Sapienza University of Rome

Ph.D. program in Computer Engineering

XXV Cycle - 2012/3

Design of Software Support Structures for High Performance Optimistic Simulations with Special Focus on Multi-Core Hosting Environment

Roberto Vitali

Sapienza University of Rome

Ph.D. program in Computer Engineering

XXV Cycle - 2012/3

Roberto Vitali

Design of Software Support Structures for High Performance Optimistic Simulations with Special Focus on Multi-Core Hosting Environment

Thesis Committee

Prof. Francesco Quaglia (Advisor)
Prof. Leonardo Querzoni

Reviewers

Prof. Christopher D. Carothers
Prof. George F. Riley

Author's address:
Roberto Vitali
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Sapienza Università di Roma
Via Ariosto 25, 00185 Roma, Italy
e-mail: vitali@dis.uniroma1.it
www: http://www.dis.uniroma1.it/∼vitali/

Contents

Abstract

Parallel Discrete Event Simulation (PDES) is a classical means to achieve high performance simulations. This is especially important on contexts entailing real-time constraints, as for the case of decision making tools, where target configurations have to be identified, via simulation, within bounded time. PDES is based on splitting the model to simulate in different simulation objects, that are mapped onto logical processes (LPs), which can process simulation events in parallel, thus potentially allowing for (large) speedups when compared to traditional serial execution of the simulation model. On the other hand, synchronization mechanisms need to be actuated in order to allow the parallel run to provide correct results, hence correct evolution of the objects' state along the simulation time axis.

In this dissertation we cope with performance of PDES systems, with special focus on deploys of the PDES platform onto multi-core architectures. These architectures represent the current CPU trend and nowadays it is common to have chips with a significant number of cores, sometimes enough for effectively performing parallel computations within a single chip. By design, these architectures share most of the CPU internal components across different cores, and physical memory across the different CPUs. This implies several drawbacks, mainly due to the contention, but at the same time, such a sharing can become a resource, if well exploited through sector specific software design.

The base building block for our work is the optimistic PDES synchronization paradigm, which has been shown to be highly promising in terms of potential for fruitful parallelism exploitation.

As a first contribution, we will present the design and implementation of optimized supports for state log/restore operations, which are at the base of the rollback procedures used to guarantee consistency in the optimistic paradigm. System requirements for these supports only entail the x86 instruction set, with no particular additional facilities to be offered by the underlying computing platform. On the other hand, the provided solution has a further reflection on performance for deploys on multi-core systems since it aims at

automatizing the optimization of a set of CPU/memory tradeoffs among which we also find virtual memory usage, which may impact physical memory (e.g. cache) contention for the case of multi-core architectures.

Secondly, we address the issue of optimizing the behavior of the caching hierarchy by providing an innovative buffer delivery mechanism suited for memory demands in optimistic PDES systems. This is a quite relevant aspect to cope with, given the intrinsic high-demand of virtual memory by optimistic PDES runs. The proposed solution tends to reduce the impact of virtual memory demand on physical memory, which is achieved in a highly general manner, thus making the approach applicable to differentiated types of hardware architectures. However, by nature, it reveals particularly suited for the case of multi-core systems, given the aforementioned issue if physical memory contention across the cores.

Finally, we provide an additional contribution, specifically tailored to the multi-core architecture, which introduces an innovative load-sharing paradigm for optimistic PDES platforms. It is aimed at improving performance and fruitful resource usage in a fully orthogonal manner with respect to traditional load-balancing. In fact, it allows the computing resources to be (dynamically) assigned to each simulation platform instance on the basis of its current workload, instead of migrating the workload from one kernel to another. This also has the advantage of not suffering from all the issues related to workload (namely LPs) remapping (i.e. data and metadata migration), leading to the simplification of the job of both application programmers and simulation-system developers.

To assess the real benefits, all the above approaches have been implemented within ROOT-Sim, an open source C-based Optimistic Simulation platform targeting the optimistic synchronization paradigm, and several experimental studies have been conducted, whose outcomes are reported in the thesis.

Chapter Organization

This dissertation is structured as follows. The first chapter provides details on the PDES approach, with the special focus on the optimistic synchronization paradigm. It also provides hints on multi-core architectures, which have been used as the reference hardware environment for this study. In chapter two, state of the art results in the context of PDES systems, which are more or less related to the presented approaches, are discussed. The third chapter provides an overview of both hardware and software environments exploited to ultimately assess the effectiveness of our proposals. Original contributions by this dissertation start form chapter four, where an Autonomic State Saving architecture is designed, implemented and tested. Chapter five introduces

our innovative memory delivery mechanism, explicitly targeted at optimistic PDES systems, and reports experimental data demonstrating its viability and advantages. Chapter six presents the load-sharing approach/model, which explicitly targets multi-core environment, its implementation and profiling/performance data showing its dynamics and effectiveness. Finally, chapter seven concludes the dissertation by summarizing the provided contributions.

Most of the material provided by this dissertation has been presented in (or has contributed to the production of) the following technical articles I have coauthored:

1 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Load sharing for optimistic parallel simulations on multi-core machines. *ACM Performance Evaluation Review.* vol.4, no.3, 2012.

2 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic state management for optimistic simulation platforms. In Preparation for submission to an international journal.

3 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. A load sharing architecture for optimistic simulations on multi-core machines. In *Proceedings of the 19th International Conference on High Performance Computing*, HiPC. IEEE Computer Society, December 2012.

4 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Assessing load sharing within optimistic simulation platforms (invited paper). In *Proceedings of the 2012 Winter Simulation Conference*, WSC. Society for Computer Simulation, December 2012.

5 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Towards symmetric multi-threaded optimistic simulation kernels. In *Proceedings of the 26th International Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 211–220. IEEE Computer Society, August 2012.

6 Roberto Vitali, Alessandro Pellegrini, and Gionata Cerasuolo. Cache-aware memory manager for optimistic simulations. In *Proceedings of the 5th International ICST Conference of Simulation Tools and Techniques*, SIMUTools, March 2012. Winner of the Best Paper Award.

7 Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. ICST, 2011.

8 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Autonomic log/restore for advanced optimistic simulation systems. In *Proceedings of the Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 319–327. IEEE Computer Society, 2010.

9 Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Benchmarking memory management capabilities within root-sim. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2009.

10 Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. DiDyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 45–53. IEEE Computer Society, 2009. Candidate for (but not winner of) the Best Paper Award.

Chapter 1

Introduction

Simulation is an attractive and well consolidated methodology to study real world phenomena. It has been exploited in a wide set of fields, including physics, biology and business-oriented processes (such as financial prediction or optimized system-configuration selection). For some application contexts, one relevant aspect relates to the timeliness according to which simulation results are provided to end-users or applications, such as when exploiting simulation as a tool supporting time-critical decision making. Hence, performance aspects while delivering simulation output is a core issue to cope with. In the next sections we overview the typical approach used to achieve high performance simulation in the context of discrete event models, namely Parallel Discrete Event Simulation (PDES), posing the attention to the PDES optimistic synchronization mechanism, which forms the base ground for the thesis. Then, we present an overview of the specific problems targeted by the thesis and of the achieved outcomes.

1.1 Parallel Discrete Event Simulation

In the context of Discrete Event Simulation (DES), high performance has been targeted via the Parallel-DES (PDES) paradigm [Fujimoto(1990)], which allows exploiting the computing power offered by (high-end) parallel/distributed platforms in order to speedup model execution and to make (very) large and/or accurate models tractable. The basic idea underlying PDES is to partition the simulation model into several distinct simulation objects, which are the core of the simulation process from the model writer's point of view. In fact, each object represents a portion of the real world being simulated, the evolution of

which is described by object state transitions, driven by a set of logical/mathematical properties. In order to represent real-world interactions, simulation objects can communicate with each other, by exchanging pieces of information in the form of events. From a technical point of view, simulation objects are handled by Logical Processes (LP), which undertake the concurrent execution of simulation events. Traditionally, a PDES run entails a number of concurrent LPs, uniquely identified by a numerical code in the range $[0, N-1]$, and the overall simulation model keeps track of the evolution of the simulated world by relying on a global simulation state, which is partitioned into various LPs' private and disjoint simulation states [1].

In PDES, simulation events are timestamped and their execution is impulsive, meaning that there is no notion of time evolution during an event processing. The current simulation time at each individual LP is known as Local Virtual Time (LVT), and can be expressed in any measure unit (i.e., one LVT unit can represent seconds, hours, or even years, depending on the actual simulation model). This notion of time is opposed to the Wall-Clock Time (WCT), which is the actual notion of time we are used to. Therefore, in one WCT unit, the LVT advancement can be of one or several units, depending on the actual complexity of the simulation model and on the efficiency of the simulation run.

During the execution of an event, other events can be generated, destined to any simulation object in the system, and are associated with a timestamp value which is greater than or equal to the one of the event currently being executed, i.e. during the execution of the event e_x associated with the timestamp t_x, a new event e_y associated with timestamp t_y can be generated and sent to another simulation object, ensuring that $t_y \geq t_x$. Therefore, event generation evolves according to a causality pattern where the present cannot affect the past.

LPs are in charge of delivering simulation events to the hosted simulation objects, via the invocation of proper event handlers. Simulation-kernel instances take care of the dispatching event processing activities across the various LPs, and of managing inter-LP communication. In particular, they handle the LPs' event queues, by reflecting the updates associated with incoming messages, and determine the best LP to be dispatched in order to optimize specific execution metrics.

On the other hand, when running the LPs concurrently on multiple CPU-

[1] Some approaches have studied variations where some portion of the state is shared across multiple LPs [Pellegrini et al.(2012)Pellegrini, Vitali, and Quaglia, Ghosh and Fujimoto(1991)]. However, the traditional case of disjointness of the LPs' states represents the most studied and exploited approach.

cores provided by the underlying parallel/distributed platform, synchronization mechanisms are required in order to ensure that the causality pattern is maintained not only for event generation but also for actual event processing at any LP. Although differentiated approaches to the definition of causally consistent execution have been devised in literature [Madhava Rao et al.(Dec)Madhava Rao, Thondugulam, Radhakrishnan, and Wilsey, Quaglia and Baldoni(1999), Fujimoto(1999), Cai et al.(2005)Cai, Turner, Lee, and Zhou], the most widely known and exploited causality criterion expresses that model execution is correct if each LP processes its input events in non-decreasing timestamp order.

To maintain causal consistency (namely local timestamp ordering at any individual LP) two main approaches have been proposed: *conservative* and *optimistic*. The conservative approach consists in avoiding at all the possibility of a causal violation, i.e., it can not happen that an event is executed out of order. To support this synchronization approach, block-until-safe policies are employed, which suspend event processing activities at an LP until it is determined that the execution of the next pending event is coherent with logical-time ordering. On the other hand, with the optimistic approach, event execution is never suspended at any LP, hence giving rise to speculative processing, under the optimistic assumption that causality is not violated. If any violation is detected, rollback recovery mechanisms are used to bring the involved LPs back to a correct state snapshot, starting from which execution of is resumed.

Literature results show that the optimistic approach is prone to higher parallelism exploitation, and to deliver performance which is less influenced by the message (event) delivery delay and by the lookahead within the simulation model [2]. These advantages are reflected also on the side of scalability, as demonstrated by the study in [Carothers and Perumalla(2010)], where very large platforms (entailing on the order of thousands of CPU-cores) are employed for a comparative analysis of conservative vs optimistic approaches. By this study, it clearly emerges that the conservative approach delivers better performance only for (very) high look-ahead values, which represent relatively uncommon cases. Further, such an advantage is reduced as the number of CPU-cores gets increased. Overall, better scalability of the optimistic approach in general contexts has been demonstrated.

[2] The lookahead expresses the capability of a model to predict the non-occurrence of events within an given interval of simulated time, starting from the current time.

1.2 Optimistic Synchronization Overview

In this section we provide an overview of optimistic synchronization, such as the Time Warp protocol presented in [Jefferson(1985)]. The overview is tailored to the description of basic dynamics proper of the optimistic approach, and of some details that are more closely related to the specific topics of interest for this thesis. A deeper discussion on literature results coping with the topics addressed by the thesis is delayed to Chapter 2.

As hinted before, the optimistic approach aims at full exploitation of the parallelism offered by the underlying computing platform, which is achieved via the avoidance of block-until-safe policies for event execution, and on the adoption of speculative processing. With this approach, causality violations can occur upon the delivery of a so called *straggler message* to any LP involved in the run. A straggler message carries a scheduled event associated with a timestamp t_1 which is lower than the timestamp t_2 of some event that has already been processed by the recipient LP. In other words, the destination LP ran far ahead in simulation time with respect to some event that should have instead affected its evolution (an example of this situation is shown in Figure 1.1). In such a case, all the events that have already been processed by the recipient LP, having timestamp t included in the interval $t_1 < t \leq t_2$ are no longer causally consistent (with respect to the local timestamp ordering criterion).

Anytime a causality violation is detected, some rollback recovery mechanism needs to be actuated which involve, at the same time, two orthogonal issues:

- restoring the state of the rolling back LP to a past snapshot that is still consistent;
- undoing the effects of inconsistent local processing activities on other LPs.

The task in the last item is generally supported via the employment of so called anti-messages, which are used to annihilate events that have been scheduled by the rolling back LP as a result of causally inconsistent processing activities. An anti-message is therefore a negative copy of a previously sent message, and is used to signal the destination to discard the original message. Clearly, in case the event carried by the original message was already processed by the destination LP, we experience a spreading of the rollback occurrence along a chain of LPs (this phenomenon is also known as *cascading rollback*).

We note that messages (events) are inserted within the system by usually employing proper APIs offered by the underlying simulation environment as a support of typical PDES programming models. This implies that the

management of anti-messages upon the occurrence of rollback does not pose particular transparency problems with respect to the application-level code. In fact, the environment can log outgoing messages and generate the corresponding anti-messages if requested (e.g. by marking a "sign" bit within the messages' meta-data). As a consequence, research efforts in the management of anti-messages are primarily related to reducing the management overhead and the need for actually sending out the anti-messages associated with given messages, such as when employing "lazy cancellation" schemes [Gafni(1985)] [3].

On the other hand, recoverability of the LPs' states poses problems on the side of both performance and application transparency. As for performance, we need to consider both CPU usage for supporting tasks enabling state recoverability and actual state recovery actions, as well as memory usage for keeping recoverability related data/meta-data. On this side, a wide literature exists that has proposed the employment of log/restore techniques (see, e.g., [Bellenot(1990), Preiss et al.(1994)Preiss, Loucks, and MacIntyre, Palaniswamy and Wilsey(1993)]), where snapshots of the state of the LPs are taken according to some (infrequent) policy in order to optimize the tradeoff between log costs and restore latency [4]. Also, these techniques deal either with non-incremental or incremental logging, the latter approach (see, e.g., [Rönngren et al.(1996)Rönngren, Liljenstam, Ayani, and Montagnat, Santoro and Quaglia(2005), West and Panesar(1996)]) being also oriented to reduce the usage of memory for log buffers. On the other hand, transparency issues deal with supporting log/restore tasks with no need for having log/restore modules to be implemented within the application-level code (hence masking the actual synchronization paradigm to the application programmer). This is a non-trivial aspect since it relates to the flexibility according to which the application programmer is allowed to organize the data structures representing the LP state image. As an example, the reliance on dynamic memory usage at the application level requires the optimistic simulation platform to entail complex memory management mechanisms [Toccaceli and Quaglia(2008)] in order to be able to make generic and scattered memory layouts to be recoverable

[3] Lazy cancellation, as opposed to aggressive cancellation, tends to avoid sending out an anti-message as soon as an original message is detected to have been produced by out-of-order computation. This helps reducing the spreading of rollback in case the message would have been reproduced after resuming computation.

[4] Taking a snapshot of the LP state less frequently increases the likelihood that the state to be restored is not logged, hence must be reconstructed by starting from a previous logged snapshot via fictitious re-processing of intermediate events. This re-processing phase is also known as *coasting forward*.

transparently and efficiently. On the other hand, this problem is exacerbated in case the target is incremental logging, since such a memory layout needs to be managed also on the side of identifying updates occurring within it, at a granularity level that allows providing advantages with respect to logging the whole state image. Recently, the memory demand problem associated with logging has been also tackled via so called *reverse computing* approaches [Carothers et al.(1999b)Carothers, Perumalla, and Fujimoto], where a state-restore operation is supported via application-level compensation logic that applies backward computation steps until the correct snapshot for resuming the LP is built. For this approach we still find issues in relation to how to generate the reverse code version transparently to the application programmer.

Still in relation to memory usage and recovery, optimistic synchronization is intrinsically linked to the notion of *Global Virtual Time* (GVT). It represents the commit horizon of the optimistic simulation run, namely the simulation time barrier currently separating the set of committed events from the ones which can still be subject to a rollback. This barrier corresponds to the minimum timestamp of not-yet-processed or in-transit messages/anti-messages, hence it is typically computed via global reduction protocols, which can be optimized for specific platform organizations (e.g., shared-memory systems vs clusters [Bauer et al.(2005)Bauer, Yaun, Carothers, Yuksel, and Kalyanaraman, Fujimoto and Hybinette(1997), Riley et al.(2000)Riley, Fujimoto, and Ammar]). Once the new GVT is available, all the memory buffers keeping the events that have been committed and the state logs related to the committed portion of the simulation can be released [5]. The procedure of recovering memory buffers is usually termed *fossil collection*. We note that the GVT protocol cannot be executed with unbounded frequency since it imposes overhead. This leads to further exacerbating the memory demand problem in optimistic simulation platforms, since they are requested to allocate and (temporarily keep) memory for information related to both speculative outcomes (e.g. speculatively scheduled events) and already committed actions (i.e. already committed events). Particularly, this may impose large memory usage especially for LPs' event and message queues, given that on the side of state recoverability infrequent logging and/or incremental schemes and/or reverse computing already tend to reduce such memory demand.

An additional central point in optimistic simulation platforms relates to the CPU-scheduling approach used to determine which LP, among the ones

[5] For infrequent logging schemes, the only exception is related to the need for keeping data/meta-data for at least one logged state image with time t less than GVT, and the events with time in between t and the GVT value, in order to be able to recover the LP state image to any point in time arbitrarily close, or coinciding with, the GVT value.

hosted by a given simulation-kernel instance, must take control for actual event processing activities. Although several proposals have been made [Som and Sargent(1998), Quaglia and Cortellessa(2002), Rönngren and Ayani(1994a), Palaniswamy and Wilsey(1994)], the common choice is represented by the Lowest-Timestamp-First (LTF) algorithm [Lin and Lazowska(1991)]. It selects the LP whose next pending event has the minimum timestamp, compared to next pending events of other LPs hosted by the same kernel. Coupled with the traditional single-threaded approach for the implementation of the simulation kernel, LTF has the advantage of avoiding the generation of causality violations across the LPs hosted by the same kernel instance. This is because these LPs are dispatched in a way similar to what would happen on top of a sequential simulation engine, which imposes a timestamp-ordered sequence of CPU-schedule operations for all the events (across all the LPs). Hence, rollbacks can be generated only in relation to events scheduled between LPs hosted by different kernels, which contributes to reduce the amount of rollbacks, and to make the optimistic paradigm effective.

On the other hand, another aspect that can potentially hamper performance, due to excessive generation of rollbacks (possibly giving rise to thrashing phenomena), is the unbalanced advancement in simulation time of LPs hosted by different kernel instances. In other words, the optimistic paradigm needs to be complemented by proper mechanisms aimed at making computing resources exploited for fruitful simulation work (not for work that is eventually rolled back, thus only causing waste of resource usage). Waste of computing resources can also be observed on long-distance basis, namely when the final portion of the computation has no inter-kernel messages exchanges. In such a scenario, the less-loaded kernels would terminate sooner with respect to others, and the associated computing resources may remain unused until the termination of the slower kernel. Similar under-utilization scenarios may occur in case some kernel instance hosts LPs that have no event to process for a specific simulation time interval. In such a case, the computing power assigned to that kernel instance remains unused for a while (namely for the wall-clock-time period required for repopulating the event queues of those LPs, depending on proper dynamics of the simulation model). Beyond traditional load-balancing solutions [Glazer and Tropper(1993), Carothers and Fujimoto(2000), Jiang et al.(1994)Jiang, Shieh, and Liu, Peluso et al.(2011)Peluso, Didona, and Quaglia], the issue of balanced advancement in simulation time across different kernel instances has also been tackled via variants of the optimistic approach based on reduction of the level of optimism (hence reduction of the possibility for some LPs to run excessively ahead of others) [Srinivasan and Reynolds(1998), Ferscha(1995)]. The tradeoff provided by these approaches usually leads to reducing the likelihood of causality violation via reduction of the exploitation of the available computing power (the typical case is when

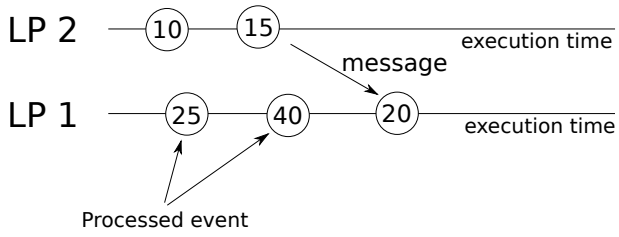


Figure 1.1: Straggler Message

execution of some LPs is explicitly throttled).

1.3 Thesis Contributions

Along the line of the provided overview on optimistic synchronization, we can identify the contributions by this thesis as related to three core topics:

State log/restore: as for this topic (see Chapter 4), we present a fully innovative state log/restore architecture, which allows time interleaved co-existence of incremental and non-incremental log/restore operating modes on a per LP basis. The architecture is based on a dual-coding scheme, which is fully transparent to the application-level programmer, and enables the application code to rely on most of the ANSI-C standard. Dual coding is achieved via instrumentation of the ELF associated with the application, hence it is tailored for UNIX based systems. Further, the architecture performs the dynamic selection of the best-suited log mode (and of its optimal parameterization, e.g., in terms of frequency of the log operation) on the basis of an innovative analytical approach, which complements existing literature results.

Buffer delivery: as for this topic (see Chapter 5), we provide an innovative buffer-delivery architecture which is explicitly oriented to reduce the impact of virtual memory usage by the simulation platform on the effectiveness of the caching hierarchy. The architecture is based on a characterization of buffer access patterns proper of the optimistic paradigm, which we also propose within the thesis. Essentially, the architecture reserves a cache partition to hot data, favoring the residence in cache of part of the working set. Although this solution is generally applicable in different architectural contexts, it gives rise to a reduction of the bus contention, which is especially relevant in multi-core and multi-processors architectures.

Balanced resource usage: as for this topic (see Chapter 6), we present a reshuffle of the design approach of optimistic simulation platforms in order to generate a final architecture oriented to maximizing the exploitation of the computing power offered by multi-core machines in order to perform fruitful work. The reshuffle is based on a symmetric multi-threading paradigm, which allows supporting load-sharing policies (as opposed to traditional load-balancing) suited for enabling balanced advancement of the LPs along the simulation time axis with no need for migrating them. Specifically, the approach that has been taken is based on dynamic reassignment of the computing power to the active kernel instances. A specific load sharing policy, based on an analytical model of the expected computing power requested to sustain the workload by the different kernel instances, is also provided.

Chapter 2

Literature Survey

In order to frame the contributions by this thesis within literature results, a sight on what is the state of the art in relation to the addressed topics is provided. Particularly, we start with an overview of results dealing with state recoverability. Then we discuss approaches explicitly oriented to cope with memory demand in optimistic simulation platforms. Finally we discuss literature solutions aimed at balanced and fruitful usage of the computing resources in optimistic synchronization, including more recent approaches based on the concept of multi-threading.

