Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

ASSESSING LOAD SHARING WITHIN OPTIMISTIC SIMULATION PLATFORMS

Roberto Vitali, Alessandro Pellegrini, Francesco Quaglia

High Performance and Dependable Computing Systems (HPDCS) Research Group
DIS - Sapienza, University of Rome
00185 Rome, Italy

ABSTRACT

The advent of multi-core machines has lead to the need for revising the architecture of modern simulation platforms. Specifically, recent proposals attempted to explore the viability of load-sharing for optimistic simulators run on top of these types of machines. In this article, we provide an extensive experimental study for an assessment of the effects on run-time dynamics provided by a load-sharing architecture that has been implemented within the ROOT-Sim package, namely an open source simulation platform adhering the optimistic synchronization paradigm. The experimental study, which is essentially aimed at evaluating possible overheads by the load sharing architecture, has been based on differentiated workloads allowing us to generate different execution profiles in terms of, e.g. granularity/locality of the simulation events.

1 INTRODUCTION

Multicore/multiprocessor machines have become a wide-spread reality, and the possibility to get access to these platforms at low costs is growing up to a greater extent, placing the need for a systematic approach concerning computational power usage, which is a key factor in high performance simulations. This has posed the need for re-evaluating the design and the architectural organization of high performance simulation platforms.

These platforms have historically been based on the partitioning of the simulation model into several distinct objects, handled by Logical Processes (LPs) (Fujimoto 1990), which are allowed to concurrently execute simulation events on, e.g., clusters of machines. On the other hand, the typical organization of the underlying simulation-kernel layer has been based on a multi-process paradigm, where each process runs in single-threaded mode and schedules its locally hosted LPs for event execution according to a time-interleaved scheme, resembling CPU-scheduling schemes typical of operating systems' technology targeted at single-core machines. For this type of organization, (dynamic) unbalance of the workload, associated with (dynamic) changes of the computational power demand across the LPs, has been tackled via load-balancing approaches targeted at migrating LPs from the overloaded simulation-kernel instances to the underloaded ones. Examples of this type of approaches, showing differentiated levels of transparency towards the application-level software, can be found in (D'Angelo and Bracuto 2009, Glazer and Tropper 1993, Carothers and Fujimoto 2000, Peluso, Didona, and Quaglia 2011)

On the other hand, the multicore/multiprocessor organization characterizing modern hardware platforms offers new potentialities that could be fully exploited by abandoning the (simple) single-threaded programming approach in favor of multi-threaded programming paradigms. Along this direction, a recent achievement presented in (Jagtap, Abu-Ghazaleh, and Ponomarev 2012) has shown how the reshuffle of multi-process optimistic simulation platforms to multi-threaded versions (particularly for the case of the ROSS open source platform (Carothers, Bauer, and Pearce 2000)), can provide noticeable benefits in terms of performance thanks to the optimization of cross simulation-kernel communication. In particular, the conjunction of (a) the usage of shared-memory to support message passing and (b) the reliance on a coherent view of virtual-addressing across the threads operating within a same process (each one implementing an instance of the simulation-kernel), has lead to a significant reduction of the amount of data to be copied when transmitting/receiving a message. However, the architectural organization proposed in (Jagtap, Abu-

Ghazaleh, and Ponomarev 2012) still relies on the traditional view according to which one instance of the simulation kernel runs as a single thread, thus having (at most) a single CPU-core as the computational power assigned to it. Another attempt has been presented in (Li-li, Ya-shuai, Yi-ping, Shao-liang, and Ling-da 2011) where a global scheduling mechanism (based on a centralized event queue) is exploited in order to assign simulation work (i.e., event processing on different LPs) to different active threads within the same platform. However, this approach may suffer from reduced scalability, thus being suited for contexts where a reduced number of CPU-cores is available (it has been evaluated with no more than 8 CPU-cores).

A more recent advance we have proposed in (Vitali, Pellegrini, and Quaglia 2012a) introduces a paradigm shift in the design of optimistic simulation kernels by providing a reference architecture based on a symmetric multi-threaded approach. Here, each instance of the optimistic simulation-kernel is able to run multiple worker threads, which can take care of the execution of whichever locally hosted LP. According to this organization, the number of worker threads within each kernel instance can be dynamically scaled up/down in a seamless manner, which opens the possibility for a new approach to the usage of the computational power. Specifically, for dynamically changing computational requirements across the LPs, re-balanced runs can be achieved by scaling down the amount of worker threads operating within under-loaded kernel instances and scaling up the number of worker threads operating within over-loaded instances. Hence, *load-sharing* policies are actually pursued, meaning that the whole computational load is shared across all the available CPU-cores which are dynamically bind to one kernel instance or the other depending on the aforementioned changes in the application requirements (and associated amounts of worker threads per kernel instance). Overall, with this approach LPs' migration facilities are no more mandatorily required in order to optimize resource usage in multicore/multiprocessor contexts, since the load-sharing approach exploits the orthogonal concept of computational power migration at the simulation-kernel level. This can not only provide different (hopefully better) tradeoffs between housekeeping overhead and resource exploitation for productive work, but can also help application transparency since the application programmer will be no longer requested to provide application level modules for, e.g., relocating the LPs across different simulation-kernel processes (hence across different address spaces), which is generally not transparently supported except for a few advanced state management architectures (Peluso, Didona, and Quaglia 2011).

Early experimental data have shown how the load-sharing architecture depicted in (Vitali, Pellegrini, and Quaglia 2012a), and implemented within the ROme OpTimistic Simulator (ROOT-Sim) package (Quaglia, Pellegrini, and Vitali 2011), namely an open source general purpose simulation platform adhering to the optimistic synchronization paradigm, can provide low overhead (e.g. for synchronizing worker threads executed within a same simulation-kernel instance) while allowing performance optimizations for dynamically changing workloads.

