PDES-A: Accelerators for Parallel Discrete Event Simulation Implemented on FPGAs

SHAFIUR RAHMAN, NAEL ABU-GHAZALEH, and WALID NAJJAR,
University of California Riverside

In this article, we present experiences implementing a general Parallel Discrete Event Simulation (PDES) accelerator on a Field Programmable Gate Array (FPGA). The accelerator can be specialized to any particular simulation model by defining the object states and the event handling code, which are then synthesized into a custom accelerator for the given model. The accelerator consists of several event processors that can process events in parallel while maintaining the dependencies between them. Events are automatically sorted by a self-sorting event queue. The accelerator supports optimistic simulation by automatically keeping track of event history and supporting rollbacks. The architecture is limited in scalability locally by the communication and port bandwidth of the different structures. However, it is designed to allow multiple accelerators to be connected to scale up the simulation. We evaluate the design and explore several design trade-offs and optimizations. We show that the accelerator can scale to 64 concurrent event processors relative to the performance of a single event processor. At this point, the scalability becomes limited by contention on the shared structures within the datapath. To alleviate this bottleneck, we also develop a new version of the datapath that partitions the state and event space of the simulation but allows these partitions to share the use of the event processors. The new design substantially reduces contention and improves the performance with 64 processors from 49x to 62x relative to a single processor design. We went through two iterations of the design of PDES-A, first using Verilog and then using Chisel (for the partitioned version of the design). We report in this article on some observations in the differences in prototyping accelerators using these two different languages. PDES-A outperforms the ROSS simulator running on a 12-core Intel Xeon machine by a factor of 3.2x with less than 15% of the power consumption. Our future work includes building multiple interconnected PDES-A cores.

CCS Concepts: • **Computing methodologies → Discrete-event simulation**; • **Hardware → Reconfigurable logic and FPGAs**; **Hardware accelerators**;

Additional Key Words and Phrases: PDES, FPGA, accelerator, coprocessor, parallel simulation

ACM Reference format:
Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2019. PDES-A: Accelerators for Parallel Discrete Event Simulation Implemented on FPGAs. *ACM Trans. Model. Comput. Simul.* 29, 2, Article 12 (April 2019), 25 pages.
https://doi.org/10.1145/3302259

This article is based on work supported by the Air Force Office of Scientific Research (AFOSR) under Award No. FA9550-15-1-0384 and the Defense University Research Instrumentation Program (DURIP) Award No. FA9550-15-1-0376.

Authors' address: S. Rahman, N. Abu-Ghazaleh, and W. Najjar, University of California Riverside, 900 University Ave, Riverside, CA, 92521, USA; emails: mrahm008@ucr.edu, {nael, najjar}@cs.ucr.edu.

1049-3301/2019/04-ART12 $15.00
https://doi.org/10.1145/3302259

1 INTRODUCTION

Discrete Event Simulation (DES) is an important application used in the design and evaluation of systems and phenomena in which the change of state is discrete. It is heavily used in a number of scientific, engineering, medical, and industrial applications. Parallel Discrete Event Simulation (PDES) leverages parallel processing to increase the performance and capacity of DES, enabling the simulation of larger, more detailed models for more scenarios and in a shorter period of time. PDES is a fine-grained application with irregular communication patterns and frequent synchronization, making it challenging to parallelize.

In recent years, researchers have developed and analyzed PDES simulators on a variety of parallel and distributed hardware platforms as these platforms have continued to evolve. The widespread use of both shared and distributed memory cluster environments has motivated the development of PDES kernels optimized for these environments, such as GTW [11], ROSS [6] and WarpIV [37]. The emergence of multi-core and many-core processors has attracted considerable interest among high-performance computing communities to explore the performance of PDES applications in these emerging platforms. Typically, these simulators [15, 27, 40] use multi-threading and develop synchronization-friendly data structures to take advantage of the low communication latency and tight memory integration among cores on same chip. PDES has been shown to scale well on many-core architectures, such as the Tilera Tile64 [21] and the Intel Xeon Phi [8, 42]. Several researchers have also explored the use of GPGPUs to accelerate PDES [7, 26, 28, 39].

In contrast to these efforts, there are very few works describing acceleration of PDES using non-conventional architectures, such as Field Programmable Gate Arrays (FPGAs). FPGA-based accelerator development platforms have recently progressed rapidly to make them available to all programmers. Amazon unveiled its EC2 F1 FPGA cluster [20], which makes high-performance FPGAs on the cloud accessible to general consumers on demand. Microsoft's Catapult [30] and the Convey Wolverine [9] are examples of recent systems that offer programmability, tight integration, advanced communication, and memory sharing with CPUs in industry standard HPC clusters. Intel has already started shipping versions of its Xeon processors with integrated FPGA support [4]. The latest evolution in memory technologies, such as Micron's Hybrid Memory Cube (HMC) [19], can offer up to 320GB/s effective bandwidth and many recent FPGAs come with built-in HMC. This can provide a huge boost to applications requiring high memory bandwidth, such as PDES, and emphasizes the need for specialized hardware support to take advantage of this bandwidth.

