Optimistic Modeling and Simulation of Complex Hardware Platforms and Embedded Systems on Many-Core HPC Clusters

Alireza Poshtkohi, M. B. Ghaznavi-Ghoushchi, and Kamyar Saghafi

Abstract—The large size and complexity of the modern digital hardware impose great challenges to design and validation. Hardware Description Languages (HDLs) and System-Level Description Languages (SLDLs) rely on sequential discrete event semantics. Parallel Discrete Event Simulation (PDES) has recently gained extensive attention for parallelizing these languages due to the ever-increasing complexity of embedded and cyber physical systems. However, PDES application has not yet reached acceptable maturity and pervasiveness for accelerating computer architecture problems. This is due to inherent complexity of hardware components that require using different advanced PDES techniques. In this paper, we look at the main problem from a radically different angle. First, we suppose there is only a single universal discrete event model of computation for simulation purely defined by distributed optimistic PDES, i.e., logical-process-based event scheduling worldview. Second, we construct a new parallel system-level simulation language called OSML for ESL. Third, we propose a unified Cloud-based CAD tool called Troodon to automatically parallelize existing hardware languages atop OSML. To the best of our knowledge, OSML is the first work on optimistic synchronization applied to hardware-specific SLDLs and hardware models at different levels of abstraction in ESL, which contain complex data structures by proposing a hybrid checkpointing scheme.

Index Terms—Parallel discrete event simulation (PDES), optimistic synchronization, system-level description languages (SLDLs), electronic system level (ESL), hybrid checkpointing

1 Introduction

Today's System-On-Chip (SoC) designs and embedded systems contain a great number of processing elements (such as multi-core processors, GPUs, DSPs and other IP cores) connected to themselves, and other elements, including, RAM and peripheral I/O devices. Traditionally, Hardware Description Languages (HDLs)—including Verilog and VHDL—and hardware-specific System-Level Description Languages (SLDLs)—in particular SystemC and SystemVerilog [1], [2]—have been used for modeling and describing an embedded system at different levels of abstraction. The models are turned into executables by a hardware-aware compiler. The compiled code is linked with a sequential simulation kernel that manages its behavior at runtime. Conventionally, the syntax and simulation semantics of all of these languages have been defined by sequential Discrete Event Simulation (DES). Today, computer simulation is a well-accepted approach to verify embedded systems. It has a myriad of other applications, for example, power and performance estimation, and design space exploration.

Parallel Discrete Event Simulation (PDES) has been exploited to speed up DES programs since its advent in the late 1970s, which is broadly divided into conservative and optimistic [3], [4]. The parallelization of hardware languages has also occasionally been under focus over the past two decades, and has been recently gaining momentum with the emergence of multi-core microprocessors. However, PDES application has not yet reached acceptable maturity and pervasiveness for accelerating computer architecture problems. This is due to the inherent complexity of hardware components that requires using different advanced PDES techniques. Time Warp [5] is widely known as the most common optimistic PDES protocol in the simulation community because it can scale up to millions of cores [6]. In hardware design, the large majority of reported works have been concentrated on conservative PDES, mainly, synchronous. Optimistic synchronization, of course, has been applied to Gate-Level (GL) models and a limited synthesizable subset of HDLs. For instance, there is no report on optimistic simulation of a comprehensive hardware blueprint, made up of processors, RAM, buses, and routers in higher levels like Register Transfer Level (RTL) and System Level (SL) beside GL. On the other hand, with the popularity of Cloud Computing [7], Electronic Design Automation (EDA) tools are moving to the Cloud [8]. Moreover, the existence of a comprehensive architectural parallel modeling and simulation platform is important to accelerate hardware/software codesign for exascale computing [9]. To address these challenges and significantly reduce the complexity, we introduce an

• *The authors are with the Department of Electrical Engineering, Shahed University, Tehran 3319118651, Iran.*
E-mail: arp@poshtkohi.ir, {ghaznavi, saghafi}@shahed.ac.ir.

Manuscript received 19 Nov. 2017; revised 28 Apr. 2018; accepted 30 June 2018. Date of publication 25 July 2018; date of current version 16 Jan. 2019.
(Corresponding author: M. B. Ghaznavi-Ghoushchi.)
Recommended for acceptance by M. M. Hayat.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2860014

optimistic PDES language and CAD tool for Electronic System Level (ESL). Consequently, the key contributions of this article are as follows:

1) For the first time, we apply optimistic PDES [5] to *hardware-specific* system-level description languages in ESL, including, SystemC, SystemVerilog and SpecC [10].
2) For the first time, we allow different hardware models at different electronic abstraction levels to be executed by optimistic synchronization, including, many-state processors, memories, buses, routers, and so on. This is achieved by proposing a hybrid state saving scheme for optimistic execution of HDLs and SLDLs.
3) We propose a new optimistic PDES language called *Optimistic System Modeling Language* (OSML) along with its distributed simulation kernel, which have built-in support for existing HDL and SLDL features. Two PDES's *systems programming* and *application programming* models are precisely defined. We benefit from OSML as an intermediate parallel simulation language (PSL) in where other hardware languages are translated into OSML. Therefore, OSML can also provide cross-language interoperability through language compilers and tools for co-simulation.
4) We implement a unified Cloud-based ESL CAD tool called *Troodon* to automate the tasks of description, visualization, compilation, massively distributed parallel simulation and performance monitoring of all the mainstream HDLs and SLDLs on many-core HPC clusters.

The rest of the paper is organized as follows. In Section 2, we study some basics, our motivation behind this article, and related work within the context of HDL/SLDL parallelization. We introduce OSML language in Section 3. Section 4 mainly discusses optimistic OSML simulation semantics and the implementation of its distributed PDES simulation kernel. Section 5 focuses on Troodon's overall architecture and tool flow. A series of Troodon and OSML case studies is described in Section 6 on a Cloud-based HPC cluster with a 40 Gb/s fiber-optic network fabric. Section 7 concludes the paper and presents our direction for further research on OSML.

2 Basics and Motivations

In this section, the principal concepts needed for this paper are discussed, and then the main problem is clearly defined.

2.1 Optimistic PDES Fundamentals

In PDES, the entire simulation is divided into a collection of smaller sub-tasks referred to as Logical Processes (LPs), and each of them is executed by a processor or node. These LPs communicate with each other by exchanging time-stamped event messages in simulated time. These events are used to synchronize the execution of LPs in parallel. An event refers to an update to simulation system state at a specific simulation time instant. LPs do not share any state variables and solely communicate through these time-stamped messages. Synchronization between LPs is violated when one of the LPs receives an out-of-order event. To overcome this problem, a lot of synchronization protocols have been proposed that are mainly fallen into synchronous and asynchronous [4]. In synchronous simulation, all LPs see a single clock and synchronize with one another through a global heavy barrier mechanism after processing events in current simulation time. In asynchronous algorithms, LPs process events at different times, namely, different local clocks. Asynchronous conservative algorithms strictly avoid causality violations. Asynchronous optimistic algorithms allow events to be processed out of order, but use a rollback mechanism to recover from such errors.

Time Warp (TW) was the first optimistic synchronization algorithm and remains the most widely used optimistic approach to this day [5]. Undoing modification of state variables can be accomplished by taking a snapshot of the state of each LP prior to processing each event. A TW-LP does need to hold a record of past input and output events. The state saving introduces the problem that one must be able to later recover the memory utilized to keep the checkpointed states. This problem is addressed by computing a lower bound on the timestamp of any future rollback that might occur, also known as *global virtual time* (GVT). Memory used for saved state variables older than GVT can be reclaimed. Optimistic synchronization is much more complicated than conservative ones, but reaches much higher speed and strong scaling [6] because processors process events with different timestamps and the property of pipeline execution is observed. It also improves the load balance by keeping idle processors busy to process future events. Effective extraction of the parallelism from simulation models has a significant impact on parallel simulation, which is usually attained by proper breaking down the design into finer processes and using a good fine-grained partitioning algorithm to balance the load and minimize inter-process communication.

One of the central issues in Time Warp synchronization is how to support fast state restoration with low-overhead checkpointing. The main proposed strategy is based on Copy State Saving (CSS). In this protocol, the state of an LP is stored into the state queue for each event execution. To reduce CSS overhead, we can save the state of an LP after executing multiple events periodically, which is referred to as Periodic State Saving (PSS). In PSS, a state to be restored may not exist. In this case, it must be computed using a previously saved state and replaying intermediate events in a coast forward, which adds a penalty time to recovery. A relatively different scheme is based on a checkpoint protocol known as Incremental State Saving (ISS). Compared to CSS, this protocol decreases both checkpoint overhead and memory usage. It keeps a history of the before-images of state variables that are added during event execution. Each before-image is recorded prior to updating that state variable. The process of restoring the associated memory requires a backward re-traversal of the recorded history and copying the before-images into their original state locations until the state to be restored is found. An important alternative technique to checkpointing is Reverse Computation (RC). In RC, the system state is recovered not by relying on memory to restore the state, but by computing backward paths from a current point of execution to the rollback point in the past. This noticeably requires the computation to be reversible, and compels the reverse code to be invoked to reach the desired point in the past at runtime.

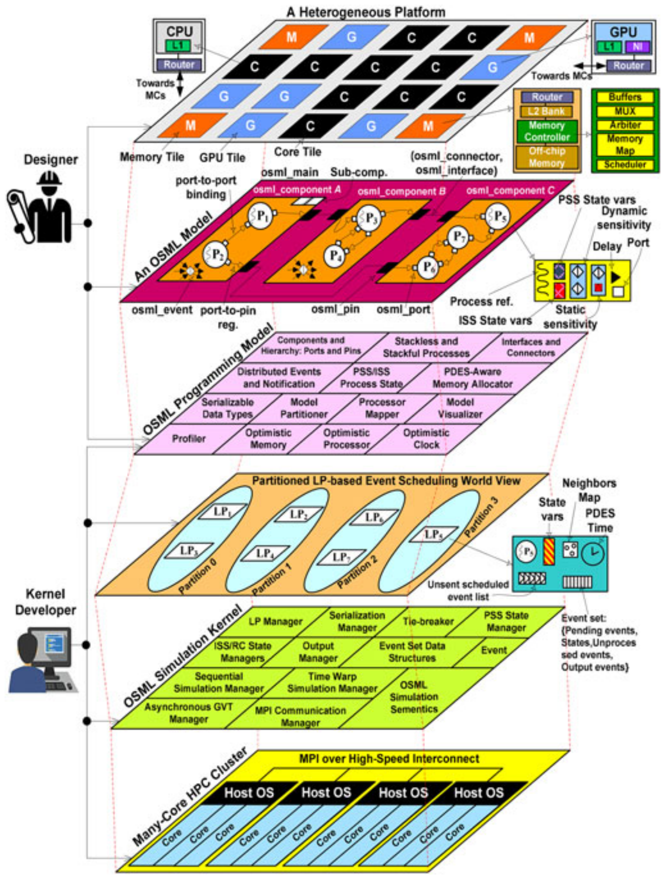


Fig. 1. Architecture of the OSML environment.

2.2 SLDL Concepts

In hardware-specific system-level design, abstract models are built to better understand the system, refinement and optimization for final implementation. Different fully-fledged object-oriented system-level languages, like SystemC, SystemVerilog and SpecC, exist for modeling and system-level description of embedded systems at various levels of abstraction. They provide a set of extensions to support system-level modeling requirements, such as executability, behavioral and structural hierarchy, concurrency, communication, synchronization, and scheduling. All of these languages benefit from a common conceptual strategy and only differ in language syntax. Their simulation semantics is defined by *evaluate*/*update* paradigm using delta cycles as a way of deterministic execution atop sequential DES.

2.3 Problem Definition

HDLs and SLDLs are collectively an integral part in the design process of hardware components. Since these components are becoming very complex, parallel simulation is one of the main ways for verification. Existing hardware languages have been designed heavily reliant on sequential DES and the same simulation semantics (delta cycles), and so they raise several obstacles for parallelization by asynchronous PDES. This paper directly addresses the complexity of the parallel simulation for electronic systems by proposing a new PDES language, which supports necessary abstractions to describe software and hardware components of a complex design. In other words, there should be a parallel simulation language for electronic systems rather than seeking how to parallelize existing languages separately. Such a PSL allows us to easily implement compilation tools in order to convert those languages to our PSL and avoid basic discoveries.