2.1 State Recoverability

As hinted, one main means to support recoverability of the LP state in optimistic simulation systems is represented by logging approaches, where the state image is (infrequently) logged in order to generate restoration points along the simulation time axis. Several studies in this direction have been aimed at providing analytical models describing the expected log/restore overhead when experiencing a given rollback pattern (e.g. in terms of frequency of rollback occurrence at the LP) and when taking state logs, namely checkpoints, at specific points of the execution (for instance each χ event executions, according to a periodic scheme) [Rönngren and Ayani(1994b), Lin et al.(1993)Lin, Preiss, Loucks, and Lazowska, Quaglia(2001)]. By monitoring the independent parameters appearing within the analytical expressions, the models can be used to (dynamically) determine the position of checkpoints in order to keep the whole log/resore overhead at minimum values. We recall that a lower number of checkpoints taken along a given execution path reduces the log cost, but is

expected to give rise to an increase in the restore cost. Particularly, the state to be recovered might not be immediately available within the log, and would need to be reconstructed by restoring an older snapshot and by reprocessing the intermediate events up to the target restoration point. Hence, the aforementioned approaches provide models for determining the well-suited balance among these two opposite overhead tendencies.

The provided models deal either with the case of non-incremental logging or with incremental logging, and some of them even cope with the case where the two approaches are used in combination (e.g. by taking incremental logs between subsequent non-incremental logs) [Soliman and Elmaghraby(1998)] or are considered comparatively [Palaniswamy and Wilsey(1993)]. However, these proposals have been mainly tailored to the evaluation of log/restore policies (once known the costs for basic operations, such as the copy of the whole or part of the LP state image into the log buffer), not to provide log/restore architectures explicitly tackling transparency of log/restore tasks to the application level code.

The issue of transparency has been dealt with by other studies. For incremental log/restore schemes this has been done by either instrumenting application level code (in order to transparently insert code portions aimed at identifying the write operations occurring onto the state image, thus allowing identification of the dirty portions of the state) [West and Panesar(1996)] or by employing operator overloading schemes, as for the case of the proposal in [Rönngren et al.(1996)Rönngren, Liljenstam, Ayani, and Montagnat] which has been tailored to object oriented technologies. For both the provided approaches there is anyhow the need for compile time identification of the memory portions forming the actual state image of the LP. Hence the approaches are not fully suited for supporting a general programming model where the memory layout of the LP state can rely on, e.g., dynamic memory allocation and/or can be updated via third party libraries. On the other hand, the solution in [Toccaceli and Quaglia(2008)] provides supports for transparency in the context of dynamic memory based state layouts, but limitedly to the case of non-incremental logging.

Other proposals have provided log architectures based on specialized hardware [Quaglia and Santoro(2003), Fujimoto et al.(1992)Fujimoto, Tsai, and Gopalakrishnan], which have been designed in order to achieve some level of transparency, while also offloading the CPU, at the price of limiting the programming model, e.g., by imposing contiguousness or static determination of the memory area maintaining the state image of the LP.

The case of dynamic memory usage at the application level has been addressed by the proposals in [speedes(), Das et al.(1994)Das, Fujimoto, Panesar, Allison, and Hybinette], which provide recoverability for memory scattered LP state images. However, the level of transparency is not maximized since

ad-hoc dynamic memory allocation/deallocation APIs are used to notify the underlying simulation platform that the corresponding operation needs to be rollbackable.

Full transparency, in combination with incremental logging, has been provided by the proposal in [Santoro and Quaglia(2005)], which has been tailored to parallel/distributed simulation platforms adhering to the High-Level-Architecture (HLA) specification [IEEE Std 1516-2000 (2000)(2000)]. This approach exploits a page-based memory update tracking mechanism relying on facilities (e.g. SEGFAULT tracking) offered by the underlying operating system. Hence the granularity according to which incremental logs are taken cannot be set arbitrarily, and cannot be optimized depending on the actual needs. Overall, these proposal are mostly suited for federations of simulation components where a middleware layer (namely the HLA Run-Time-Infrastructure) is used to operate distributed coordination and data exchange, whose overhead tends to mask the one imposed by the page-based logging approach. They result less suited for traditional PDES platforms, relying on highly optimized low-overhead engine level coordination and data exchange facilities.

An approach to state recoverability which is orthogonal to the aforementioned solutions has been provided in [Carothers et al.(1999b)Carothers, Perumalla, and Fujimoto]. Instead of relying on state logs, this proposal is based on reverse computing schemes where the forward execution code (namely the native implementation of the application level simulation code) is coupled with a reverse code version which is in charge of backward compensating (hence undoing) the updates occurred onto the LP state in case a rollback occurs. The issue of automation of the generation of the reverse code, which targets transparency to the application programmer, is also faced. The reverse computing approach has been recently exploited, and has been demonstrated to be effective, for several applications [Bauer and Page(2007), Seal and Perumalla(2011)]. Particularly, one main advantage by this approach is the reduction of memory demand for state-log buffers, while also nullifying the log overhead (since logs are not taken at all). On the other hand, the tradeoff is towards a potential increase of the restore latency in case very long rollbacks occur, which would require log reverse computing paths to achieve the restoration point. On the other hand, this approach can be complemented with periodic state logging in order to both (a) avoid excessively log backward computation phases, and (b) to deal with non-reversible operations (such as flat assignments within the state image).

Advancement by the thesis. As for state recoverability, the thesis advances the state of the art by providing a log/restore solution, inspired to the

autonomic computing paradigm, which does not rely on any specialized hardware, and jointly addressed transparency and performance issues by exhibiting all together the following features:

- It allows the application level programmer to use standard constructs for dynamic memory allocation/deallocation operations, hence allowing the LP state to be scattered across non-contiguous memory chunks.
- It transparently enables phase-interleaved adoption of incremental and non-incremental log/restore modes.
- It runs each log/restore mode in a highly optimized fashion, via the adoption dual-coding approaches and of classical schemes for the optimization of typical parameters determining the actual overhead for each mode.
- It dynamically (and transparently) switches to the best suited operating mode (incremental vs non-incremental) depending on proper execution dynamics of the optimistic simulation run.

While individual, or subsets, of the above points are dealt with by literature results, none of these proposals fully covers the whole set of listed issues.

2.2 Memory Management

Although optimized approaches and/or architectures supporting state recoverability can also be considered as solutions aimed at reducing the memory demand by the optimistic simulation environment (this is especially true when considering reverse computing approaches), such a reduction can be considered as a reflection of the whole optimization process leading to well suited tradeoffs between log and restore overheads. On the other hand, the memory demand problem in optimistic simulation platforms, and its reflection on the effectiveness of the underlying caching hierarchy and virtual memory system, have also been addressed via sector specific approaches.

As hinted, the memory demand by optimistic platforms does not only involve log operations, but the need for temporary maintaining events that are not yet detected as already committed (since GVT is typically re-evaluated periodically), and the need for supporting speculative scheduling of future events. The latter aspect may entail high frequency for buffer allocation requests just due to the fact that purely optimistic approaches can allow the LPs to run far ahead of the currently committed horizon given the absence of blocking or throttling strategies.

As for memory requirements related to the already committed portion of the computation, some advanced fossil collection mechanisms have been proposed [Chetlur and Wilsey(2006)] that, by means of dissemination of information about causality relations among events, are aimed at the identification of the fossils (hence of memory to be recovered) in an complementary manner compared to the classical ones based on GVT computation.

The effects of the caching hierarchy and of the underlying virtual memory system on the performance of specific tasks (such as state saving) and/or of the overall simulation run has also been studied by several works [Carothers et al.(1999a)Carothers, Perumalla, and Fujimoto, Akyildiz et al.(1993)Akyildiz, Chen, Das, Fujimoto, and Serfozo, Akyildiz et al.(1992)Akyildiz, Chen, Das, Fujimoto, and Serfozo, Das and Fujimoto(1997a)]. Outcomes by these studies show how both caching and virtual memory may have a relevant impact on performance, thus posing the need for optimizing platform level configuration and/or design in order to limit the performance degradation phenomenon.

Interesting proposals aimed at the integration of advanced memory management schemes specifically tailored to optimistic PDES platforms can be found in [Lin and Preiss(1991), Das and Fujimoto(1997b), Jefferson(1990), Preiss and Loucks(1995)]. Here the authors propose techniques, such as cancelback, pruneback or artificial rollback, which are aimed at achieving efficient executions of the optimistic paradigm when considering limited available memory. This is achieved by, e.g., artificially squashing portions of speculated computation in order to avoid maintaining the related information (such as the record for speculatively scheduled events) when memory demand becomes critical (such as when swapping phenomena within the virtual memory system would tent to appear). With this type of integrations, the optimistic approach has been shown to be able to complete the run at reasonable performance by using an amount of memory similar (or slightly larger than) the one requested by a sequential, non-speculative run of the same simulation application. Further, the performance tradeoffs associated with these schemes have been thoroughly investigated both analytically and empirically (see, e.g., [Das and Fujimoto(1993)]).

Different approaches, still tailored to the tradeoff between memory management and performance, relate to the reduction of the number of memory copies for supporting event exchanges within the optimistic platform. Particularly, the proposal in [Swenson and Riley(2012)] provides a so called zero-copy message passing approach, suited for both conservative and optimistic simulation, which allows reducing the whole memory demand due to message buffering on shared memory architectures, thanks to the reduction of the amount of virtual memory buffers used for keeping the messages.

The only work in literature that directly faces the cache hierarchy misuse in optimistic simulators is [Fuj95], whose main target is to point out the rele-

vance of buffer delivery mechanisms that are cache-friendly in shared memory contexts. The work exclusively accounts for buffers reserved for exchanged messages. It presents a new approach that partitions the memory destined to messages in a way that the pages are accessed only by the two processes that participate in the communication, providing a reduction of the cache-coherence overhead and the cache invalidation. Our proposal is orthogonal since it does not account for message-related buffers only, and can be deployed on both shared and non-shared memory platforms.

Advancement by the thesis. In this thesis we address the effects on performance due to the memory system in an orthogonal manner with respect to all the aforementioned solutions. Particularly, we concentrate on caching phenomena and propose an innovative architecture which supports a cache-aware memory delivery mechanism specifically tailored to optimistic simulation platforms. Hence our approach could be used in combination with other approaches which have been targeted to the reduction of virtual memory usage by the platform.

We must say that cache-aware memory allocation is not a new topic in general (namely, outside the PDES area). However, our approach differs from existing solutions. As an example, when considering cache-aware buffer delivery within operating system kernels (such as the LINUX Slab-allocator [Bonwick(1994)]), the main objective is the minimization of the RAM/cache data movement (e.g. via the minimization of false cache sharing on multi-core machines). Compared to these approaches, we further target the concept of hot vs cold data in the context of optimistic simulation runs, and tailor cache usage in order to optimize the management of hot data (namely buffers associated with information that requires more frequent access).

As for the maximization of cache hits, some buffer allocation policies have been proposed in [Chilimbi et al.(2000)Chilimbi, Hill, and Larus]. These are based either on proper APIs to notify the memory management system that different memory areas are correlated in terms of expected future accesses (hence they should be located such in a way to be loaded into, e.g., the same cache line) or on run-time profiling mechanisms aimed at clustering memory accesses and reorganizing the layout to maximize cache efficiency. Compared to these approaches we do not require the usage of proper APIs (hence the simulation platform can rely on traditional memory allocation services) and do not require run-time profiling (hence we avoid the associated costs) since buffer relevance in terms of cache hits/misses are predefined via a general scheme tailored to optimistic PDES systems.

In terms of employed cache management methods, the work more close to our approach is the one in [Mueller(1995)] which presents a software-

partitioned cache system to allow predictable task execution for real-time environments. However, the final target of this work is completely orthogonal to our one since it is focuses on memory latency predictability, achieved at the cost of performance drops caused by the non-minimal number of cache partitions that are typically required to support the different real-time tasks. Conversely our goal is to increase exploitation of the cache system, which is achieved by partitioning the cache to host buffers accessed in different execution patterns in two different cache partitions. The two-partition division reduces cache pollution due to infrequently accessed buffers, favoring the permanence of (part of) the working set in one of the two partitions.

2.3 Balanced and Fruitful Usage of Resources

Balancing the usage of resources and its reflection on balanced and performance-effective advancement in simulation time by different LPs is a fundamental issue to be addressed for both conservative and optimistic simulation systems. While in conservative simulation a non-balanced scenario could lead to underutilization of the resources, in optimistic simulation platforms we may also experience thrashing phenomena due to excessive rollback generation. This issue has been traditionally tackled via load-balancing schemes where the LPs are dynamically migrated from one simulation kernel instance to another depending on the actual load they impose in a specific phase of the simulation run. Along this direction we can find several solutions, such as [Glazer and Tropper(1993), Carothers and Fujimoto(2000), Choe and Tropper(1999), Meraji et al.(2010)Meraji, Zhang, and Tropper], which share the common feature of being targeted to provide metrics for detecting imbalance scenarios and policies to re-balance the load, but that are not targeted to provide architectural solutions for full transparency of the LP migration task to the application programmer. Other studies have addressed the issue of load re-balance in optimistic simulation systems for specific application fields, such as logic circuit simulation [Meraji and Tropper(June)], hence providing sector specific policies.

As for transparency, a recent work in [Peluso et al.(2011)Peluso, Didona, and Quaglia] provides a global memory management architecture that allows reinstalling the LP state image (even in case it is formed by memory scattered dynamically allocated chunks) across different kernel instances with no intervention by the application level code. The architecture has been then combined with already existing balancing policies (particularly the one in [Carothers and Fujimoto(2000)]) in order to provide fully fledged supports for the re-balance problem in general application contexts.

All the above approaches have been targeted to scenarios where the dif-

ferent kernel instances can be run on either shared memory or distributed memory systems. Hence they do not explicitly consider the possibility of exploiting the actual sharing of memory across different processes/threads within the simulation platforms. Given the relevance of the multi-core architectural organization, which natively offers sharing facilities, more recent approaches have been tailored to balanced and fruitful resource usage in the context of shared-memory platforms.

Along this path, the work in [Chen et al.(2011)Chen, Lu, Yao, Peng, and Wu] presents a global schedule mechanism relying on a distributed event queue that can be accessed by different threads, which is used to fairly distribute the actual simulation load across the whole set of CPU-cores within the underlying platform. However, the global event queue represents a synchronization point which tends to hamper scalability of the proposed solution. In fact, the effectiveness of the approach has been tested limitedly to the case of a maximum of 8 CPU-cores. Similar considerations can be made for the case of the ThreadedWarped architecture in [Miller(2010)], which uses a global priority queue across subsets of the threads within the platform. While this approach reduces contention (given that a queue is shared by a subset of threads, rather than all the active ones), it does not allow to redistributed the load (i.e. the LPs) across different subsets of threads. Hence, execution of a specific LP remains confined onto a given subset of the CPU-cores within the platform.

Finally, one recent work specifically oriented to improve the performance of simulation platforms on multi-core machines can be found in [Liu and Wainer(2012)]. This work is targeted at the IBM Cell processor [Kahle(2005)], thus not being immediately suited for differentiated multi-core platforms.

Advancement by the thesis. As for balanced and fruitful usage of resources, the thesis provides an innovative approach explicitly targeted to multi-core machines, which is based on a paradigm shift from load-balancing to load-sharing. Particularly, in our proposal, a simulation kernel instance is a dynamic entity structured according to a symmetric multi-threaded paradigm, which has the ability to acquire or release computing power depending on the actual workload to be sustained. As a result, the hosted LPs can potentially run on any CPU-core (given that any core can be incorporated for usage by the kernel instance hosting the LP). Hence higher flexibility is provided in terms of mapping of LPs to CPU-cores, and hence in terms of determining a well suited balance of the whole workload over the underlying platform.

We note that this approach is fully orthogonal to load-balancing, hence they could be used in combination, for example for optimizing resource usage in clusters of multi-core machines (where load-sharing can optimize intra-machine dynamics, while load-balancing and the associated LP migration

schemes can optimize inter-machine dynamics).

Finally, we note that the architectural proposal for the reshuffle to load-sharing has been also complemented with an innovative analytical model for the determination of how to reassign the computing power to the different kernel instances, which constitutes an additional contribution by the thesis.

Chapter 3

Reference Environment

In this chapter we illustrate the hardware and software environment that has been exploited for integrating and experimentally evaluating the proposals by this thesis. We will start by over-viewing the hardware platform and the lower-level software facilities (such as the operating system and the used compiling tools). Then we present the target open source optimistic simulation platform where the solutions provided by the thesis have been integrated, namely ROOT-Sim. Finally, we describe the applications that have been used as benchmarks in the experimental studies.

3.1 Hardware and Base Software

Our reference computing platform is an HP ProLiant server equipped with four 2GHz AMD Opteron 6128 processors working at 64-bits. Each processor is composed by 8 cores, for a total amount of 32 cores. Each core has a private 128 KB L1 cache (64 KB data-cache and 64 KB instruction-cache) and a private 512KB L2 cache. The last level of cache (LLC), having 5118 KB capability, is shared among four cores within a single processor, for a total of 10236 KB within the same processor. The machine is equipped with 64 GB of RAM based on a NUMA (Non-Uniform Memory Access) architecture, where each group of cores that share the LLC sees 8 GB as *close memory* and the remaining 56 GB as *far memory* [1]. The schematized machine-architecture is

[1] In NUMA architectures several CPU-cores share memory resources, and the memory is split in a way that each bank is "close to" a subset of the CPU-cores, commonly called node. Each node accesses its close memory banks in fast way, while slower access is experienced for

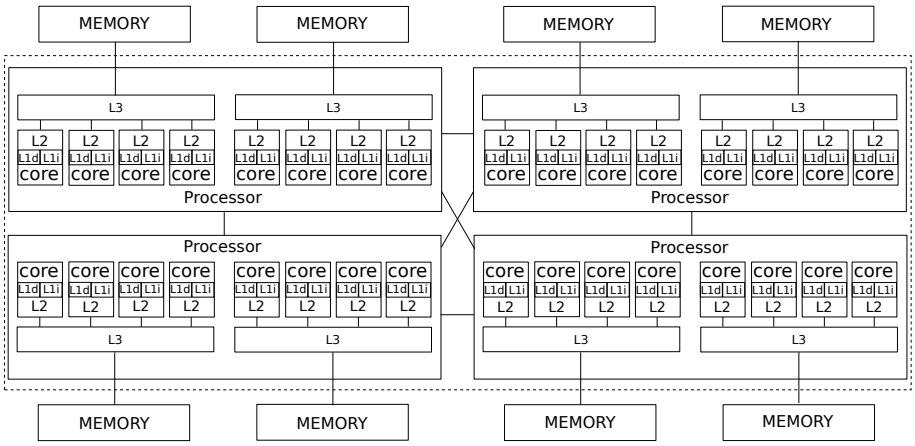


Figure 3.1: Schematized machine architecture

	L1 Data	L1 Inst	L2	L3
Ways of associativity	2	2	16	48
Type	Data	Instruction	Unified	Unified
Size (KB)	64	64	512	5118
Coherence line size (B)	64	64	64	64
Shared vs Private	Private	Private	Private	Shared (4 cores)

Table 3.1: Opteron cache details

shown in Figure 3.1, while details related to the caching system are provided in Table 3.1.

The operating system installed on the machine is 64-bit *Debian* 6, with *Linux* Kernel version 2.6.32.5. The compiling and linking tools that have been exploited are *gcc* 4.3.4 and *binutils* (*as* and *ld*) 2.20.0.

memory banks that are close to others node. The access type is called non-uniform because all the nodes see the whole memory, but each node accesses different memory portions with different latencies.

3.2 The ROOT-Sim Platform

The simulation platform that has been taken as the reference for integrating all the proposals by this thesis is the ROme OpTimistic Simulator (ROOT-Sim) [Pellegrini et al.(2011)Pellegrini, Vitali, and Quaglia, HPDCS Research Group(2012)]. This is a C/MPI-based open source optimistic simulation platform based on the Time Warp protocol [Jefferson(1985)] and tailored for UNIX-like systems. This platform has been designed as a general purpose solution, hence being suited for supporting differentiated simulation models adhering to a very simple programming model (as we shall also discuss in Section 3.2.1). The platform transparently handles all the mechanisms associated with parallelization, (e.g., mapping of LPs on different kernel instances) and optimistic synchronization (e.g., state recoverability). A schematization of the internal architecture of ROOT-Sim, as it was standing before integrating the proposals by this thesis, is shown in Figure 3.2.

At the core of the architecture, there is an event-queue manager that maintains multiple input/output queues storing incoming (or already processed) and outgoing simulation events. Each pair of input/output queues is logically associated with a same locally-hosted LP. The interaction between the event-queue manager and the MPI layer, in order to support event notification across different instances of the ROOT-Sim kernel, is mediated by a messaging manager which multiplexes ROOT-Sim defined message tags (e.g., `EVENT` or `ANTI_EVENT`) travelling across different ROOT-Sim instances over the same MPI channel. The scheduling sub-system gives control to the application layer along the same thread running the scheduler. Hence, simulation events (and the associated LPs) are dispatched for execution according to a classical time-interleaved mode, where the scheduling priority for the next-to-be-executed event across all the hosted LPs is based on the Lowest-Timestamp-First (LTF) algorithm. The scheduler can run in two differentiated modes. The first one is a stateless $O(n)$ mode, resembling the linux-2.4 scheduler, which queries the event-queue manager at each dispatch operation for getting information about the next-event timestamp (and hence the scheduling goodness) of all the LPs. The second mode [Santoro and Quaglia(2010)] operates in constant time (at least statistically), and is based on pre-populated meta-data that are constantly kept updated by reflecting the updates of the input queues of the locally hosted LPs. It results well suited for (very) large models, for which the advantages in terms of reduced scheduling latency overstep any overhead for scheduler-state maintenance.

As for state recoverability (and hence of data structures maintained at the application level), which is a crucial aspect for the design of effective optimistically synchronized environments, two main architectural approaches have been adopted. First, dynamic memory allocation and release via the standard

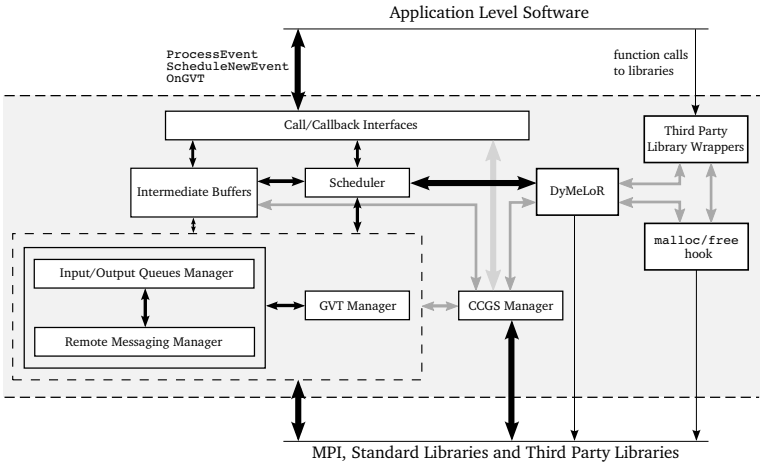


Figure 3.2: ROOT-Sim architecture

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 allocation/deallocation operations by maximizing memory locality for the state layout of each single LP, and by maintaining meta-data allowing the memory map to be recoverable to past values [Toccaceli and Quaglia(2008)]. This aspect will be further discussed in Chapter 4 since the internal architecture of DyMeLor has been exploited as a starting point for our autonomic log/restore proposal. Overall, thanks to DyMeLoR, ROOT-Sim complies to a model where an LP is a (dynamically allocated) set of data structures updated by subsequent calls to an event handler. This semantic will be maintained via the autonomic management.