In particular, our interest in FPGAs stems from the fact that they do not limit the datapath organization of the accelerator, allowing us to experiment with how the computation should ideally be supported. In addition, the end of Dennard scaling and the expected arrival of dark silicon makes the use of custom accelerators for important applications one of the few remaining directions for continued improvement of computing performance. Many types of accelerators have already been proposed for a large number of important applications, such as deep learning [25, 35] and graph processing [43]. The exploration of accelerator architecture for PDES can yield similar benefits and inform the design decision of custom accelerators for many simulation applications.

An FPGA implementation of PDES offers two primary advantages:

- *Low-latency and high-bandwidth on-chip communication:* An FPGA can support fast and high bandwidth on-chip communication, substantially alleviating the communication bottleneck that often limits the performance of PDES [41]. On the other hand, the memory latency experienced by FPGAs is very often high (but the available bandwidth is also high), necessitating approaches to hide the memory access latency.
- *Specialized, high-bandwidth datapaths:* General-purpose processing provides high flexibility but at the cost of high overhead and a fixed datapath. A specialized accelerator, in contrast,

can more efficiently execute a required task without the unnecessary overheads of fetching instructions and moving data around a general datapath. These advantages have been estimated to yield over 500x improvement in performance for video encoding, with 90% reduction in energy [16]. Moreover, an FPGA can allow high parallelism limited only by the available memory bandwidth, number of processing units, and the communication bandwidth available between them.

We believe that PDES is potentially an excellent fit for these strengths of FPGAs. PDES exhibits *ordered irregular parallelism* (OIP) with the following three characteristics: (1) total or partial order between tasks; (2) dynamic and unpredictable data dependencies; and (3) dynamic generation of tasks that are not known beforehand [29]. It has inherent parallelism that is difficult to exploit in a traditional multi-processor architecture without careful implementation. To preserve order among the tasks and maintain causality, hardware-based speculative implementations such as thread-level speculation (TLS) often introduce false data dependencies, for example, in the form of a priority queue [23]. Runtime communication overheads limit the scalability of PDES [17]; these overheads may be lowered and masked in the context of an FPGA [41]. For models with a computationally expensive processing task, FPGA implementations are likely to yield to more streamlined customized processors. On the other hand, if the event processing is simple, FPGAs can accommodate a larger number of event processors, increasing the raw available hardware parallelism. Finally, FPGAs have exceptional energy properties compared to general-purpose graphical processing units (GPGPUs) and many-cores.

In this article, we present our design of a PDES accelerator, which we call *PDES-A*. We show that PDES-A can provide excellent scalability for Phold with up to 64 concurrent event processors. We explore the design trade-off space and explore alternatives in the design of the critical structures supporting the simulation. Our initial prototype outperforms a similar simulation on a 12-core, 3.5GHz Intel Xeon CPU by 2.5x. Analysis of the baseline design shows that contention for the shared structures of the datapath starts to substantially limit performance when the number of event processors increases, accounting for 30% of the event processing time. Therefore, we rearchitect PDES-A to relieve contention by partitioning all shared structures, such as event and state queues, into multiple substructures and mapping each logical process (LP) to one of these different partitions. Although causality is not maintained directly among the partitions, we use the rollback mechanism to recover. Partitioning alleviates contention, resulting in another 25% improvement in throughput; while the baseline design scaled to 49x with 64 event processors, the new design is able to reach 62x. As a result, it outperforms the 12-core Intel Core i7 by 3.2x while consuming less than 15% of the power (i.e., over 22x improvement in performance per watt). We show that there remain several opportunities to further optimize the design. Moreover, we show that multiple PDES-A accelerators can fit within the same FPGA chip, allowing us to further scale the performance in the future.

PDES-A is in the vein of prior studies that explore customized or programmable hardware support for PDES. Fujimoto et al. proposed the Rollback chip, a special-purpose processor to accelerate state saving and rollbacks in Time Warp [14]. Most similar to our work, Herbordt et al. [18] explored FPGA implementation of a specific PDES model for molecular dynamics. In the area of logic simulation, the use of FPGA offers opportunities for performance since the design being simulated can simply be emulated on the FPGA [38]. Noronha et al. explore the use of a programmable network interface card to accelerate GVT computation and direct message cancellation [24]. Our work differs in the way that a general optimistic PDES is implemented completely in hardware.