In this article we complement such an early study by providing an extensive experimental characterization of the effects of the load-sharing architecture when considering differentiated application-level settings, which are essenitally aimed at determining potential sources of ovehrheads by the load sharing organization. In particular, we report a set of measures that, in a complementary manner to the coarse grain measures reported in (Vitali, Pellegrini, and Quaglia 2012a), allow a fine grain comparative analysis of the actual run-time dynamics of the load-sharing architecture and of those achievable with a traditional multi-process organization of the simulation platform.

The remainder of this paper is organized as follows. In Section 2 we provide an overview of the organization of the load-sharing architecture. A discussion on the structure of the experimental assessment, and on related target parameters to be observed, is provided in Section 3. The results of the experimental study are presented in Section 4.

2 OVERVIEW OF THE LOAD-SHARING ARCHITECTURE

As hinted, the load-sharing architecture we have presented is based on a symmetric multi-threaded approach where each worker thread running within each single simulation-kernel instance has the ability to execute both in application- and kernel-mode, and can control and take care of the execution of whichever locally hosted LP. This type of approach has relations with what happens in operating systems targeted at multicore/multiprocessor machines, where the CPU-scheduler controlling a specific CPU-core is generally allowed to dispatch whichever ready-to-run thread. On the other hand, different LPs may mutually issue

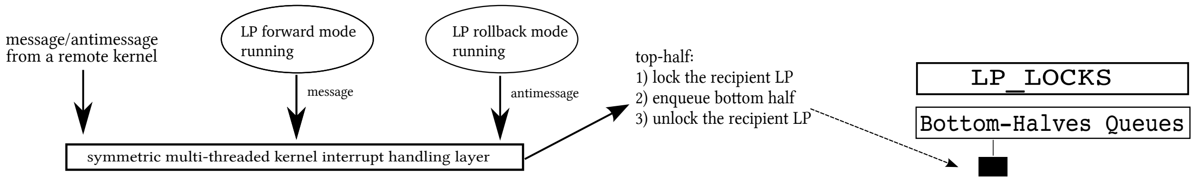


Figure 1: Outline of the Top/Bottom-Halves Architecture.

interactions, e.g., via event scheduling services, that are actually supported by entering the kernel mode along the thread taking care of running the LP that issues the interaction.

However, while the execution in user mode intrinsically relies on data partitioning, since, in compliance with the original specification of the optimistic synchronization protocol, namely the Time Warp protocol (Jefferson 1985), each LP operates on private (per-LP) data structures implementing its current state within the simulation run, this is not the same when worker threads operate in kernel mode. In particular, kernel-mode execution may require that a worker thread taking care of running LP_a needs to access the kernel level meta-data associated with whichever locally hosted LP_b, e.g., the event-queue associated with LP_b, given that LP_b may figure out as the recipient of an issued interaction, namely a newly scheduled event from LP_a. Depending on whether a different worker thread is allowed to concurrently operating in kernel mode exactly on that queue, which is the case for the symmetric organization, a critical section must be guaranteed, requiring some form of synchronization. We note that, depending on proper dynamics related to the specific simulation model, such a situation may occur frequently, thus imposing the need for a properly tailored management of synchronization aspects within the symmetric multi-threaded organization.

In order to prevent kernel-level synchronization phases from becoming a bottleneck, in our load-sharing proposal we have devised that each cross-LP interaction logically represents an interrupt-event, which does not get atomically finalized upon its acceptance, hence avoiding the need for acquiring locks in target data structures according to a possibly adverse timing (e.g. when the data structure is already locked for the finalization of a concurrently issued interaction). Instead, the finalization of the interaction takes place by adhering to a top/bottom-half scheme, resembling the scheme typically used for the implementation of interrupt-drivers in operating systems targeted at multicore/multiprocessor machines.

A graphical representation of the outcoming architecture is provided in Figure 1. When an interaction is issued, a lightweight top-half module is executed which only inserts a bottom-half task into a queue associated with the destination LP, in order to allow finalization at a later instant of time, more conveniently wrt synchronization. We note that an interaction to be treated according to the top/bottom-half scheme may be issued in three different scenarios:

- When an LP runs in forward mode and produces events to be destined to other locally-hosted LPs.
- When an LP runs in rollback mode (i.e., the kernel layer is currently recovering its state due to the occurrence of a causality violation). In this case an interaction associated with an anti-event might be destined to some locally hosted LP ([1]).
- When the messaging layer locally notifies an interaction, namely an event or an anti-event, whose source LP is hosted by a different simulation-kernel instance.

Basically, our approach can be supported by relying on a spin-lock array, named `LP_LOCKS`, having one entry for each LP hosted by the multi-threaded simulation-kernel. `LP_LOCKS[i]` is used to implement a fast critical section for the access to the bottom-half queue associated with the i-th LP hosted by the kernel, either for inserting a new bottom-half task to be eventually flushed, or for taking care of unlinking the current chain, in order to flush the pending bottom-halves.

With this organization, each worker thread can (in principle) take care of flushing the pending interactions currently recorded within the bottom-half queue of any LP, which can be done at convenient time instants, namely when no other worker thread is already doing this same job. In this way, wait-phases for exclusive

[1]An anti-event, also known as anti-message, is a *negative* copy used to annihilate a previously issued interaction, namely an already scheduled event. Anti-events are used to propagate the effects of causality errors across the LPs by retracting events scheduled during the causal inconsistent portion of the simulation.

accesses/updates to the kernel-level data structures associated with whichever LP get eliminated. The support for this type of operation would simply consist in an additional per-LP spin-lock, accessible in non-blocking mode. Successful non-blocking access would give rise to the possibility for the worker thread to operate the flush of the bottom-half queue for that LP. On the other hand, unsuccessful non-blocking access would simply tell the worker thread to skip taking care of that LP.