As discussed in Section 2.4, optimistic synchronization has only been applied to a limited subset of HDLs, more precisely, only synthesized models or synthesizable data paths. In addition, optimistic PDES has not yet been employed for the simulation of hardware-specific SLDLs. The main reason can be attributed to the complexity of these languages. A hardware model at abstraction levels higher than the gate has a variety of state variables in form of complex data structures that must be handled by PDES executive. With a little analysis of hardware models written in an HDL or SLDL consisting of data and control path, bus, and memory, we find they have different state variables that cannot be expressed singly by one of CSS, PSS, ISS and RC techniques for optimal simulation. Fig. 1 illustrates architecture of the OSML environment. User describes the model using optimistic PDES-aware primitives of the OSML language and makes right decisions for optimal execution. The OSML's parallel programming model is the only thing that the modeler needs to be aware of. The model is mapped onto the logical process pattern by distributed optimistic OSML kernel, which implements the Time Warp protocol. Each LP executes OSML simulation semantics. This semantics is purely defined for optimistic synchronization and guarantees deterministic simulation. As outlined in Fig. 1, let us consider how designer can specify a many-core heterogeneous NoC on top of the OSML language for an optimistic run. This model is made up of different components, which are modeled at various levels of abstraction, detailed in Table 1. Those components that contain a few state

TABLE 1
Specifications of the Heterogeneous Platform

Component	Abstraction	Specification Details	Checkpointing Mode
CPU	Behavioral RTL	Single process, FSM: 200 state variables, synchronous	Hybrid (PSS+ISS)
GPU	Structural RTL	Multiple processes, Pipeline, synchronous	PSS
L1 Cache/L2 Bank	Behavioral RTL	Single process, asynchronous	Hybrid (PSS+ISS)
Router	Behavioral RTL	Multiple processes, array-based buffer, synchronous	Hybrid (PSS+ISS)
Arbiter	Behavioral RTL	Multiple processes, synchronous	Hybrid (PSS+ISS+RC)
Network Interface (NI)	System Level - BFM	Multiple processes, synchronous	Hybrid (PSS+ISS+RC)
Off-Chip Memory	Behavioral RTL	Single process, synchronous	Hybrid (PSS+ISS)
Buffer	Behavioral RTL	Single process, array-based buffer, asynchronous	Hybrid (PSS+ISS)
MUX	Structural RTL	Multiple processes, asynchronous	PSS
Memory Map	Behavioral RTL	Single process, synchronous	Hybrid (PSS+ISS+RC)
Scheduler	System Level - BFM	Single process, synchronous	Hybrid (PSS+ISS+RC)

TABLE 2
Comparison Between OSML and Other PSLs

Language	Domain	Protocol	Worldview	Programming	State Saving	Granularity	Mapping	Abstract Data Type	PDES-aware Compiler	Visualization
OSML	ESL, General	TW, distributed	LP, Process interaction	Fully dynamic OO, System & user levels	Hybrid, Adaptive	Process	Automatic & Manual: HWG/ profile-driven	Reversible containers & memory	Not needed, PDES-aware language	Cloud-based, UML/XML
APOSTLE	General	BTB, shared	Process interaction	Basic static OO, User level	ISS	Entity	Round-robin	Basic	Mandatory	None
Parsec	General	Spacetime, distributed	Process interaction	Structured, User	CSS	Entity	Manual	Basic	Mandatory	C-based IDE
TeD	Telecom.	TW, shared	Process interaction	Basic static OO, User level	ISS	Entity	Round-robin	Basic	Mandatory	Java-based IDE

variables are implemented using PSS checkpointing. We take advantage of HSS for the rest of components. As an example, a processor modeled using FSM, a router with an internal queue for packet requests modeled as an array-based priority queue and a synchronous clocked memory are described by a hybrid checkpointing scheme. Here, component pins and other state variables are introduced to the OSML kernel respectively by PSS, and ISS/RC.

2.4 Related Work

In hardware design, PDES has predominantly been used for parallel simulation of gate-level logic circuits expressed by netlists due to its simplicity. Numerous works explore different conservative and optimistic protocols along with partitioning and load-balancing algorithms at GL [11], [12], [13], [14], [15], [16]. PDES has also applied to a limited synthesizable subset of HDLs (e.g., data path) that are close to GL using Time Warp through CSS and PSS [17], [18]. For instance, a complete system written in an HDL has not yet been simulated by Time Warp that consists of processors, memory modules and buses at RTL. Recently, the parallelization of hardware-specific SLDLs (i.e., SystemC, SystemVerilog and SpecC) has gained widespread attention mostly using the synchronous PDES on multi-core hosts [19], [20], [21], [22], [23], [24], [25], [26]; and of course, there is no optimistic implementation of SLDLs (SystemC, SystemVerilog and SpecC) in ESL yet. Furthermore, there are a number of reports that utilize conservative PDES to accelerate various computer architecture simulations, including, multi-core and many-core systems, NoC and CMP [27], [28], [29], [30], [31]. A complete survey can be found in [32].

As stated by Fujimoto in 2016, one of the six major research challenges in Parallel and Distributed Simulation (PADS) is to make it widely accessible to the general Modeling and Simulation (M&S) community by simplifying the development of simulation models and supporting Cloud Computing services [3]. Three main approaches used to address this problem are: (1) PDES kernel libraries; (2) development of automatic parallelization methods for sequential simulation languages (SSLs); and (3) new parallel simulation languages (PSLs). To date, progress has been made greatly in areas (1) and (2). There are three low-level optimistic PDES libraries in the public domain [33], [34], [35], each of which thoroughly supports only a single state-saving scheme. These libraries facilitate the development of simulation models to some degree, although working with them is hard even for experts in parallel computing who are unfamiliar with PDES. A fundamental issue is that the optimal development of a model, as discussed throughout this paper, is heavily dependent on the use of a combination of state-saving techniques. In this paper, we present a layered methodology based on operating system concepts for separation of concerns to build PDES kernels, in which many low-level parallel processing functionalities and optimistic PDES remain in the PDES kernel layer and the rest of the services like state management are moved to OSML kernel. On the other hand, since programming with PDES libraries is cumbersome for modelers unaccustomed to PDES and they do not support modular design, a few of PSLs were created in the 1990s. However, their development was abandoned because they did not cover the broad requirements of underlying parallel processing (such as the lack of a uniform infrastructure for state saving, and reversible complex data structures) and fully-fledged object-oriented programming constructs available in languages like C++ and Java. If they resolved these problems, it would be greatly favorable to convert SSLs, such as SystemC, DEVS, etc., into those PSLs. For example, parallelization techniques of all the hardware languages either use a third-party optimistic PDES library or extend their kernel relied on conservative PDES. Therefore, new modeling languages coupled with mechanisms to automatically translate modeling abstractions to efficient parallel simulation code are an important avenue for M&S community. Suitable intermediate representations are needed to provide descriptive yet precise specifications of model state and behavior [3]. OSML is an essential attempt to develop a new optimistic PDES language for ESL in this regard, of course, OSML defines a powerful programming interface that can be used to model other DES problems like large-scale networked systems [36]. PSLs were usually built by adding primitives or library functions to a sequential simulation language to specify parallel execution. However, OSML has been designed from the ground up with the aim of exposing explicit parallelism in mind.

Table 2 compares OSML with other PSLs. Objected-oriented (OO) PSLs only supports a very basic part of OO concepts, in which each entity contains a set of processes communicating by exchanging messages. They are deficient in common OO concepts such as templates, virtual methods, operator overloading, polymorphism, abstract classes, etc., which are compulsory for system-level design. They need a PDES-aware compiler that is used to generate the underlying LP code atop a typical PDES library written in a language like C or C++. Consequently, the user is prohibited from low-level access to PDES engine if he is willing to make changes for performance optimization. Provided that the intermediate code is available, it is difficult to be understood and modified (because the user has no knowledge of its execution semantics). In addition, the extra emitted code can cause significant cache misses and hurt the efficiency, because each process has a different LP implemented in the source code. More precisely, this approach injects and replicates PDES simulator codes into the model code. Cache read misses from an instruction cache generally cause the largest delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory. They lack a model elaboration phase and so we call them as *static languages*. This exposes the limitation on the models that cannot be created dynamically at runtime. This feature is essential for reusing third-party model libraries and design space exploration (e.g., an NoC to be reconfigured at runtime). The granularity of these languages is at the entity level in which an entire entity is mapped to an LP, and thus they can dramatically reduce the degree of parallelism in hierarchical models. All of them provide only a single state-saving scheme, simple data types and fixed-size one-dimensional arrays at compile time. Namely, they make it impossible to allocate dynamic memory and thus to build complex data structures such as lists, trees and queues that are required in system-level and hardware design (because the modeler also has no low-level PDES access to build them on his own right, and even there were no compiler support for this). They hand the model-to-processor mapping over to the user (due to lack of an elaboration phase, he is made map entities to processors manually, which is a very time-consuming task for large-scale models) or use the round-robin algorithm of the underlying PDES simulator. APOSTLE and Parsec languages extended the C language [37], while TeD was a domain-specific language for telecommunication networks and used a mixture of the two VHDL and C languages [38] (which was problematic for the user, because two different languages must be learnt and used to write structures and behaviors of the models). The GUI of these languages only supports visualizing the structure of the models not their behaviors through simple tools built before the current-day status of web and cloud computing [39], [40].

The OSML language directly targets the above issues. OSML is a fully-fledged object-oriented language (with a dedicated elaboration phase) and has a PDES-aware syntax, and therefore can be compiled by a PDES-unaware compiler like GCC. On the one hand, OSML is a *PDES systems programming language* that allows systems programmers and compiler writers (of SSLs) to write and emit optimized codes with direct access to OSML exokernel. OSML implements a wide range of reversible data containers using an optimistic dynamic memory manager, in where system programmers can extend them for PDES application programmers. On the other hand, OSML is *a PDES application programming language* that lets the users unfamiliar with PDES easily develop their own models by instantiating the OSML's reversible objects and primitive data types for parallel simulation. In OSML, there is only a single instance of the LP code at any moment of time that is shared among all the model processes at runtime on each physical core. OSML takes advantage of a

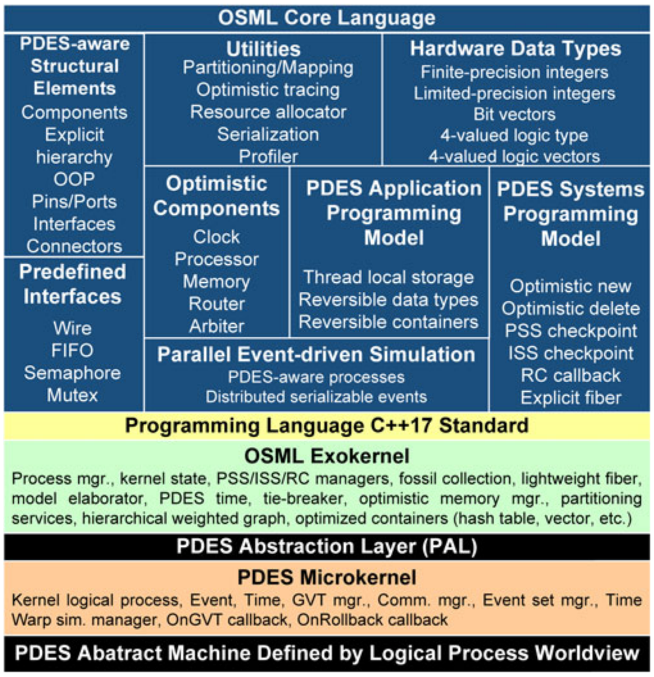

OSML Core Language
PDES-aware Structural Elements
Components
Explicit hierarchy
OOP
Pins/Ports
Interfaces
Connectors
Utilities
Partitioning/Mapping
Optimistic tracing
Resource allocator
Serialization
Profiler
Hardware Data Types
Finite-precision integers
Limited-precision integers
Bit vectors
4-valued logic type
4-valued logic vectors
Optimistic Components
Clock
Processor
Memory
Router
Arbiter
PDES Application Programming Model
Thread local storage
Reversible data types
Reversible containers
PDES Systems Programming Model
Optimistic new
Optimistic delete
PSS checkpoint
ISS checkpoint
RC callback
Explicit fiber
Predefined Interfaces
Wire
FIFO
Semaphore
Mutex
Parallel Event-driven Simulation
PDES-aware processes
Distributed serializable events
Programming Language C++17 Standard
OSML Exokernel
Process mgr., kernel state, PSS/ISS/RC managers, fossil collection, lightweight fiber, model elaborator, PDES time, tie-breaker, optimistic memory mgr., partitioning services, hierarchical weighted graph, optimized containers (hash table, vector, etc.)
PDES Abstraction Layer (PAL)
PDES Microkernel
Kernel logical process, Event, Time, GVT mgr., Comm. mgr., Event set mgr., Time Warp sim. manager, OnGVT callback, OnRollback callback
PDES Abatract Machine Defined by Logical Process Worldview

Fig. 2. OSML language architecture.

profile-driven sequential execution for partitioning purposes and choosing the appropriate type of state saving for state variables of the model in parallel simulation. The structure of a model is accessible to the modelers and OSML exokernel as a *hierarchical weighted graph* (HWG) in elaboration phase, which is used to implement automatic and manual partitioning algorithms reliant on the weight information of graph nodes. This mapping is performed at the granularity of process level not OSML components. OSML, in addition to being a general-purpose PDES language, provides a broad spectrum of language constructs for ESL, including, parallel hardware execution semantics, hardware-specific data types, hardware timing models, interfaces, and optimistic VCD trace files. Furthermore, powerful OSML primitives allow the programmer to use the facilities present in common parallel programming languages like MPI—including, data marshaling for distributed execution, efficient resource allocation on network nodes when partitioning, parallel execution statistics at different levels such as PDES and MPI for adaptive runs in optimistic mode, and hybrid PDES and OpenMP (or CUDA) parallel programming model for data-parallel applications (e.g., we can program a system-level accelerator for signal processing in this style).