An earlier version of this article appeared in SIGSIM-PADS 2017 [31]. The current version extends the original paper in the following ways:

- We identified contention as a major bottleneck in the design, and developed a completely new design that alleviates contention by partitioning the datapath structures in multiple independent components that share the event processors.
- We present an evaluation of the new design showing that it alleviates contention, leading to near-perfect speedup (62x with 64 processors).
- The new design was rewritten using Chisel, a new higher-level hardware description language; we discuss our experience in using Chisel versus Verilog, which is important for future designers of accelerators.
- We extend and improve the description of several aspects of the design and analysis.

The remainder of this article is organized as follows. Section 2 presents some background information related to PDES and introduces the Convey Wolverine FPGA system that we used in our experiments. Section 3 introduces the design of our PDES-A accelerator and its various components. Section 4 contains an overview of some implementation details and the verification of PDES-A. Section 5 presents a detailed performance evaluation of the design. Our optimizations on the baseline design is discussed in detail in Section 6. Section 7 shows a comparison with a CPU-based PDES simulator for better understanding of the performance gain. In Section 8, we explore the overhead of PDES-A and project the potential performance if we integrate multiple PDES-A accelerators on the same FPGA chip. Finally, Section 9 presents our concluding remarks.

2 BACKGROUND

In this section, we provide some background information necessary for understanding our proposed design. First, we discuss PDES and then present the Convey Wolverine FPGA application accelerator that we use in our experiments.

2.1 Parallel Discrete Event Simulation

A discrete event simulation (DES) models the behavior of a system that has discrete changes in state. This is in contrast to the more typical time-stepped simulations in which the complete state of the system is computed at regular intervals in time. DES has applications in many domains, such as computer and telecommunication simulations, war gaming/military simulations, operations research, epidemic simulations, and many more. PDES leverages the additional computational power and memory capacity of multiple processors to increase the performance and capacity of DES, allowing the simulation of larger, more detailed models and the consideration of more scenarios in a shorter amount of time [12].

In a PDES simulation, the simulation objects are partitioned across a number of *logical processes* (LPs) that are distributed to different Processing Elements (PEs). Each PE executes its events in simulation time order (similar to DES). Each processed event can update the state of its object and possibly generate future events. Maintaining correct execution requires preserving timestamp order among dependent events on different LPs. If a PE receives an event from another PE, this event must be processed in time-stamped order for correct simulation.

To ensure correct simulation, two synchronization algorithms are commonly used: conservative and optimistic synchronization. In conservative simulation, PEs coordinate with each other to agree on a *lookahead* window in time where events can be safely executed without compromising causality. This synchronization imposes an overhead on the PEs to continue to advance. In contrast, optimistic simulation algorithms such as Time Warp [22] allow PEs to process events without synchronization. As a result, it is possible for an LP to receive a *straggler* event with a timestamp earlier than their current simulation time. To preserve causality, optimistic simulators maintain checkpoints of the simulation and rollback to a state in the past earlier than the time of

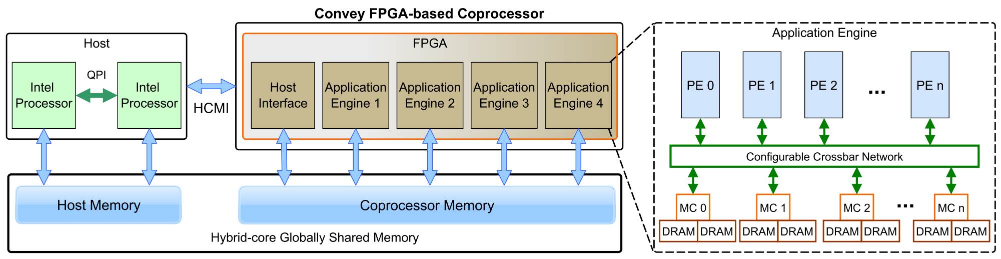

Fig. 1. Overview of the Convey Hybrid-Core architecture.

the straggler event. The rollback may require the LP to cancel out any event messages that it generated erroneously using *anti-messages*. This approach uses more memory for keeping checkpoint information, which needs to be garbage collected when no longer needed to bound the dynamic memory size. A *Global Virtual Time* (GVT) algorithm is used to identify the minimum simulation time that all LPs have reached: checkpoints with a time lower than the GVT can be garbage collected, and events earlier than the GVT may be safely committed.