Additionally, in order to improve the efficiency of the caching hierarchy, we have proposed a temporary binding mechanism that assigns each worker thread the responsibility to flush the bottom-half queues of only a subset of the locally hosted LPs. Such a thread also has the responsibility to dispatch its bound LPs for event execution in time-interleaved mode. In this way, direct cache contention while handling user level or kernel level execution of the LPs may only occur for LPs bound to the same worker thread (hence operating on top of the same CPU-core), depending on the interleave of the operations performed by the worker thread (e.g., forward execution operations) related to one or the other of these LPs. In other words, cache contention across different worker threads (e.g., mutual cache invalidation) may only occur in relation to the access/manipulation of kernel-level synchronization data structures, such as the bottom-half queues. In (Vitali, Pellegrini, and Quaglia 2012a), rules for periodically selecting the LPs to be bind to a specific worker thread are defined, together with some rules adopted to reassign the computational power (i.e., the CPU-cores) to the different simulation-kernel instances by scaling up/down the amount of worker threads per instance. These are particularly suited for the case of dynamic workloads.

3 OVERVIEW OF THE EXPERIMENTAL ASSESSMENT

In this section we provide an overview of the experimental assessment we have performed of the load-sharing architecture. Actually, what we will present are data related to a specific implementation of this architecture which has been integrated within the ROOT-Sim package (Quaglia, Pellegrini, and Vitali 2011). As a consequence, beyond the evaluation of the general mechanisms at the base of the load-sharing approach, we will also focus on specific effects due to the integration of load-sharing within ROOT-Sim since, as we will show, this imposes some constraints on run-time dynamics properly related to specific ROOT-Sim's subsystems.

Anyway, all the parameters that will be object of experimental assessment will be discussed in this section, by also motivating why they have been selected in the analysis. To this end, a brief recall on the structure and capabilities of ROOT-Sim are presented, so to provide the basis for easing the comprehension of the discussion. As a final preliminary note, our reference architecture for the assessment will be represented by the original multi-process version of ROOT-Sim.

3.1 The architecture of ROOT-Sim

ROOT-Sim is an open source C/MPI-based simulation package targeted at POSIX systems, which implements a general-purpose parallel/distributed simulation environment relying on the optimistic (i.e., rollback-based) synchronization paradigm. It offers a very simple programming model based on the classical notion of simulation-event handlers (both for processing events and for accessing a committed and globally consistent state image upon GVT ([2]) calculations), to be implemented according to the ANSI-C standard, and transparently supports all the services required to parallelize the execution.

As for management and recoverability of LPs' state, which are crucial aspects for the design of effective optimistically synchronized environments, two main architectural approaches have been adopted. First, dynamic memory allocation/release by the application, performed via the standard malloc library, are *hooked* by the kernel and redirected to a wrapper. Second, the simulation platform is "*context-aware*", i.e., it has an internal state which distinguishes whether the current execution flow belongs to the application-level code or the platform's internals. In the former case, the hooked calls are redirected via the wrapper to an internal Memory Map Manager (called DyMeLoR), which handles per-LP allocation/deallocation operations by maximizing memory locality of the state layout for each single LP, and by maintaining meta-data identifying the state memory map and making it correctly recoverable to past values (Toccaceli and Quaglia 2008).

[2]GVT - Global Virtual Time - represents the commitment horizon. No causality violation can even occur for processed events whose timestamp falls before the current GVT value. GVT updates typically trigger memory recovery procedures, e.g., of obsolete state logs.

Concerning GVT calculation, ROOT-Sim relies on an optimized asynchronous approach based on a message acknowledgment scheme to solve the well-known transient message problem. Within this scheme, each kernel instance keeps track of all the messages sent to the other instances in an aggregate manner (i.e., via counters). Also, to reduce the communication overhead, each instance sends cumulative acknowledgment messages according to a window-based approach. Finally, to overcome the simultaneous reporting problem, each kernel instance temporarily stops sending acknowledgment messages during the execution of the GVT protocol.

ROOT-Sim also supports a very peculiar service that, once a new GVT value is available, transparently rebuilds a Committed and Consistent Global Snapshot (CCGS), formed by a collection of individual LPs' states (Cucuzzo, D'Alessio, Quaglia, and Romano 2007). This occurs via update operations applied to local committed checkpoints of individual LPs so to eliminate mutual dependencies among the final-achieved state values. Once the CCGS is built, each LP gains control via an ad-hoc callback within the API, by also having access to the copy of its state image belonging to the CCGS. Such a service can support, e.g., termination detection schemes based on global predicates evaluated on a committed and consistent global snapshot.

3.2 Evaluating the effects on caching and memory accesses

It is clear that the different internal organization of the load-sharing architecture can impact locality, which may give rise to variations of the effectiveness of the caching hierarchy. This may occur, e.g., due to the presence of kernel-level data structures shared across multiple threads, which are instead avoided in traditional multi-process platforms. In addition, we note that concurrent accesses can produce a strong impact on bus contention, due to locking operations needed for synchronizing threads' execution in critical sections. In order to provide quantitative data related to potential variations of the execution locality and its effects, we have decided to focus on three parameters:

- The latency for taking a checkpoint of the LP state.
- The latency for reloading a previously taken checkpoint in case of rollback.
- The event execution latency.

The first two parameters are associated with memory intensive operations, since each log or restore operation entails spanning across the LP's state or the log buffer in read mode. They represent therefore a good test case for determining how efficiently these read operations are supported thanks to the effects of the caching hierarchy.

On the other hand, the event execution latency is a reflection of the locality expressed by the application, and of how well such a locality is supported via the caching system. Hence we have decided to include also this parameter in the assessment of the effects of the load-sharing architecture.