3 OSML Language

In this section, we examine the syntax and structures of OSML. We show how the designer can leverage the underlying parallelism and explicitly specify process states in a hybrid fashion based on OSML primitives. One of the primary goals of OSML language is to enable optimistic, explicitly parallel system-level modeling—that is, real parallel modeling of systems above the RTL that might be implemented in hardware, software or a combination of the two. Of course, RTL and GL modeling are also possible in OSML. Fig. 2 illustrates OSML language architecture. OSML syntax is aware of PDES. In the design of OSML, we

TABLE 3
Osml Language Primitives

Feature	PDES-aware OSML Constructs
Hierarchy/Modularity	*osml_component: register_subcomponent(), osml_pin: register_pin()*
Processes	*osml_process p = register_process(method,mode,name), osml_port(osml_inport, osml_outport,osml_gport), p.register_port()*
Interfaces /Channels	*osml_connector, osml_nonshared_interface: osml_wire, Offload(p), fetch(), put(val,p), osml_shared_interface(osml_fifo, osml_mutex, osml_semaphore)*
System-Level Synchronization	Distributed *osml_event:register_initiator, register_subscriber, osml_wait(expr,p), osml_notify(expr,p)* *sensitivity list (p.register_sensitivity)*
Optimistic Execution Semantics	PSS*: osml_process_state(clone_state, restore_state, free_state), p.register_state()*, ISS: *osml_iss_state_manager, save_address(addr,len),* RC*:* reverse callbacks, optimistic dynamic memory, reversible ADTs
Distributed Shared Memory Execution Semantics	Serializable composite data type system (*serialize, deserialize*), interface function calls (*CreateInstance/DestroyInstance, CloneValue, CopyValue, DeleteValue*)
Model Partitioning/ Processor Mapping	*osml_partitioner(partition), osml_mapper(map), osml_profiler*
Simulation	*osml_simulator: start(cores,time,partitioner,mapper), pause(), stop()*

decided to minimize PDES kernel services and transfer the remains to OSML kernel, which could be implemented at higher layers. This enabled us to seamlessly implement different checkpointing mechanisms in OSML kernel to support hybrid state saving. As a result, OSML kernel is independent of a particular PDES kernel. The least-defined services for an optimistic PDES engine make up a *microkernel*. State-saving operations are totally moved to OSML kernel, and so we refer to it as *OSML exokernel*—since OSML allows the programmer to directly access to state-saving utilities without knowing the complex details of the underlying parallel mechanisms. PAL defines a standard wrapper facade for interaction between both the kernels, which can be implemented simply on top of various PDES libraries. PDES kernel informs the OSML kernel of rollback and GVT points through two callbacks. Therefore, OSML kernel will be able to easily manage different state-saving schemas and implement fossil collection by itself. The language introduces two PDES systems and application programming models to the user for the sake of OSML exokernel. In the former, programmer has direct access to state-saving routines (PSS, ISS and RC), and subsequently can implement his application optimally. In the latter, modeler can reuse reversible containers and data types implemented by OSML. Moreover, OSML hardware data types are reversible and support distributed execution through data marshaling. OSML also implements widely used hardware components as reusable optimistic templates.

3.1 Language Constructs

OSML language has been crafted in C++17 and so inherits all C++ features, such as inheritance and encapsulation. Table 3 lists OSML primitives. It introduces a component-based hierarchical worldview to the modeler. Components can include any number of processes or sub-components. The hierarchy of a model is constructed by connecting component pins, which are undirected and not used for data transfer between components. The programmer must register pins and sub-components of each component to it. A process represents the behavior of a part of the system and communicates with other processes through ports. The user must register ports to processes, and makes the processes sensitive to the input ports if necessary. Processes are split into two types of stackless and stackful. A stackful process has memory (state) and allows context switches through *osml_wait* routine. Ports and pins are connected through a pair of connector/interface. Interfaces actually describe a part of the design connectivity and are containers to store data. An interface equips OSML for creating new instances of its own for those ports that have been bound to it at parallel runtime (*CreateInstance* method). It also provides some methods for copying their data and sending to their processes through OSML kernel and applying them to the receipt processes' ports (e.g., the *osml_wire* interface implements such functionalities). All OSML data types must override and implement two methods of *serialize/deserialize*, which are used to convert data into a format suitable to be transferred over a network, for distributed execution.

The structure of a model is accessible to the user as a process graph, which is obtained by traversing model hierarchy through pin and ports during OSML elaboration phase. Therefore, the modelers can develop different partitioning algorithms to optimally assign processes to partitions by extending OSML primitives. Ports also guarantee the sender/receiver relationship for logical processes in PDES layer. OSML events provide a flexible mechanism for inter-process communication. Because the source and target of these events must also be present in PDES layer, they define an initiator-subscriber pattern. Subsequently, an OSML event has a number of initiators and subscribers that must be registered to it (*register_initiator* and *register_subscriber* methods). A process can suspend itself on an OSML event by calling the *osml_wait* method and return control to the kernel, and wake up when another process triggers it through *notify* method. Since the source of PDES messages must be present, all OSML primitives that generate system-level events whether directly or indirectly must specify the reference of issuer process (such as *notify*, *put* and timed-*wait* functions).

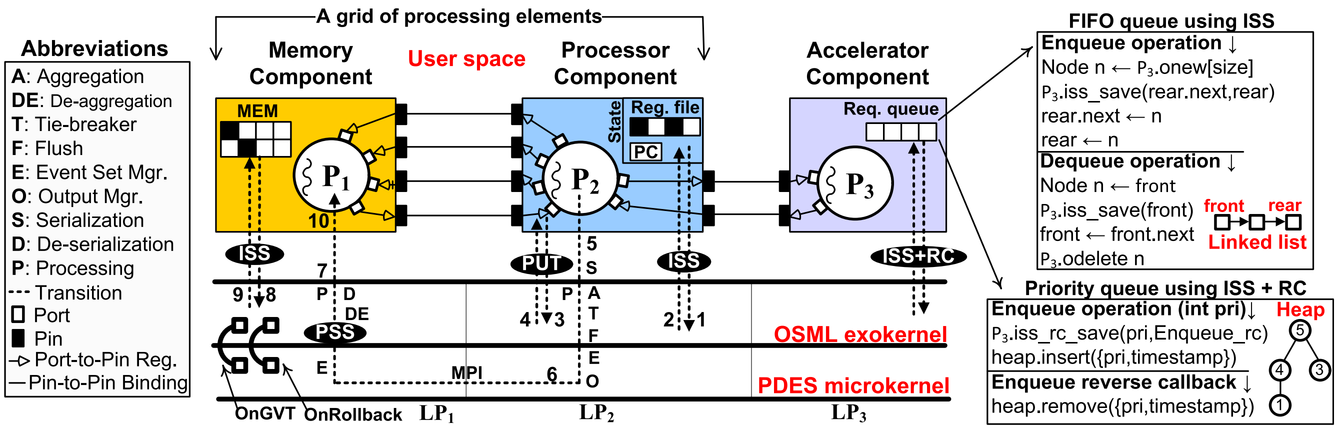


Fig. 3. A many-core architecture of PEs and its interaction with optimistic PDES engine by hybrid checkpointing. All processors are connected to a single accelerator. We assume that components reside on different physical cores for parallel simulation.

Each process has a set of state variables that must be registered by the user to the kernel based upon their characteristics. By default, all input ports of a process are considered as a part of its state and handled by PSS manager at runtime. We can explicitly define state variables of a process based on PSS scheme by extending *osml_process_state* class, where the programmer must implement methods to prepare a copy of the state, restore the state and free the memory allocated for the state (these are automatically done for primitive C++ and OSML data types). If modelers want to use hybrid checkpointing for a process, they can associate an instance of *osml_iss_state_manager* class to that process. ISS manager stores a range-based memory location from an address in memory relied on the length that the user specifies through *save_address* method to the kernel. OSML kernel uses this information to prepare a checkpoint or restore the state when a rollback occurs. RC callbacks can be registered with each process as well.

3.2 The OSML's PDES Systems Programming Model to Build Optimistic Hardware Components Using Explicit Hybrid Checkpointing

In this section, we examine how a user can benefit from OSML's PDES systems programming model with low-level access to optimistic dynamic memory and hybrid state saving for developing hardware models. The interactions of the model with PDES layer are also discussed. Fig. 3 portrays an overall view of a many-core architecture in some parts. Each processor component with its local memory is considered as a Processing Element (PE) arranged in a grid topology. All processors are connected to a single accelerator and communicate with each other through a router not shown. The accelerator serves incoming requests by two scheduling algorithms: FIFO and priority queue (where a PE identifier is regarded as a priority). Each processor has a state made up of one register file (as an array) and a program counter (PC). Let's first consider optimistic modeling of the memory component in detail. An abstract memory module has a collection of inputs, output and an internal memory region requested from the host RAM to load and store hardware words. Accordingly, this component cannot be modeled by CSS or PSS because of the internal memory as an immensely large state variable. To solve this problem, the internal memory is manually checkpointed by ISS using rewriting the original model. All other inputs are automatically checkpointed by PSS. The memory component modeled by optimistic PDES appears in Fig. 4. Fig. 4a is a synchronous memory module described in Verilog with a large internal array. Fig. 4b shows a snapshot of state variables recorded by CSS. Within 10 clock cycles, at least 10 GB of states is accumulated in the host RAM by

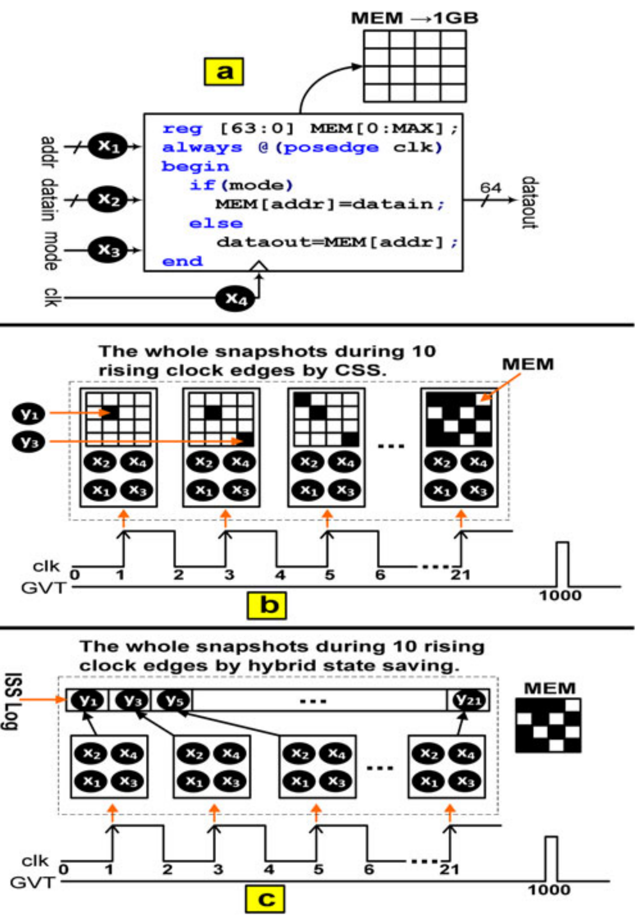

```
reg [63:0] MEM[0:MAX];
always @(posedge clk)
begin
   if(mode)
     MEM[addr]=datain;
   else
     dataout=MEM[addr];
end
```
a
MEM →1GB
addr
datain
mode
clk
X1
X2
X3
X4
64
dataout
b
The whole snapshots during 10 rising clock edges by CSS.
MEM
Y1
Y3
clk
GVT
0 1 2 3 4 5 6 21 1000
c
The whole snapshots during 10 rising clock edges by hybrid state saving.
ISS Log
Y1
Y3
Y5
Y21
MEM

Fig. 4. Modeling an optimistic memory component in OSML: (a) a typical Verilog implementation, (b) the state snapshots of the memory model captured by CSS, and (c) the state snapshots of the memory model recorded by HSS.

```
1 template <class T, int size>
2 class osml_sync_memory : public osml_component {
     // Pin and port definitions
 3    osml_pin<osml_wire<bool > > clk; ...
 4    osml_inport<osml_wire<bool > > _clk;...
 5    process_state pss_state; // PSS-based state definition
 6    T *array = nullptr; osml_process *p;
 7    osml_sync_memory(const std::string &name){
 8       register_pin(clk, _clk); ... // Register pins to ports
 9       // Register a stackful process and do its settings
10       p = register_process(process, true, &pss_state, name);
11       p->register_on_partitioning(on_partitioning);
12       p->register_port(_clk); ...
13       p->register_sensitivity(_clk, OSML_POS_EDGE);
14    }
      // This method is invoked after partitioning
15    static void on_partitioning(osml_process &owner){
16       auto myComp = (osml_sync_memory *)owner.get_component();
17       myComp->array = new T[size];
18    }
};
```

Fig. 5. Definition of an optimistic memory written in OSML.