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 (see [Cucuzzo et al.(2007)Cucuzzo, D'Alessio, Quaglia, and Romano]). This occurs via update operations applied to local committed checkpoints of individual LPs so to eliminate mutual dependencies among the final-achieved state values. The checkpoint update operation is completely transparent since ROOT-Sim realigns the logged state images by triggering the execution of event handlers natively present within the application code, by passing as input already committed events not yet discarded by memory recovery procedures. 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.

3.2.1 Exposed API

Beyond auxiliary functions, e.g., for accessing recoverable random number generators, ROOT-Sim supports the following API:

(A) `int ProcessEvent(int me, time_type now, int event_type, void *content, int size, void *state)` - a callback to be implemented within the application layer, which provides control to the application for the actual processing of simulation events. `me` is the identifier of the LP being dispatched, `now` is the current value for the local clock, `event_type` is the numerical code for the event to be processed, `content` is the buffer maintaining the event payload (made of `size` bytes), and `state` is the pointer to the top data structure forming the LP state layout.

(B) `int ScheduleNewEvent(int where, time_type timestamp, int event_type, void *content, int size)` - this function allows injecting a new simulation event within the system, to be destined to whichever simulation object identified via `where` (the other parameters have the above described meaning).

(C) `int onGVT(int me, void *snapshot)` - this callback passes control to the application by providing the LP snapshot belonging to the CCGS.

By the above description, it is clear that the application programmer is requested to reason on no aspect in relation to parallelism of the model execution. The programmer is only requested to understand that what is coded within the `ProcessEvent` callback will be executed speculatively. Hence, any audit on the simulation model state-trajectory (when considering committed state updates) will need to be carried out via inspection on the LPs' states

through the `onGVT` callback. No other aspect in relation to the actual execution mode and synchronization dynamics is seen by the programmer.

3.2.2 Code Examples

In this section we present some code snippets to implement a working ROOT-Sim application which models a set of N nodes connected as a mesh, sending packets randomly to each other. The simulation model implementation is straitforward thanks to the fact that all the tasks related to parallelization or other housekeeping operations are completely hidden. The first important thing is to define the possible events handled by the model, the content of an event message, and the structure of the state:

```
#include <ROOT-Sim.h>

#define INIT    0
#define PACKET 1

#define PACKETS 1000000

extern seed_type master_seed;

typedef struct _event_content_type {
   time_type sent_at;
} event_content_type;

typedef struct _lp_state_type{
   int        pckt_count;
   seed_type seed_state;
} lp_state_type;
```

Notice that in this application we allow just two events: `INIT`, sent by the simulation kernel to startup the simulation, and `PACKET`, which identifies the transit of a packet in the mesh. `PACKETS` is a macro that will be used in the termination check, while `master_seed` is an initial seed for random functions exposed by the platform.

Then, we must specify the actual logic for the `ProcessEvent()` callback. This is the only entry point at application level for processing events, so we must rely on a `switch` construct to demultiplex them:

```
void ProcessEvent(int me, time_type now, int event_type,
                  event_content_type *event_content, void *ptr) {

   event_content_type new_event_content;
   lp_state_type *pointer = (lp_state_type*)ptr;
   time_type timestamp;

   switch(event_type) {
      case INIT:
         pointer = (lp_state_type *)malloc(sizeof(lp_state_type));
         pointer->pckt_count = 0;
         pointer->seed_state = master_seed;

         timestamp = (time_type)(20*Random((unsigned long *) \
                     &pointer->seed_state));
         ScheduleNewEvent(me,me,timestamp,PACKET,NULL,0);

         break;

      case PACKET: {
         pointer->pckt_count++;
         new_event_content.sent_at = now;

         int rcv = (N_PRC_TOT * Random((unsigned long *) \
                   &pointer->seed_state ) );
         timestamp = now + (Expent(((unsigned long *) \
                     &pointer->seed_state), DELAY));
         ScheduleNewEvent(rcv,me,timestamp,PACKET, \
                   &new_event_content,sizeof(new_event_content));
      }
   }
}
```

The logic in the code is fairly simple: upon `INIT` event, the LP's state is `malloc`'d and initialized, and an initial packet is sent to the LP itself. Whenever a `PACKET` event is received, a local counter is increased, and a packet is sent back to a random LP in the simulation environment. Timestamps associated to these events are computed according to an exponential distribution, exploiting the internal `Expent()` function.

`CheckTermination()` is the second callback to be implemented in the application-level code, and it performs a local check on the LP's state. In particular, if the number of packets passed in the LP is smaller than `PACKETS`, it tells the simulation platform that the simulation cannot be halted yet:

```
int CheckTermination(lp_state_type *snapshot, int gid) {
   if (snapshot->pckt_count < PACKETS)
      return 0;
   return 1;
}
```

3.2.3 ROOT-Sim audience

ROOT-Sim is a general purpose simulation platform, it implies that the possible audience is represented by eachone needs to perform a simulation. Its intrinsic parallelism make it suited in particular to be adopted when the model to simulate is complex, both from a computational point of view or from resource requirements one, or when there are some real-time constraints to be matched. The reduced API and the trasparency of all the housekeeping operations make it suited also by all that scientists who do not have good programming skills.

3.3 Benchmark Applications

We exploited two different benchmark applications in this thesis, namely:

- PCS (Personal Communication System), and
- Traffic

Both the applications are described in details in what follows.

3.3.1 Personal Communication System

Personal Communication System (PCS) models a cellular (*connected*) network adhering to GSM technology, where each LP models the evolution of the state of an individual cell, and the whole set of cells provides wireless coverage on a square region of variable size (which depends on the total number of cells included in the model, each one assumed to cover an hexagonal region, as in typical modeling approaches [Boukerche et al.(1999)Boukerche, Das, Fabbri, and Yildz]). Each cell handles a parameterizable number N of wireless

channels, thus the whole model can be tailored for being representative of micro-cell vs macro-cell technology. Wireless channels are modeled in a high fidelity fashion via explicit simulation of power regulation/usage and interference/fading phenomena on the basis of the current state of the corresponding cell. The power regulation model has been implemented according to the results in [Kandukuri and Boyd(2002)].

The event types which can occur at any LP are:

- *Start Call*, which is intended to simulate a new call installation on a target cell;
- *End Call* which is intended to simulates a call termination;
- *Handoff Leave* which is intended to simulate the leave of an on-going call (i.e. of an active device) from the current residence cell;
- *Handoff Receive* which is intended to simulate the installation of a call handed-off from an adjacent cell;
- *Recompute Fading*, which is intended to simulate the effects of climatic variations onto the fading (and consequently interference) phenomena for ongoing calls.

Upon the start of a call destined to a mobile device currently hosted by a given wireless cell, a call-setup record is instantiated via dynamically-allocated data structures, which gets linked to a list of already active records within that same cell. Each record gets released when the corresponding call ends or is handed-off towards an adjacent 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 current call-setup to achieve the threshold-level SIR value. Data structures keeping track of fading coefficients are also updated while scanning the list, according to a meteorological model defining climatic conditions (and related variations). The climatic model accounts for variations of the climatic conditions (e.g. the current wind speed).

This application is highly parameterizable, so that several different configurations have been exploited in this thesis in order to better tailor the experiments to the specific objective. Beyond the already mentioned number N of wireless channels per cell, the set of configurable application parameters entails:

- τ_A, which expresses the inter-arrival time of subsequent calls to any target cell;

- $\tau_{duration}$, which expresses the expected call duration;
- τ_{change}, which expresses the residual residence time of a mobile device into the current cell, as evaluated since the time instant of installation of the call within the cell.

Different distributions can be used for determining samples of the above parameters, among which this thesis mostly rely on the exponential one.

A relevant dependent parameter within PCS models is the channel utilization factor, expressing the percentage of time during which the channel is busy. Particularly, the channel utilization factor depends on other parameters according to the following expression:

$$utilization factor = \frac{\tau_{duration}}{\tau_A * N}$$

The value of this parameter impacts the granularity of the events since the more the busy channels, the more power-management records are allocated and consequently scanned (or updated, as when fading gets recomputed on the basis of climatic variations) during the processing of different events. On the other hand, higher values of the channel utilization factor lead to higher memory requirements for the state image of individual LPs. Both the above dependencies (namely, CPU demand and memory) are anyhow bounded depending on the total number N of per-cell managed channels.

3.3.2 Traffic

This application simulates a complex highway system (at a single car granularity), where the topology is a generic graph, where nodes represent cities or junctions and edges represent the actual highways. Every node is described in terms of car inter-arrival time and car leaving probability, while edges are described in terms of their length.

At startup phase, the simulation model is asked to distribute the highway's topology on a given number of LPs. Every LP therefore handles the simulation of a node or a portion of a segment, the length of which depends on the total highway's length and the number of available LPs.

Cars enter the system according to an Erlang probability distribution, with a mean interarrival time specified (for each node) in the topology configuration file. They can join the highway starting from cities/junctions only, and are later directed towards highway segments with a uniform probability. Whenever a car is received, it is enquequed in the LP's list of traversing cars, and its speed (for the particular LP it is entering in) is determined according to a Gaussian probability distribution, the mean and the variance of which are specified at startup time. Then, the model computes the time the car will

need to traverse the node, adding traffic slowdowns which are again computed according to a Gaussian distribution. In particular, the probability of finding a traffic jam is a function of the number of cars which are currently passing through the node.

Accidents are derived according to a probability function as well. In particular, they are more likely to occur when the amount of cars traversing an LP is about half of the cars which can be hosted altogether. In fact, if few cars are in, accidents are less frequent. Similarly, if there are many, the traffic factor produces a speed slowdown, entailing the probability of an accident to occur to be reduced. Therefore, the model discretizes a Normal distribution, computing the Cumulative Density Function in a contour defined as $cars\ in\ the\ node \pm \frac{1}{2}$, having as the mean half of the total number of cars which are at the current moment in the system, and as variance a factor which can be specified at startup. The total number of cars which can be hosted by an LP is computed according to the actual length of the simulated road, which is determined when the model is initialized. When an accident occurs, the cars are not allowed to leave the LP, until the road is freed. The duration of an accident phase is determined according to a Gaussian distribution, the mean and the variance of which are again specified at startup.

For the experiments in this thesis, we have exploited this application to simulate the whole Italian highway network on top of 1024 LPs. We have discarded the highways segments in the islands in order to simulate an undirected connected graph, which allows to have the actual workload migrating overall the highway. The topology has been derived from [AUTOMAP()], and the traffic parameters have been tuned according to the measurements provided in [Autostrade per L'Italia S.p.A.()]. The average speed has been set to 110 Km/h, with a variance of 20 Km/h, and accident durations have been set to 1 hour, with 30 minutes variance. This model has provided results which are statistically close to the real measurements provided in [ACI()].

Chapter 4

Autonomic Log/Restore

As shown by literature studies (see, e.g., [Palaniswamy and Wilsey(1993)]), depending on the application profile (e.g., in terms of memory access pattern) and on synchronization related dynamics (e.g., the frequency of causality violations to be recovered), no one among the traditional non-incremental or incremental logging schemes can be considered as the unique winner solution. The autonomic log/restore approach we present in this chapter exactly aims at tackling this issue, which is done via time-interleaved co-existence of both non-incremental and incremental modes, with per-LP dynamic selection of the best suited mode (and of the associated optimal parameterization) based on an analytical approach. Such an approach also exhibits the feature of explicitly accounting for stability of each log mode versus fluctuations of the simulation model execution dynamics. The presented solution provides complete transparency towards the final programmer, supporting (as already done by ROOT-Sim via DyMeLoR) an ANSI-C oriented programming model that can make use of dynamically allocated memory provided through standard allocation/deallocation functions (namely `malloc` and `free` services) in order to determine the memory layout of the LP state.

4.1 Co-existence of Different Log/Restore Modes

4.1.1 Starting from Non-Incremental State Saving Supports: DyMeLor Details

The autonomic state saving architecture has been built on top of the memory manager already present in ROOT-Sim, namely DyMeLor, which sup-

ports transparent log/restore facilities for LPs with generic memory layout. DyMeLoR offers the possibility to allocate/deallocate memory chunks via standard memory allocation/deallocation API, supporting a general programming model where the state of an LP can be scattered on dynamically allocated memory chunks. It operates as a wrapper of ANSI-C `malloc/free` services, which is completely transparent thanks to the adoption of ad-hoc compile/linking time directives. For each LP to be handled, the library maintains a metadata table of `malloc_area` entries. Each `malloc_area` handles a set of same-sized chunks. As soon as a new allocation request is performed, if there is no free chunk to be delivered, a block of chunks is newly allocated (pre-reserved) and linked to the `malloc_area`. Pre-reserving provides good locality, resulting as an efficient solution for both, caching alignment and memory access. In addition, contiguousness of the memory chunks to be delivered upon LP (future) requests allows an efficient management, by the usage of a simple bitmap for the recognition of in-use and free 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.

4.1.2 Incremental State Saving Supports

Our solution extends DyMeLor allowing application transparent identification of memory chunks that have been updated during event processing for supporting Incremental State Saving. This is achieved via a software instrumentation approach. Particularly, a software instrumentation tool (IT) has been built which parses and modifies the application object files. It is 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. IT parses the object file, identifies all the write-to-memory instructions, and inserts a `call` instruction to an `update_tracker` module right before the memory-write instruction, edited in assembly language, which performs the recognition of the exact memory areas that are being written (in terms of base address and size of the write operation).

The insertion of the call to `update_tracker` before a memory-write instruction leads to shift of all the subsequent instructions, to a resize of the sections associated with the object file and to the shift of other memory locations inside the object layout. Hence, IT 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.

Given that the `update_tracker` module must operate in real-time, the instruction decode is not performed at run-time, which could be onerous, especially due to the complexity and variable format/length of the x86 (and x86-64) instruction set. Instead, at compile and linking time IT creates, populates, and links with the simulator code a table of disassembled memory-write operations, providing a hybrid approach where some compile-time tasks help reducing run-time overhead. In particular, what is demanded to run-time computation only consists of (a) the identification of the memory address of the area to be updated, since it may depend on information not known at compile-time, and (b) the branch correction for all those branches that depend on dynamic values, i.e., register values.

In order for IT to be able to build the table of `update_tracker_entry` records via correct insertion of the absolute addresses of memory-writing instructions, 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 symbols associated with application level software inside the executable layout.