In addition, we have decided to measure scheduling operations' latency in order to assess the effects of multithreading on data structures which are sparsely accessed during the simulation's execution. We have explicitly decided to rely on a $O(n)$ Shortest Timestamp First (STF) scheduler, which determines which is the next event to be processed by going over every LP's input queue for selecting the event associated with the minimum LVT among the pending events for all the LPs currently hosted by the simulation kernel. We consider this to be a significant measure for a large set of operations which are essential in a simulation platform, such as message queues scanning, log chains traversing, and other ones which rely on many data structures, thus providing a viable study to assess optimistic simulation platform's access pattern operations' dynamics.

3.3 Evaluating the impact on MPI operations

If LPs' interaction is related to those hosted by different kernels, instead of relying on the top/bottom-half scheme, a message is presented to the MPI layer. Given that MPI does not support multi-threading, accesses have been serialized by exploiting again critical sections supported via spin-locking. The same has been done for probing MPI and issuing message receive operations by the worker threads, which are ultimately reflected in the execution of a top-half module (see again Figure 1).

Clearly this approach may induce delays on the worker threads when compared to the multi-process scheme. However we note that when the kernel is organized according to the multi-threaded scheme, we expect a reduced amount of interactions to be handled via MPI. This is because, once fixed the total amount of threads (and hence of CPU-cores) running the simulation platform, in either multi-process or multi-threaded mode, there is a non zero likelihood that two LPs hosted by different kernel instances within the multi-process organization are hosted by the same kernel instance when running in multi-threaded mode. Hence the mutual interactions between these LPs will not require passing via MPI, instead they will be handled via the top/bottom-half architecture.

Overall, to account for the above effects we have decided to evaluate:

- The time spent while interacting with the MPI layer.
- The time spent while managing the data structures supporting interactions via the top/bottom-half architecture, which, we recall, might include the time spent while synchronizing concurrent worker threads within the access to bottom-half queues.

A joint analysis of the two above parameters would allow understanding dynamics related to the actual handling of the interactions across the LPs involved within the simulation model.

We note that a possible approach to reduce the synchronization costs in the load-sharing architecture while interacting with the MPI layer would be represented by message aggregation. In fact, messages (namely events and anti-events) destined to remote multi-threaded kernel instances could be aggregate into local buffers and only periodically sent towards the destination. This can reduce the frequency of interactions with the MPI layer, thus favoring a reduction of the overhead when considering the case of synchronized accesses to MPI by multiple worker threads. Given that we have not yet adopted a similar approach, for what concerns the interaction with MPI, the assessment we will provide can be related to a kind of worst case architectural configuration.

3.4 Evaluating the impact on GVT and global snapshot operations

As for the computation of GVT, in the symmetric multi-threaded version of ROOT-Sim the GVT subsystem has been modified in order to account, within the global reduction determining the new GVT value, for the timestamps of events/anti-events that have not yet been reflected into the event queues of the recipient LPs due to the fact that they are still pending within bottom-half queues. These events/anti-events represent a sort of in-transit information, exhibiting similarities (and hence requiring similar management approaches) with traditional in-transit messages travelling via the messaging subsystem (MPI in our case) across different kernel instances.

Beyond the above issue, another aspect that required relatively significant intervention while integrating the load-sharing approach within ROOT-Sim is related to the CCGS subsystem. As hinted, this subsystem is in charge of reconstructing, upon GVT calculations, committed and consistent global states, formed by collections of individual LPs' states. These individual states are then passed in input to an application level callback where the programmer is allowed to inspect the committed computation results and to perform unrecoverable actions/operations, such as I/O.

In the original multi-process version of ROOT-Sim, each active thread, individually representing an active kernel instance, is allowed to process those callbacks since they are intrinsically sequentialized along the execution of that same thread. Instead, for the symmetric multi-threaded organization, the active worker threads are not all allowed to do this same job since this would lead to inconsistencies on the content of the (default) file used for tracing the output on each kernel instance. As a consequence, we have decided to synchronize all the worker threads operating within the same kernel instance in such a way to allow a single worker thread to run CCGS facilities. This reduces the computational power of the load-sharing architecture during the phases where the CCGS protocol is run. Hence we have decided to report in the assessment the latency observed when running GVT plus CCGS protocols upon committing a new portion of the simulation in order to quantify this phenomenon.

3.5 Evaluating the effects on the overall rollback pattern

The proposed approach for sharing the load in terms of computational resources reassignment among the various instances of simulation kernels being run might affect rollback patterns. In fact, since a rollback operation happens upon receiving an out-of-order event to be executed, we note that load-sharing architectures can benefit from the fact that usually the gap between different LPs' local clocks is due to a different workload being processed. Therefore, if the workload is dynamically redistributed evenly among the various simulation kernel instances being run, since events' scheduling is performed according to a STF policy, local clocks are expected to diverge less, and in case a rollback operation must be performed, the rollback length (i.e., the amount of executed events which must be undone in order to reach the correct LVT to restart the execution from) is expected to be reduced.

In order to evaluate the secondary effects by our architecture on the rollback pattern, we have explicitly measured:

- Rollback probability, measured as the ratio between rollback operations and executed events.
- Rollback length, expressed in average number of undone events
- Efficiency, which is measured as the ratio between committed and executed events.

4 EXPERIMENTAL RESULTS

4.1 Benchmark Applications and Setting

In order to significantly benchmark our load-sharing architecture and to show the actual costs of internal operations and data structures accesses supporting our load-sharing simulation kernel, we have relied on the PCS (*Personal Communication Service*) simulation model, thoroughly described in (Vitali, Pellegrini, and Quaglia 2012b)), a parameterizable GSM communication model — explicitly modeling cells' response upon different call arrival rate — which produces a workload relatively uniform across the various LPs, with a communication pattern which involves sending messages only to each LP's neighbours.