OSML kernel, while only about 400 bytes of RAM for states are occupied by the hybrid protocol in Fig. 4c.

Fig. 5 depicts definition of the optimistic memory class implemented in OSML. A number of pins and input and output ports are declared for this component (lines 3 through 4). The PSS-based state includes the interfaces bound to input ports in addition to an integer indicating the number of stores (lines 5). Because OSML considers the contents of input ports as part of the process state, we do not lay them in *process_state* class. Pins are registered to the component inside the class constructor, and then a stackful process is defined. Each process has an associated ISS manager. Since an OSML model is fully instantiated on all processors in an HPC cluster (this is due to the MPI programming model used in the implementation of OSML kernel), the internal memory of our component can waste intrusively physical memory of the hosts. To cope with this issue, a callback can be registered with every process to specify which processor activates it so that the programmer can only once allocate internal objects of that process (line 11, and lines 15 through 18). The implementation of the memory process comes in Fig. 6. Loading a word from memory is normally done in lines 7 to 8; however, when writing to a memory location if its content is being updated, we must perform a checkpoint of this location by ISS manager before the real write to it (line 10). OSML processes allow context switches in a model. This is achieved through fibers in which the registers and stack of the process must be saved before switching. Thus, stackful process checkpointing is very costly in terms of pressure on the RAM of host nodes. To tackle this new dilemma, OSML proposes extremely lightweight stackful processes and explicitly gives the full control over context switches to the user, including, saving the current execution location and local variables of the process before suspension, and restoring the full process state after awakening. Since each process accesses to its own state by *get_state* method, there is no need for a heavily inexplicit context switch. This means we no longer need to allocate a fiber of execution for each process, but rather this is the user who decides which part of a process should be executed by adding control transfer instructions to the code. The *osml_process_state* class

```
1 static void process(osml_process &owner){
2   auto myComp = (osml_sync_memory *)owner.get_component();
3   auto myState = (process_state *)owner.get_state();
4   int addr = myComp->address->fetch();
5   switch(myState->get_label()) { case 0: goto L0; }
6   while(true) {
7     if(!myComp->_we->fetch() ...) // Read from memory
8       myComp->_data_out->put(myComp->array[addr], owner);
9     else if(myComp->_we->fetch() ...) { // Write to memory
        // We must checkpoint before performing the real write
10      owner.iss_save(&myComp->array[addr], sizeof(T));
        // Now, we perform the write to the memory
11       myComp->array[addr] = myComp->data_in->fetch();
12       myState->writes++;
13    }
14    osml_wait(owner); myState->set_label(0);return;L0:
15  }
}
```

Fig. 6. The process implementation of the memory component.

embraces two *get_lable* and *set_lable* methods that can be used to load and store the address of current program location when a suspension/resumption is issued by the *osml_wait* function. In Fig. 6, we implement this mechanism by using *switch* and *goto* commands.

Now, we are interested in Fig. 3 to examine a transaction issued by the processor for storing a word in memory and sending a job to the accelerator to be computed. P_2 first saves its state (updated registers and PC) by issuing a system call to OSML kernel using the ISS manager. Then, P_2 delivers the contents to the interfaces bound to P_1 by PUT system call, which are stored by OSML in a list as serialized prior to real delivery. After *osml_wait* function is invoked by the model, control of execution is transferred from userspace to OSML exokernel. OSML kernel implements parallel hardware execution semantics. First, the messages stored in step 3 are turned into an aggregate event. Next, OSML tie-breaker, to deal with simultaneous events, attaches a unique timestamp to the event. Extra information such as sender and receipt identifiers is added to the event as well, and the message is offloaded to PDES microkernel via PAL. PDES kernel conveys this event to LP_1 through MPI—output manager also maintains a copy of the message to send anti-messages later.

On the LP_1 side, PDES kernel fetches and executes the events one by one. The events are delivered by invoking a callback pre-registered to OSML kernel. OSML kernel initially gives control of execution to PSS manager in order to check if a PSS checkpoint is needed, which is done by calling the *clone* routine of state variables and input port-bound interfaces. The received event is then de-aggregated. For instance, messages accompanying interface-related data are first deserialized by invoking the methods written in userspace interfaces (like wires) and then committed to the interface. During this operation, OSML kernel processes the messages one by one based on OSML execution semantics to see whether P_1 is triggered or not. If triggered, control of execution is returned to the point in userspace where P_1 has already suspended itself by calling *osml_wait*. At this moment, the user code is executed as seen in Fig. 6. When the processor sends a request to the accelerator, actions similar to steps 1 to 10 are taken in LP_2 and LP_3. The accelerator puts requests in a queue. Since the queue uses an internal data structure much more complex than the memory component, we cannot merely benefit from the described mechanism for arrays using ISS. We need optimistically

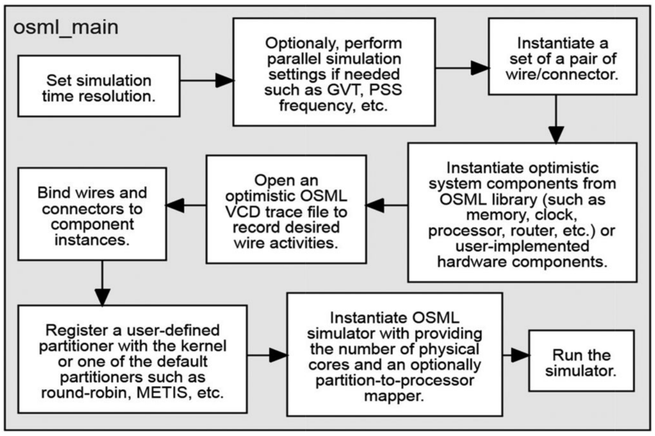

osml_main
Set simulation time resolution.
Optionaly, perform parallel simulation settings if needed such as GVT, PSS frequency, etc.
Instantiate a set of a pair of wire/connector.
Instantiate optimistic system components from OSML library (such as memory, clock, processor, router, etc.) or user-implemented hardware components.
Open an optimistic OSML VCD trace file to record desired wire activities.
Bind wires and connectors to component instances.
Register a user-defined partitioner with the kernel or one of the default partitioners such as round-robin, METIS, etc.
Instantiate OSML simulator with providing the number of physical cores and an optionally partition-to-processor mapper.
Run the simulator.

Fig. 7. Typical steps needed to prepare a model to run in *osml_main* function.

synchronized dynamic memory to implement a linked-list-based FIFO queue. Fig. 3 shows some fragments of a modified FIFO algorithm for reversible *enqueue* and *dequeue* operations. In enqueue, we allocate an optimistic memory block by calling the *onew* operator from P_3. OSML keeps one heap associated with each process for memory allocations. The optimistic memory manager returns blocks of the heap marked by *odelete* operator to the free list when an *OnGVT-Callback* occurs. Also, when a rollback is performed, all memory blocks are freed to the point of rollbacked event. After allocating the node *n*, addresses of the two *rear* and *rear.next* nodes are recorded using ISS, and the remaining operations are followed routinely. To implement a priority queue through an array-based heap sort, we cannot use ISS alone, because many ISS requests must be made when swapping array elements, which is not efficient. Instead, we make careful use of a combination of both ISS and RC techniques. Priority values are saved, and we associate a reverse computation callback with P_3. If P_3 is rollbacked, the reverse callback is concurrently executed in conjunction with ISS. This callback performs the inverse function of insertion, i.e., removal. A coalescence of the priority and current P_3 simulation time (which has three flags, see Section 4) is used to break potential ties within the queue elements in parallel execution. Following the development of optimistic components, the programmer connects them together in a function called *osml_main*. The typical steps come in Fig. 7.

3.3 The OSML's PDES Application Programming Model Using Transparent Hybrid Checkpointing

The PDES systems programming model introduced in the previous section provides low-level access to state management capabilities of a process for optimal implementation. Working with this style of system modeling may be a bit challenging for users unfamiliar with PDES in most cases. To solve this problem, OSML proposes the *PDES application programming model*. The key idea is that the user must comply with a standard way to define state variables so that OSML kernel can manage process states transparently. In this model, each stackful process must define its state as an optimistic thread-local storage (TLS). This TLS is actually a global memory visible to a process, and all used objects and called functions inside it. A modeler cannot use primitive C++ data types or data structures built by them inside a TLS. Instead, this model implements all the C++ data types as reversible (like *rint*, *rstring*, etc.). They can be handled as PSS or ISS/RC based on OSML kernel statistics or user adjustment. Additionally, relied on the rules studied in Fig. 3, OSML presents a wide variety of reversible data structures using the PDES systems programming model, including, multi-dimensional arrays, lists, queues, trees and graphs. Also, OSML supplies an immensely lightweight fiber facility to hide the explicitly emulated fiber in Fig. 6. The central idea here is to use the instruction pointer (IP) register (such as EIP and RIP registers in x86 architecture) for performing context switches, because TLS is available to all execution paths of a process. The optimistic fiber checkpoints the IP register into the PSS state region of a stackful process. When that process should be woken up, OSML kernel will jump to the address of IP using the *goto* command. Accordingly, the user has no knowledge of fiber details. If the modeler wants to build a new reversible data container, he can extend the PDES systems programming interfaces.

4 The Optimistic OSML Simulation Exokernel

The system-level description written in OSML language must finally be simulated by a kernel that implements OSML's parallel execution semantics. This kernel must be aware of the LP pattern and exposes the underlying parallelism to the modeler by hiding the details of optimistic synchronization. Additionally, it must support the language extensions introduced in Section 3. The life cycle of the distributed OSML exokernel is made up of several tasks, including, (i) running sequential simulation on a single processor to obtain some information for partitioning and HSS techniques used at parallel runtime, (ii) distributing the model on processor nodes, performing OSML elaboration phase to flatten the model hierarchy, discover processes and build HWG, (iii) executing the partitioning phase on HWG to get a process-to-processor mapping, and invoking user-defined routines to allocate resources during partitioning, (iv) allocating internal data structures per process (e.g., instantiating interfaces, the state variables of OSML execution semantics, optimistic TLS and heap), (v) setting up the rollback and GVT callbacks with PDES microkernel, (vi) running the simulation, and (vii) termination. Fig. 8 shows the UML class diagram of the optimistic OSML kernel. The OSML kernel provides an interface to construct logical processes through PAL. The definition of *LogicalProcess* class is divided into two parts. The first part is a set of methods that PDES kernel provides to that LP. These methods are presented by PDES kernel for communication to the LPs (*SendEvent* and *GetNextEvent*), querying the kernel for information (*GetSimulationTime*) and accessing to its state (*GetState*). The second part includes those methods that must be overridden. *Initialize* and *Finalize* methods are invoked before simulation starts on each LP and at the end of simulation respectively, which allow LPs to perform initialization and clean-up. *ExecuteProcess* method is invoked by the PDES kernel when each LP has at least one event to process. Time is expressed using the *PdesTime* class to deal with simultaneous events. These classes provide merely a low-level model definition for simulation, and the OSML