2.2 Convey Wolverine FPGA Accelerator

The Convey Wolverine FPGA Accelerator is an FPGA-based coprocessor that augments a commodity processor with processing elements optimized for key algorithms that are inefficient to run in a conventional processors. The coprocessor contains a standard FPGA that uses a standard x86 host interface to communicate with a Intel Xeon-based host processor.

The Wolverine WX2000 integrates three major subsystems: Application Engine Hub (AEH), Application Engines (AEs), and the Memory Subsystem (MCs) [9]. Figure 1 shows an abstracted view of the system architecture. AEs are the core of the system and implement the specialized functionality of the coprocessor. There are four AEs in the system implemented on a Xilinx Virtex-7 XC7V2000T FPGA. The AEs are connected to memory controllers via 10GB/s point-to-point network links availing up to 40GB/s of bandwidth in an optimized implementation. The clock rate of the FPGAs (150MHz) is much lower than that of the CPU, but they can implement many specialized functional units in parallel. With proper utilization of the memory bandwidth, the throughput can be many times that of a single processor. This makes the system ideal for applications benefiting from high computation capability and large high-bandwidth memory. The number of processing elements in an AE is limited by the resources available in the FPGA chip used. The processing elements can connect to the memory subsystem through a crossbar network that allows each processing element to access any part of the physical memory.

The AEH acts as the control and management interface to the coprocessor. The Hybrid Core Memory Interconnect (HCMI), implemented in the AEH, connects the coprocessor to the processor to fetch instructions and to process and route memory requests to the MCs. It also initializes the AEs, programs them, and conveys execution instructions from the processor.

The memory subsystem includes 4 memory controllers supporting 4 DDR3 memory channels providing a high bandwidth, but also high-latency, connection between memory and application engines [10]. The memory subsystem provides simplified logical memory interface ports that connect to a crossbar network, which, in turn, connects to the physical memory controller. Another important part of the memory system is the Hybrid Core Globally Shared Memory architecture. It creates a unified memory address space where all physical memory is addressable by both the processor and the coprocessor using a virtual address. The memory subsystem implements the address translation, crossbar routing, and configuration circuits.

The architecture of the Convey system can present some advantages in the design of a PDES system. Memory access latency and communication overhead prevent most PDES models from achieving high throughput. The high bandwidth parallel data access capability in the Convey system can be exploited to bypass the bottleneck by employing a large number of event processors. In this way, while one event processor waits for memory, others can be active, enabling the system to effectively use the high memory bandwidth. Also, the reconfigurable fabric allows us to implement optimized datapaths, including the communication network among event processors to reduce communication and synchronization overheads. Finally, leveraging the standard x86 interface, multiple Convey servers can be interconnected, which opens up the possibilities to scale up the PDES accelerator to a large cluster-based implementation.

3 PDES-A: DESIGN OVERVIEW

FPGAs are progressing quickly in terms of both capabilities and integration with computing platforms, making them increasingly accessible to programmers. However, concerns regarding longer development time and different development tools, as well as lack of flexibility and portability, are significant impediments to FPGA adoption. Considering these concerns, our goal is to enable simulation of different applications within an easy-to-use framework. An interesting characteristic of PDES simulation algorithms is that, despite the irregular nature of the dependencies, the algorithm itself has a fairly simple and clean execution semantics, iterating over the event list to schedule events, processing these events, and then scheduling any events that result from their execution. Most of the complexity lies in the data structures to manage the event lists and those for handling synchronization and causality, which are both common to any PDES application. In contrast, application-specific event processing often is computationally and logically contained, and for many simulation models, they are simple. The Simian project [34] shows that a completely functional PDES engine can be implemented in less that 500 lines of Python code. These properties can be leveraged by an FPGA-based PDES engine to make it modular and scalable so that experts in any domain can simulate their application models by simply defining the state transition and event-processing logic, not requiring hardware development expertise.

In this article, we present an overview of the unit PDES accelerator (PDES-A), the building block of our PDES accelerator. Each PDES-A accelerator is a tightly coupled high-performance PDES simulator in its own right. However, hardware limitations—such as contention for shared event and state queue ports, local interconnection network complexity, and bandwidth limit—restrict the scalability of this tightly coupled design approach. These properties suggest a design in which multiple interconnected PDES-A accelerators together work on a large simulation model and exploiting the full available FPGA resources. In this article, we explore and analyze only PDES-A and not the full architecture consisting of many PDES-A accelerators.

In an FPGA implementation, event processing, communication, synchronization, and memory access operate differently from how they work in general-purpose processors. Therefore, both performance bottlenecks and optimization opportunities differ from those in conventional software implementations of PDES. We developed a baseline implementation of PDES-A and used it to identify performance bottlenecks. We then used these insights to develop improved versions of the accelerator. We describe our design and optimizations in this section.