Calls inter-arrival time (τ_A) is exponentially distributed, and average duration is set to 2 minutes. Three different configurations of the model have been executed, namely with τ_A set to 0.4, 0.8, and 1.2 respectively, to achieve channel utilization factor on the order of 35%, 15%, and 10% respectively, while the residence time of an active device within a cell has been set a mean value of 5 min and follows the exponential distribution.

For the above scenario, we have run experiments with 1024 wireless cells, modeled as hexagons covering a square region, each one hosting 1000 wireless channels. The checkpointing interval χ has been set to a value of 20, in order to avoid fluctuations in the resulting assessment, due to self-adjusting policies, e.g., autonomic ones. In order to clearly show the actual overhead due to the load-sharing architecture, we have run our experiments in a static fashion, i.e. by forcing the power reassignment procedure within our kernel not to modify the initial even allocation of worker threads to kernel instances. This allows us to check what is the overhead associated with monitoring, managing, and reassignment operations without any benefit from the actual load sharing approach.

We have run our set of experiments on a HP ProLiant 64-bit NUMA server equipped with four 2GHz AMD Opteron 6128 processors and 64GB of RAM. Each processor has 8 cores (for a total of 32 cores) that share a 10MB L3 cache (5 MB per each 4-cores set), and each core has a 512KB private L2 cache. The operating system is 64-bit Debian 6, with Linux Kernel version 2.6.32.5.

4.2 Results

4.2.1 Assessing Operation Costs

Top/Bottom Halves Processing Recalling that a higher number of worker threads is related to a lower number of concurrent kernel instances, since the number of LPs handled by the simulation system is constant, the logical contention on data structures increases.

In order to ensure correctness, whenever a top/bottom half operation must be performed by some worker thread, a lock on the per-LPs queue must be taken (although in the case of bottom-halves processing, the lock is only needed for dequeueing an event from the list). If the number of concurrent worker threads

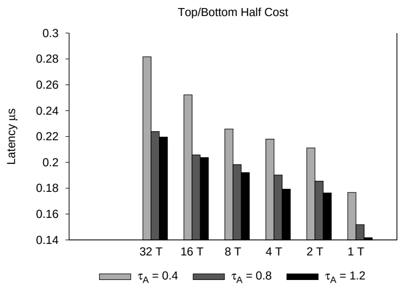


Figure 2: Top/Bottom Halves

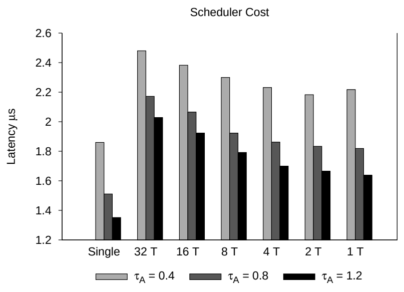


Figure 3: Scheduling

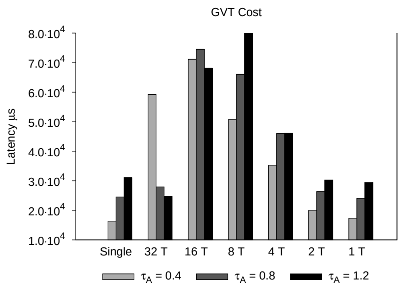
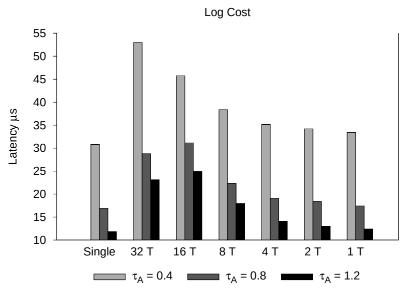


Figure 4: GVT and CCGS



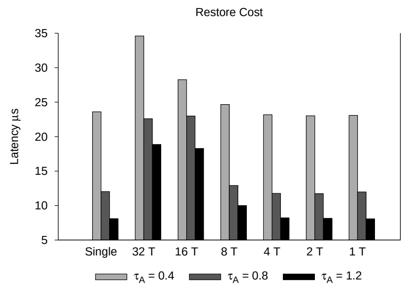


Figure 5: Log/Restore Operations

grows, the contention on the queues is increased, since the worker threads synchronizing on this resource must wait for the lock to be granted to them. At the same time, since a higher number of available worker threads entails a higher number of handled LPs, this statistically reduces contention on per-LP queues, so that this latency is expected to grow, but up to a certain (not large) extent. Additionally, in our implementation we explicitly relied on pre-allocation for reserving buffers used to keep track of bottom halves. This choice is guided by the fact that relying on the malloc library to allocate nodes can result in a costly operation when executed in a multi-threaded environment, since its internal synchronization relies on futexes. If the size of the pre-allocated buffer is well-tuned, the contention on top-half registration is reduced to the minimum.

In Figure 2 we show the per-event latency related to the management of the top/bottom halves. By the plots we see that when the number of per-kernel worker threads increases, the related cost increases just linearly and moderately, given the above considerations.

Scheduling In Figure 3, the per-event latency related to scheduling operations is provided. By the plots, we can see that the non-multithreaded implementation (which we refer to as "Single") shows a latency which is smaller than the load-sharing one on the order of 15%. This small overhead is related to the fact that the load-sharing architecture implements a mapping between LPs handled by a certain worker thread and the actual thread. In order to access data structures related to LPs to perform scheduling operations, a worker thread must first check which are the LPs it is currently handling. This is an operation which is not executed in the non-multithreaded implementation. Nevertheless, this difference is not enough to justify a 15% latency increase: In fact, we note that a significant additional difference between the two implementation relies in the fact that the load-sharing architecture has a large use of locking primitives for ensuring correctness. This entails a higher number of in-memory accesses for trying to acquire spinlocks, which in turn increases memory bus contention[3] and can affect procedures which access large data structures sparsely, as the scheduling operation does.