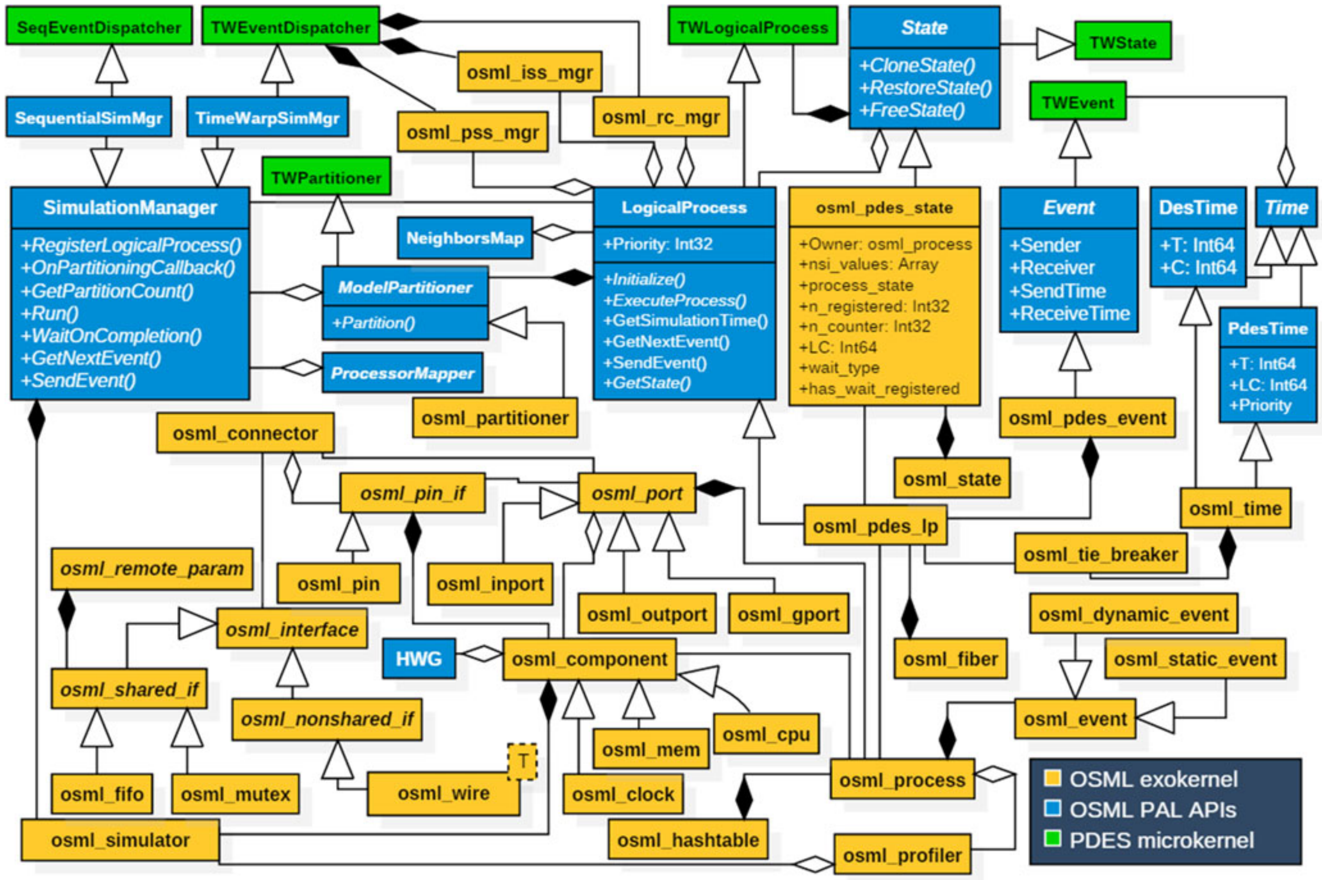


Fig. 8. UML class diagram of the optimistic OSML simulation kernel.

kernel implements control structures of the real simulation, which is presented through *SimulationManager* class. It must perform any kind of setup tasks and preparing the simulation environment through PDES kernel. For partitioning effectively, the kernel requires that LPs store their neighbors in a map through *ModelPartitioner* class. WARPED optimistic simulation kernel [35], which only implements PSS, was extended to conform to the PAL wrapper APIs. We modified WARPED to support a minimum set of PDES services and made it as a PDES microkernel. OSML simulation manager makes use of a hybrid state saving for each LP as proposed in Fig. 9. When a straggler message is received, rollback is performed in three mixed steps. First, ISS manager restores the before-images of state variables by a reverse traversal from the end of ISS log to the event greater than the last checkpointed event and ahead of the straggler message (each LP has a callback named *OnRollback*, which is invoked upon receiving a straggler by ISS manager). Second, RC manager invokes the reverse event handlers upon completion of every ISS operation to backtrack the computation. Third, PSS manager is activated and performs coast forward in which the LP is executed up to the rollback point to generate intermediate process states. We should disable ISS manager, to save ISS-based updated states throughout the coast forward, and RC event handlers; because it is unnecessary to mess up the ISS log, with those states that are already there, and forward computation. Fossil collection also consists of two parts where PSS and ISS managers release the memory up to GVT point. OSML kernel works in three main PDES-aware phases: distributed initialization, optimistic simulation and distributed termination. OSML elaboration phase prepares a model to run on a PDES environment. It must transform high-level structures of a hierarchical, modular model into a low-level representation that is only understood by LP-

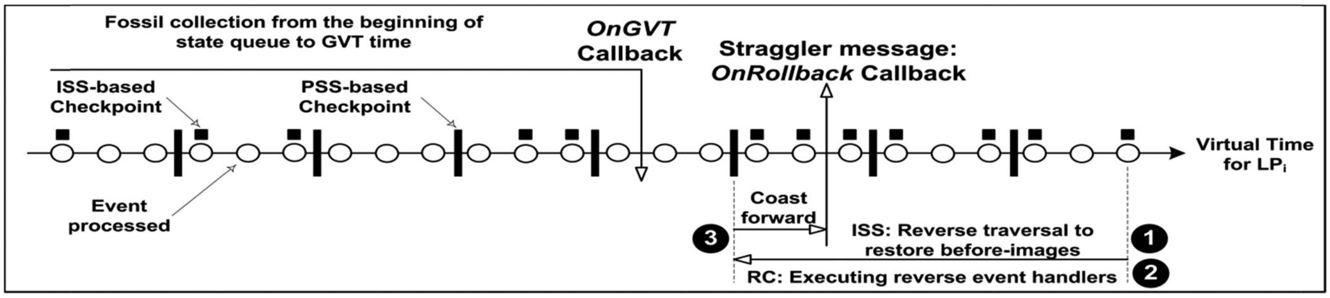

Fossil collection from the beginning of state queue to GVT time
OnGVT Callback
Straggler message: OnRollback Callback
ISS-based Checkpoint
PSS-based Checkpoint
Event processed
Virtual Time for LPi
Coast forward
3
ISS: Reverse traversal to restore before-images
1
2
RC: Executing reverse event handlers

Fig. 9. Hybrid state saving in OSML exokernel.

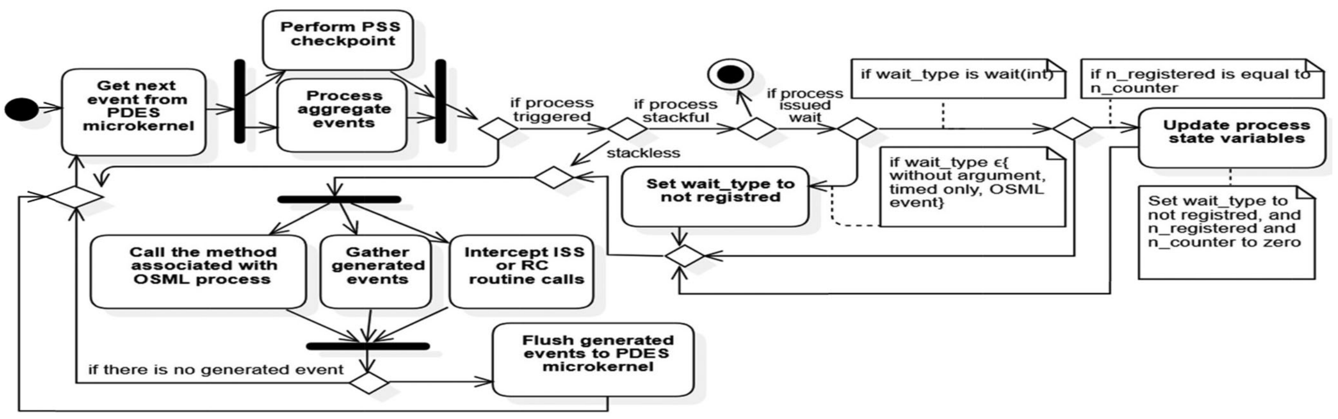

Get next event from PDES microkernel
Perform PSS checkpoint
Process aggregate events
if process triggered
if process stackful
if process issued wait
stackless
if wait_type is wait(int)
if n_registered is equal to n_counter
Update process state variables
Set wait_type to not registred
if wait_type ϵ{ without argument, timed only, OSML event}
Set wait_type to not registred, and n_registered and n_counter to zero
Call the method associated with OSML process
Gather generated events
Intercept ISS or RC routine calls
Flush generated events to PDES microkernel
if there is no generated event

Fig. 10. The main algorithm of parallel OSML execution semantics when an event is received from PDES microkernel for an LP.

based event-scheduling world view. This is performed by traversing the model hierarchy, which is made by pin-to-pin binding and port-to-pin binding through connectors. The resultant output of this stage is a logical process network (LPN). Finally, a new instance of a non-shared interface (NSI) by *CreateInstance* method is associated with each port.

After preparing the internal data structures of the OSML kernel to build LPN in elaboration phase, the parallel simulation begins. Since OSML kernel is a decentralized scheduler due to the inherent requirement of optimistic PDES, OSML semantics is accomplished by each LP that locally manages a single system-level process. The lifespan of an individual LP consists of multiple concurrent stages: (i) initialization, (ii) consuming an event, extracting messages from the event, decision making on the type of the messages, applying changes to input ports, updating LP (simulator) state variables to satisfy OSML execution semantics; invoking the process, managing PSS/ISS/RC routine calls for state management and collecting generated events during process execution; serializing generated events and aggregating them into a contiguous event, and flushing the event to PDES microkernel, (iii) executing the semantics of PSS/ISS/RC managers upon being invoked rollback and GVT callbacks from PDES microkernel, (iv) and cleanup.

When the processes run for the first time at initialization stage, they generate events which are captured by OSML and sent. These initial events cause the kernel to execute the *ExecuteProcess* method of other LPs. Fig. 10 shows the steps taken by this method. Each process retrieves its next event to process by the *GetNextEvent* method. Then, PSS state manager is executed. If the process is triggered by processing the event through the *ProcessAggregateEvents* method, different decisions are made. If the process is stackful and has already suspended itself by the function *osml_wait(expr)*, it is activated by invoking its function. For *osml_wait(int n)* mode, an additional step is required to check whether the number of events occurred on that stackful process has reached the value *n* or not. If the process is stackless, its function is called directly. While a process is executing, the kernel gathers generated events and also intercepts ISS or RC routine calls. An aggregation mechanism is used to reduce traffic congestion of the transmitted events. If a process schedules several events for another process (e.g., issuing multiple *put* or *notify* operations), these events can be sent as a single aggregate message. For this purpose, when events are issued in processes, they are placed into a list. Upon returning the control of execution from the process to OSML kernel, these events are flushed and sent. The structure of an aggregate message is shown in Fig. 11. It is made up of two parts. The first part contains the message information sent. The second part has the actual aggregate events. Current OSML kernel defines three types of events, (i) NSI events that are created by committing non-shared interfaces, (ii) those events that originate from *osml_wait(T)* (in this case, an LP schedules a timed event for itself), and (iii) OSML events that are issued by *notify(expr)*. Each NSI event includes port number of the destination process and the contents committed to its interface, where the data length is obtained from the interface's *ValueSize* method in target, and therefore it is not required to be specified in the message header. Fig. 12 depicts the flow of *Flush* function for NSI events. An array-based bounded hash table, to increase locality of reference, is used to store events sent to adjacent LPs. The key of each node in the hash Table 1s the destination process identifier. Because OSML uses a unique event ordering mechanism, the order of scheduled events is preserved in this table, and events are sent in the order that appear in the code. The time of events must be prepared by the tie-breaker before dispatch in order to deal with simultaneous events. Fig. 13 demonstrates the algorithm of processing the aggregated events. All events are extracted from the aggregate message and are handled based on the type of events. For example, the port interface of the receipt process is determined, and the event data is copied to the interface for an NSI event. OSML events are divided into three groups: single-event, and-event list and or-event list. For a single-event type, it is checked to see if the event identifier is equal to the OSML event already registered by *osml_wait(e)*, then the process must be triggered. A similar technique is used for event lists, although we must examine a vector of events instead.

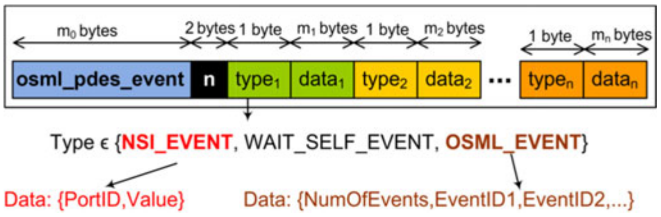


Fig. 11. The header format of an aggregate OSML message.