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 instruction. The instruction opcode reveals 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 disassembling process, IT 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;
};
```

- `ret_addr` indicates to `update_tracker` where control will be returned after its execution. It corresponds to the memory address of the write instruction which immediately follows the current instance of the call to `update_tracker`;

- `size` indicates the size of the memory area that is being written by the instruction [1];

- `flags` is used to identify which of the four parameters potentially defining the target memory address are actually relevant and should be considered by `update_tracker` for computing the exact address for the memory-write operation.

The remaining fields (`base, index, scale, displacement`) have the same meanings as the corresponding fields in the memory access instruction.

The access to the `update_tracker_entry` occurs each time a write-to-memory is executed during event processing by any LP, thus it is a performance critical operation directly impacting the event execution costs. Also, the percentage of write-to-memory instructions can be quite huge depending on the specific application logic. For this reason we have decided to adopt a fast search hash-with-buckets table.

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 completed the search the base address and the size of the memory area being written can easily be computed by a few machine instructions.

Actually, IT 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 pre-specified threshold. In such a case, IT 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 (thanks to the tendency towards $O(1)$ time complexity, as the best case).

IT avoids to process instructions associated with memory accesses related to automatic variables (those allocated inside the stack). These instructions

[1] The only exception is for string movement instructions (`movs` and `stos`), 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`.

can be recognized since they address memory via stack pointer registers (rbp, rsp registers, in 64-bit architectures and ebp, esp in 32-bit architectures). Automatic variables are not considered given that they do not belong to the actual LP state image, since they do not survive across different invocations of the event handler.

In IA-32/x86-64 processors the destination addresses for some `jmp` instructions, namely *register jumps*, are computed at run-time, thus they cannot be corrected at compile-time by rewriting relocation tables. For *register jumps*, also referred to as *indirect branches*, the destination address is dynamically identified via the content of CPU registers.

To deal with this type of instructions, we have implemented a complementary monitoring module, called `brach_corrector`, for supporting on-the-fly correction of *indirect branches* destination addresses. It works similarly to the aforementioned monitoring mechanism since a `call` to the `brach_corrector` is inserted right before any indirect branch, whose objective is to correct the destination address. It relies on another hash table, similar to the previously described one, where each entry corresponds to a different *register jump* in the original code, and keeps information regarding which are the registers used for the computation of the destination address for that instruction. This table is again built and populated at compile-time to avoid run-time overhead. The `brach_corrector` then corrects the destination address depending on shifts operated to the instructions. It determines the shift value thanks to another table, again generated at compile-time, where all the shifts originated during the instrumentation phase are listed as pairs $< address, offset >$. The $address$ represents the original linear address within the ELF from which a shift operation had to take place due to instrumentation, while $offset$ is the actual generated shift. The table is ordered on the basis of the $address$ field, and the `branch_corrector` performs a logarithmic-cost binary search to retrieve the interval containing the original destination for the *register jump* to correct. The jump correction is not performed by modifying the register value since it would result in an incorrect processor status, instead at compile-time we substitute all the *register jumps* with *offset jumps*, and the `branch_corrector` just computes the destination address and overwrites the offset field. In order to support run-time instruction rewrite, without impacting typical settings associated with memory protection, the *offset jump* instruction has been moved to an ELF section which is writable at run-time, properly created by exploiting compiler/linker facilities. Also, at compile-time the *register jump* in the original code has been substituted with a jump-label instruction that points to the *offset jump* present in the run-time writable section. This multi-layered jump allows run-time modification of the destination address by maintaining all the application code non-writable, except for the ad-hoc section just containing the *offset jump* instruction, which is updated

whenever required by the `branch_corrector` in order to give rise to correct branches.

We note that efficient solutions for correcting *register jumps* (e.g. via the avoidance of run-time disassembling) have practical relevance since *register jumps* are typically generated by standard compilers (e.g. `gcc` version 3) for machine language translation of *Switch-Case* constructs. These constructs are quite relevant in simulation applications (e.g. for flow control inside the event handler on the basis of the type of event to dispatch), which supports the relevance of our optimizations aimed at limiting the cost of on-the-fly address correction.

The original DyMeLoR data structures and modules managing the LP memory map have been extended/modified 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 previously described. 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, 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.

4.1.3 State Log Operations

Via the exploitation of the additional fields inside each `malloc_area`, and of the dirty bitmaps, logging activities 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.

4.1.4 State Restore Operations

Each log is stamped with the current simulation time, and all the logs (full and incremental) are linked together in a timestamp-ordered 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.

4.1.5 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 our architecture. 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, 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.

[2] Flexibility in memory management via partitioning/aggregating free memory buffers according to the so called "boundary tagging" scheme [Lea(1996)] is anyway inherited from the original DyMeLoR design thanks to per-chunk headers used at the level of the underlying malloc library.

4.1.6 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 our architecture 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 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 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 of all those library functions which explicitly allocate memory and/or have an internal state.

4.1.7 The Dual-Coding Scheme

We have expanded the above design in order to provide an optimized coexistence of the two different log modes. One way to possibly achieve such a co-existence would have been to simply add a (per LP) flag indicating to the memory-update tracking monitor whether the logging layer is currently executing in incremental mode or not. In the latter case, the monitor does not actually need to perform identification of the memory chunk to be dirtied, and to flag the corresponding dirty-bit. It could simply return right after checking the flag value. However, this solution would actuate the non-incremental logging mode by running application level modules that actually experience part of the overhead associated with the memory-update tracking monitoring

mechanism (caused by a set of machine instructions, which include an explicit call for flow control variation and the associated stack-frame setup). This would mean running a non-optimized application layer configuration vs the current operating mode of the underlying log layer.

We have instead adopted a different approach where automatic ELF rewriting schemes have been again used in order to create, starting from the same set of application level modules, two different text sections within the ELF, one containing a non-instrumented version of the compiled modules, and the other one containing the instrumented counterpart. These two sections are then transparently placed within different virtual memory sections thanks to standard `ld` facilities. However, the corresponding symbol tables are modified by our preprocessing/instrumenting tool in order to expose the application interface requested by the underlying simulation kernel, namely the event handler callback, via differentiated symbols. The rodata sections corresponding to the two different text sections are modified in order to provide correct adjustment of the displacement information associated with the position of code and data within the virtual memory addressing. Also, the replicated data/BSS sections associated with the two versions of the application object code have been collapsed on the same virtual addressing range in order to provide a single actual copy of initialized and non-initialized data, accessible by both the generated code versions. A schematization of the whole process supporting such a dual-version code generation is provided in Figure 4.1, where we explicitly indicate the steps carried out by our ad-hoc automatic compile/linking time instrumentation tool. Once the executable is finally built and run, a kernel level switch between the two different log modes simply involves reassigning the event-handler callback pointer to the entry point symbol associated with the corresponding version of the duplicated application executable modules. Adopting this solution, each log mode is supported according to an optimized run-time scheme where any overheads are at all avoided while processing simulation events in case no tracking of memory update operations is requested by the currently active log mode.

The above scheme would only entail additional virtual addresses consumption due to the presence of two versions of the executable modules associated with the application layer. However, this should not represent a real problem when considering the tendency of vendors towards 64-bit processors, enabling extremely wide span of virtual memory addressing, and the fact that text sections usually fill a reduced percentage of the available virtual addresses.

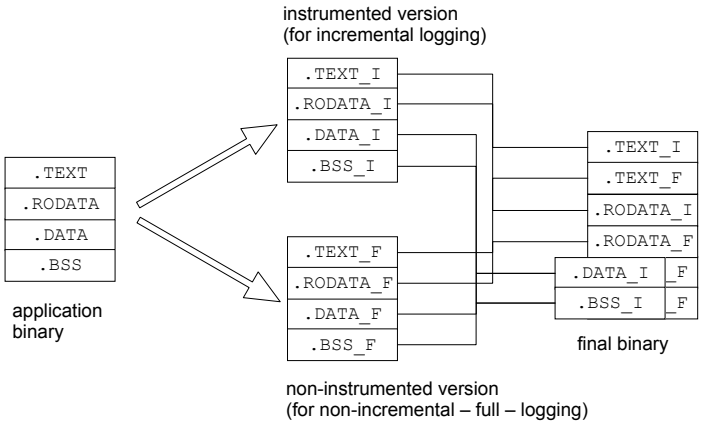


Figure 4.1: Dual-version code generation via compile/linking time ELF rewriting

4.2 Log/Restore Overhead Modeling

After having enabled the optimized co-existence of incremental and non-incremental log/restore modes, as explained in the previous section, we provide the models assessing the corresponding overhead per event (due to both log and restore operations). These models borrow from the one presented in [Rönngren and Ayani(1994b)] for periodic non-incremental logging, for which we provide both (i) a specialization to capture internal mechanisms proper of our advanced memory-map manager (i.e. the cost of managing meta-data identifying scattered memory layouts), and (ii) an extension to accommodate the case of incremental logging as supported by our architecture. Note that the model in [Rönngren and Ayani(1994b)] describes the log/restore overhead on a per-LP basis. We inherit this feature in our modeling approach, thus providing an autonomic scheme allowing dynamic optimization of the log/restore mode for any individual LP. Consequently, from now on, overhead modeling and autonomic optimization are implicitly referred to what experienced for each single LP.

For the non-incremental case, borrowing from [Rönngren and Ayani(1994b)] and recalling the aforementioned specialization, the log/restore overhead per event can be expressed as

$$OH_F = \frac{S_F}{\chi_F}\delta_{LB} + P_r(S_F\delta_{RB} + \frac{\chi_F - 1}{2}\delta_e) \tag{4.1}$$

where

δ_e is the average event execution cost.

S_F is the average size of a full (non-incremental) log.

δ_{LB} is the average cost for logging a single byte belonging to the state image, which we consider to also include the per-byte cost for logging the meta-data maintained by the memory-map manager.

δ_{RB} is the average cost for restoring a single byte from the log, which we again assume to include the per-byte cost associated with the restore of the state layout meta-data.

P_r is the rollback probability (frequency of rollback occurrences over event executions).

χ_F is the selected log interval when operating according to the non-incremental mode, which determines the expected length of the coasting forward phase, occurring after the latest log preceding the causality violation is reloaded.

By the result in [Rönngren and Ayani(1994b)], the above overhead gets minimized for $\chi_F = \left\lceil \sqrt{\frac{2}{P_r}\frac{\delta_{LB}S_F}{\delta_e}} \right\rceil$, and we denote as χ_F^{opt} the optimal non-incremental log-interval according to this equation.

For the incremental mode, as supported by our architecture, 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. Accordingly, state reconstruction at whichever simulation time can be supported via a mix of state restore from the log, and classical coasting forward. Also, full logs can be (infrequently) interleaved with incremental logs to enable fossil collection of incremental log records with timestamp less than the timestamp of the latest committed full log. These full logs are anyway exploitable during recovery procedures since, while backward traversing the log chain, the restore operation of a complete state image gets finalized by extracting from the log all the in-use chunks that have not yet been restored via the scan of incremental logs, and putting them back in place within the state layout. To account for such optimized internal mechanisms offered by the memory-map manager, the above equation can be adapted as shown below to model the log/restore overhead for the incremental mode

$$\begin{aligned} OH_I &= \frac{S_P}{\chi_I}\delta_{LB} + \frac{(S_F - S_P)}{\chi_I \chi_{I,F}}\delta_{LB} + \\ &\quad P_r\left[S_F\delta_{RB} + \frac{\chi_I - 1}{2}(\delta_e + \delta_m)\right] + \delta_m \end{aligned} \tag{4.2}$$

where the additional/different terms in the equation have the following meaning

S_P is the average size of a partial (incremental) log.

X_I is the selected log-interval when operating according to the incremental mode, which again determines the expected length of the coasting forward phase after the reload of the latest valid state image from the log.

$X_{I,F}$ is the interleave step between full and incremental logs (number of incremental log operations after which a full-log is taken).

δ_m is the cost for running the memory-update tracking module.

In equation (4.2), the term $S_F\delta_{RB}$ accounting for the cost of state reload from the log is comparable to the one in equation (4.1), due to the aforementioned mechanism, according to which all the in-use chunks belonging to the state image are restored (by retrieving them either from the incremental logs along the log chain, or the first full log found during the log chain backward traversing procedure). Further, each event is charged with the memory-update monitoring overhead δ_m, which also appears during costing forward. By exploiting the same arguments used in [Rönngren and Ayani(1994b)] for the minimization of the overhead vs the log interval in the context of non-incremental logging, we get that the optimum value for the interval of incremental logs can be computed as $\chi_I = \left\lceil \sqrt{\frac{2}{P_r}\frac{\delta_{LB}S_P}{\delta_e+\delta_m}} \right\rceil$, and we denote as χ_I^{opt} the optimal interval according to this equation. Also, by the benchmarking results in [Vitali et al.(2009)Vitali, Pellegrini, and Quaglia], a well suited value for $\chi_{I,F}$, providing no significant additional overhead due to full logs, while ensuring efficient memory recovery during fossil collection, is on the order of 10. We have used such a value as a configuration setting for the autonomic log/restore layer.

4.3 Autonomic Optimization

By the analysis in the previous section we have, for each of the two co-existing log modes, the description of their overhead, together with the indication of the optimum value of the corresponding independent parameters, namely the log-intervals. In our autonomic log/restore architecture, these models are not used to simply select as the best operating log mode the one for which the corresponding expected overhead is minimal (once identified the best log-interval value). Instead, the best suited mode is identified as the one providing the best performance despite plausible fluctuations that can affect

the parameters appearing within the overhead models (e.g. the expected event execution cost δ_e), which cannot be directly controlled since they depend on proper run-time dynamics related to the simulation model execution within the optimistic run. This set involves all the parameters appearing within the performance models, except the log-intervals χ_F^{opt} and χ_I^{opt} (or $\chi_{I,F}$), that can be controlled at run-time by the autonomic log/restore architecture.

Such an approach, actually aimed at pro-actively providing stability of the optimal performance, exactly matches characterizing aspects of the innovative autonomic-computing paradigm. Also, it well fits performance optimization when the set of possible operating modes is differentiated, each of them providing different overhead sensibility vs parameter fluctuations and/or variations. Literature approaches for log/restore optimization do not cope with such a multiple operating-mode scenario, which is the reason why sensibility of the a-priori uniquely selected operating mode vs parameter variations did not require to be addressed. Overall, our autonomic scheme for the selection of the best suited operating mode is based on a cost function $CF(\chi_F^{opt}, \chi_I^{opt})$ defined as

$$CF(\chi_F^{opt}, \chi_I^{opt}) = OH_F(\chi_F^{opt}) - OH(\chi_I^{opt}) \tag{4.3}$$

and on the result of the integration of this cost function over a multi-dimensional domain defined by the values of the parameters $(\delta_e, \delta_m, \delta_{LB}, \delta_{RB}, P_r, S_F, S_P)$. The integral function allows us to take into account the possible fluctuations of the parameters the cost function is evaluated on.

$$\int \cdots \int_D CF(\chi_F^{opt}, \chi_I^{opt}) dD =$$

$$\int \cdots \int_D \left(\frac{S_F}{\chi_F^{opt}} - \frac{(\chi_{I,F}-1)S_P + S_F}{\chi_I^{opt}\chi_{I,F}}\right)\delta_{LB} + P_r\left(\frac{\chi_F^{opt} - \chi_I^{opt}}{2}\delta_e - \frac{(\chi_I^{opt}-1)}{2}\delta_m\right) - \delta_m \, dD \tag{4.4}$$

$$where \; D = \{\delta_e, \delta_m, \delta_{LB}, \delta_{RB}, P_r, S_F, S_P\}$$

For each parameter x defining a dimension of the integration domain, we integrate the cost function over the interval $\bar{x} \pm \alpha\bar{x}$, where we suggest $\alpha = 0.1$ to capture statistically relevant fluctuations of the parameters that can be envisaged at the time the dynamic selection is carried out. If the integration result is negative, then the selected operating mode is non-incremental (with the log-interval set to χ_F^{opt}), otherwise the incremental mode is selected (with log-interval set to χ_I^{opt}). Assuming the independence of the parameters defining the integration domain (which is reasonable in our approach since the mean values are operatively determined by direct sampling of the corresponding stochastic processes - see Section 4.3.1), the integral function for

$CF(\chi_F^{opt}, \chi_I^{opt})$ is a polynomial, having the following simple form, which allows non-costly evaluation

$$\left(\frac{S_F^2}{2\chi_F^{opt}} - \frac{(\chi_{I,F}-1)S_P^2 + S_F^2}{2\chi_I^{opt}\chi_{I,F}}\right)\frac{\delta_{LB}^2}{2} + \frac{Pr^2}{2}\left(\frac{\chi_F^{opt}-\chi_I^{opt}}{2}\frac{\delta_e^2}{2} - \frac{\chi_I^{opt}-1}{2}\frac{\delta_m^2}{2}\right) - \frac{\delta_m^2}{2} \tag{4.5}$$

After the substitution of the integral domain variables we obtain the following result:

$$\left(\frac{2\alpha\bar{S_F}}{\chi_F^{opt}} - \frac{(\chi_{I,F}-1)2\alpha\bar{S_P} + 2\alpha\bar{S_F}}{\chi_I^{opt}\chi_{I,F}}\right)2\alpha\bar{\delta_{LB}} + 2\alpha\bar{Pr}\left(\frac{\chi_F^{opt}-\chi_I^{opt}}{2}2\alpha\bar{\delta_e} - \frac{\chi_I^{opt}-1}{2}2\alpha\bar{\delta_m}\right) - 2\alpha\bar{\delta_m} \tag{4.6}$$

Given that we only need to determine the sign of the above expressed value, we have finally divided it by 2α, in order to get rid of some machine instructions for multiplications.

The above optimization procedure requires defining a trigger for the evaluation of the integral function in order to dynamically actuate the selection of the best suited log-mode. In our autonomic system, we assume that the simulation run is partitioned into a startup phase and a normal phase. For the startup phase one of the two possible log modes is selected by default, and is kept until the end of that phase. Then, before starting the normal phase, the integral function is evaluated by using the mean $\bar{x}$ and the corresponding relevant statistical fluctuation $\alpha\bar{x}$ for the above parameters defining the integration domain, on the basis of samples observed during the startup phase. Actually, the mean can be computed in a very fast incremental manner not requiring the store of individual samples, thus not even impacting memory consumption.

Once the best suited log mode is selected at the end of the startup phase, subsequent re-selections can occur during the normal phase. The re-selection trigger is based on the current value of the mean $\bar{x}$ of any of the parameters defining the integration domain, and a predicate involving the values $\bar{x}^*$ and $\alpha\bar{x^*}$ that were used upon the last log mode autonomic selection. If for whichever parameter x the expression $|\bar{x} - \bar{x}^*| > \alpha\bar{x^*}$ becomes verified during the run, then the integral function is recalculated on the basis of current mean values. The reason for such a trigger is that the last dynamic selection of the

best suited log mode has been actuated on the basis of statistical parameter values $\bar{x}^*$ and $\alpha\bar{x}^*$ that can be considered no more representative of actual run-time dynamics and related fluctuations. In case the current mean goes outside the integration interval for the corresponding parameter, it is likely that some relevant variation has actually occurred within the run time dynamics, which requires re-evaluating the decision about the best suited log/restore mode. In other words, fluctuations (around expected parameter values) accounted for in last log-mode selection step are no more representative of the current system behavior. As a last observation, instead of using the arithmetic mean, we relied on the exponential mean, with weighting parameter set to 0.1, which allows better reactiveness of the mean value vs variations of the corresponding stochastic process.

4.3.1 Run-time Parameter Sampling

As hinted, our approach relies on the mean value of the parameters appearing in the performance models in (4.1)-(4.2), which are used to define the integration boundaries within the corresponding multi-dimensional integration domain. We rely on a run-time sampling process for computing the mean of each parameter. One relevant difficulty is related to the fact that the mean value of every parameter x appearing in the performance models actually requires to be tracked by the sampling process over time, independently of the current operating mode of the log layer (incremental vs non-incremental). This is because the mean is used both to trigger the re-selection process of the best suited log mode, and to determine the actual outcome of the selection. Accordingly, the parameters δ_m and S_P, specifically used to capture run-time costs proper of the incremental log mode, require to be sampled even when the non-incremental mode is currently operating. Ad-hoc schemes to address this issue will be provided and discussed in this section. We do not explicitly address the issue of sampling the value of P_r since we rely on typical approaches (such as [Rönngren and Ayani(1994b)]) based on counting the number of rollbacks over a given interval of executed events.

Event and Memory-Update Tracking Costs

To determine event and memory-update tracking costs $\delta_{tracking}$, our autonomic layer implements a sampling mechanism based on the hardware tick counter, offering a single clock tick granularity. This approach is not intrusive at all since it relies on a single machine instruction, namely `rdtsc`, to retrieve the current number of ticks from the machine start up.

A per-LP counter *Count* internal to the autonomic layer is kept, which

is used to determine the number of invocations of the memory-write tracking routine occurring during the processing of each event. In case the current log mode is incremental, the application level modules whose execution is currently triggered with invocation of the proper callback entry point (according to the dual-version-code scheme presented in Section 4.1.7) embeds the memory-update tracking routine, which increments *Count* upon its execution.

Given that the monitor is active only when running in incremental log mode, its value must be estimated during a non-incremental phase. To this end, we have slightly modified the dual-version code generation procedure in such a way that the code version running when non-incremental log/restore is active embeds a very light instrumentation scheme where each memory-write instruction is preceded by a single `ADD-r/m32,imm8` assembly instruction allowing the update of *Count*. In this way, we can infer the value of δ_m^i by simply multiplying the i-th sample of the counter value by an estimated value $\delta_{tracking}$, exactly as if the incremental mode were active. We note that this approach requires instrumenting memory-writes via a negligible overhead (just thanks to the single machine-instruction instrumentation approach), hence not altering the validity of the overhead model in expression (4.1), describing the case of non-incremental logging, which excludes costs associated with instrumenting instructions within the application code.

