters, with measures obtained for a small-sized test-bed benchmark configuration formed by four simulation objects (each one hosted by one instance of the simulation kernel running on the Quad-Core machine): (A) The average latency for the execution of a simulation event. (B) The average latency for a log operation. (C) The average latency for restoring the memory map to a logged state. (D) The average size for a taken log.

As pointed out in Section 3.2.2, the latency of a state restore operation in Di-DyMeLoR directly depends on the interleaving between full logs and incremental logs along the log chain. Hence, for the parameters in points (B), (C) and (D), the plots refer to different interleaving steps between full and incremental logs, namely incremental logs taken at each event and full logs taken every 20 and every 50 log operations, respectively.

By the results, we see that the overhead caused by the memory update tracking mechanism on the event execution latency is very limited. Also, CPU and memory requirements for each log operation in the instrumented case are definitely lower than those observed for non-instrumented software. The latter configuration actually provides a gain

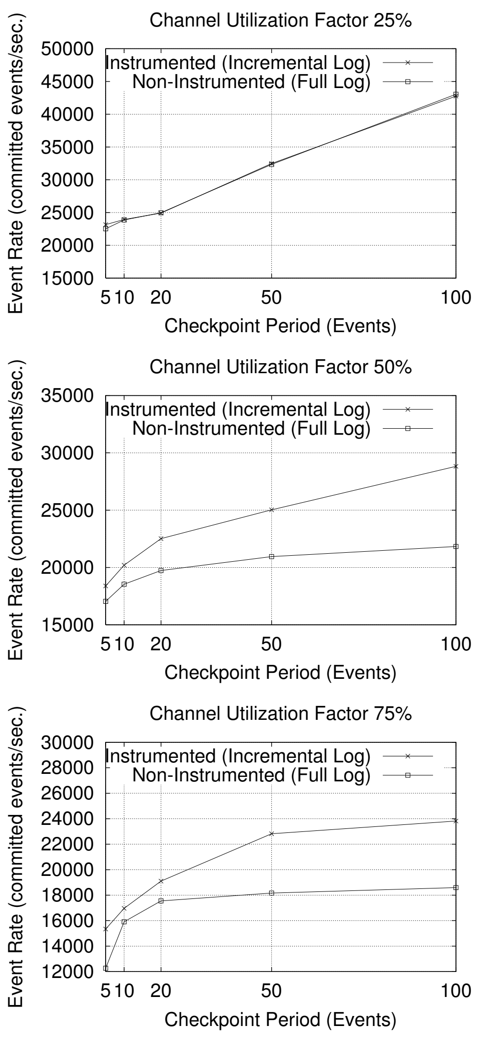

Figure 3. Event Rate vs the Checkpoint Period for Three Different Channel Utilization Factor Values (1024 Simulation Objects).

for state restore operations. Anyway, by the plots we see that performance decrease in the state restore for the case of instrumented software can be controlled (while maintaining the advantages on the side of logging) via proper selection of a non-oversized interleaving step between full and incremental logs.

In order to assess the overall benefits provided by Di-DyMeLoR, in Figure 3 we also report plots related to the event rate (committed events per wall-clock time unit) while varying the state log interval, also known as checkpoint period. This time the curves refer to a much larger configuration of the aforementioned benchmark, with 1024 macro-cells evenly distributed on the four simulation kernel instances hosted by the Quad-Core machine.

Concerning the checkpoint period, namely the independent parameter, we recall that it directly affects the trade-off between state log and coasting forward overheads, and potentially affects the overall memory locality due to variations in the memory usage for checkpoints. For Di-DyMeLoR, a state restore (occurring before coasting forward, if any) additionally depends on the interleaving step between full and incremental logs, which has been set to 20 in this study. Also, the plots are reported for three different channel utilization factors (25%, 50% and 75%) in order to observe the overall performance while varying the application software CPU/memory requirements.

By the results we see that, as soon as the application exhibits non-minimal memory requirements (namely when the channel utilization factor is non-minimal, hence inducing an increase in the simulation object state size), the incremental approach provides significant performance advantages. We note that these advantages come from a direct reduction in the cost of state logs (this can be noted especially for small values of the checkpoint period) and from increased locality (this can be noted especially for increased values of the checkpoint period, where the event rate curve for the incremental case does not stay flat, as instead occurs for the non-incremental case). Concerning the latter point, the frequency of GVT calculation and related memory recovery operations has been set in a way to never exceed 60/70% of RAM usage, so not to incur swapping phenomena that would alter the reliability of the reported measures. Hence, improved locality does not even include potential (further) advantages from incremental logging thanks to the avoidance of swap phenomena, which are more likely to occur with non-incremental logs in case of (excessively) lazy settings for GVT operations.

6 Summary

In this paper we have presented a software architecture complementing an existing open source layer supporting transparent log/restore operations for optimistic simulation objects with state layout based on standard dynamic memory allocation/deallocation services. The new capabilities added through the presented architecture entail: (i) Lightweight run-time monitoring mechanisms (based on ad-hoc software instrumentation facilities) for tracking memory update references inside the current memory map associated with the state of each simulation object - (ii) Optimized log/restore based on incremental copies of dirty chunks inside the memory map. Some experimental results have also been reported for an evaluation of the benefits achievable through the provided approach. Planned future work encompasses: (A) Supports for completely transparent state management, based on incremental logging, in the context of application software integration with third party libraries - (for which we have currently provided a partial solution only in case of coping with stateless libraries, and libraries not directly interacting with lower level memory allocation/deallocaton APIs) - (B) The design of (autonomic) mechanisms for dynamic switching between incremental and non-incremental operating modes, in order to further improve the system run-time behavior by optimizing the trade-off between the cost of memory update tracking (to be paid in case the incremental mode is switched on) and the cost of (full or incremental) log operations.

References

[1] GDB: The GNU Project Debugger. *http://www.gnu.org/software/gdb/.*

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

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

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[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, 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] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: a Time Warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 1332–1339, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[8] J. Fleischmann and P. 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.

[9] S. Franks, F. Gomes, B. Unger, and J. Cleary;. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 72–79. IEEE Computer Society, June 1997.

[10] S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

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

[12] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO*, pages 135–148, 2006.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] J. Steinman. Incremental state saving in SPEEDES using C plus plus. In *Proceedings of the Winter Simulation Conference*, pages 687–696. Society for Computer Simulation, 1993.

[20] 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.

[21] R. Wahbe, S. Lucco, and S. L. Graham. Practical data breakpoints: Design and implementation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 1993.

[22] 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.

[23] Q. Zhao, R. M. Rabbah, S. P. Amarasinghe, L. Rudolph, and W.-F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In L. J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2008.