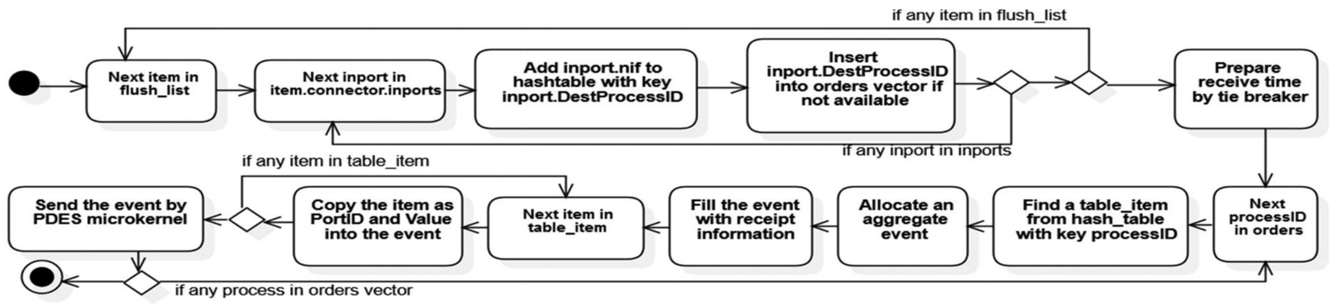

Next item in flush_list
Next inport in item.connector.inports
Add inport.nif to hashtable with key inport.DestProcessID
Insert inport.DestProcessID into orders vector if not available
if any item in flush_list
if any inport in inports
Prepare receive time by tie breaker
Next processID in orders
Find a table_item from hash_table with key processID
Allocate an aggregate event
Fill the event with receipt information
Next item in table_item
Copy the item as PortID and Value into the event
if any item in table_item
Send the event by PDES microkernel
if any process in orders vector

Fig. 12. The algorithm of scheduling and flushing an aggregate event, generated by the process execution, down the PDES microkernel.

Two or more events that are scheduled to occur at the same point in time are called *simultaneous events*. They are seen in SLDL simulations frequently. A number of tie-breaking rules are necessary to decide which simultaneous event should be executed first for a deterministic simulation. Particularly, optimistic protocols allow the time to creep into the past and make the problem more critical. In OSML kernel, time is represented by the triplet $t = \langle T, LC, Pr \rangle$ in parallel simulation, where tie-breaking can be performed in a straightforwardly decentralized fashion. T is the occurrence time of an event. Pr is the LP priority scheduling the event. LC is calculated based on Lamport algorithm [41] in which each LP before sending an event increments a counter associated with the LP. When an LP sends an event, this value is attached to the message. On receiving an event, the receipt LP updates its counter to the maximum of its current counter and the received LC, and then increments it by 1. Logical clocks only establish the happened-before relations. Two events that are processing by an LP may have equal LCs received from adjacent LPs. For example, when two sender LPs do not communicate with each other due to data independence, their logical clocks are not synchronized. To break the ties in this state, the third flag Pr is added to this definition. Sorting is preformed based on the following rule:

$$t_1 < t_2 \Leftrightarrow T_1 < T_2 \vee (T_1 = T_2 \wedge LC_1 < LC_2) \vee (T_1 = T_2 \wedge LC_1 = LC_2 \wedge Pr_1 < Pr_2).$$

5 TROODON'S TOOL FLOW

Fig. 14 shows Troodon's tool flow. Designers write their model's blueprint using one of the hardware description languages such as OSML, SystemC and UML. They are connected to the Cloud-based Troodon tool through a browser and do their time-consuming simulation tasks. Models are graphically displayed to the users, and they can view

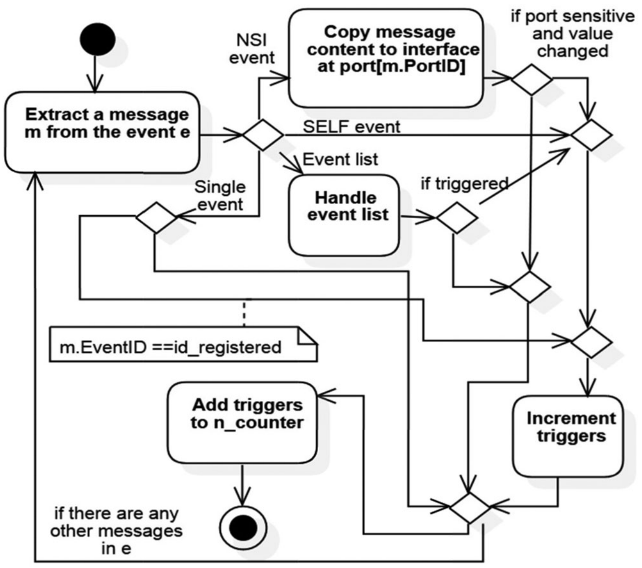

Extract a message m from the event e
NSI event
Copy message content to interface at port[m.PortID]
if port sensitive and value changed
SELF event
Event list
Single event
Handle event list
if triggered
m.EventID ==id_registered
Add triggers to n_counter
Increment triggers
if there are any other messages in e

Fig. 13. The algorithm of processing an aggregate event.

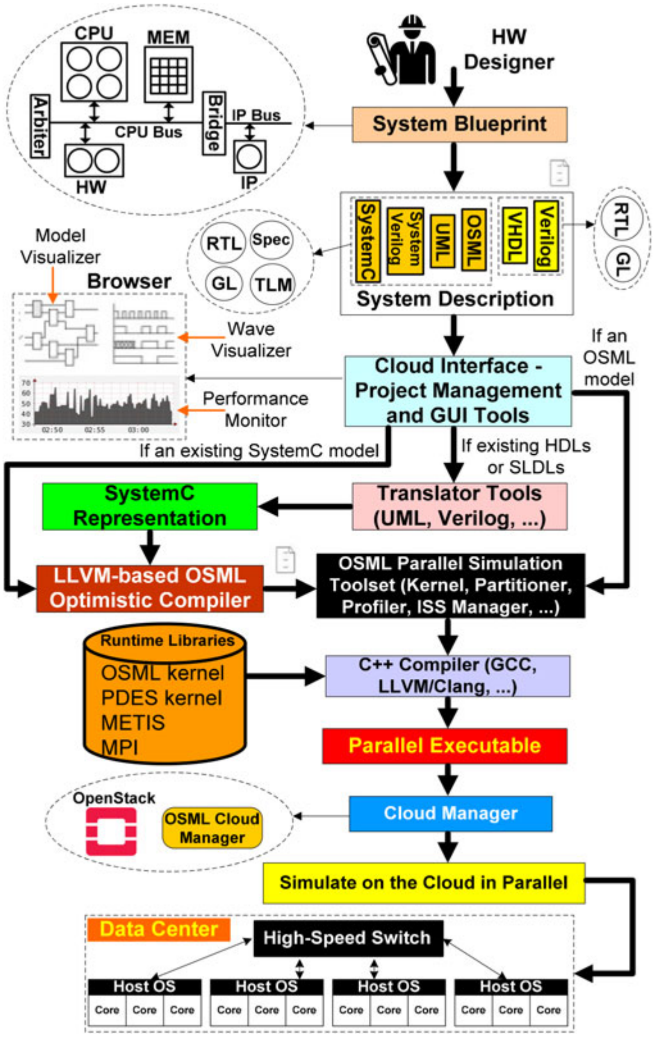
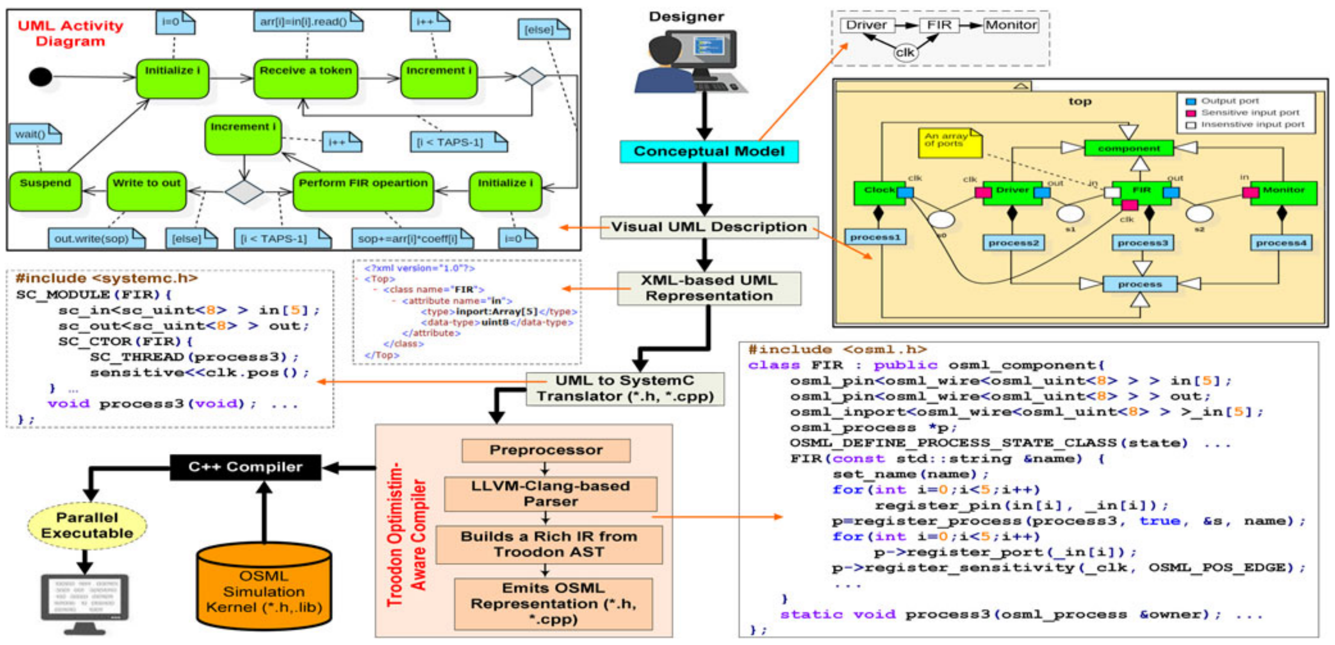


Fig. 14. The design flow of Troodon.



```
#include <systemc.h>
SC_MODULE(FIR){
    sc_in<sc_uint<8> > in[5];
    sc_out<sc_uint<8> > out;
    SC_CTOR(FIR){
        SC_THREAD(process3);
        sensitive<<clk.pos();
  } ...
  void process3(void); ...
};
```

```
<?xml version="1.0"?>
- <Top>
  - <class name="FIR">
    - <attribute name="in">
        <type>inport:Array[5]</type>
        <data-type>uint8</data-type>
      </attribute>
    </class>
  </Top>
```

```
#include <osml.h>
class FIR : public osml_component{
    osml_pin<osml_wire<osml_uint<8> > > in[5];
    osml_pin<osml_wire<osml_uint<8> > > out;
    osml_inport<osml_wire<osml_uint<8> > >_in[5];
    osml_process *p;
    OSML_DEFINE_PROCESS_STATE_CLASS(state) ...
    FIR(const std::string &name) {
        set_name(name);
        for(int i=0;i<5;i++)
            register_pin(in[i], _in[i]);
        p=register_process(process3, true, &s, name);
        for(int i=0;i<5;i++)
            p->register_port(_in[i]);
        p->register_sensitivity(_clk, OSML_POS_EDGE);
        ...
    }
   static void process3(osml_process &owner); ...
};
```

Fig. 15. UML-based design flow for optimistic simulation in Troodon.

simulation wave forms and performance monitoring plots inside a single window in real time. They can edit their models by Graphical User Interface (GUI) components in addition to uploading their files. Upon submission of a design to the Cloud, Troodon automatically carries out the remaining tasks for parallel simulation. SystemC is considered as a high-level intermediate language in this flow. The models specified in UML, VHDL, Verilog and SystemVerilog are first converted into equivalent SystemC codes by source-to-source translators. If the model is entirely written in OSML, no extra work is done on it at this stage. Then, an optimistic compiler developed atop LLVM/Clang [42], which meets the requirements of OSML semantics, is invoked to transform the SystemC model into OSML. It is composed of a frontend and a backend: the former constructs an abstract syntax tree (AST), and the latter performs the AST mapping to OSML codes. Because none of the mentioned languages, including SystemC, can directly be mapped into OSML, multiple transformations are made on the input SystemC codes. Section 3 highlighted some OSML functionalities that are not available in traditional languages. OSML codes are compiled by a conventional C++ compiler and linked with parallel runtime libraries (e.g., OSML kernel), and a parallel simulation executable is generated. OSML cloud manager runs on the IaaS-based OpenStack platform [43], which distributes PDES executive across the data center and does monitoring tasks.