Log/Restore In Figure 5(a) we present the single log operation cost. By the plots, we can see that, independently of the workload, a higher number of worker threads (i.e., a smaller number of concurrent

[3]We note that this result is expected to be different on hardware architectures which rely on *cache locking*, i.e., a cache-coherence protocol which ensures atomicity of in-cache operations by relegating accesses to the highest available levels, therefore avoiding bus contention if data accessed by other threads is not related at all.

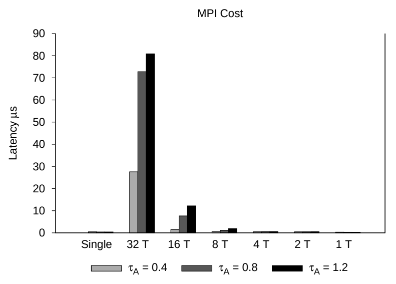


Figure 6: Inter Communication

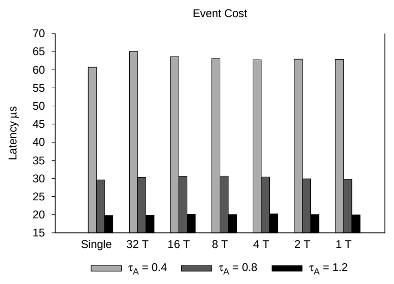


Figure 7: Event Execution

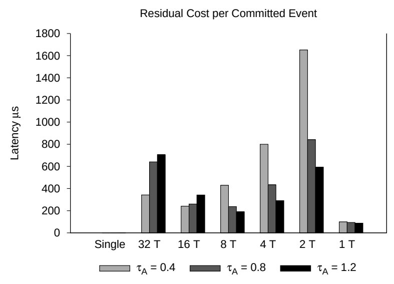


Figure 8: Residual Cost

kernel instances) presents a higher latency. In particular, the cost starts growing when running with 4 kernel instances, each one handling 8 worker threads, with the greatest difference on the order of 30% (15 μs). As mentioned before, we note that this latency increase wrt the number of available worker threads is influenced to the aforementioned internal synchronization at malloc library's level.

In Figure 5(b) the single restore operation latency is shown. Fluctuations between different workloads are on the order of 25%, which is due to the fact that a higher number of threads entails a higher number of in-cache buffers invalidations.

GVT and CCGS Computation Figure 4 shows the plots related to the GVT and CCGS execution latency. As hinted in section 3.4, the load-sharing version of our simulation kernel allows one single worker thread to perform CCGS operations, due to critical races on output which is allowed on the reconstructed committed state.

At the same time, during the GVT operations a procedure for computing the actual workload which the various kernel instances are following through is executed. This can be seen as a distributed agreement among the kernels to determine which is the best number of worker threads active on each instance to evenly share the current workload. Again, this procedure is based on MPI message exchanging.

By the plots, we can see that when running with a small number of worker threads the latency is comparable with the one presented by the single-threated one. On the other hand, a larger number of available worker threads entails a higher latency. This is related to the serialization required for consistently accessing even queues which are needed to compute the GVT reduction and to evaluate the future (expected) LPs' workload. Additionally, inter-kernel (MPI-based) communication is exploited to correctly follow through the distributed agreement on the best-suited number of worker threads. We additionally note that when running with 32 worker threads, the latency is reduced, since in this configuration there is no actual need to rely on MPI for executing the agreement procedure, since data structures are already locally available.

Inter-Kernel Communication As for Inter-Kernel Communications, by the plots in Figure 6 we note that the executions relying on 32 and 16 worker threads have a latency which is almost two and one orders of magnitude greater than other configurations, respectively. This produces a smaller throughput, as depicted in Figure 9. This is related to the fact that these configurations process a larger number of committed and uncommitted events (as reported in Figure 14), since most of them are rolled back. In fact, in Figure 12 we can see that these configuration show, among the others, the higher rollback probability, along with a non-minimal rollback length. This produces a low efficiency (which is reported in Figure 13).

Therefore, the high Inter-Kernel communication exhibited by these configurations is related to the high contention on the MPI component due to the large amount of message exchange which is related to a larger number of events and anti-events generated (for the 16 threads configuration), to a higher number of GVT phases as described in Section 4.2.1 (for both configurations), and for the higher number of MPI Probing (for both configurations).

Events' Execution In Figure 7 we show per-event execution latency. Although different values of the τ_A parameter produce different events granularities (due to the different simulation model's load, which produces a higher amount of data to be processed for power allocation computation and SIR ratio regulation), different configurations do not affect significantly the event's latency, with small differences in the order of 3-5%.

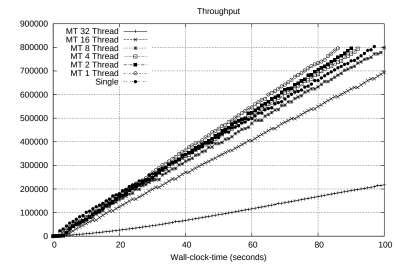


Figure 9: Throughput ($\tau_A = 0.4$)

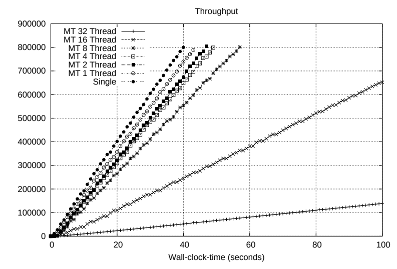


Figure 10: Throughput ($\tau_A = 0.8$)