3.1 Design Goals

PDES-A provides a modular framework in which various components can be adjusted independently to attain the most effective datapath flow control across different PDES models. Since the time to process events in different models will vary, we designed an event-driven execution model that does not make assumptions about event execution time. We decided to implement an

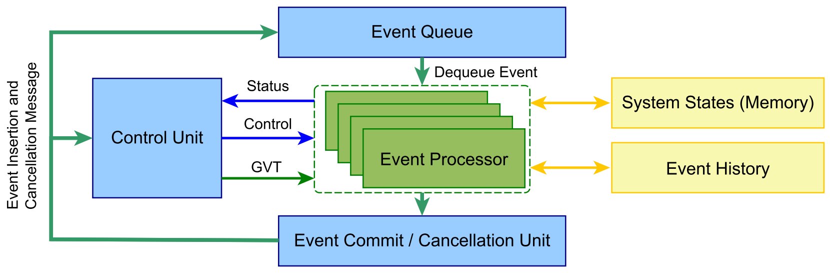

Fig. 2. Block diagram of basic control and dataflow in a PDES system.

optimistically synchronized simulator to allow the system to operate around the large memory access latencies. However, the tight coupling within the system should allow us to control the progress of the simulation and naturally bound optimism. We avoided any model-specific tuning to retain generality of the accelerator.

3.2 General Overview

The simulator is organized into four major components: (1) Event Queue—stores the pending events; (2) Event Processors—custom datapaths for processing the event tasks in the model; (3) System State Memory—holds relevant system state, including checkpointing information; (4) and the Controller—coordinates all aspects of operation. The first three components correspond to the same functionality in traditional PDES engines in any discrete event simulator; the last one oversees the event processors to ensure correct parallel operation and communication.

Communication between different components uses message passing. We currently support three message types: Event messages, anti-messages, and GVT messages. These three message types are the minimum required for an optimistic simulator to operate, but additional message types could be supported in the future to implement optimization or to coordinate between multiple PDES-A units.

Figure 2 shows the major components of PDES-A and their interactions. The event queue contains a list of all of the unprocessed events sorted in ascending order of their timestamp. Event processors receive event messages from the queue. After processing events, additional events that may be generated are sent and inserted into the event queue for scheduling. The system needs to keep track of all of the processed events and the changes made by them until it is guaranteed that the events will not be rolled back. When an event is received for processing, the event processor checks for any conflicting events from the event history. Anti-messages are generated when the event processor discovers that erroneous events have been generated by an event processed earlier. Since the state memory is shared, a controller unit is necessary to monitor the event processors for possible resource conflict and manage their correct operation. Another integral function of the control unit is the generation of GVT, which is used to identify the events and state changes that can be safely committed. The control unit computes GVT continuously and forwards updated estimates to the commit logic. These messages should have low latency to limit the occurrence of rollbacks and to control the size of the event and message history. In the remainder of this section, we describe the primary components in more detail.

3.3 Event Queue

The event queue maintains a time-ordered list of events to be processed by the event processors. It needs to support two basic operations: *insert* and *dequeue*. An *invalidate* operation can be included to facilitate faster cancellation of events that have not been processed yet. However, this function was not considered in our preliminary implementation to avoid circuit complexity.

The event queue structure and its impact on PDES performance has been studied in the context of software implementations [33]; however, it is important to understand suitable queue organizations implemented in hardware. Prior work has studied hardware queue structures supporting different features. Priority queues offer attractive properties for PDES such as constant time operation, scalability, low area overhead, and simple hardware routing structures. Simple binary heap–based priority queues are commonly used in hardware-based implementations but require $O(log(n))$ time for enqueue and dequeue operations. Other options have other drawbacks; for example, Calendar Queues [3] support $O(1)$ access time but are difficult and expensive to scale in a hardware implementation. QuickQ [32] uses multiple dual-ported RAM in a pipelined structure that provides easy scalability and supports constant time access. However, the access time is proportional to the size of each stage of RAM. Configuring them to achieve a small access time necessitates a large number of stages, which leads to high hardware complexity. For these reasons, we selected a *pipelined heap* (P-heap for short) [2] structure as the basic organization in our implementation, except for a few modifications described later. P-heap uses a pipelined binary heap to provide constant access time of two cycles while having a hardware complexity similar to binary heaps.

The P-heap structure uses a conventional binary heap, with each node storing a few additional bits to represent the number of vacancies in the subtree rooted at the node (Figure 3). The capacity values are used by *insert* operations to find the path in the heap through which it should percolate. P-heap also keeps a *token* variable for each stage, which contains the current operation, target node identifier, and value that is percolating down to that stage. During an insertion operation, the value in token variable is compared with the target node: a smaller value replaces the target node value and a larger value passes down to the token variable of the following stage. The id value of the next stage is determined by checking the capacity associated with the nodes.