6 Case Studies

We study a number of experiments to evaluate the capability and performance of Troodon. The tests were carried out on an HPC cluster of 17 nodes managed by OpenStack. Each machine had 12 cores operating at 3.33 GHz, 12 GB RAM and 12 MB L3 cache. The machines were connected to a 40 GB/s fiber-optic network. Each node ran CentOS 7 Linux with a kernel 3.10.0-327. OSML speedups are reported in comparison with Accellera sequential SystemC reference simulator. All tested models were compiled by GCC with the O3 flag. MVAPICH2.3 was installed in the cluster and all the models on compute nodes had access to test bench files via NFS. The hardware components used in all benchmarks are written by the rules mentioned in Section 3.

6.1 UML-Based Hardware Modeling and Simulation in Troodon

As the first example, we look at how Troodon helps the designer automatically prepare a model from a high level of abstraction for optimistic simulation. Fig. 15 shows Troodon's complete design flow to transform a model described in Unified Modeling Language (UML) and generate a parallel executable file. Troodon benefits from the utilities of StarUML tool to translate a model specified in UML to a kind of SystemC coding style which is OSML compliant. To enable the modeling and specification of a design by supporting translatable extensions to OSML language, a UML profile was developed for SystemC in StarUML. Each element of the SystemC model has a relationship with UML metaclasses. To translate a model written on top of the Troodon's standard UML profile, we exploit the ability of StarUML to generate code using its templates. The developed translator includes all the code generation rules that are written in JavaScript language and uses the available functions in StarUML to access the elements and symbols in it. Now, let's assume the user intends to specify and verify an FIR filter. He must first draw the static structure of his design through UML Class Diagram by using Troodon's schematic tools. The model's subsystems must be derived from *component* class and can be connected through UML ports. The model hierarchy is constructed by binding UML interfaces to the ports at the UML Model *top*. Each component can define different processes. The behavior of each process is expressed by using the UML Activity Diagram. Fig. 15 depicts the activity diagram of third process for FIR component. The details about each step of the flow, namely the operations that must be performed, are precisely defined by UML Note. Also, if the modeler uses complex data structures such as queues and trees, he must reuse from reversible ADT counterparts of the UML profile. The next

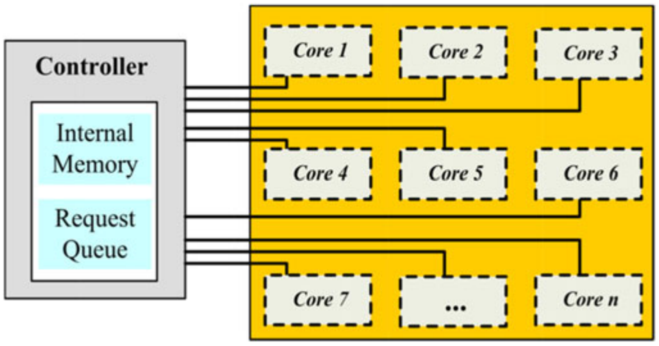


Fig. 16. Architecture of the many-core accelerator.

steps in Troodon's design flow are done automatically. The UML structure of a hardware model is stored and retrieved as an XML file to be processed by Troodon. This XML representation is turned into a C++-based SystemC description by a textual translator. In fact, SystemC is treated as a common high-level intermediate language for OSML among existing languages such as UML and Verilog. Because OSML has been developed atop the C++ standard, the transformation of SystemC codes to OSML can be straightforward. However, a new critical problem emerges; because C++ is a complex language, requires developing a complete C++ frontend that must be able to understand SystemC structural semantics as well. On the contrary, since SystemC is a sequential simulation language, has not been founded upon the notion of logical processes and has no way to define process states; this frontend gets more complicated. Troodon implements an optimistic PDES-aware compiler infrastructure to map the codes of a SystemC model into OSML. Compilation is performed in three stages:

Preprocessing Phase. Because SystemC models uses C macros to define a part of the model structure and Troodon frontend can only process the C++ language, the program code is first preprocessed and expanded with its libraries. Since the final code is a large file, which includes the declarations of the model code and SystemC library, it is analyzed in an additional step using its metadata, and the original model code is reconstructed without macros by removing extra parts such as SystemC library and STL.

Compiler Analysis Phase (Frontend). Troodon primarily needs to analyze the structure of a SystemC model and extract it as an IR. Since Clang is only able to parse C++ and does not understand SystemC structures, Troodon

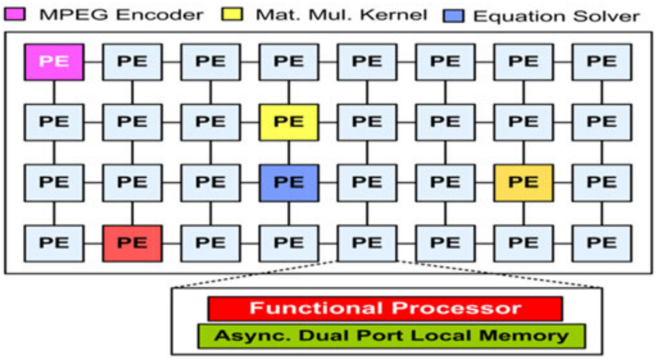


Fig. 17. Architecture of the many-core BFM platform.

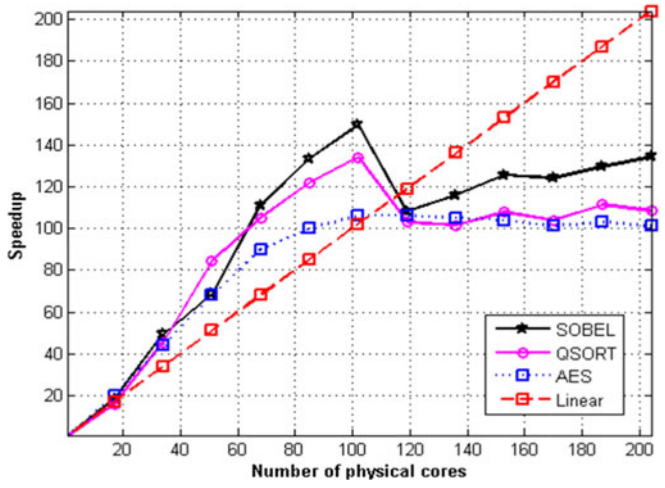


Fig. 18. Speedups of the different system level models.

frontend was implemented on top of the Clang frontend. For each SystemC model file, Troodon visits all AST nodes using pre-order depth-first search (DFS). This procedure is done with syntactic knowledge of a SystemC model. Furthermore, this frontend performs a comprehensive analysis on a SystemC model in order to discover necessary information unavailable inside the original model—for example, to determine the ports of an OSML process accurately—and construct an OSML equivalent model.

Source-to-Source Transformation Phase (Backend). After a rich IR is constructed in phase 2, SystemC transformation phase into OSML begins, which is the most complex part of the Troodon compiler. For this transformation, Troodon utilizes the *Rewrite* library of the Clang frontend. Only those parts of the input source file that require applying transformations are rewritten. An OSML model embodies far more information than its SystemC representation that is compulsory for parallel simulation by the OSML simulation kernel. Troodon extracts the whole state variables of a SystemC design for optimistic execution by analyzing module members, global variables, and local variables inside threads. The backend puts these state variables down separately into a state class for each process. Finally, OSML codes, with OSML exokernel, PDES microkernel and MPI library, are converted to a parallel executable by a traditional C++ compiler.

6.2 Performance Evaluation

Figs. 16 and 17 show high-level architecture of the developed benchmarks. In the many-core accelerator, controller broadcast tasks to different cores for processing. Two sets of experiments are performed by this accelerator. In the first set, each core implements a behavioral model of an algorithm, including, Sobel filter, quick sort for integer lists, and AES cryptography. In the second set, each core contains a cycle-accurate RTL model of a pipeline triple DES IP core. Fig. 17 implements a buss functional model (BFM) in which each PE executes a computational kernel. Each PE has a local memory connected to a functional processor via wires. The estimated time of each processor is modeled by inserting timed-waits into the code. All the models are partitioned directly using METIS [44] except for the BFM model and mapped to physical processors through MPI. In the BFM model, due to heavy interactions of each processor with its local memory, a

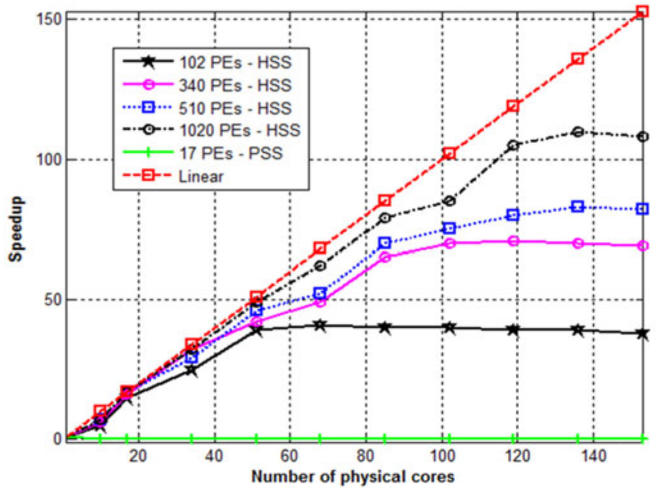


Fig. 19. Speedups of the BFM model with respect to the number of PEs.

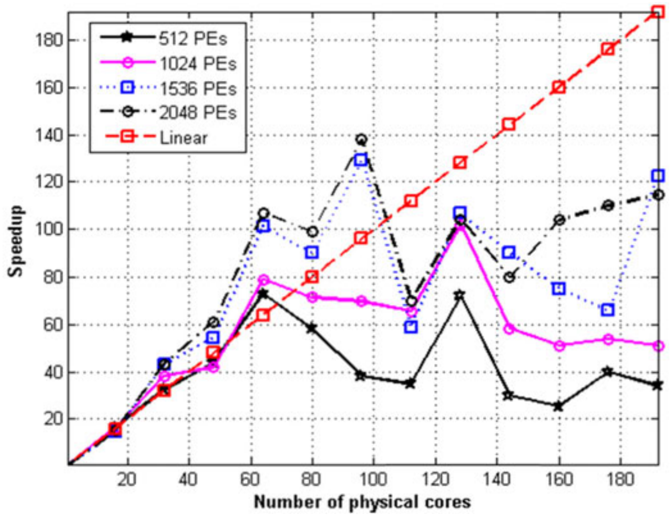


Fig. 20. Speedups of the many-core triple DES RTL model with respect to the number of PEs.

customized partitioner was developed. It first assigns all processes of each PE to a single partition, and then the new mapped model is re-partitioned by METIS for load balancing. In fact, the resultant processes from each PE are considered as a super LP and the final partitioning is done reliant on it.

Fig. 18 illustrates simulation results of the behavioral models for three algorithms. As seen, these models show super-linear speedups. The behavior can be attributed to large processor caches, in which the model fits into them in parallel simulation unlike sequential simulation on a single core. The speedup is almost constant with a large increase in the number of processors. The reason is related to the reduction in the number of processes that can be assigned to processors. Because every processor does not have enough workload to run, it will spend much of its time for inter-processor communication; of course, this can lead to load imbalances. Fig. 19 shows the results of the BFM model with respect to the different number of PEs, where the increase in the model complexity results in higher simulation speedups. As seen, the speedup behavior is close to linear for larger models; in fact, super-linear speedup is not observed. This is because the memory module does not perform any useful computation, and PEs use it solely for loading and storing memory words.

For PSS mode, there is no speedup gain as the simulation is out of memory and abnormally terminated even with 17 PEs. To demonstrate the impact of PSS on performance, we performed a simple test to copy 1 GB buffer into another by the *memcpy* C function on one of the HPC nodes. It took 0.3 second until completion. If we assume that the memory component receives 100-milion clock events and PSS frequency is 100, state saving is activated at least one million times and the simulation of a single memory process takes 83 hours to complete! Fig. 20 demonstrates the results of the RTL model with respect to the number of PEs. This model has much higher processes than the previous models due to low level of abstraction and its fine granularity. Super-linear speedups are observed by increasing the model complexity. Unlike previous tests, this model has fluctuations in speedups specially by increasing the number of processors. This is because of the small amount of RAM installed per node on the cluster. The DES model involves a highly irregular topology surrounded by many feedback loops. Accordingly, the METIS partitioner that treats the resultant directed LP graph as an undirected one cannot perform uniform mapping for large processor counts, and then it can result in load imbalances, especially when activity is high around the feedback loops. We observed that the fine granularity of the RTL model and the load imbalance caused by the partitioner for large processor counts (and in a number of specific counts) leads to increased rollbacks and also lack of enough memory space (because the simulation is not time-bound, but input-data-bound; therefore, LPs advances their virtual times too much fast giving rise to many extra primary and secondary rollbacks), which hurt performance significantly.