Of course, to estimate an accurate value of monitor's execution time when running in non-incremental mode, some samples coming from real executions of the monitoring routine should be used. To cope with this issue, we can starup the simulation by adopting the incremental mode as the default initial mode and exploit the estimation of $\delta_{tracking}$ performed during the initial phase.

We note that the above mechanisms based on real-time clocks accessed by `rdts` directly fits cases where the computing platform is dedicated to the parallel simulation run, as typical of scenarios where performance is a critical factor. In case of time-sharing with other applications, such an approach needs to be complemented with solutions based on code pre-analysis and lightweight run-time profiling such as the one discussed in [Quaglia(2001)].

Size of Full and Partial Logs

Samples S_F^i of the size of full (non-incremental) logs can immediately be taken by the autonomic layer independently of the currently active log mode since the memory-map manager maintains meta-data (i.e. an accumulator) recording at any time the real memory occupancy of the object state image (in terms of the amount of bytes associated with currently allocated chunks). Hence, S_F^i samples can be taken by simple querying the memory-map manager for the value of the accumulator. In our implementation, the autonomic layer queries

the memory-map manager each time a log (incremental or non-incremental) is taken.

A different approach is instead required for taking S_P^i samples of partial (incremental) logs. Specifically, when the currently active log mode is incremental, the memory-map manager updates a second accumulator accounting for the amount of bytes associated with chunks that have been dirtied since the last log. The accumulator is updated on the basis of actual memory-write operations that are tracked at run-time. This accumulator was already included within the extended DyMeLoR design we have presented since it was used to determine the size of the buffer destined to keep the incrementally dirtied chunks. The value of this accumulator is therefore directly used as a valid S_P^i sample when the incremental log mode is active.

In case the current log mode within the autonomic scheme is non-incremental, the above accumulator does not get updated. Hence, we have decided to infer the value of S_P^i according to the following different approach. Each $K \times \chi_F^{opt}$ non-incremental log operations, we flag the corresponding LP so that, after the subsequent event is executed by the LP, we compare chunk by chunk the current memory image content after the event with the last one packed within the log buffer. The comparison is carried out only over chunks that belong both to the memory image packed within the log buffer, and to the current memory image, hence taking into account the portion of the state layout that is stable across the two subsequent snapshots.

Obviously, the cost of this operation depends on the value of K and on specific optimizations for the comparison of each couple of chunks. As for the second aspect, we have not employed the traditional `memcmp()` service since, depending on the implementation, it might not provide early stop upon detection of the first different byte between the two memory chunks. We have therefore developed efficient, ad-hoc assembly modules that iteratively compare memory areas by fully exploiting the size of CPU registers at each compare-step, and that exactly implement the early stop procedure upon the detection of the first different byte between the two chunks. This matches the chunk-based granularity offered by the log/restore approach within the autonomic layer. Also, these modules are optimized in order to maximize the likelihood of actual early stop in case of different chunks between the two snapshots according to the following scheme. Small size chunks are checked within the comparison process by starting from the top byte, and then going towards the bottom. Instead, for large chunks, we have implemented a procedure that checks the bytes in an interleaved mode starting from the top and from 3/4 of the chunk size. The above approach well fits typical programming practices, which tend to structure records in such a way that the most frequently touched data are at the top of the record and/or at the bottom (see, e.g., pointers for linking between memory scattered dynamically allocated records). Hence, for

large chunks, it is better to check top/bottom portions with higher priority. Also, starting from 3/4 of the chunk size accounts for internal chunk fragmentation, due to the typical un-correlation between the size of the record to be placed by the application software within the allocated chunk, and the actual size of the chunk that best fists the allocation request, among those managed by the memory management subsystem (defined according to power-of-two values). Once identified the dirty chunks according to the above scheme, on the basis of the aforementioned stable portion of the snapshot, the corresponding percentage p of dirty bytes is applied to the total current state size S_F^i to generate the j-th S_P sample as $S_P^j = p \times S_F^i$.

Concerning the value of K, namely the second factor determining the actual overhead due to the estimation of S_P samples when the non-incremental log mode is active, we have used a static approach where K is set to the value 20. Given that the cost associated with the estimation procedure for a single sample is, at worst, comparable with the one for a full-log operation [3], this would simply increase the real overhead experienced when the non-incremental mode is active by, at worst, 5% of the corresponding logging overhead.

By the above optimizations, the overhead for determining S_P samples when the non-incremental mode is operative is expected to be negligible, thus again not altering the validity of the non-incremental overhead model in equation (4.1).

Per-Byte Log/Restore Costs

The last parameters involved in the sampling process are the per-byte log/restore costs, namely δ_{LB} and δ_{RB}. However, δ_{RB} does not appear in the final formula and we concentrate on δ_{LB}. To sample δ_{LB}, we have again exploited `rdts`, in combination with the sampling process of S_F and S_P depicted in the previous section. In particular, the i-th log operation latency, say Δ_{log}^i, is sampled via `rdts` and is normalized to either the corresponding S_F sample, or the corresponding S_P sample, just depending on the currently active log mode. Given that Δ_{log}^i also accounts for the cost of manipulating and logging meta-data associated with the logged chunks, the normalization allows taking samples for δ_{LB} actually expressing how the meta-data management cost is charged on the log operation of each single byte.

[3] Memory compare operations are in fact similar in cost to memory copies, since they both involve similar memory/register data moves. Also, the early stop for chunk compare operations should additionally favor the latency of comparing the chunks across the stable portion of the snapshot.

4.4 Experimental Results

In this section we report an experimental study for the assessment of the effectiveness of the autonomic log/restore proposal. To this end, we use two different configurations of the PCS application. In one configuration we simulate 1024 cells, each one managing up to 1000 wireless channels, where the expected duration of a call $\tau_{duration}$ has been set to 120 sec, the residual residence time for an active call in the current cell τ_{change} has been set to the value 300 sec, while the inter-arrival time τ_A has been varied during the simulation so to generate a configuration where the actual load on the cells depends on the period of the day. Specifically, 17 hours of operativity of the cellular system have been simulated (from 00:00 AM to 17:00 PM) with variations of τ_A in the interval [0.64, 3.20], with peak intensity of the workload during the morning till lunch time, and minimum load very early in the morning (around breakfast). Consequently, the utilization factor has been varied in the interval [0.31, 0.06]. For this configuration of the PCS model, climatic conditions have been set as good and steady, thus not causing the need for frequent recalculation fading coefficients. On the other hand, the second configuration of PCS that has been considered has been parameterized by having the expected inter-arrival time τ_A fixed to the value 0.8 (which gives rise to channel utilization values on the order of 25%), which leads to focusing the simulation on a morning operativity scenario, but where the climatic conditions exhibit variations that lead to periods where frequent recalculation of fading coefficients need to be operated. Both the above configurations lead to run time dynamics that vary, e.g., in terms of event granularity and portion of the LP state that needs to be updated by the events, however this is achieved in different manners in the different scenarios.

For the two configurations, we report in Figure 4.2 and in Figure 4.3 the cumulated committed events achieved by the parallel run vs the wall-clock-time. These values refer to what observed on a single simulation kernel instance (over the 32 active instances, which exhibit anyhow very similar dynamics) and have been computed as the average over ten runs (done with different pseudo-random seeds), with a minimal variance observed across different runs. This parameter (and the pendency of the associated curve) indicates the speed according to which a given platform configuration commits events, and hence how fast the configuration allows model execution. We report three plots referring to (i) the case in which the autonomic layer is active (ii) the case in which the autonomic layer is active, but we always force the incremental log/restore mode, with the corresponding optimized value for χ_I and (iii) the case in which the layer is active but the non-incremental (full) log/restore mode is forced, with the corresponding optimized value for χ_F. The plots for cases (ii) and (iii) express performance levels that could be achieved via an optimized

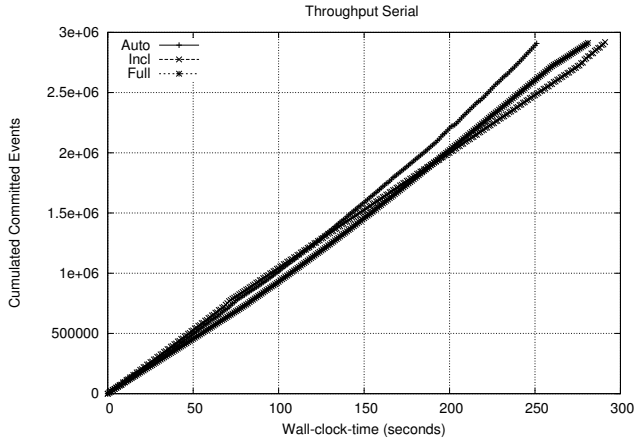


Figure 4.2: Simulation throughput for Autonomic Memory Manager: PCS benchmark variable τ_A

log/resotre mode (adaptive in the selection of the log interval) based on either the incremental or the non-incremental log mode, but not allowing autonomic switch between the two modes on the basis of run-time dynamics.

By the results, we see that, depending on the specific phase within the simulation run, (e.g. early morning vs lunch time for the case of variable τ_A) forced-incremental and forced-full modes alternately exhibit better execution speed (which is indicated by the different pendency of the cumulated committed events curve while the run is in progress). Anyway, the most important outcome by the cumulated event rate plots is that the autonomic configuration always switches to the best performing mode (incremental vs non-incremental) depending on the currently simulated period, and hence depending of the actual dynamics (e.g. in terms of state size, event granularity, memory update pattern and so on). The overall effect is that the autonomic mode actually allows faster execution, on the order of 10% to 14% over the other modes for the case of the variable τ_A configuration, and on the order of 11% to 27% for the case of fixed τ_A and variable climatic conditions. Given that the other modes represent anyway optimized configurations, the achieved improvements support high effectiveness by the autonomic approach.

We also report in Table 4.1 the execution time values for running the PCS applications (same identical code used for the parallel runs) in serial mode on top of a calendar queue scheduler. By the data we see how the parallel runs with the autonomic log/restore scheme allow significant speedups, especially for the case of fixed τ_A and variable climatic conditions (since the application is less local in terms of accesses to the LP state layout thus further not favoring the sequential run due to reduced effectiveness of the cache). Overall, the

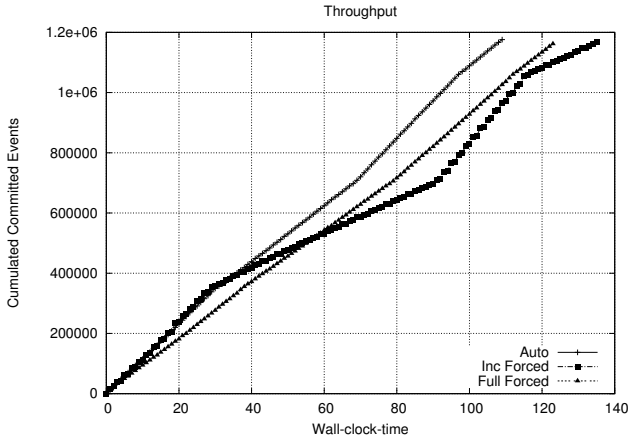


Figure 4.3: Simulation throughput for Autonomic Memory Manager: PCS benchmark fixed τ_A and variable climatic conditions

PCS configuration	execution time	speedup by the parallel run with autonomic log/restore
variable τ_A	6400	25.6
fixed τ_A - frequent fading recalc.	4442	40

Table 4.1: Results for serial execution of the PCS application and speedup values

experimental study has been carried out in a scenario entailing competitive parallel dynamics.

The above discussed results provide a view of the overall performance achievable by the autonomic proposal. To complement these results, we report two additional plots related to the goodness of internal tasks/dynamics within the autonomic layer. In particular, we report in Figure 4.4 data related to the estimation of the dirty portion of the LP state, when executing according to the non-incremental mode. These data refer to the case of PCS configured with fixed τ_A and variable climatic conditions, since with this configuration we have a relatively stable size of the whole LP state but with a very variable read/write access pattern within the state image, which represents a good test case for the target objective. We recall that the estimation is based on chunk comparison between successive state images only over the stable portion of the state snapshot (see Section 4.3.1). In particular, we report the ratio between the estimated size of the dirty portion of the LP and the actual size

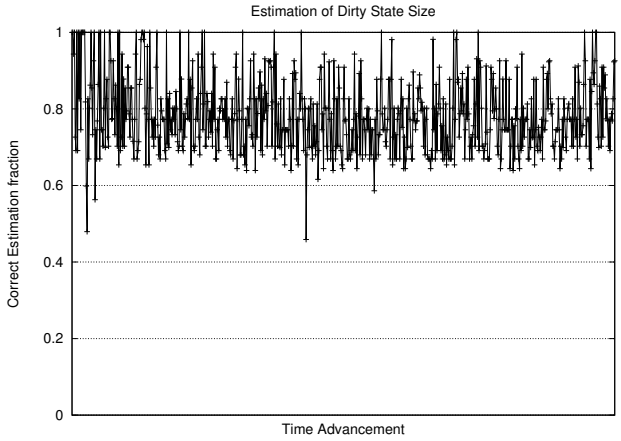


Figure 4.4: Estimation of the dirty portion of the LP state

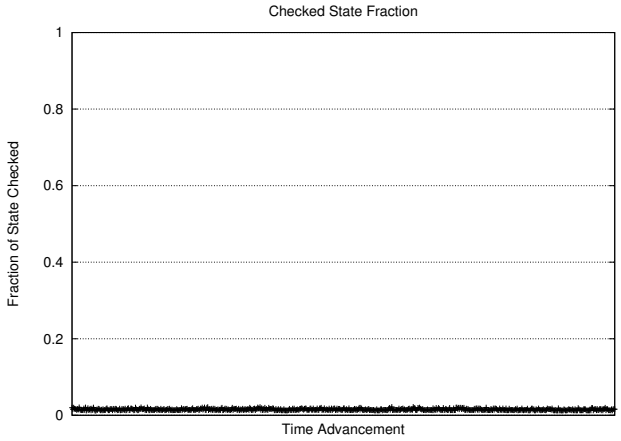


Figure 4.5: Effects of early stopping schemes

(as observed by actually tracking memory updates while executing according to the incremental mode). The plotted curve refers to a fraction of the whole simulated time interval, however, the data are representative of the overall simulation model execution dynamics. By the data we see that the error in the estimation process is on the order of no more than 20%. Finally, in Figure 4.5 we show, for the same PCS configuration, the effects of the early stop approach to chunk comparison (see again Section 4.3.1). In particular, we report the ratio between the actual number of compared bytes (across two subsequent state images) and the total amount of bytes forming the dirty portion of the reached state image. Given that our implementation of the GSM system collocates frequently accessed fields associated with on-going call records at the top of the records, the early stop approach provides actual

advantages by allowing chunk comparison to be executed only over 5% of the stable portion of the snapshot. We expect such an optimization to even provide scale-up advantages for generic simulation models entailing scaled-up LP state size, compared to the simulated GSM system, provided that the above, common field-collocation approach onto the used memory chunks is adopted.

Chapter 5

Cache Aware Memory Delivery Mechanisms

In this chapter we present an approach for improving the exploitation of the caching system in the context of optimistic PDES platforms. We envisage two motivations for this type of work. First, optimistic simulation platforms may suffer from non-local behavior, since several housekeeping tasks, which are mandatory for correctness of the execution, may exhibit access to endemically sparse data structures, thus potentially leading to in-cache replacement of access-intensive data structures. These can be replaced in cache by data that could be no longer accessed while the run is in progress (as for the case of a state-log which is not eventually used for recovery purposes). Second, the current trend towards many-core architectures can be expected to give rise, in the long period, to scenarios characterized by a tighter match between the amount of LPs and cores. This would be expected even for relatively large simulation models, and for model execution on top off-the-shelf machines. As a consequence, an increased amount of per-LP available cache-storage is expected, which, if effectively exploited, could provide further enhancement of the overall system performance. Further, the relevance of effective cache exploitation in optimistic PDES systems hosted by multi/many-core machines also arises since cache-misses, beyond directly leading to increased memory access latency, also give rise to higher bus contention.

5.1 Cache-Aware Memory Manager Design

5.1.1 Rationale

In order to increase the portion of the working set which is resident in cache, we propose to allocate the buffers used by the optimistic PDES system, which are expected not be further accesses (or to be accessed with reduced probability), in a way such that they will collide with each other (i.e their addresses will be mapped to the same cache regions) leaving untouched cache areas that are meant to contain data structures which are likely to be accessed more frequently. To this purpose, we have to discriminate between memory buffers which are *access-intensive* and *access-mild*. In order to perform this classification correctly, we have to follow through our analysis in a separate way for the application level and the simulation kernel.

By the typical organization and the typical tasks supported by the optimistic PDES system, we can classify the input message queues of the LPs as access-intensive data structures. In fact, upon the scheduling of any new event within the system (which is an inherently frequent operation) the target queue must be scanned for insertion and/or re-balanced, according to its actual implementation. Other data structures are instead commonly accessed at particular positions, except for cases where rollback is actually triggered or when periodic memory management (e.g., recovery) activities are executed. As an example, the output queues are typically accesses only at the tail (upon logging a newly scheduled event) and are partially scanned only when generating antimessages or when collecting fossils from the queue. Therefore giving these structures the possibility to invalidate cache buffers which can be accessed in the near future can make the system suffer from secondary effects.

As far as the application-level software is concerned, an a-priori decision is hard, given that we want to provide the user with complete transparency, and considering that we target general purpose optimistic simulation platforms. Therefore, the actual access pattern of the simulation model cannot be inferred with no additional knowledge. However, considering that event execution relies only on the LP state layout to produce advancements of computation, we can conservatively assume that the whole simulation state is formed by access-intensive buffers. This might seem a strong assumption, but in fact it provides no performance decrease even when the application shows a completely nonlocal behaviour. In fact, if we suppose that a simulation model has no locality at all in its memory accesses, then cache usage will become similar to the one shown by an application which makes no assumptions at all. The exhibited performance will therefore be similar to the one of a simulation not using our Memory Manager. Thus, provided that most operations in the simulation kernel have a large (sparsely/infrequently-accessed) working set as well,

memory accesses will show to be more cache-miss prone. In order to increase the in-cache resident set, we propose a Memory Management subsystem which partitions the cache between access-intensive and access-mild memory buffers, trying to embank the cache-miss phenomenon typical of optimistic simulation platforms. Basically, our proposal preallocates a cache-aligned portion of the available address space and serves memory requests in a differentiated way depending on their expected access rate, as it will be discussed in the next section.

5.1.2 Design Details

This memory manager works preallocating a cache-aligned memory area used to serve requests, called `stock`. The `stock` is split in portions, called `block`s, with a size equals to the lower-level cache available in the system. Each `block` is split in two regions, one for hosting access-intesive buffers, and the other for access-mild buffers. Memory buffer requests are served from fixed-size chunks within memory blocks, and depending on their access patterns, they are clustered in a way such that they will be mapped to separate cache regions of different sizes (i.e. the access-mild region will be smaller than the access-intensive one). Following this policy, whenever the execution undergoes a housekeeping or management operation, memory accesses will not invalidate application-level data related to the actual simulation working set in the caching architecture, thus providing an enhancement in event execution data locality. In order to reduce internal fragmentation, chunk size contained into a memory block is determined at runtime upon receiving a request. In this way, a memory block can contain chunks of different sizes for the application and the kernel levels. This is important because usually kernel and application level memory requirements are different, simulation models requiring smaller buffers and simulation kernels requiring larger ones. If a memory block were to contain same-sized chunks for both layers, internal fragmentation could arise, since blocks containing application-level chunks of a certain size would not likely contain any kernel-level chunk of the same size, and vice versa. Of course, in order to enhance even more cache locality, memory chunks should be allocated in a cache-stripe aligned way. This choice allows the underlying hardware architecture to reduce the number of cache misses and, whenever one is encountered, the number of memory chunks replaced in cache is minimum, as a chunk is overlapped to more than one stripe only if its size is greater than the stripe's. We note that if the whole preallocated memory is cache-aligned, then this behaviour can be easily obtained by fine tuning chunks' sizes as powers of 2. Given that the cache is partitioned between access-intensive and -mild buffers, we note that to avoid interfering with the alignment, the separa-

tion between the two regions must be stripe-aligned as well. The last general consideration we want to point out is concerned about the transient behaviour at the beginning of the simulation execution, which can in turn affect the entire simulation as well (in the context of multicore cache sharing). In fact, considering that simulation kernels are handling memory addresses separately (i.e. there is no run-time agreement between different kernel instances on how the memory should be allocated), at the beginning of the execution, different simulation kernel instances will start allocating buffers which will be mapped to the same cache locations, as long as the memory manager's behaviour is deterministic. If the application-level execution pattern is likely to allocate memory during the whole execution (i.e. the simulation state can grow indefinitely) and operate uniformly at random on it, this transient behaviour will produce an increase in cache invalidations mostly in the initial part of the simulation. On the other hand, let us consider the case where the application level allocates the whole simulation state at simulation startup (i.e. an operational behaviour where the simulation state is non-growing). Cache conflicts related to allocation determinism will produce a bias in cache exploitation which will produce a performance much smaller than the one generated by a common allocator which tries not to optimize with respect to the caching architecture. In fact, different kernel instances' (i.e. processes') buffers will conflict during the whole simulation, producing a large number of cache misses. This problem can be faced by forcing the memory manager to serve memory requests starting from different memory addresses according to the actual simulation kernel instance it is running within, following some circular allocation policy.

5.2 Details on Actual ROOT-Sim Integration

Integration within an operating environment requires allocation/deallocation requests to be wrapped in order to redirect them towards the cache-aware memory manager. If the simulation platform is "context aware", i.e., it can distinguish when running in kernel mode or in application mode, the integration is straightforward. ROOT-Sim already relies on a `malloc` hooking facility, which allows to determine whether an invocation is issued by the application-level software or the simulation kernel. In such a scenario, we need anyhow to discriminate among *input queue* allocations and all other kernel-level allocations. To accomplish this task, new function called `intensive_buffer(int true))` has been added within the system, which is used to override the invocation-context-based decision, where the parameter indicates whether the allocation should be served via an access-intensive or access-mild buffer. This approach is still transparent to the application level.

The core data structure of the cache-aware memory allocator is the

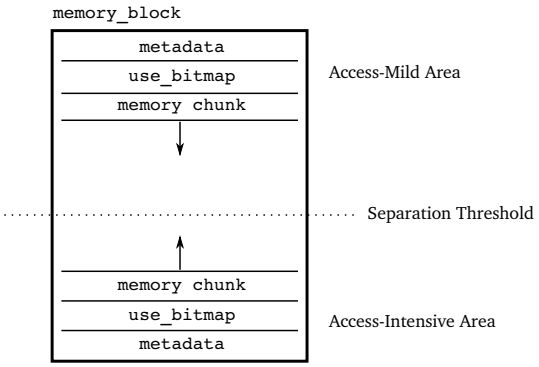


Figure 5.1: `memory_block` structure

`memory_block` (shown in figure 5.1). These blocks are preallocated as in an array to form a `stock`. Each `memory_block` is split in two region, one for serving access-intensive buffers and one for serving access-mild ones. Depending on the current request to serve a memory chunk is taken by the former or the latter region. Each region within a memory block has a different chunk size for serving different memory requests. In fact, as explained in section 5.1, the two regions can have different chunks sizes. The part of `memory_block` reserved for one area or the other is based on a threshold set at compile time. The importance of the threshold selection will be experimentally shown in section 5.3. Particularly, a too large access-intensive area would leave some cache entries unused, that could have been exploited by kernel-level data structures, on the other hand a too small access-intensive area, could increase the collisions within this area, that is counterproductive. In fact, thanks to the locality behavior of the data within this area, less cache-misses could have been achieved sharing the whole cache with the (non-local) kernel-level data structures.

Upon a `stock` allocation all its `memory_block`s are marked as invalid. In fact, initialization of `memory_block`s' internal data structures is performed in a lazy way, delaying it until a memory request actually requires a chunk to be acquired from the not-yet-valid `memory_block`. Both the areas in a `memory_block` maintain some header information, including a bitmap for the chunks status, put at the head, or at the tail, of the `memory_block`. The header is structured as follows:

```
struct memory_block {
    void *init_address;
    int num_chunks;
```

```
        int busy_chunks;

        size_t chunk_size;

        int next_chunk;

    };
```

- `init_address` indicates where the chunks being part of the `memory_block` start. As depicted in Figure 5.1, chunks are allocated stacked upon each other, with opposite growing direction depending on whether they are regarded as access-intensive or not. It is used also as a valid flag, in fact if it is `null`, it means that the block is not valid;
- `num_chunks` indicates the number of chunks being part of the block. In fact, given that the chunks size within blocks is variable, each block would shown its own number of chunks;
- `busy_bitmap` is a bitmap indicating if the corresponding chunk is in use or not;
- `chunks_size` indicates the chunk size for the current block;
- `next_chunk` is used as starting point for the identification of the next available chunk;
- `next_size - prev_size` are used to link `memory_block`s in an ordered list, where, for each (active) size, the firstly activated block belongs to the list;
- `next_same_size - prev_same_size` are used to link the `memory_block`s of the same size in a bucket-list.

The manipulation of `next_chunk` is based on the algorithm used by Linux for the identification of a process next free file descriptor. In particular it gets increased upon a new chunk reservation, to avoid scanning the bitmap from the beginning for any new allocation, and upon a release it is set to the released chunk index because it will be the first free position. It uses a best-fit policy, although all the chunks within the block are same sized, which is aimed at reducing both free-chunks and bitmap fragmentation by having allocated chunks mostly aggregated in the initial part of the block.

The `next_size - prev_size` are used to fast detect a block of a certain area. Upon a new allocation request, the list ordered on chunk-size it is scanned. It keeps a pointer to the first active block for each size of already in use chunks, thus returning the `memory_block` that can deliver the request reply, avoiding a scan on all the active blocks. If no-block is present for the

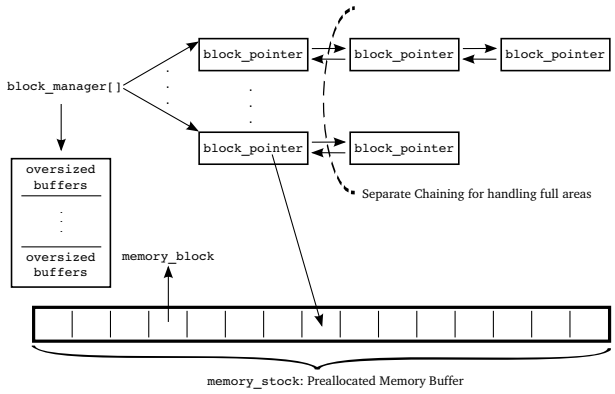


Figure 5.2: Cache-aware memory manager architecture

specified size, a non-active block is activated and linked to the list. If the block found is full, thanks to the fields `next_size - prev_size`, it is linked to all the same-sized active blocks, thus providing a search only on the blocks that fit the request, bypassing all the blocks that cannot match the requested size. If also in this list of same-sized `block`s, no available chunks can be found, again a non-active block is activated and inserted in the bucket-list, this organization is depicted in Figure 5.2. If no more `memory_block` is, a new `memory_stock` is allocated, relying again on `posix_memalign()`.

We have decided to not rely on a per-chunk meta-data structure to save space and enhance locality even more. Note that `memory_block`-wise meta-data, thanks to the strategic position, at the head or at the tail of the block, and to the fact that each one is cache-aligned, can be fast accessed with a few arithmetics operations (even upon a `free` operation).

In particular, as stated above, memory stocks must be cache-aligned, and memory blocks must be cache-sized. To this purpose, the `/proc` file system is accessed, in order to obtain the actual lowest-level cache size. This information is later used to determine which is the most suitable `memory_block`'s size, so that an accurate mapping between memory buffers and cache regions can be created.

The cache information retrieved is also used to insert some padding within the `block`'s metadata, to reduce possible conflicts between these and the boundary chunks, other than for putting chunks cache-aligned. In fact, we consider that metadata are more likely to be accessed frequently since, independently of the application-level allocation patterns, they are used also for allocating all the data structures necessary for the platform execution, in both the regions, i.e., access-intensive and access-mild. The choice of inserting some padding would further reduce data conflicts among metadata accesses

and chunk accesses (independently of whether they belong to the same metadata's `memory_chunk` or not).

The last point of the auto-setup phase deals with the computation of the first chunk associated with the blocks belonging to each kernel. As indicated in Section 5.1 this choice helps avoiding possible biasing problems with the allocation addresses associated with cache stripes. The first chunk to be mapped for each kernel is computed as:

$$\frac{cache\_size}{num\ cores} \cdot core$$

where *core* is the numerical id of the core hosting a kernel instance.

The `memory_stock` size is again decided at compile-time, its allocation relies on the `posix_memalign` allocator, which in simulation-dedicated environments supports the effectiveness of the access-mild vs access-intensive areas' separation. Actual conflicts between the two areas might arise since the cache lower levels operate on conventional architectures behind hardware Memory Management Units, therefore working at physical address level. This case is avoided since internally `posix_memalign` relies on standard `malloc` that, for very large request, uses the `map()` system call to find addressable address space. This system call directly interacts with Linux Kernel's Buddy System, which is likely to provide physical alignment (at requested size) and contiguousness [1]. Therefore, this guarantees to have the two regions completely disjoint, thanks to `memory_block`'s size being the same as the last cache level, as depicted in Figure 5.3. Additionally, the exploitation of `posix_memalign` , with an alignment equals to the cache stripe size, allows reducing the false cache sharing phenomenon at higher level of caches, which conversely rely on virtual addresses.

The last point to consider is related to an allocation request larger than the maximum chunk size available (which depends on the last level cache size). These allocations are managed via an apposite data structure, concatenating the different requests in a chain, which are delivered according to the standard `malloc` policy. Anyhow given that this behavior might degrade performance since these buffers would conflict with almost all the cache stripes regardless of whether they should be used for access-intensive or access-mild data, we have decided to provide a compile-time flag which can be raised in order to reject these requests, leading the software to return a `null` pointer that should be explicitly handled by the programmer, as for the standard `malloc` facility.

[1] An alternative (not being the privilege of simulation environments) is to guarantee alignment at large sizes via *huge pages*, on architectures where the supported hugepages's size is greater than the last level cache size

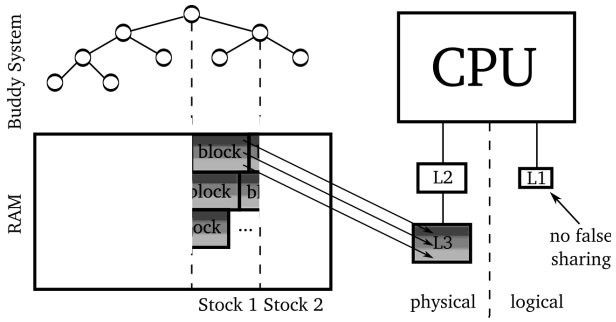


Figure 5.3: `posix_mem_align` behaviour

5.3 Experimental Results

5.3.1 Benchmark Parametrization

To asses the goodness of the proposed solution we have again exploited the PCS benchmark application, which has been configured to run with 64 LPs, each one modeling one macro-cell managing up to 5000 channels. We believe this configuration, with reduced LPs per-core, is relevant since it would tend to emulate the aforementioned long-term architectural scenario (already typical of very large supercomputing platforms), where an increased amount of per-LP cache storage is expected, thanks to the more tight coupling between the amount of LPs and the CPU-cores in very large many-core machines. This is expressed by the ratio of $2:1$ in our case [2].

The application has been parameterized by selecting τ_A in order to give rise to a normal operation mode, with utilization factor of the channels set on the order of 20%, when considering that the average duration of a call has been still set to 2 min. Also, the expected residual residence-time within the macro-cells has been set to 10 min.

To assess the behavior of the proposed architecture different runs have been executed, which differ in the parametrization of the separation threshold that determines the `blocks` devoted to host access-intensive buffers and access-mild buffers. The test compares the performance, again evaluated as the cumulated event rate, by the classical allocator and four configurations of the cache-aware allocator, associated with four different ratios of the cache portions destined to the access-mild vs access-intensive buffers. Again we

[2] The benchmark is structured for simulating a squared-region, thus requiring a squared number of LPs, which is the reason why we used 64 macro-cells instead of 32, which would equal the number of available cores.

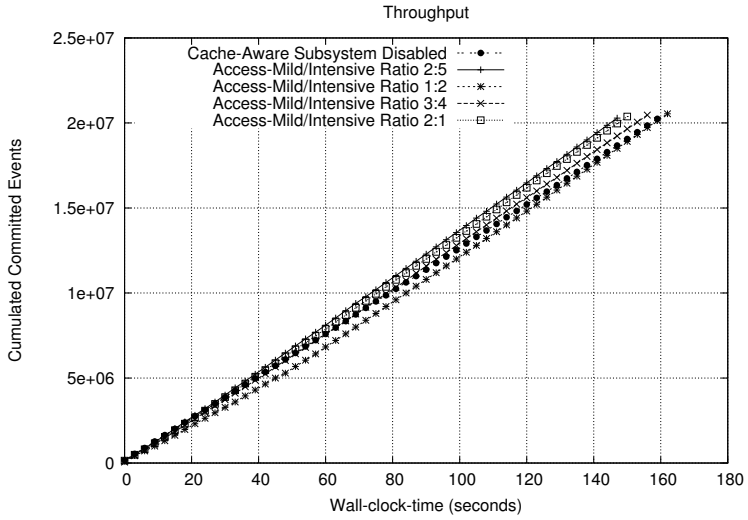


Figure 5.4: Simulation throughput for the PCS benchmark with the cache-aware allocator

have reported samples that have been averaged over ten runs. For these runs we have kept the autonomic log/restore subsystem presented in Chapter 4 active, since an optimized scheme determines the actual locality on relation to the buffers used at kernel level for keeping state recovery related information, which is a relevant parameter determining the final effectiveness of the cache aware approach. Also, GVT has been computed each one second along the run.

By the plots, shown in Figure 5.4 (where this time the y-axis expresses the sum of events committed along time by the whole set of 32 simulation kernel instances) we see how some configurations of the cache aware architecture favor performance, with a gain up to 11%. However, maybe surprisingly, the better performance is achieved either when largely favoring cache hits for access-intensive data (such as when partitioning the cache according to 2:5) or when favoring cache hits for access-mild data (which however become hot data during specific phases such as fossil collection), while anyhow guaranteing a non-minimal cache partition dedicated to forward mode hot data. This is not achieved by cache-unaware memory management, which therefore tends to squash too much the forward mode hot data from cache during the execution of specific (periodic) housekeeping tasks. Finally, we note that the average event granularity for this benchmark configuration is on the order of 200 microseconds, which would give rise to a lower bound on the achievable

speedup (considering a serial run with no housekeeping costs) by the parallel runs on the order of 28 for the case of the best configuration of the cache aware system.

Chapter 6

Load-Sharing on Multi-Core Machines

An additional contribution by this thesis consists in the introduction of an innovative load-sharing architecture, explicitly targeted at balanced and fruitful resource usage when running optimistic PDES applications on top of multi-core machines. We start by providing some technical details in relation to peculiarities of traditional load-balancing schemes, inherently coupled with traditional organizations of the PDES environment, emphasizing their limitations and/or drawbacks we applied in the context of multi-core architectures. Then we introduce the load-sharing innovative model, and present an architectural reshuffle in order to provide a PDES optimistic kernel organization suited for supporting load-sharing at very reduced cost (e.g., in terms of overhead for the management of the tasks associated with load-sharing policies). Finally, an experimental assessment of the whole proposal is provided.

6.1 Base Discussion

The traditional approach to the design and actual implementation of optimistic PDES platforms consists in having multiple LPs being run within a same single-threaded simulation-kernel process (see, e.g., [Carothers and Fujimoto(2000)]). As a consequence, the LPs hosted by this process are dispatched and run on top of an individual CPU-core, according to a classical time-interleaved mode. By this organization, the typical literature approach aimed at achieving effective simulation runs, by optimizing the exploitation of

the available computing resources, is *load balancing*. This technique is based on migrating the application load (i.e. LPs) amongst different simulation-kernel processes while the run is in progress. No other means to dynamically re-balance the load can be employed since each simulation-kernel process has fixed computing power allocated to it, namely one CPU-core.

Clearly, this approach needs to rely on a distributed protocol in order to determine whether a re-balance action is required. This typically maps onto a master/slave protocol, with $O(k)$ message complexity, where the master simulation-kernel process gathers statistics on the current load profile from the other $k-1$ processes, and then notifies the new configuration to be adopted, if any. Computing the current load profile typically requires to sort the LPs according to some reward metric (e.g. the percentage of non-rolled back work) so to be able to determine which LPs need to be migrated. This can be achieved with $O(n \cdot log\ n)$ complexity, where n expresses the number of LPs.

However, beyond the above costs, actual re-balance additionally requires reinstalling onto the destination process' address space the image of any migrated LP. This operation has per-LP latency Δ_m whose lower bound is:

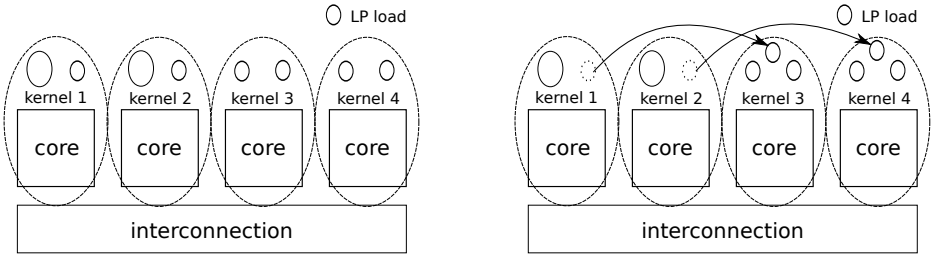
$$\Omega(\Delta_m) = \delta_t \cdot \left[S_{state} + \sum_{i=1}^{N_P} S_{evt}^i \right] \tag{6.1}$$

where we denote with: δ_t the average per-byte transfer time between source and destination simulation-kernel processes; S_{state} the migrating LP's state size; N_P the number of pending events for the migrating LP; S_{evt}^i the size of the i-th pending event for the migrating LP.

With the above lower bound, we do not intend to capture aspects associated with, e.g., event-queue implementation and related scan/update costs. We do not even include the latency for transferring data needed to support correct recovery in case of rollback[1]. Anyway, by Equation (6.1), there is a clear dependency between the actual cost for supporting re-balance and the complexity of the simulation model, in terms of both size of the state of individual LPs to be migrated and event density along the simulation time axis.

In this thesis we change the perspective and propose an orthogonal approach targeted at optimistic PDES systems run on top of multi-core machines, which is based on computing power (expressed in terms of CPU-cores) dynamic reallocation over time towards the different active simulation-kernel

[1]This data includes, e.g., already processed but uncommitted events (which might be required to be reprocessed in case of rollback of the LP after the migration phase) and state log information to correctly reconstruct past LP's state snapshots onto the destination kernel-process.

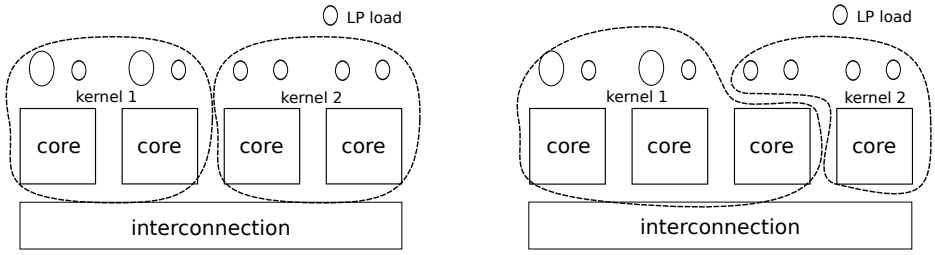


(a) Unbalanced Situation (b) Load Balancing

Figure 6.1: Unbalanced single thread scenario

processes. This is achieved by scaling up/down the number of worker threads operating within each kernel instance, depending on whether locally hosted LPs increase/decrease their computing power demand. Overall, we put in place a *load sharing* approach that ultimately redistributes the whole simulation load across the whole set of available computing resources, without the need for actual migration of the LPs across the different kernel instances. In the essence, the difference between traditional load-balancing and our load-sharing proposal can be outlined by considering the examples depicted in Figure 6.1 (case of load-balancing) and in Figure 6.2 (case of load-sharing). By Figure 6.1 we see how, in case some LP becomes heavy weight (represented as larger in size) and requests more computing power for advancing in simulation time, an explicit migration is required (involving lighter LPs in the example) in order to recreate balanced advancement across all the single threaded kernel instances. On the other hand, in Figure 6.2 we show how, according to the load-sharing approach (where the kernel instance is supposed to natively run according to a symmetric multi-threaded scheme), no LP migration needs to be actuated. Instead, the temporary overloaded kernel acquires an additional CPU-core for usage by a worker thread that gets newly activated within the kernel instance.

The redistribution rule of the cores to the kernels will be based in our proposal on an innovative algorithm/model specifically targeted at capturing productive usage of resources in the context of optimistically synchronized simulators. Although this approach requires a distributed protocol similar in complexity to the aforementioned master/slave one, plus some local sorting of LPs' related information, for determining whether and how to reconfigure the system, it does not pay any LP transfer cost upon system's reconfiguration. In fact, the only additional paid costs relate to worker-thread suspension/reactivation (and associated cache refill), which are anyhow not directly dependent



(a) Unbalanced Situation (b) Load Sharing

Figure 6.2: Unbalanced multithread scenario

on the aforementioned complexity of the simulation model (e.g. in terms of event density along the simulation time axis).

Obviously, the final effectiveness of such an approach depends on how well the worker threads can concurrently operate within a same simulation-kernel process during normal execution phases. To this end, we also provide a reference architectural organization, based on a symmetric multi-threading paradigm, which makes inter-thread synchronization costs affordable. Like all the other contributions by this thesis, also this proposal has been actually implemented within the ROOT-Sim reference platform, and the implementation has been exploited for demonstrating both the viability of the symmetric multi-threading paradigm, when actuated according to our architectural indications, and the effectiveness of load-sharing, when supported via the proposed methodology.

6.2 The Load-sharing Model

Let us denote with C_{tot} the amount of available CPU-cores, and let us assume that we have a set of K_{tot} active simulation-kernel instances in the simulation run, with $K_{tot} < C_{tot}$. Our first objective is to determine the amount of CPU-cores C_i to be assigned to each kernel instance k_i (with $i \in [1, K_{tot}]$) for a given wall-clock-time window, so to improve resource exploitation for fruitful processing.

In our proposal, the re-evaluation of C_i values is carried out periodically, upon computing a new GVT value (or after a set of subsequent GVT computations) since it exploits information on the event rate (committed events per wall-clock-time unit) achieved by each kernel instance k_i, which we denote as evr_i. This quantity is a measure for the fruitful (non-rolled back)

amount of simulation work carried out by each kernel instance. In an ideal scenario where the efficiency is maximized (i.e. where the undone computation is negligible), each kernel instance k_i should use an amount of computing power that suffices to execute exactly evr_i events per wall-clock-time unit. In fact, an excess of computing power could lead to over-optimism and hence to rolled back computation, thus moving run-time dynamics far from the above depicted ideal case. So the idea behind the determination of C_i values is to dynamically assign an amount of CPU-cores to kernel k_i which is proportional to the actual requirements of k_i for the achievement of its relative event rate, compared to the one by the other kernel instances. To also take into account possible differences in the event granularity across the LPs hosted by different kernel instances, which is the indicator of the real usage of computing power for committing the events, the evr_i metric can be refined by weighting it via the average CPU time required for processing the committed events on a specific kernel instance k_i, which we denote as Δ_i. Hence we express the weighted event rate as $wevr_i = evr_i \times \Delta_i$.

In other words, $wevr_i$ values observed during the last wall-clock-time period express the relative CPU requirements of each kernel instance in order to carry out productive simulation work, in relation to the activities of the other kernels and to actual synchronization dynamics. Hence, assigning a computing power proportional to the relative weighted event rate would tend to lead to the situation where each kernel instance allows advancing its LPs in simulation time in a "synchronization suited" manner according to what the other kernels are able to do on their own. This part of the dynamic reallocation scheme would therefore tend to avoid significant presence of overoptimistic kernel instances during the various phases of the run.

It is anyway typical that performance can be further enhanced even in cases where the efficiency is already maximized (or optimized), for example by further reassigning the computing power depending on the real weight of the workload associated with the hosted LPs. As an example, for loosely synchronized models, we may have two or more groups of LPs that do not interact, or stop interacting during the run (hence eventually not directly impacting synchronization and efficiency), exhibiting different speed of advancement in simulation time due to, e.g., different weights of the corresponding events in terms of CPU requirements. In such a case, the completion of the simulation would be delayed by the slowest group. Therefore, within the dynamic scheme for resource assignment, an increase of computing power should also be envisaged for all those kernel instances exhibiting larger CPU requirements to advance in simulation time. To this end we include in our scheme the parameter $wcta_i$, which indicates the wall-clock-time required by kernel k_i to advance a single simulation time unit.

Finally, the amount of cores C_i to be assigned to kernel k_i should anyway

be bounded by the maximum degree of parallelism that can be accomplished by k_i, which is a function of the amount of locally hosted LPs. In fact, each LP is an intrinsically sequential entity, which is not further parallelized, thus not being allowed to simultaneously use multiple CPU-cores for its execution.

Overall, we devise the following rules for dynamically defining the amount of CPU-cores to be reassigned to each kernel instance k_i in order to optimize the usage of the available computing power:

1. For each simulation-kernel instance k_i we compute the parameter $\alpha_i = \frac{wevr_i}{\sum_{j \in [1,K_{tot}]} wevr_j}$.

2. A first estimation of C_i is then evaluated as $\widehat{C}_i = max(\lfloor \alpha_i \cdot C_{tot} \rfloor, 1)$.

3. For each kernel instance k_i for which the condition $\widehat{C}_i \geq N_i$ is verified (where N_i identifies the number of LPs hosted by k_i), then C_i is definitively set to N_i. In fact, additional CPU-cores could not be effectively exploited for parallelization of the locally hosted LPs.

4. At this point, there could be some CPU-cores left to be assigned, which we decide to assign on the basis of (A) the request for allocation remainder of kernel k_i, namely $r_i = max([(\alpha_i \cdot C_{tot}) - \widehat{C}_i], 0)$ and (B) the parameter $wcta_i$. In particular, we order the kernels for which the finalization of C_i values still needs to be performed (so the ones already finalized in point 3 are excluded) according to decreasing values of the product $r_i \cdot wcta_i$, and we assign the remaining CPU-cores according to a round-robin rule following the priority defined by such an ordering.

Each of the above steps is an implementation of the rationales discussed above in terms of suited CPU-core assignment vs specific performance aspects. However, once selected final C_i values, we need to determine how to optimize the usage of the assigned CPU-cores within each single simulation-kernel instance k_i. This problem translates into defining which LPs locally hosted by k_i needs to be assigned to each of the C_i worker threads running in parallel within kernel instance k_i. We will refer to the assigned LPs as being *affine* to the worker thread, and still scheduled for event execution along this thread according to LTF.

For the j-th LP hosted by kernel k_i, which we denote as $LP^j_{k_i}$, we compute the total amount of CPU-time required for committing its events during the last observation period. We refer to this metric as $cpu^j_{k^i}$. The maximum $cpu^j_{k_i}$ value across all the locally hosted LPs represents in our scheme a reference knapsack, and the corresponding $LP^j_{k_i}$ is assigned to a given worker thread. Then we exploit the greedy approximation approach proposed by

George Dantzig in [Dantzig(1957)] which allows a maximum "*overflow*" of about 30% over the reference knapsack, in order to build the other knapsacks of LPs (hence knapsacks characterized by sums of $cpu^*_{k_i}$ values) to be assigned to the remaining worker threads. We can do this by applying a variant of the original scheme, where the knapsacks are filled according to a round-robin approach. The procedure is then iterated until no more LP needs to be further bind to any worker thread within k_i.

As a preliminary note to the complexity analysis presented in the subsequent section, the computation of $wevr_i$ and $cpu^j_{k_i}$ values can be embedded within the fossil collection algorithm. In particular, while scanning the input queues of the LPs for releasing the buffers related to the already committed portion of the simulation, per-event execution costs (typical logged while processing the events within the same buffers as a form of audit) can be accessed and accumulated to determine the actual values of the parameters $wevr_i$ and $cpu^j_{k_i}$. Hence, these values can be made available to the algorithm supporting the above described load sharing policy with no variation of the asymptotic cost of fossil collection.

6.2.1 Asymptotic Costs Analysis

Solving the load sharing model can ultimately rely on a distributed master/slave protocol where every kernel instance k_i sends to the master kernel a message containing the values of the parameters $wevr_i$ and $wcta_i$, and the master kernel sends to k_i a message notifying the newly computed value of C_i. This entails $O(K_{tot})$ message complexity, namely linear complexity vs the number of kernel instances.

The local execution cost at the master kernel for the determination of C_i values, associated with steps 1–4 described above, relates to performing per-kernel analysis of the statistics collected during the master/slave communication phase, and to sorting the kernel instances on the basis of $r_i \cdot wcta_i$ values (see step 4). This leads to an asymptotic $O(K_{tot} \cdot log\ K_{tot})$ complexity.

Once received the notification of the computed C_i value from the master, every kernel instance k_i must sort the locally hosted LPs on the basis of $cpu^j_{k_i}$ values, which entails $O(N_i \cdot log\ N_i)$ time, and must then solve a 0-1 Knapsack's problem, which can be done in pseudo-polynomial time in the number N_i of locally hosted LPs. Hence we get an overall cost of $O(N_i \cdot log\ N_i)$ for local operations to be executed on each simulation kernel instance k_i.

Given that $N_i \geq 1$, we get $\sum_{i \in [1,K_{tot}]} N_i \geq K_{tot}$, hence $O(K_{tot} \cdot log\ K_{tot})$ is bounded by $O(N_{tot} \cdot log\ N_{tot})$, where N_{tot} represents the sum of individual N_i values, namely the total amount of LPs within the run.

In the end, we get that the determination of the new configuration can be

achieved with linear message complexity, vs the number of kernel instances, and $O(n \cdot log\ n)$ local processing complexity vs the number n of LPs within the simulation model. This is the same complexity as for typical load balancing schemes, whose asymptotic costs have been discussed in Section 6.1. However, as already pointed out in the same section, our load sharing approach does not entail actual LP transfer operations, but only activation/deactivation of worker threads within the different kernel instances. We remark again that the cost of this operation does not directly depend on the complexity of the simulation model, in terms of state size of the LPs and event density in simulation time, as instead it occurs for LP transfer operations proper of load balancing approaches.

6.3 Architectural Aspects

The core requirement for the effectiveness of the load sharing approach, is related to how efficiently the different worker threads running within the same simulation-kernel process (hence within the same address space) can synchronize with each other. Specifically, while different worker threads inherently execute according to data partitioning paradigms once entered application mode (since, in accordance with what specified in [Jefferson(1985)], each LP handles its own application-level data structures), care must be taken to avoid excessive synchronization costs when running housekeeping tasks involving shared data structures.

Most notably, the shared data structures requiring frequent updates, to be performed coherently via proper synchronization mechanisms, are the input queues of the LPs. Essentially, these data structures represent the core of cross-LP dependencies, thus involving update operations caused not only by the activities executed by the worker thread currently taking care of running the "queue-owner LP", but also by the activities carried out by worker threads taking care of running other LPs. Synchronizing the access to these data structures via a conventional locking mechanism would give rise to scalability problems, exactly due to such a strict coupling. Further, it would give rise to critical sections whose duration would depend on the actual time-complexity of the queue-update operation. The access to the LPs' state queues (either for saving or restoring a state image) does not induce thread synchronization issues since the need for state log/restore operations is only an indirect reflection of cross-LP coupling, caused by events scheduled across the LPs. In other words, a single worker thread is allowed to safely operate on the state queues of its affine LPs at any time, since it is the only worker thread that can take care of dispatching those LPs for either forward or rollback execution. Similar considerations can be made for the output queues, which are essentially used

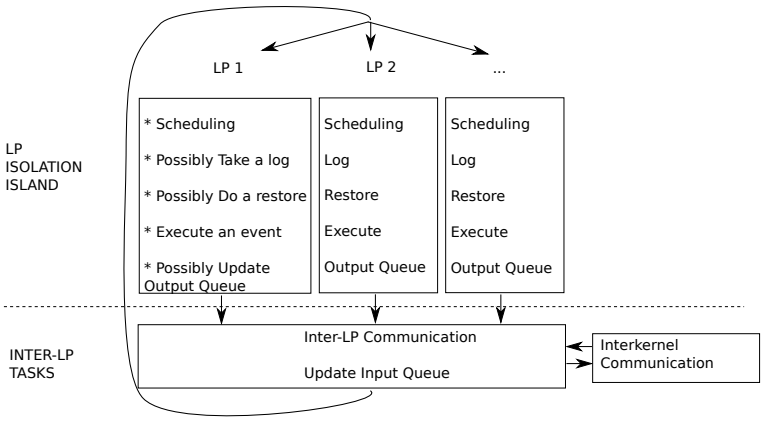


Figure 6.3: LP isolation island

for auditing the messages sent out by the LPs in order to undo them via antimessages in case of rollback. On the other hand, the rollback operation and the generation of antimessages, via consultation of audit information within the output queue, are performed (if requested) by the unique worker thread for which a specific LP is currently affine. Overall, as also schematized in Figure 6.3, isolation islands exist in relation to LP execution along different worker threads, except for LPs' input-queues management and for the reflection of event scheduling on the side of cross-kernel message passing (since the message passing layer might not be immediately targeted at managing multi-threading, such as when relying on MPI).

The architectural organization we propose in this paper to cope with the reduction of synchronization costs borrows from the design principles proper of multi-processor/multi-core Operating Systems. Specifically, all the worker threads that are active within the same kernel instance operate symmetrically, by having access to any housekeeping functionality. On the other hand, any housekeeping task potentially crossing the boundaries of individual LPs' data structures is dispatched according to the same rules employed to structure modern Operating System drivers, by organizing it according to top/bottom-half activities. Hence, whenever the need for the execution of such a task arises, it (logically) takes place as an interrupt to be eventually finalized within a bottom-half module. More in details, upon the interrupt occurrence, we do not immediately finalize the task, thus not immediately locking (or waiting for the lock) on the target data structure. Instead we simply execute a light top-half module which registers the bottom-half function (and its parameters) associated with the interrupt finalization within a per-LP bottom-half queue,

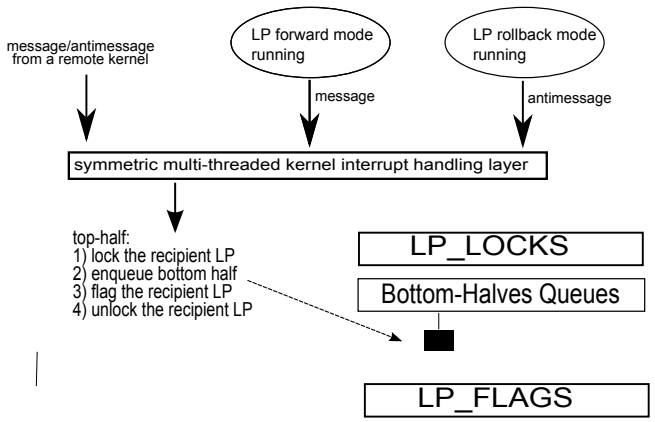


Figure 6.4: The top/bottom-halves architecture

resembling the Linux task queue. The critical section accessing the bottom-half queue takes constant-time since each new bottom-half associated with the LP is recorded at the tail of the queue. Also, when the bottom-half tasks currently registered for a given LP are flushed, the corresponding chain of records is initially unlinked from the corresponding bottom-half queue, which is again done in constant time by unlinking the head element within the chain from its base pointer [2]. Given that the access to the LP bottom-half queue represents in our architectural organization the only frequently occurring synchronization point, constant-time for the corresponding critical sections directly leads to minimizing synchronization costs.

The schematization of our proposal is presented in Figure 6.4. 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[j]` is used to implement the critical section for the access to the bottom-half queue associated with the j-th LP hosted by any kernel k_i, namely $LP^j_{k_i}$, 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.

Let us now depict when (logical) interrupts to be handled via this type of organization occur. Basically, an interrupt occurs as soon as any worker thread currently active within kernel k_i becomes aware of a new message/antimessage destined to some locally hosted $LP^j_{k_i}$. In such a case, the worker thread needs to accesses the j-th bottom-half queue within a critical section

[2] Actual data structure updates can be safely performed out of the critical section, provided that a single worker-thread at any time is in charge of flushing the bottom-halves of any LP that is affine to it.

that performs the insertion of the corresponding message/antimessage delivery task. To provide additional details, awareness by a worker thread of a new message/antimessage destined to a locally hosted LP arises in three different circumstances:

(i) The worker thread is currently running $LP_{k_i}^j$ in forward mode, and this LP produces a new event to be scheduled for the locally hosted $LP_{k_i}^t$. Thus the worker thread enters housekeeping for actuating the delivery of the corresponding message to $LP_{k_i}^t$'s input-queue. (Note that j might be equal to t, in which case sender and receiver coincide.)

(ii) The worker thread is currently running the locally hosted $LP_{k_i}^j$ in rollback mode (hence it is performing housekeeping operations associated with revealed causality errors), which gives rise to the production of an antimessage destined to $LP_{k_i}^t$, which again requires access to $LP_{k_i}^t$'s input queue for annihilating the original message. (Also in this case we might have $j = t$.)

(iii) The message passing layer notifies the worker thread (e.g. via an explicit message receive operation executed by this thread according to a traditional polling scheme) about a new message/antimessage incoming from some remote kernel instance, which is destined to a locally hosted LP.

As shown in Figure 6.4, we logically mark all the above three circumstances as interrupts, which will be treated homogeneously, and whose associated message/antimessage delivery operation will be finalized via the bottom-half mechanism.

We note that spin-locks may anyhow exhibit non-minimal costs since they require the corresponding operations to be performed via sequences of atomic instructions (e.g. via the LOCK prefix for the IA-32 instruction set). Additionally, since they are shared and accessed by different threads, cross-cache invalidation effects can be induced as soon as one worker thread gains control on the spin-lock. To reduce these effects, we devise the presence of an additional array of flags `LP_FLAGS` (see again Figure 6.4), where `LP_FLAGS[j]` indicates whether the corresponding bottom-half queue, namely the one associated with the j-th locally hosted LP, is not empty. `LP_FLAGS[j]` gets updated within the critical section protected by `LP_LOCKS[j]`, either when a new bottom-half is inserted within the corresponding queue (in this case the flag is raised), or when the queue is flushed (in this case the flag is reset). However, `LP_FLAGS[j]` is also accessed before trying to lock the bottom-half queue in order to avoid spin-lock operations in all the cases where the queue

would reveal empty once accessed within the critical section leading to flush operations. The exact scheme is:

```
TOP-HALF:                    BOTTOM-HALF:
 lock(&LP_LOCKS[j]);          if (LP_FLAGS[j])
 <log bottom-half>;             if (try_lock(&LP_LOCKS[j])){
 LP_FLAGS[j] = TRUE;               <unlink bottom-halves>;
 unlock(&LP_LOCKS[j]);             LP_FLAGS[j] = FALSE;
                                   unlock(&LP_LOCKS[j]);
                                   <perform bottom-halves>;}
```

Being `LP_FLAGS[j]` checked non-atomically wrt lock acquisition when attempting to perform bottom-halves, we might experience false negatives in case the top-half finalizes the insertion of the bottom-half task concurrently with the check. However, this does not represent a safety problem since the flag will be rechecked periodically in subsequent attempts to flush the corresponding bottom-half queue, thus eventually falling in the case where the bottom-half queue is correctly reflected into the state of the input queue of the destination LP. Such a reflection might therefore experience only a delay, which resembles delays introduced by traditional single-threaded kernels while reflecting the content of cross-kernel messages into the system state, which is typically affected by the polling period according to which the messaging layer is accessed for acquiring not yet delivered messages. Further, as hinted, when allowing a single worker thread at a time to manage flush operations for its affine LPs, no false positives will ever be experienced.

6.3.1 Details on Actual ROOT-Sim Integration

Integration of load sharing within ROOT-Sim, according to the architectural indications provided above, has been based on `pthread` technology, and on the reorganization of housekeeping data structures in order to (i) provide per-thread private data, and (ii) cache aligned memory buffers, so to avoid false cache sharing across the worker threads within the same kernel instance. The latter objective has been achieved by exploiting the `posix_memalign` API, plus the usage of proper padding schemes allowing cache alignment for sequences of records, such as arrays of values. As for the accesses to the MPI layer, in our architecture they can be symmetrically issued by any of the worker threads operating within a given kernel instance. Given that the MPI does not natively support multi-threading, we have included a wrapper that synchronizes these accesses transparently to the worker threads via the embedding of critical sections protected by spin-locks.

As far as GVT computation and fossil collection are concerned, we implemented a symmetric scheme in which the worker threads operating within a same kernel instance run a race. The race winner computes the local reduction and interacts with the master kernel in order to determine the globally reduced value representing the new GVT. However, once defined the new GVT value, all the worker threads operating within the same kernel instance are allowed to perform fossil collection operations in parallel, each one fossil collecting obsolete information for its affine LPs.

6.4 Overhead Oriented Experimental Assessment

In this section we aim at providing an experimental study oriented to assess the internal dynamics of the symmetric multi-threaded reorganization of the optimistic kernel, and their impact on run-time aspects such has locality and contention of hardware resources. Overall, we focus on the overhead by the reorganization, under workloads where no advantage is expected due to a natively balanced distribution of the workload across different kernels. 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 this imposes some constraints on run-time dynamics properly related to specific ROOT-Sim subsystems, as hinted before.

All the parameters that will be object of the experimental assessment will be discussed in this section, by also motivating why they have been selected in the analysis. As a final preliminary note, our reference architecture for the assessment will be represented by the original multi-process (single-threaded) version of ROOT-Sim.

6.4.1 Overview of the Assessment

Evaluating the Effects on Caching and Memory Accesses

It is clear that the 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 an 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 good metrics 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.

In addition, we have decided to measure scheduling operations' latency in order to assess the effects of multi-threading on data structures which are accessed sparsely. To this purpose, we have configured ROOT-Sim to rely on the O(n) variant of the LTF scheduler, which determines the next event to be processed by going over LPs' input queues for identifying the pending event associated with the minimum timestamp. We consider this to be a measure representative of the dynamics proper of a large set of operations which are essential in a simulation platform, such as queues scanning and log-chains traversing.

Evaluating the Impact on MPI Operations

In case of interactions between LPs hosted by different kernel instances, instead of relying on the top/bottom-half scheme, messages are directly provided in input to the MPI layer. As said, 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.

Clearly this approach may induce delays on the worker threads when compared to the multi-process scheme. However, 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, if any, will not require passing via MPI. This likely leads to a reduced amount of interactions to be handled via MPI, since some interactions will be locally treated at the level of the top/bottom-half subsystem. 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 embedded a similar optimization within ROOT-Sim, for what concerns the interaction with MPI, the experimental assessment can be related to a kind of worst case architectural configuration.

Evaluating the Impact on GVT and Global Snapshot Operations

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 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. 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 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.

Evaluating the Effects on the Overall Rollback Pattern

Since a rollback happens upon receiving an out-of-order event to be executed, this can more likely arise for larger gaps between different LPs' local clocks possibly caused by a different workload being processed across different simulation-kernels. Therefore, if the computing power is dynamically redistributed among the various simulation-kernel instances in order to achieve more balanced runs, as it occurs in the load-sharing architecture, 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 Local-Virtual-Time to restart the execution from) is expected to be reduced. On the other hand, even for balanced workloads, the employment of the top/bottom-half architecture generates a different timing of actions, in terms of information reflection within the LPs' input queues, which can secondarily impact the rollback pattern. In order to evaluate these secondary effects, we have explicitly measured the following parameters:

- Rollback probability, evaluated as the ratio between the amount rollback operations and the amount of executed events.

- Rollback length, expressed as the average number of undone events per rollback operation.

- Efficiency, which is measured as the ratio between the amount of committed and executed events.

6.4.2 Benchmark Application Setting

The experimental assessment has been based on PCS, which has been configured with 1024 wireless cells evenly distributed across the different simulation kernel instances running on the underlying 32-core machine, each one managing up to 1000 wireless channels. The simulated workload has been configured to produce a load relatively uniform across the various LPs.

The exponentially distributed call inter-arrival time has mean value τ_A, and the average call 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 factors on the order of 35%, 15%, and 10% respectively, while the residence time of an active device within a cell has been set to a mean value of 5 min and still follows the exponential distribution. The variations of τ_A determine model instances with different profiles in terms of both event granularity and memory requirements. Specifically, the lower the value of τ_A, the larger CPU/memory requirements. Also, lower values for τ_A imply greater computation to communication ratios.

In this study, we rely on non-incremental state saving, with checkpointing interval χ set to the fixed value of 20, in order to avoid run-time dynamics fluctuations potentially caused by self-adjusting checkpointing policies. 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 the multi-threaded 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. Finally, the data reported have also this time been computed as average values over ten runs of the reference configuration.

6.4.3 Results

Top/Bottom-Halves Processing

In order to ensure correctness, whenever a top/bottom-half operation must be performed by some worker thread, a lock on the bottom-half queue associated with the destination LP must be taken (although in the case of bottom-halves processing, the lock is only needed for de-queueing the events' chain currently registered within the queue). If the number of concurrent worker threads

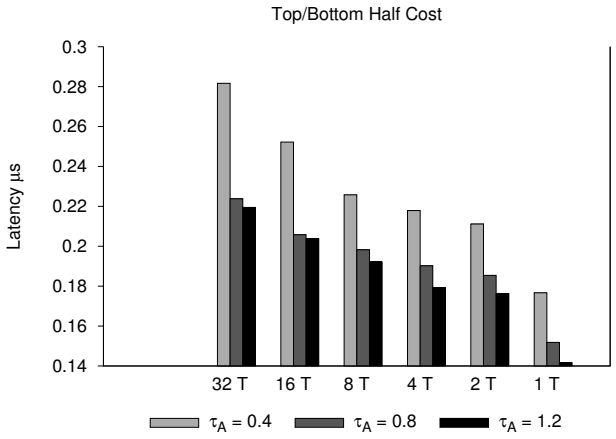


Figure 6.5: Top/bottom-halves

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 per-kernel instance, 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 6.5 we show the per-event latency related to the management of 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 6.6, the per-event latency related to scheduling operations is provided. By the plots, we can see that the non-multi-threaded implementation (which we refer to as "Single") shows a latency which is on the order of 15% smaller than the load-sharing one. This small overhead is related to the fact that the load-sharing architecture implements a mapping between LPs han-

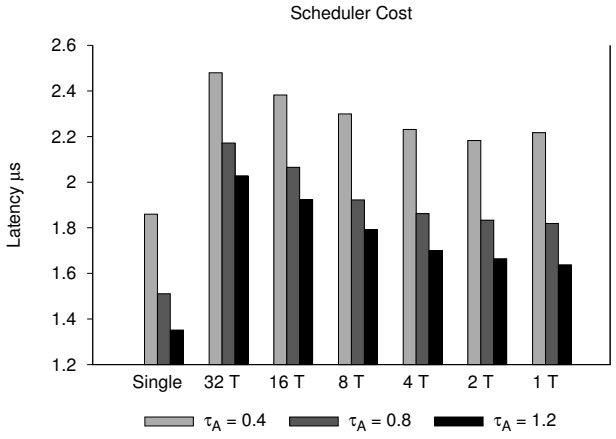


Figure 6.6: Scheduling

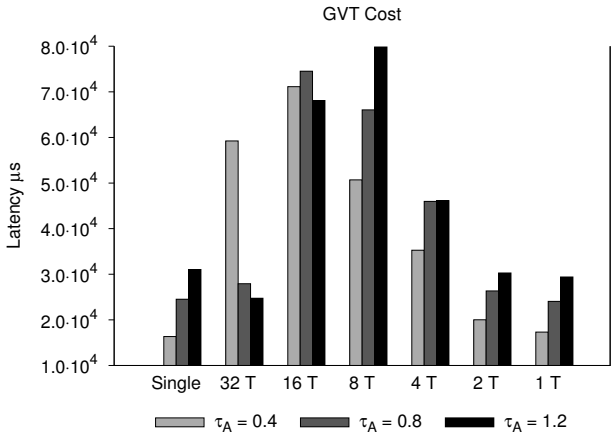


Figure 6.7: GVT and CCGS

dled by a certain worker thread and the actual thread. Therefore, in order to perform CPU 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-multi-threaded implementation. Nevertheless, this difference is not enough to justify 15% latency increase. In fact, a significant additional difference between the two implementations relies on the fact that the load-sharing architecture makes large use of locking primitives for ensuring correctness. This entails a higher number of in-memory accesses for trying to acquire spin-locks, 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 6.8(top) we present the per-log 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 kernel instances) presents a higher latency. In particular, the cost starts growing when running with 4 kernel instances, each one handling 8 worker threads, and the greatest difference is on the order of 30% (15 μs). This latency increase wrt the number of worker threads is also influenced by the aforementioned internal synchronization at the level of the malloc library.

In Figure 6.8(bottom) the per-restore 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 6.7 shows the plots related to the GVT and CCGS execution latency. As hinted in Section 6.4.1, the load-sharing version of our simulation-kernel allows a single worker thread to perform CCGS operations, due to critical races on the output.

At the same time, during 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

[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.

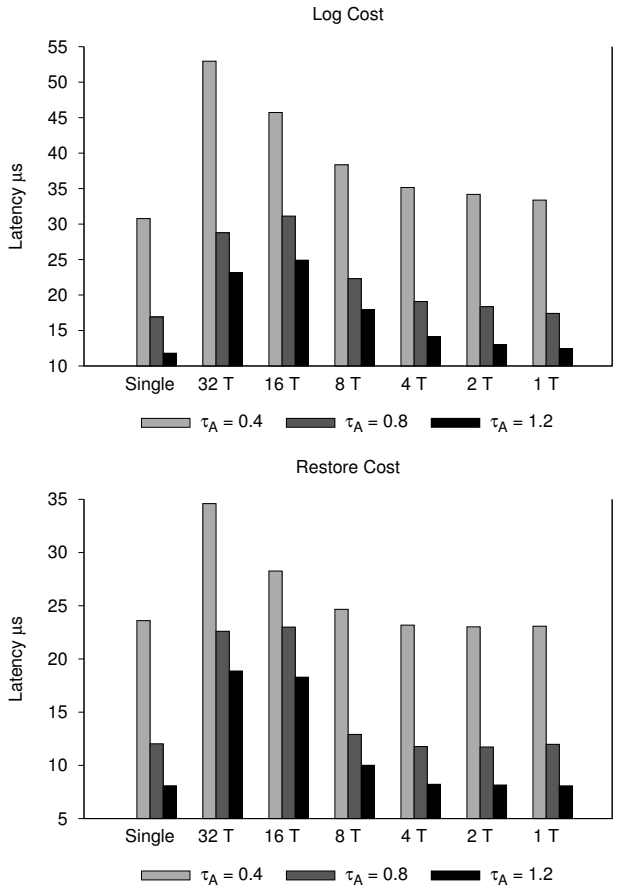


Figure 6.8: Log/restore operations

determine which is the best number of worker threads per-kernel instance to evenly share the current workload. Again, this procedure is based on MPI message exchanges.

By the plots, we can see that, when running with a small number of worker threads, the latency is comparable with the one achieved by the single-threaded kernel. On the other hand, a larger number of worker threads entails a higher latency. 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.

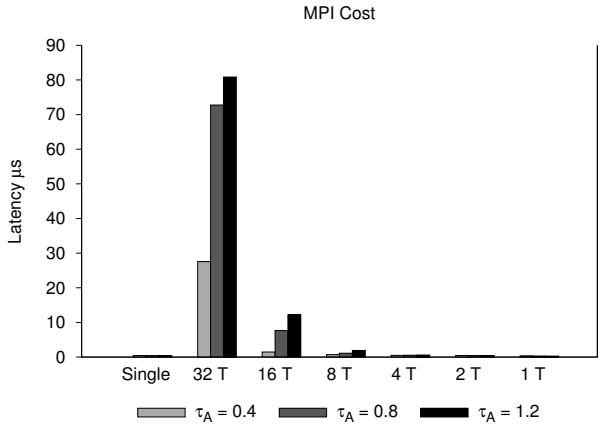


Figure 6.9: Inter-kernel communication

Inter-Kernel Communication

By the plots in Figure 6.9, the executions relying on 32 and 16 worker threads exhibit inter-kernel communication latency which is almost two and one orders of magnitude greater than other configurations, respectively, thus giving rise to a smaller event throughput, as depicted in Figure 6.12. This is related to the fact that these configurations process a larger number of uncommitted events (as reported in Figure 6.17), since most of the processed events are rolled back. In fact, in Figure 6.15 we can see that these configurations show, among the others, the higher rollback probability, along with a non-minimal rollback length. This gives rise to low efficiency (as reported in Figure 6.16).

The high inter-kernel communication latency exhibited by these configurations is related to the higher contention on the MPI layer due to the large amounts of message exchanges 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 6.4.3 (for both configurations), and to the higher number of MPI probe operations (for both configurations). As for the latter aspect, in the 32-threads configuration, we left the simulation-kernel to perform probe operations towards the MPI even though no message will ever income (given that the run relies on a single kernel instance). This has been done just to observe the effects of the interactions with the MPI layer when scaling up the number of worker threads to the maximum value admitted in relation to the amount of CPU-cores available from the underlying computing platform.

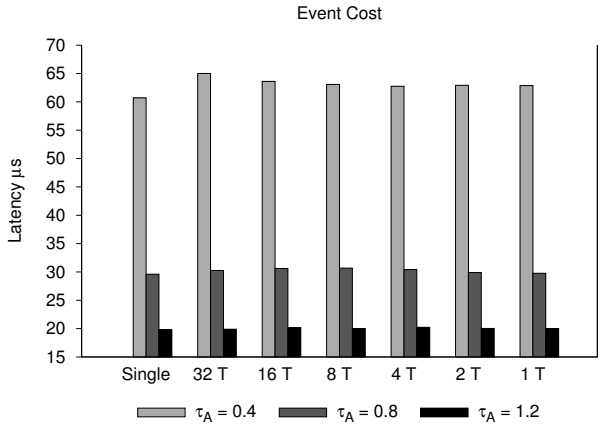


Figure 6.10: Event execution

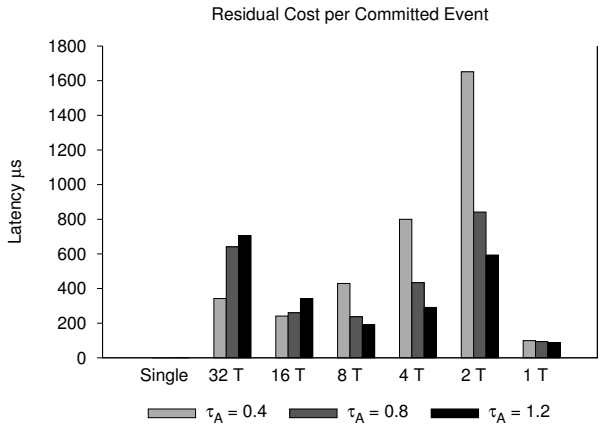


Figure 6.11: Residual cost

Events' Execution

In Figure 6.10 we show the per-event execution latency. Although different values of the τ_A parameter produce different events granularity values (due to the different simulation model's load, which produces a higher amount of data to be processed for SIR regulation), different configurations do not affect significantly the event's latency, with small differences that are in the order of 3-5%. This emphasizes the fact that the load-sharing architecture does not affects locality more significantly than the non-multi-threaded one, as far as events' execution is concerned.

Residual Cost

To complete the punctual assessment of our load-sharing architecture, in Figure 6.11 we present the plots for the residual cost. This includes all the per-committed-event costs which do not appear in the above measurements. Essentially, the residual cost shows which is the time spent for scanning/processing input/output-queues, and for all the 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 subject to secondary effects related to memory contention, as depicted in the previous analysis.

As for housekeeping operations, the input queue management and the ack-handling subsystems have a great importance. The former entails calling 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 in charge of facing the well-known transient message issue for GVT computation, for which 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 it can be seen by the plots, the highest residual cost 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 inter-kernel message exchange volume which entails trying to acquire the lock more frequently. The one-thread configuration does not show this contention effect, while increasing the number of threads reduces the need to update the window, thus reducing contention as well.

To show how the so-far described costs impact on the overall performance of the load-sharing platform's execution, in Figure 6.14 we present an aggregation showing the percentage of time spent in the various operations, normalized

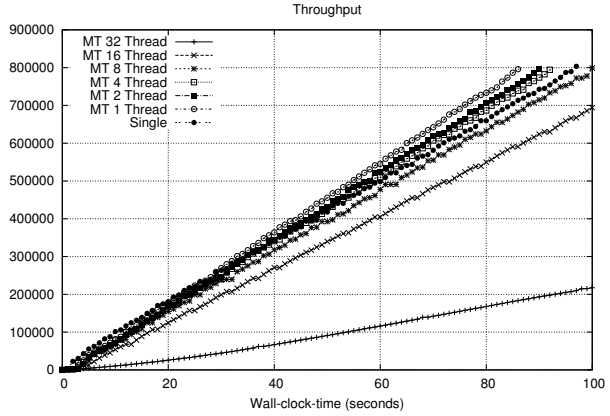


Figure 6.12: Simulation throughput for Load-sharing: PCS benchmark $\tau_A = 0.4$

on committed event's execution. By the plots we can see that the non-multi-threaded execution, independently of the workload, spends almost 70% of the time in events' processing, similarly to what the load-sharing architecture configured with one worker thread does.

In addition, we can see that as the number of worker threads increases, 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 it was already clearly illustrated in Section 6.4.3.

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 it was depicted in the previous section.

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), again used as the indicator of the speed of the optimistic simulation run.

In particular, Figures 6.12 and 6.13 show the corresponding throughput values for the benchmark configurations related to the τ_A parameter set to 0.4 and 0.8, respectively. By Figure 6.13 we can see how the so-far discussed over-

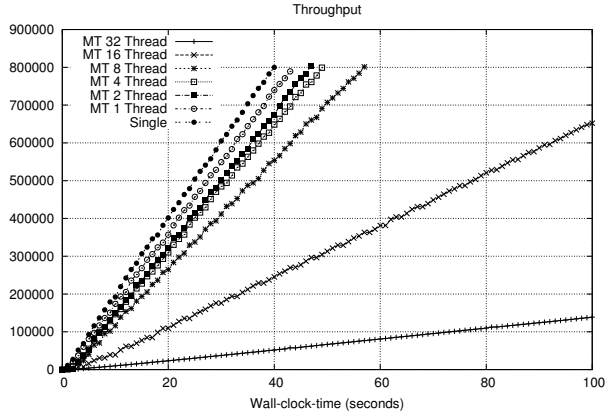


Figure 6.13: Simulation throughput for Load-sharing: PCS benchmark $\tau_A = 0.8$

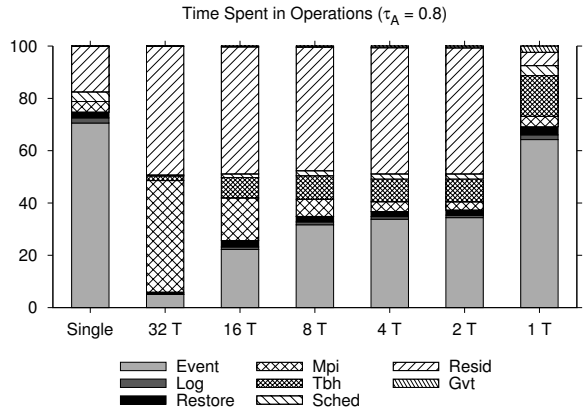


Figure 6.14: Operation weights

heads produce a decreasing throughput when the number of worker threads is increased (we remind that in this experimentation, the workload is constant and evenly distributed, and the re-balancing procedure is forced not to re-assign worker threads to kernel instances, in order to evaluate which are the intrinsic costs of the presented architecture). As it was explained before, the configurations associated with 16 and 32 threads do not scale well, particularly because of the high costs related to MPI operations, and due to a large amount of executed events which get rolled back.

Figure 6.12 shows the configuration with a different (higher) workload. We note that some load-sharing configurations present a throughput slightly higher than the non-multi-threaded 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 6.16.

Overall, limited or null overhead is noted in general for configurations entailing up to 4 or 8 worker threads per kernel instance, which leads to some degree of flexibility in the possibility to reassign resources in an architectural context not providing significant penalties while handling the supports for making the reassignment operative.

6.5 Evaluating the Effectiveness of Load-Sharing

In the previous section we concentrated on overhead aspects by the load-sharing architecture when considering balanced workload. In this section we change the perspective by carrying out an experimentation based on the highly variable Traffic application. This would lead to observe the effectiveness of load-sharing in contexts where dynamic decisions aimed at improving balanced and fruitful usage of resources are highly welcome.

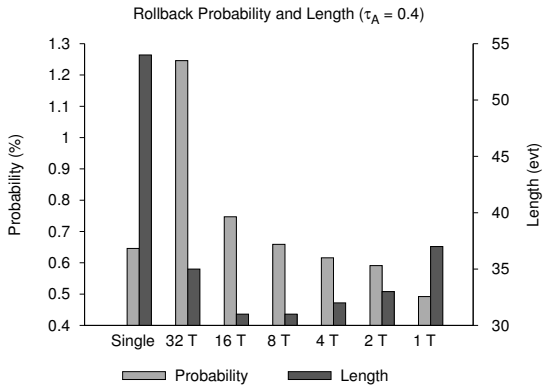
Traffic has been exploited in the exact configuration described in Chapter 3, and for this application we have compared the throughput (cumulated committed events) by our load-sharing architecture with the one by a classical single-threaded organization, with the one related to serial execution of the same application-level software running on top of a calendar-queue scheduler, and with results by a load-balancing architecture based on a migration approach, still implemented into the same ROOT-Sim platform, as presented in [Peluso et al.(2011)Peluso, Didona, and Quaglia].