For the dequeue operation, the value of the root node is dequeued and replaced by the smaller between its two child nodes. The same operation continues to move through the branch, promoting the smallest child at every step. During any operation, any two of the consecutive stages are accessed; one read access and the other write access. As a result, a stage can handle a new operation every two cycles since the operation of the heap is pipelined with different insert and/or dequeue operations at different stages in their operation [2].

P-heap can be efficiently implemented in an FPGA. Every stage requires *a Dual Port RAM*. Depending on the size of the stage, it can be synthesized with registers, distributed RAM, or block RAM to maximize resource utilization. An arbitrary number of stages can be added (limited by resource availability) as the performance is not hurt by the number of stages in the heap, making it straightforward to scale.

In an optimistic PDES system, it is possible that ordering can be relaxed to improve performance, while maintaining simulation correctness via rollbacks to recover from occasional ordering violations. This opens up possibilities for optimization of the queue structure. For example, multiple heaps may be used in parallel to service more than one request in a single cycle. In an approach similar to [18], we can use a randomizer network to direct multiple requests to multiple available heaps (Figure 4). There is a chance that two of the highest-priority events may reside in the same heap and an ordering violation will occur when a lower-priority event from the same LP is dequeued from another queue during multiple dequeues. However, as the number of LPs and events grow, the probability that two events from the same LP are at different queue heads decreases. Thus, the number of such events will be low enough to result in a net performance gain. We used the simple P-heap model in the baseline implementation and explored the effect of this version in

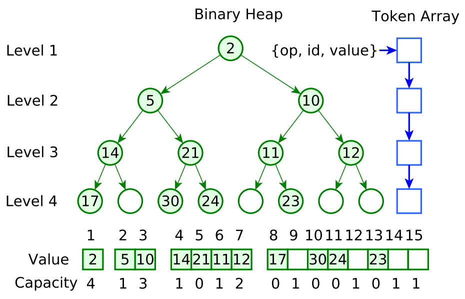

Fig. 3. The P-heap data structure [2].

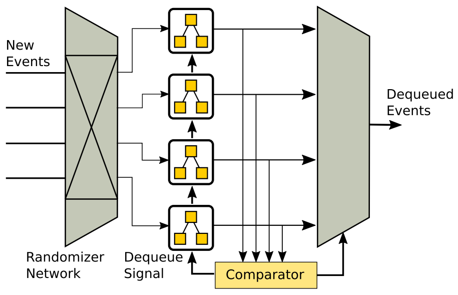

Fig. 4. Multiple event issue priority queue.

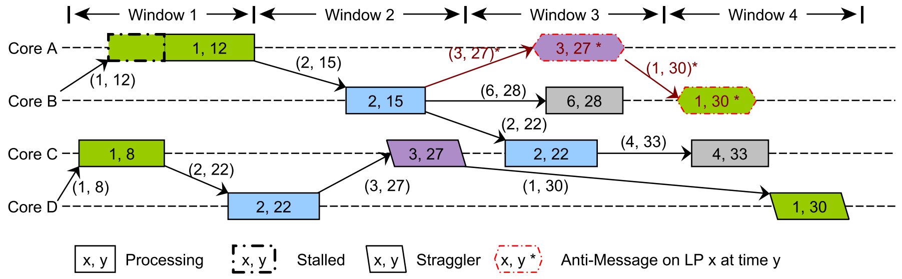

Fig. 5. Simplified timeline representation showing scheduling of events in the system.

our optimized architecture, discussed in Section 6. Other structures that sacrifice full ordering but admit higher parallelism, such as Gupta and Wilsey's lock-free queue, may also be explored [15].

The queue stores a key-value pair. Event timestamp acts as the key and the value contains the id of the target LP and a payload message. In case the payload message is too large, we store a pointer to a payload message in memory.

3.4 Event Processor

The event processor is at the core of PDES-A. The front end of the processor is common to all simulation models. It is responsible for the following general operations: (1) to check the event history for conflicts; (2) to store and clean up state snapshots by checking GVT; (3) to support event exchange with the event queue; and (4) to respond to control signals to avoid conflicting event processing. In addition, the event processors execute the actual event handlers that are specialized to each simulation model to generate the next events and compute state transitions.