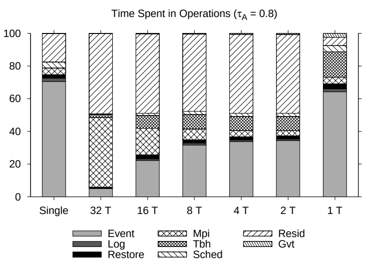
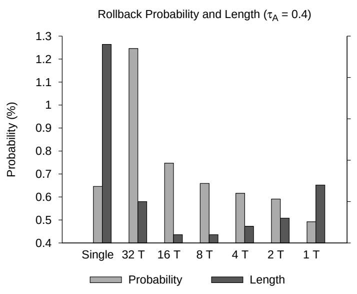
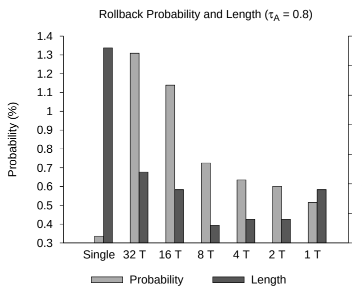


Figure 11: Operation Weights


(a) $\tau_A = 0.4$


(b) $\tau_A = 0.8$

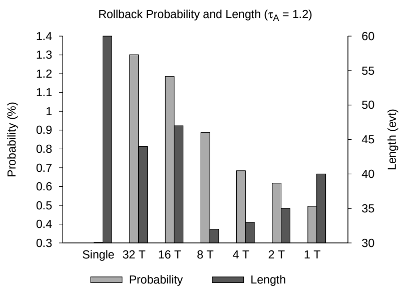

(c) $\tau_A = 1.2$

Figure 12: Rollback Probability and Length

This emphasizes the fact that the load-sharing architecture does not affects locality more significantly than the non-multithreaded one, as far as events' execution is concerned.

Residual Cost To complete the punctual assessment of our load-sharing architecture, in Figure 8 we present the plots for the residual cost, i.e. all the per-committed-event costs which do not appear in the above measurements. Above all, the residual cost shows which is the time spent for scanning/processing input/output-queues, and all other housekeeping operations needed to let the simulation correctly advance. These operations are performed by worker threads on a per-LP basis, but entail accessing memory sparsely, being subjected to secondary effects related to memory contention, as depicted in the previous analysis.

Among the housekeeping operations, the input queue management and the ack-handling subsystem have a great importance. As described before, the former entails a call to the malloc library for reserving memory buffers which are used to store messages destined to locally handled LPs. Concurrently requesting memory buffers to the malloc library involves synchronization mechanisms based on futexes, which are likely to increase the overhead related to the registration of messages. The latter is a subsystem which is in charge of facing the well-known transient message issue for GVT computation, a window-based ack mechanism has been adopted. The implementation relies on a lock for each time window (one per kernel), which is acquired during an update operation. As can be seen by the plot, the highest latency is associated with the two-worker-threads configuration. In fact, in this case there is a small contention wrt the number of threads, but since there are 16 kernel instances running, there is a higher message passing which entails trying to acquire the lock more frequently. The one-threaded configuration does not show this contention effect, while increasing the number of threads reduces the need to update the window, thus reducing the contention as well.

4.2.2 Operation Weights

To show how the so-far described costs impact on the overall performance of the load-sharing platform's execution, in Figure 11 we present an aggregation showing the percentage of time spent in the various operations, normalized on committed event's execution. By the plots we can see that the non-multithreaded execution, independently of the workload, spends almost 70% of the time in events' processing, as the load-sharing configured with one worker thread does. This measure enforces the idea that the load-sharing adds a minimal overhead wrt the non-multithreaded architecture.

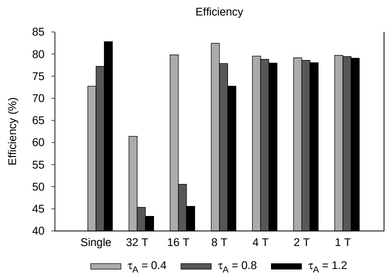


Figure 13: Efficiency

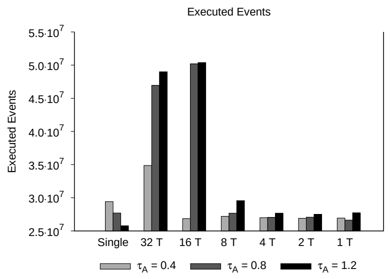


Figure 14: Executed Events

In addition, we can see that as the number of worker threads increases (i.e., we move towards a more parallel configuration) the amount of time spent in the top/bottom halves processing decreases, due to the fact that a larger number of LPs is handled by a single kernel instance. At the same time, inter-kernel communication decreases since a higher number of events can be delivered to local LPs. The 32-worker-threads configuration shows an amount of time spent in MPI operations which reduces to the minimum the time spent for event processing, as was already clearly illustrated in Section 4.2.1.

The high residual time's relevance in the load-sharing architecture running with more than one thread is again related to the window-based ack mechanism, as was depicted in the previous section.

4.2.3 Global Assessment

To assess our proposed architecture globally, we have measured the cumulated event rate (expressed as the amount of cumulated committed events per Wall-Clock-Time unit), which is a classical indicator of the speed of the optimistic simulation run.

In particular, Figures 9 and 10 show the throughput for the benchmark configurations related to the τ_A parameter set to 0.4 and 0.8, respectively. By Figure 10 we can see how the so-far discussed overheads produce a decreasing throughput when the number of worker threads is increased (we remind that in this configuration, the workload is constant and evenly distributed, and the rebalancing procedure is forced not to reassign worker threads to kernel instances, in order to evaluate which are the intrinsic costs of the presented architecture). As was explained before, the configurations associated with 16 and 32 threads do not scale, particularly because of the high costs related to MPI operations, due to a large amount of executed events which get discarded.

Figure 9 shows the configuration with a different (higher) workload. We note that some load-sharing configurations present a throughput slightly higher than the non-multithreaded version. This is related to benefits derived from a better exploitation of the caching architecture. Additionally, this is reflected in a higher efficiency, as depicted in Figure 13. As hinted, in this paper our focus is on the overhead analysis for the laod sharing arhitecure in balanced contexts. Experimental data related to the benefits from load sharing with dynamically varying workloads can be found in (Vitali, Pellegrini, and Quaglia 2012a).