By the data shown in Figure 6.18, the parallel approaches provide a super-scalar speedup. The multi-threaded versions of the simulation kernels all provide a speedup wrt the single-threaded one, which ranges in between 40% (for the 4 kernels configuration) and 55% (for the 8 and 16 kernels configuration). In particular, we note that the execution with 4 kernel instances shows a reduced speedup due to several reasons: (i) the re-balancing is more likely to

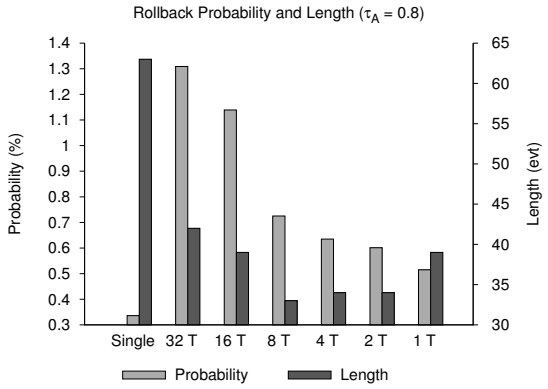
map a worker thread on a core which is not actually sharing any level of cache; (ii) a worker thread can access remote memory with a higher probability (we recall that the experiments have been run on a NUMA machine); (iii) worker threads are more subject to false cache sharing effects.

As for the execution with 32 multi-threaded kernels, the speed down is in the order of 15%. This is related to the fact that in this configuration no actual re-balancing is possible (in fact, each simulation kernel must have at least one worker thread in order to proceed in the simulation run). Therefore, in this configuration we are again measuring the architecture's overhead, which is indeed comparable to the one shown when running the PCS (balanced) benchmark.

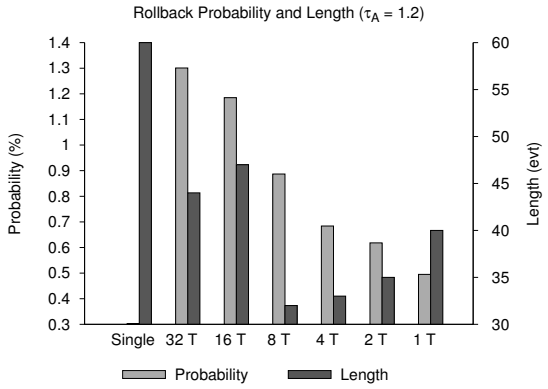
The last comparison shown by the plot is the one wrt the load balancing configuration, referred to as migrator. Although we note that this configuration provides a speedup in the order of 40% wrt the single-threaded approach, it's throughput is comparable with the 4 kernels multi-threaded configuration, while the 8 and 16 kernel configurations of the multi-threaded architecture are still 30% faster than the migrator configuration. This is related to the fact that the migrator approach does not assign resources to the simulation kernels, instead it migrates LPs from one instance to the other, entailing complex marshalling and communication operations. We emphasize that these two approaches are orthogonal and do not exclude each other, considering that when relying on clusters, a migration approach merged with a load-sharing approach can result in a significant benefit for the simulation's throughput, as they face different issues.



(a) $\tau_A = 0.4$



(b) $\tau_A = 0.8$



(c) $\tau_A = 1.2$

Figure 6.15: Rollback probability and length

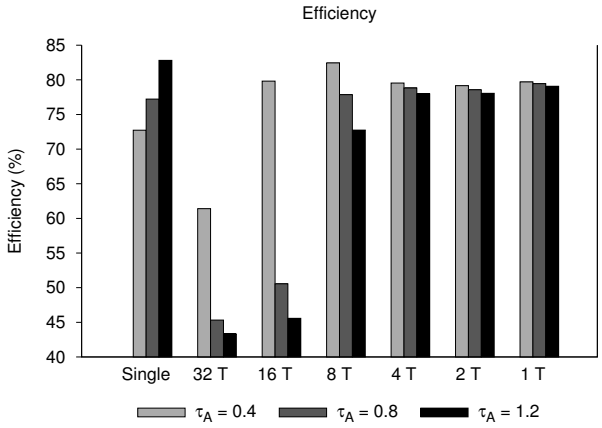


Figure 6.16: Efficiency

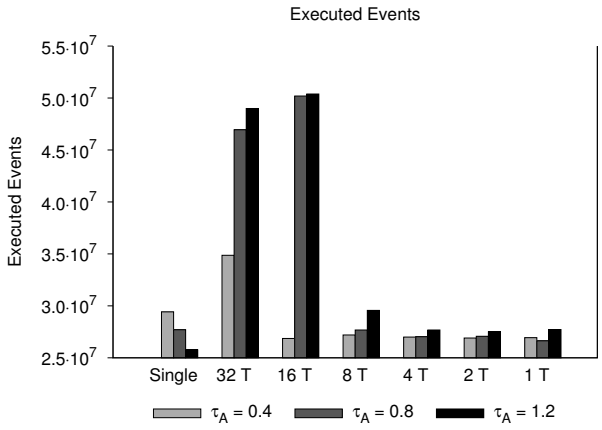


Figure 6.17: Executed events

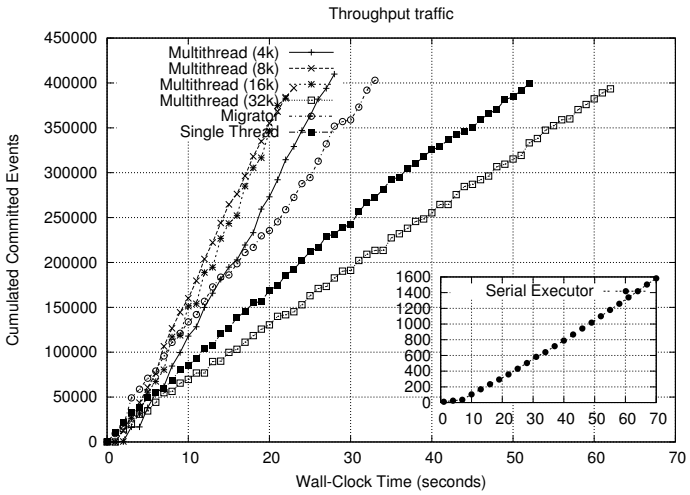


Figure 6.18: Simulation throughput for the Traffic benchmark

Chapter 7

Conclusion

Optimistic Parallel Discrete Event Simulation (PDES) platforms are known as a means for achieving speedup while executing complex models thanks to the employment of speculative processing schemes. However, the optimistic paradigm poses a set of issues in relation to the complexity of the supports that are required in order to achieve fruitful exploitation of the underlying computing resources, while jointly masking synchronization dynamics (e.g. the squashing of part of the computation when causality errors are detected) to the application programmers.

In this thesis we have tackled three core issues in relation to optimistic PDES systems, namely state recovery, memory access optimization, and balanced and productive usage of resources in the context of multi-core machines. As for state recoverability, we have provided an innovative log/restore architecture inspired to the autonomic computing paradigm, which is expected to represent a highly general solution, thanks to the joint employment of differentiated log modes (in time interleaved mode) and stability oriented performance optimization schemes for the tuning of the parameters determining log/restore dynamics. As for memory access optimization, we have studied a cache-aware approach, where the delivery of memory buffers is tailored for optimizing the memory access latency to the data structures that from the core of this type of platforms, depending on their access profile in typical runs. Finally, balanced and productive resource usage, which is mandatory for keeping the system far from thrashing phenomena that would lead the optimistic paradigm to reveal un-successful (due to excessive waste of computation), has been addressed by proposing a load-sharing approach specifically tailored for PDES systems run on top of multi-core machines. Although inspired to typical load-sharing schemes actuated by conventional operating system kernels, our proposal had

to face complications related to the intrinsic strict coupling of the activities executed at the different concurrent simulation object (compared to the level of coupling of typical threads/processes running on top of the operating system).

We have tackled all the above issues both on the methodological and the architectural sides. Particularly, most of the proposals rely on (or exploit) innovative performance models. Also, all the proposals have been developed, according to properly defined/reshuffled design approaches, and have been integrated within a real operating environment, namely an open source optimistic PDES platform available to the research community.

List of Figures

List of Tables

References

[ACI()] ACI. Dati e statistiche. http://www.aci.it/?id=54.

[Akyildiz et al.(1992)Akyildiz, Chen, Das, Fujimoto, and Serfozo] Ian F. Akyildiz, Liang Chen, Samir Ranjan Das, Richard Fujimoto, and Richard F. Serfozo. Performance analysis of "time warp" with limited memory. In *SIGMETRICS*, pages 213–224, 1992.

[Akyildiz et al.(1993)Akyildiz, Chen, Das, Fujimoto, and Serfozo] Ian F. Akyildiz, Liang Chen, Samir Ranjan Das, Richard Fujimoto, and Richard F. Serfozo. The effect of memory capacity on time warp performance. pages 411–422, 1993.

[AUTOMAP()] AUTOMAP. Atlante stradale italia. http://www.automap.it/.

[Autostrade per L'Italia S.p.A.()] Autostrade per L'Italia S.p.A. Reportistica sul traffico. http://www.autostrade.it/studi/studi_traffico.html.

[Bauer and Page(2007)] David W. Bauer and Ernest H. Page. An approach for incorporating rollback through perfectly reversible computation in a stream simulator. In *21st International Workshop on Principles of Advanced and Distributed Simulation*, pages 171–178. IEEE Computer Society, 2007.

[Bauer et al.(2005)Bauer, Yaun, Carothers, Yuksel, and Kalyanaraman] David W. Bauer, Garrett Yaun, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Seven-oclock: A new distributed gvt algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation*, pages 39–48. IEEE Computer Society, 2005.

[Bellenot(1990)] Steven Bellenot. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 122–127, January 1990.

[Bonwick(1994)] Jeff Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 6–6, 1994.

[Boukerche et al.(1999)Boukerche, Das, Fabbri, and Yildz] Azzedine Boukerche, Sajal K. Das, A. Fabbri, and O. Yildz. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 166–173. IEEE Computer Society, May 1999.

[Cai et al.(2005)Cai, Turner, Lee, and Zhou] Wentong Cai, Stephen J. Turner, Bu-Sung Lee, and Junlan Zhou. An alternative time management mechanism for distributed simulations. *ACM Trans. Model. Comput. Simul.*, 15(2):109–137, April 2005. ISSN 1049-3301. doi: 10.1145/1060576.1060577. URL http://doi.acm.org/10.1145/1060576.1060577.

[Carothers and Fujimoto(2000)] Christopher D. Carothers and Richard M. Fujimoto. Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 11(3):299–317, 2000.

[Carothers and Perumalla(2010)] Christopher D. Carothers and Kalyan S. Perumalla. On deciding between conservative and optimistic approaches on massively parallel platforms. In *Winter Simulation Conference*, pages 678–687, 2010.

[Carothers et al.(1999a)Carothers, Perumalla, and Fujimoto] Christopher D. Carothers, Kalyan S. Perumalla, and Richard Fujimoto. The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In *Winter Simulation Conference*, pages 1624–1633, 1999a.

[Carothers et al.(1999b)Carothers, Perumalla, and Fujimoto] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, july 1999b.

[Chen et al.(2011)Chen, Lu, Yao, Peng, and Wu] Li-li Chen, Ya-shuai Lu, Yi-Ping Yao, Shao-liang Peng, and Ling-da Wu. A well-balanced time warp system on multi-core environments. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS, pages 1–9. IEEE Computer Society, 2011. ISBN 978-1-4577-1363-7. doi: 10.1109/PADS.2011.5936752.

[Chetlur and Wilsey(2006)] M. Chetlur and Philip A. Wilsey. Causality information and fossil collection in time warp simulations. *Winter Simulation Conference*, 0:987–994, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/WSC.2006.323186.

[Chilimbi et al.(2000)Chilimbi, Hill, and Larus] Trishul M Chilimbi, Mark D Hill, and James R Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[Choe and Tropper(1999)] Myongsu Choe and Carl Tropper. On learning algorithms and balancing loads in time warp. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 101–108. Springer Verlag, 1999.

[Cucuzzo et al.(2007)Cucuzzo, D'Alessio, Quaglia, and Romano] Diego Cucuzzo, Stefano D'Alessio, Francesco Quaglia, and Paolo Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. *Distributed Simulation and Real Time Applications, IEEE/ACM International Symposium on*, 0:227–234, 2007. ISSN 1550-6525. doi: http://doi.ieeecomputersociety.org/10.1109/DS-RT.2007.18.

[Dantzig(1957)] George B. Dantzig. Discrete-variable extremum problems. *Operational Research*, (5), 1957.

[Das and Fujimoto(1993)] Samir R. Das and Richard M. Fujimoto. A performance study of the cancelback protocol for time warp. *SIGSIM Simul. Dig.*, 23(1):135–142, 1993. ISSN 0163-6103. doi: http://doi.acm.org/10.1145/174134.158476.

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

[Das and Fujimoto(1997b)] Samir R. Das and Richard M. Fujimoto. Adaptive memory management and optimism control in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 7(2):239–271, 1997b. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/249204.249207.

[Das et al.(1994)Das, Fujimoto, Panesar, Allison, and Hybinette] Samir R. Das, Richard M. Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. GTW: a time warp system for shared memory multiprocessors. In *WSC '94: Proceedings of the 26th conference on Winter simulation*,

pages 1332–1339. Society for Computer Simulation International, 1994. ISBN 0-7803-2109-X.

[Ferscha(1995)] A. Ferscha. Probabilistic adaptive direct optimism control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 120–129. IEEE Computer Society, June 1995.

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

[Fujimoto(1999)] Richard M. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 46–53. IEEE Computer Society, May 1999.

[Fujimoto and Hybinette(1997)] Richard M. Fujimoto and Maria Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation*, 7(4):425–446, October 1997.

[Fujimoto et al.(1992)Fujimoto, Tsai, and Gopalakrishnan] Richard M. Fujimoto, J.J. Tsai, and G. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, 1992.

[Gafni(1985)] A. Gafni. Space management and cancellation mechanisms for Time Warp. *Tech. Rep. TR-85-341, University of Southern California, Los Angeles (Ca,USA)*, unknown 1985.

[Ghosh and Fujimoto(1991)] Kaushik Ghosh and Richard Fujimoto. Parallel discrete event simulation using space-time memory. In *ICPP (3)*, pages 201–208, 1991.

[Glazer and Tropper(1993)] D. W. Glazer and Carl Tropper. On process migration and load balancing in time warp. *IEEE Transactions Parallel Distrib. Syst.*, 4(3):318–327, 1993. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.210814.

[HPDCS Research Group(2012)] HPDCS Research Group. ROOT-Sim: The ROme OpTimistic Simulator - v 1.0. http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/, October 2012. URL http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/.

[IEEE Std 1516-2000 (2000)(2000)] IEEE Std 1516-2000 (2000). IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)

– Framework and Rules. Technical report, Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 2000.

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

[Jefferson(1990)] David R. Jefferson. Virtual Time II: storage management in conservative and optimistic systems. In *Proceedings of the 9th annual ACM symposium on Principles of distributed computing*, pages 75–89. ACM, 1990. ISBN 0-89791-404-X. doi: http://doi.acm.org/10.1145/93385.93403.

[Jiang et al.(1994)Jiang, Shieh, and Liu] M.-R. Jiang, S.P. Shieh, and C.-L. Liu. Dynamic load balancing in parallel simulation using time warp mechanism. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pages 222–229. IEEE Computer Society, 1994. ISBN 0-8186-6555-6.

[Kahle(2005)] Jim Kahle. The cell processor architecture. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 3–, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: 10.1109/MICRO.2005.33. URL http://dx.doi.org/10.1109/MICRO.2005.33.

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

[Lea(1996)] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html, 1996.

[Lin and Lazowska(1991)] Yi-Bing Lin and Edward D. Lazowska. Processor scheduling for Time Warp parallel simulation. In *Proceedings of the 23rd SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 11–14. IEEE Computer Society, January 1991.

[Lin and Preiss(1991)] Yi-Bing Lin and Bruno R. Preiss. Optimal memory management for time warp parallel simulation. *ACM Transactions on Modeling and Computer Simulation*, 1:283–307, 1991.

[Lin et al.(1993)Lin, Preiss, Loucks, and Lazowska] Yi-Bing Lin, Bruno R. Preiss, Wayne M. Loucks, and Edward D. Lazowska. Selecting the checkpoint interval in time warp simulation. *SIGSIM Simul. Dig.*, 23(1):3–10, 1993. ISSN 0163-6103. doi: http://doi.acm.org/10.1145/174134.158460.

[Liu and Wainer(2012)] Qi Liu and Gabriel Wainer. Multicore acceleration of discrete event system specification systems. *SIMULATION*, 88:801–831, july 2012. doi: 10.1177/0037549711412237.

[Madhava Rao et al.(Dec)Madhava Rao, Thondugulam, Radhakrishnan, and Wilsey] D. Madhava Rao, N.V. Thondugulam, R. Radhakrishnan, and P.A. Wilsey. Unsynchronized parallel discrete event simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1563–1570 vol.2, Dec. doi: 10.1109/WSC.1998.746030.

[Meraji and Tropper(June)] S. Meraji and C. Tropper. Optimizing techniques for parallel digital logic simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 23(6):1135–1146, June. ISSN 1045-9219. doi: 10.1109/TPDS.2011.246.

[Meraji et al.(2010)Meraji, Zhang, and Tropper] Sina Meraji, Wei Zhang, and Carl Tropper. A multi-state q-learning approach for the dynamic load balancing of time warp. In *Principles of Advanced and Distributed Simulation (PADS)*, pages 1–8, 2010. ISBN 978-1-4244-7292-5.

[Miller(2010)] Ryan J. Miller. Optimistic parallel discrete event simulation on a beowulf cluster of multi-core machines. Master's thesis, University of Cincinnati, 2010.

[Mueller(1995)] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, November 1995. ISSN 0362-1340. doi: 10.1145/216633.216677. URL http://doi.acm.org/10.1145/216633.216677.

[Palaniswamy and Wilsey(1993)] Avinash C. Palaniswamy and Philip A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and distributed simulation*, pages 127–134. ACM, 1993. ISBN 1-56555-055-2. doi: http://doi.acm.org/10.1145/158459.158475.

[Palaniswamy and Wilsey(1994)] Avinash C. Palaniswamy and Philip A. Wilsey. Scheduling Time Warp processes using adaptive control techniques. In *Proceedings of the 1994 Winter Simulation Conference*, pages 731–738. Society for Computer Simulation, December 1994.

[Pellegrini et al.(2011)Pellegrini, Vitali, and Quaglia] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. The ROme OpTimistic Simulator: Core internals and programming model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools. ICST, 2011.

[Pellegrini et al.(2012)Pellegrini, Vitali, and Quaglia] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. Transparent and efficient shared-state management for optimistic simulations on multi-core machines. In *Proceedings 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, pages 134–141. IEEE Computer Society, August 2012.

[Peluso et al.(2011)Peluso, Didona, and Quaglia] Sebastiano Peluso, Diego Didona, and Francesco Quaglia. Application transparent migration of simulation objects with generic memory layout. In *Proceedings of the 25th Workshop on Principles of Advanced and Distributed Simulation*, pages 169–177. IEEE Computer Society, june 2011. doi: 10.1109/PADS.2011.5936755.

[Preiss and Loucks(1995)] Bruno R. Preiss and Wayne M. Loucks. Memory management techniques for time warp on a distributed memory machine. In *PADS*, pages 30–39, 1995.

[Preiss et al.(1994)Preiss, Loucks, and MacIntyre] Bruno R. Preiss, Wayne 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.

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

[Quaglia and Baldoni(1999)] Francesco Quaglia and Roberto Baldoni. Exploiting intra-object dependencies in parallel simulation. *Inf. Process. Lett.*, 70(3):119–125, 1999.

[Quaglia and Cortellessa(2002)] Francesco Quaglia and Vittorio Cortellessa. On the processor scheduling problem in time warp synchronization. *ACM Transactions on Modeling and Computer Simulation*, 12, July 2002. ISSN 1049-3301.

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

[Riley et al.(2000)Riley, Fujimoto, and Ammar] George F. Riley, Richard Fujimoto, and Mostafa H. Ammar. Network aware time management and event distribution. In *PADS*, pages 119–126, 2000.

[Rönngren and Ayani(1994a)] Robert Rönngren and Rassul Ayani. Service oriented scheduling in Time Warp. In *Proceedings of 1994 Winter Simulation Conference*, pages 1340–1346. Society for Computer Simulation, December 1994a.

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

[Rönngren et al.(1996)Rönngren, Liljenstam, Ayani, and Montagnat] Robert Rönngren, M. Liljenstam, Rassul 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.

[Santoro and Quaglia(2005)] Andrea Santoro and Francesco 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, June 2005.

[Santoro and Quaglia(2010)] Tiziano Santoro and Francesco Quaglia. A low-overhead constant-time LTF scheduler for optimistic simulation systems. In *Proceedings of the IEEE Symposium on Computers and Communications*, pages 948–953, 2010.

[Seal and Perumalla(2011)] Sudip K. Seal and Kalyan S. Perumalla. Reversible parallel discrete event formulation of a tlm-based radio signal propagation model. *ACM Trans. Model. Comput. Simul.*, 22(1):4:1–4:23, December 2011. ISSN 1049-3301. doi: 10.1145/2043635.2043639. URL http://doi.acm.org/10.1145/2043635.2043639.

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

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

[speedes()] speedes. SPEEDES. http://www.speedes.com, 2005.

[Srinivasan and Reynolds(1998)] S. Srinivasan and P.F. Reynolds, Jr. Elastic time. *ACM Transactions on Modeling and Computer Simulation*, 8(2): 103–139, April 1998.

[Swenson and Riley(2012)] Brian Paul Swenson and George F. Riley. A new approach to zero-copy message passing with reversible memory allocation in multi-core architectures. In *PADS*, pages 44–52, 2012.

[Toccaceli and Quaglia(2008)] Roberto Toccaceli and Francesco 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. ISBN 978-0-7695-3159-5. doi: http://dx.doi.org/10.1109/PADS.2008.23.

[Vitali et al.(2009)Vitali, Pellegrini, and Quaglia] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. Benchmarking memory management capabilities within root-sim. In *Proceedings of the 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*. IEEE Computer Society, 2009.

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