The task processing logic is designed to be replaceable and easily customizable to the events in different models. It appears as a black box to the event processor system. All communications are done through the preconfigured interface. The event processor passes event messages and relevant data to the core logic by populating FIFO buffers. Once the events are processed, the core logic uses output buffers to load the generated events. The core logic has interfaces to request state memory by supplying addresses and sizes. The fetched memory is placed into a FIFO buffer to be read from the core. The interface to the memory port is standard and provided in the core to be easily accessible by the task-processing logic.

3.5 Event Scheduling and Processing

Figure 5 shows a representative event execution timeline in the system. Events are assigned to the event processors in order of their timestamps; in the figure, the event is represented by a tuple (x,y) where x is the LP number and y is the simulation time. When a second event (1, 12) is scheduled while another event (1, 8) associated with the same LP is already being processed, the core is stalled by the controller unit until the first event completes. At the completion of an event, the controller unit allows the earliest timestamp among the waiting (stalled) events for that core to proceed as shown in window 1. Each event generates one or more new events when it exits, which are scheduled at some time in the future when a core is available. Occasionally, an event is processed after another event with a later timestamp has already executed (i.e., *a straggler event*). It needs to be rolled back to restore causality. Windows 2 in Figure 5 shows one such event, (2, 22), which executes before event (2, 15). We use a lazy cancellation and rollback approach. Event-processing logic detects the conflict by checking the processed events list and initiates the rollback. The new event will restore the states and generate events it would have normally scheduled (6, 28) along with anti-messages (3, 27*) for all events generated by the straggler event, and new event (2, 22) that reschedules the straggler event itself.

The anti-messages may get processed before or after the target event is done. An anti-message (3, 27*) checks the event history and if the target event has already been processed, it rolls back the states and generates other anti-messages (1, 30*) to *chase* the erroneous message chain much like a regular event, as shown in window 3 of Figure 5. If the target message is yet to arrive, the anti-message is stored in the event history table. The target message (1, 30) cancels itself upon discovery of the anti-message in the history and no new event is generated, as shown in window 4.

3.6 Event History

An important component for maintaining order of execution in an optimistic simulation is the checkpointing and state restoration mechanism. To revert back the changes done by an event, we keep records of it until it is guaranteed to be committed. To be able to do this, we store the processed events along with a set of information required for rollbacks in an Event History Memory. The memory is organized as a *free list* of memory blocks in the on-chip memory, each block containing space for 4 entries and pointer metadata. The history entries contain event data and a rollback data structure defined by the programmer. Hardware structure generation is done by the framework based on the computed size of the entries after the user defines the data structure.

Each LP has a list for history where blocks are appended as history grows and entries are stored in order of their execution. The on-chip memory gives fast access to the history but puts a restriction on how large it can be. Thus, when the on-chip memory is close to being full, we allocate memory blocks in the off-chip RAM. This will add latency to event history access and put a burden on the memory bandwidth. To prevent this, when the event history starts spilling into off-chip memory, we initiate a *flush* of the processing cores. A *flush* would temporarily prevent issuing new events for processing until all event processors are idle. This allows the GVT to be recomputed, and the pending event sets to be properly reordered. As a result, stale history entries can be removed and the event history shrinks in size.

The history list is maintained and pruned at the processor. The list is pulled into the processor as a whole, then the stale events with timestamps older than GVT are removed from the list. If there are events violating causality, they are removed and passed to the rollback module. The active event in the processor is appended to the rest of the list, and the list is stored to memory when event processing is complete. The memory allocation and communication, when done in the hardware, is cheap and does not add much overhead.

3.7 Rollback and Cancellation

The state restoration or reverse computation logic for rollback needs to be defined by the programmer. The programmer also defines the data structure for the event history and which values should be stored during forward computation. The model-specific logic populates the data structure with correct states and the framework takes care of storing the checkpoint data to the history. On causality violation, the processor receives a list of violating events in reverse order of execution. The programmer needs to define the way that the checkpoint data is used to restore the states. For smaller states, state value may be saved in the event history and later restored directly. For more complex cases, the user can store other information, such as a random number generator seed, that can be used to reverse compute the states and create anti-messages. We save the history for each event and rollbacks are done for every event executed out of order.

Rolling back an event also reinserts the rolled-back event to the event queue and generates anti-messages to revert the events that it wrongly created. This anti-messages are scheduled like normal events and they trigger cancellation during the checking of event history in processors, as described in Section 3.5. Each event message, along with its payload, carries a unique identifier which is a tuple (processor ID, sequence number of active event) that serves as an identifier for its parent event. The sequence number simply indicates the number of events processed in a processor and each new event received in a processor gets a unique sequence number from a counter in the processor. The unique identifier for the parent event is stored in its history entry along with the rest of the event data. When the parent event is reversed, the rollback process emits anti-messages carrying this identifier with timestamp and LP of the target events to be canceled. The anti-message can be matched with the target event using the unique identifier and event timestamp.