5 CONCLUSIONS AND FUTURE WORK

In this work we have presented a deep study on runtime dynamics related to a symmetric multithreaded architecture for optimistic simulation kernels. In particular, the most typical issues related to concurrency have emerged, along with some secondary effects which can significantly effect overall performance.

By our experimental results, we proved that the overhead associated with the hereby-proposed symmetric architecture scales well wrt the number of worker threads used during the simulation, except for particular aspects like MPI communication. Nevertheless, this particular overhead can be faced with complementary techniques, like the one proposed in (Jagtap, Abu-Ghazaleh, and Ponomarev 2012).

Future work entails studying an ad-hoc rollback management in the multi-threaded environment, since bottom-halves management can produce a delay in message delivery, thus entailing an efficiency reduction.

REFERENCES

Carothers, C. D., D. W. Bauer, and S. Pearce. 2000, May. "ROSS: a High Performance Modular Time Warp System". In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, 53–60: IEEE Computer Society.

Carothers, C. D., and R. Fujimoto. 2000. "Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms". *IEEE Trans. Parallel Distrib. Syst.* 11 (3): 299–317.

Cucuzzo, D., S. D'Alessio, F. Quaglia, and P. Romano. 2007. "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*, 227–234.

D'Angelo, G., and M. Bracuto. 2009. "Distributed Simulation of Large-Scale and Detailed Models.". *International Journal of Simulation and Process Modelling (IJSPM)* 5 (2): 120–131.

Fujimoto, R. M. 1990, October. "Parallel Discrete Event Simulation". *Communications of the ACM* 33 (10): 30–53.

Glazer, D. W., and C. Tropper. 1993. "On Process Migration and Load Balancing in Time Warp". *IEEE Trans. Parallel Distrib. Syst.* 4 (3): 318–327.

Jagtap, D. A., N. Abu-Ghazaleh, and D. Ponomarev. 2012. "Optimization of Parallel Discrete Event Simulator for Multi-core Systems". In *Proceedings of the 26th Parallel and Distributed Processing Symposium*: IEEE Computer Society.

Jefferson, D. R. 1985, July. "Virtual Time". *ACM Transactions on Programming Languages and System*:404–425.

Li-li, C., L. Ya-shuai, Y. Yi-ping, P. Shao-liang, and W. Ling-da. 2011. "A Well-Balanced Time Warp System on Multi-Core Environments". In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, 154–162.

Peluso, S., D. Didona, and F. Quaglia. 2011. "Application Transparent Migration of Simulation Objects with Generic memory Layout". In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, 169–177: IEEE Computer Society.

Quaglia, F. and Pellegrini, A. and Vitali, R. 2011, October. "ROOT-Sim: The ROme OpTimistic Simulator: http://www.dis.uniroma1.it/∼hpdcs/ROOT-Sim/".

Toccaceli, R., and F. Quaglia. 2008. "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*, 163–172. Washington, DC, USA: IEEE Computer Society.

Vitali, R., A. Pellegrini, and F. Quaglia. 2012a. "Towards Symmetrc Multi-Threaded Optimistic Simulation Kernels". In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation*: IEEE Computer Society.

Vitali, R., A. Pellegrini, and F. Quaglia. 2012b. "Towards Symmetric Multi-threaded Optimistic Simulation Kernels". In *26th International Workshop on Principles of Advanced and Distributed Simulation*: IEEE Computer Society.

AUTHOR BIOGRAPHIES

ROBERTO VITALI is currently a PhD student in *Dipartimento di Informatica e Sistemistica* at *Sapienza Universit di Roma*. He is a member of *High Performance and Dependable Computing Systems* group. He achieved the bachelor's degree in Computer Engineering in 2007 and the master's degree in Distributed Systems and Computer Architectures in 2009. His research interests are primarly Distribuited Simulation, Computer Architectures and Operating Systems. His email address is `vitali@dis.uniroma1.it`.

ALESSANDRO PELLEGRINI is currently a PhD student in *Dipartimento di Informatica e Sistemistica* at *Sapienza Università di Roma*, collaborating with the *High Performance and Dependable Computing Systems*, where he is working in the area of Distributed Systems and Parallel Simulation. He achieved the bachelor's degree in Computer Engineering in 2008 and the master's degree in Distributed Systems and Computer Architectures in 2010. His other research interest include Code Parallelization, Autonomic Computing, and Machine Learning. His email address is `pellegrini@dis.uniroma1.it`.

FRANCESCO QUAGLIA received the Laurea degree (MS level) in Electronic Engineering in 1995 and the PhD degree in Computer Engineering in 1999 from the University of Rome "La Sapienza". From summer 1999 to summer 2000 he held an appointment as a Researcher at the Italian National Research Council (CNR). Since January 2005 he works as an Associate Professor at the School of Engineering of the University

of Rome “La Sapienza”, where he has previously worked as an Assistant Professor since September 2000 to December 2004. His research interests span from theoretical to practical aspects concerning distributed systems and applications, distributed protocols, middleware platforms, parallel discrete event simulation, federated simulation systems, parallel computing applications, fault-tolerant programming, transactional systems, Web-based systems and performance evaluation of software/hardware systems. He is an active member in the international research community, serving as a referee for several international conferences and journals. He regularly serves as Program Committee Member for prestigious international conferences. He has also served as Program Co-Chair of ACM PADS 2002, as Program Co-Chair of IEEE NCA 2007, and as General Chair of ACM PADS 2008. Since 2004, he is an Editorial Board Member of the International Journal of Simulation and Process Modelling (IJSPM). His email address is `quaglia@dis.uniroma1.it`.