Table 4 shows sequential and parallel execution times, and simulation efficiency for some of the bench-marks.

TABLE 4
Some Runtime Statistics for Sequential and Parallel Simulations

Model	Sequential SystemC (sec)	Sequential OSML (sec)	Parallel Execution Time (PET) (sec) and Simulation Efficiency (%)									
			34 cores		64 cores		80 cores		96 cores		112 cores	
			PET	Efficiency	PET	Efficiency	PET	Efficiency	PET	Efficiency	PET	Efficiency
SOBEL	1147	1468	23	98.1	10.7	97.8	9.1	97.9	7.9	97.4	8.6	97.5
AES	519	652	11.8	99.8	6.1	99.62	5.3	99.35	4.9	99.29	4.8	97.52
BFM (1020)	6375	6249	187	97.16	108	98.22	83	95.49	76	95.66	61	93.82
DES (512)	555	531	17.3	97.2	7.6	95.6	9.5	82.39	14.4	79.8	16	72.35

Simulation efficiency is defined by decrementing one by the ratio of rollbacked events to the total processed events, and the result multiplied by one hundred. It is worth noting that for large-scale models like SOBEL and AES, sequential OSML execution time is slightly larger than sequential SystemC. This is because OSML makes use of the sequential WARPED simulation manager that instead emulates the execution of the LP-based model as an event list-based implementation. Since the sequential OSML simulator maintains additional data structures for the LP pattern, it cannot benefit from cache locality on a single core. Therefore, the reported speedups in Figs. 18 through 20 are the worst-case scenarios calculated with respect to the sequential SystemC runs. Moreover, the low efficiency of the DES RTL model in a large number of processors accompanies the analysis for the oscillations given in the previous paragraph.

7 Conclusion and Future Work

In this paper, we proposed a new optimistic PDES language called OSML that inherits all the features of existing hardware languages, and discussed the details of its distributed simulation kernel. Because optimistic PDES has significant potential to harness maximum degree of parallelism from DES programs on many-core microprocessors and supercomputers, as well as current-day hardware models are intractable to be simulated by optimistic synchronization, a hybrid checkpointing scheme was developed for OSML. To address parallelization challenges of the existing HDLs and SLDLs, we presented a unified CAD tool, referred to as Troodon, to automatically accelerate them on Cloud-based HPC clusters. The introduced architecture allows researchers to easily explore different asynchronous PDES algorithms for distributed OSML simulation. On this basis, we plan to investigate various simulation optimizations for SLDLs, including dynamic load balancing and adaptive PDES protocols. In addition, further research will be devoted to build comprehensive simulation models specific to other domains like computer networks in OSML.

Acknowledgments

The authors would like to thank gratefully the anonymous reviewers for their helpful comments that greatly improved the clarity of this paper. They are also grateful to Miss. Elham Tadayon and Miss. Fatemeh Motefaker for their assistance to develop a UML-to-SystemC translator for Troodon.

References

[1] I. C. Society, "IEEE standard 1666-2011 for standard SystemC language reference manual," ed: IEEE, 2011. [Online]. Available: https://standards.ieee.org/findstds/standard/1666-2011.html

[2] I. C. Society, "ANSI/IEEE Standard 1800-2012— IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," ed: IEEE, 2012. [Online]. Available: https://standards.ieee.org/findstds/standard/1800-2012.html

[3] R. M. Fujimoto, "Research challenges in parallel and distributed simulation," *ACM Trans. Model. Comput. Simul.*, vol. 26, 2016, Art. no. 22.

[4] S. Jafer, Q. Liu, and G. Wainer, "Synchronization methods in parallel and distributed discrete-event simulation," *Simul. Model. Practice Theory*, vol. 30, pp. 54–73, 2013.

[5] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Languages Syst.*, vol. 7, pp. 404–425, 1985.

[6] P. D. Barnes Jr, C. D. Carothers, D. R. Jefferson, and J. M. LaPre, "Warp speed: Executing time warp on 1,966,080 cores," in *Proc. 1st ACM SIGSIM Conf. Principles Adv. Discrete Simul.*, 2013, pp. 327–336.

[7] A. Poshtkohi, M. B. Ghaznavi-Ghoushchi, and K. Saghafi, "The parvicursor infrastructure to facilitate the design of grid and cloud computing systems," *Comput.*, vol. 99, pp. 979–1006, Oct. 2017.

[8] N. Sehgal, J. M. Acken, and S. Sohoni, "Is the EDA industry ready for cloud computing?," *IETE Tech. Rev.*, vol. 33, pp. 345–356, 2016.

[9] J. Shalf, D. Quinlan, and C. Janssen, "Rethinking hardware-software codesign for exascale systems," *Comput.*, vol. 44, pp. 22–30, 2011.

[10] R. Dömer, A. Gerstlauer, and D. Gajski, "SpecC language reference manual," in *SpecC Technology Open Consortium*, vol. 2, ed., 2002. [Online]. Available: http://users.ece.utexas.edu/~gerstl/publications/SpecC_LRM_20.pdf

[11] L. Zhu, G. Chen, B. K. Szymanski, C. Tropper, and T. Zhang, "Parallel logic simulation of million-gate VLSI circuits," in *Proc. 13th IEEE Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2005, pp. 521–524.

[12] S. Meraji, W. Zhang, and C. Tropper, "On the scalability and dynamic load-balancing of optimistic gate level simulation," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 29, no. 9, pp. 1368–1380, Sep. 2010.

[13] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with GPUs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, 2011, Art. no. 29.

[14] S. Meraji and C. Tropper, "Optimizing techniques for parallel digital logic simulation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 6, pp. 1135–1146, Jun. 2012.

[15] M. L. Bailey, J. V. Briner Jr, and R. D. Chamberlain, "Parallel logic simulation of VLSI systems," *ACM Comput. Surveys*, vol. 26, pp. 255–294, 1994.

[16] E. Gonsiorowski, J. M. Lapre, and C. D. Carothers, "Automatic model generation for gate-level circuit PDES with reverse computation," *ACM Trans. Model. Comput. Simul.*, vol. 27, 2017, Art. no. 12.

[17] D. E. Martin, R. Radhakrishnan, D. M. Rao, M. Chetlur, K. Subramani, and P. A. Wilsey, "Analysis and simulation of mixed-technology VLSI systems," *J. Parallel Distrib. Comput.*, vol. 62, pp. 468–493, 2002.

[18] L. Li, H. Huang, and C. Tropper, "Dvs: An object-oriented framework for distributed verilog simulation," in *Proc. 17th Workshop Parallel Distrib. Simul.*, 2003, pp. 173–180.

[19] D. Becker, M. Moy, and J. Cornet, "Challenges for the parallelization of loosely timed SystemC programs," in *Proc. IEEE Int. Symp. Rapid Syst. Prototyping*, 2015, pp. 54–60.

[20] J. H. Weinstock, L. G. Murillo, R. Leupers, and G. Ascheid, "Parallel SystemC simulation for ESL design," *ACM Trans. Embedded Comput. Syst.*, vol. 16, 2016, Art. no. 27.

[21] N. Ventroux and T. Sassolas, "A new parallel SystemC kernel leveraging manycore architectures," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2016, pp. 487–492.

[22] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous parallel systemc simulation on multi-core host architectures," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, 2010, pp. 241–246.

[23] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC acceleration on GPU architectures," in *Proc. 49th ACM/EDAC/IEEE Des. Autom. Conf.*, 2012, pp. 115–120.

[24] S. Reder, C. Roth, H. Bucher, O. Sander, and J. Becker, "Adaptive algorithm and tool flow for accelerating SystemC on many-core architectures," *Microprocessors Microsyst.*, vol. 39, pp. 1063–1075, 2015.

[25] W. Chen, X. Han, C. Chang, G. Liu, and R. Domer, "Out-of-order parallel discrete event simulation for transaction level models," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 12, pp. 1859–1872, Dec. 2014.

[26] T. Schmidt, G. Liu, and R. Dömer, "Exploiting thread and data level parallelism for ultimate parallel SystemC simulation," in *Proc. 54th Annu. Des. Autom. Conf.*, 2017, Art. no. 79.

[27] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, et al., "Hornet: A cycle-level multicore simulator," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 31, no. 6, pp. 890–903, Jun. 2012.

[28] J. Wang, Z. Dong, S. Yalamanchili, and G. Riley, "FNM: An enhanced null-message algorithm for parallel simulation of multicore systems," *ACM Trans. Model. Comput. Simul.*, vol. 26, 2016, Art. no. 11.

[29] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The wisconsin wind tunnel: Virtual prototyping of parallel computers," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, 1993, pp. 48–60.

[30] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, et al., "The structural simulation toolkit," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, 2011.

[31] M. Chidester and A. George, "Parallel simulation of chip-multiprocessor architectures," *ACM Trans. Model. Comput. Simul.*, vol. 12, pp. 176–200, 2002.

[32] J. Zarrin, R. L. Aguiar, and J. P. Barraca, "Manycore simulation for peta-scale system design: Motivation, tools, challenges and prospects," *Simul. Model. Practice Theory*, vol. 72, pp. 168–201, 2017.

[33] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low-memory, modular time warp system," *J. Parallel Distrib. Comput.*, vol. 62, pp. 1648–1669, 2002.

[34] A. Pellegrini, R. Vitali, and F. Quaglia, "The rome optimistic simulator: Core internals and programming model," in *Proc. 4th Int. ICST Conf. Simul. Tools Techn.*, 2011, pp. 96–98.

[35] S. Gupta and P. A. Wilsey, "Quantitative driven optimization of a time warp kernel," in *Proc. ACM SIGSIM Conf. Principles Adv. Discrete Simul.*, 2017, pp. 27–38.

[36] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale hpc network systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, pp. 87–100, 2017.

[37] Y.-H. Low, C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, et al., "Survey of languages and runtime libraries for parallel discrete-event simulation," *Simul.*, vol. 72, pp. 170–186, 1999.

[38] K. Perumalla, R. Fujimoto, and A. Ogielski, "TED—a language for modeling telecommunication networks," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 25, pp. 4–11, 1998.

[39] R. A. Meyer and R. L. Bagrodia, "MVPE: Visual design of parallel simulation models," in *Proc. 4th Int. Workshop Model. Anal. Simul. Comput. Telecommun. Syst.*, 1996, pp. 227–230.

[40] K. S. Perumalla and R. M. Fujimoto, "Interactive parallel simulations with the Jane framework," *Future Generation Comput. Syst.*, vol. 17, pp. 525–537, 2001.

[41] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, 1978.

[42] C. Lattner, "LLVM and Clang: Next generation compiler technology," in *Proc. BSD Conf.*, 2008, pp. 1–2.

[43] (2017). *OpenStack is Open Source Software for Creating Private and Public Clouds*. [Online]. Available: http://www.openstack.org/

[44] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

Alireza Poshtkohi received the BSc degree from the Azad University of Qazvin, Qazvin, Iran, in 2006, and the MSc degree from Shahed University, Tehran, Iran, in 2011, both in electrical and electronic engineering. His current research interests include operating systems, distributed systems, parallel and distributed simulation, grid and cloud computing, system-level design and methodologies, embedded computer systems, specification and modeling languages, computer architecture, computational physics, and differential geometry. He has authored a textbook in Grid and Cloud Computing with Taylor & Francis, Florida USA.

M. B. Ghaznavi-Ghoushchi received the BSc degree from the Shiraz University, Shiraz, Iran, in 1993, the MSc and PhD degrees both from the Tarbiat Modares University, Tehran, Iran, in 1997, and 2003 respectively. During 2003-2004, he was a researcher in the TMU Institute of Information Technology. He is currently an associate professor with Shahed University, Tehran, Iran. His interests include VLSI Design, Low-Power and Energy-Efficient circuit and systems, Computer Aided Design Automation for Mixed-Signal and UML-based designs for SOC and Mixed-Signal.

Kamyar Saghafi received the B.Sc. degree in electrical engineering from the Iran University of Science and Technology, Tehran, Iran, in 1991, and the MS and PhD degrees in electrical engineering from Tarbiat Modares University, Tehran, Iran in 1994 and 1999, respectively. He is currently an associate professor with Shahed University, Tehran, Iran. His current research interests are numerical simulation of semiconductor microelectronic, nanoelectronic, and optoelectronic devices.