4 IMPLEMENTATION OVERVIEW

We used a full RTL implementation on a Convey WX-2000 accelerator for prototyping the simulator. The current prototype fits comfortably in a Virtex-7 XC7V2000T FPGA. The event history table and queue were implemented in the BRAM memory available in the FPGA. The on-board 32GB DDR3 memory was used for state memory implementation, although very little memory was necessary for our prototype. The system uses a 150MHz clock rate. The host server was used to initialize the memory and events at the beginning of the simulation. The accelerator communicates through the host interface to report results and other measurements that we collected to characterize the operation of the design. For any values that we wanted to measure during runtime, we instrumented the design with hardware counters that keep track of these events. We complemented these results with other statistics, such as queue and core occupancy, that we obtained from a functional simulator of the RTL implementation using Modelsim.

Since our design is modular, we can scale the number of event processors easily. However, as the number of processors increases, we can expect contention to arise on the fixed components of the design such as the event queue and the interconnection network. We experiment with cluster sizes from 8 to 64 in order to analyze the design trade-offs and scalability bottlenecks. The performance of the system under variable numbers of LPs and event distribution gives us insight into the most effective design parameters for a system. We sized our queues to support up to 512 initial events in the system. The queue is flexible and can be expanded in capacity or can even be made to handle overflow by spilling into the memory.

4.1 Design Language and Application Modeling

The baseline design was implemented using Verilog. However, during optimization, we reimplemented the system using Chisel [1], a hardware construction language. This decision was driven primarily by the design goal of lowering the barrier for domain modelers to build and run simulation models on the framework. Chisel is based on Scala, which is an easy-to-use and already familiar language to most domain experts. This would reduce development time and effort for anyone inclined to use the framework. The encapsulation property of the object-oriented approach of Chisel also allows us to separate modeling and framework development code, which requires less understanding of the framework from the end users. Moreover, the generated code is highly parameterizable with metaprogramming. This gives the users the capability to configure size and some basic parameters without the in-depth understanding of the language or design. Effectively, the hardware implementation becomes almost similar to the CPU-based PDES engines from the perspective of the user.

Additionally, quicker development and easier maintenance of the framework was possible thanks to the object-oriented nature and metaprogramming capabilities of Scala language. The code base was reduced in size by about 40%. The simplicity in code leads to better maintainability. Unlike industrial applications, research projects are usually developed without a fixed set of constraints in mind; thus, constraints keep evolving rapidly throughout each experiment. In the standard HDL languages, small adjustments in one component tend to ripple through the whole design, requiring extensive edits, which is not congenial to rapid prototyping. Because of the metaprogramming capabilities of the Chisel framework, effort and time required for such a task can be reduced significantly, which makes many explorations feasible. This is tangential to the architecture or performance of the accelerator, but we still offer these comments for the benefit of future research.

Models: Our goal in the evaluation is to present a general characterization of this initial prototype of PDES-A. We used the Phold model for our experiments because it is widely used to provide general characterization of PDES execution that is sensitive to the system. On Convey machines, the memory system provides high bandwidth at the cost of high latency (a few hundred cycles) which end up dominating event execution time. To emulate event processing, we let each event increment a counter up to a value picked randomly between 10 and 75 cycles to represent computation complexity. The model generates memory accesses by reading from the memory when the event starts and writing back to it again when it ends. Phold is state oblivious, but we still use a dummy state holding a counter and restore it during rollback so that we can analyze the effects of rollback in our system. New outgoing agents are generated to a random LP using a random number generator. We also use the *Airport* model [13] on our optimized system to analyze performance for an application with multiple event type and larger state. We developed our model to represent the *Airport* model included with ROSS models [5].

Design Validation: Verification of hardware design is complex since it is difficult to peek into the simulation running on the hardware. However, the hardware design flow supports a logic level simulator of the design that we used to validate that the model correctly executes the simulation. In particular, the ModelSim simulator was used to study the complete model, including the memory controllers, cross bar network and the PDES-A logic. Since the design admits many legal execution paths, and many components of the system introduce additional variability, we decided to validate the model by checking a number of invariants. In particular, we verified that no causality constraints are violated in the full event execution trace of the simulation under a number of PDES-A and application configurations.

5 PERFORMANCE EVALUATION

In this section, we evaluate the design under a number of conditions to study its performance and scalability. In addition, we analyze the hardware complexity of the design in terms of the percentage of the FPGA area that it consumes. Finally, we compare the performance to PDES on a multi-core machine and use the area estimates to project the performance of the full system with multiple PDES-A accelerators.

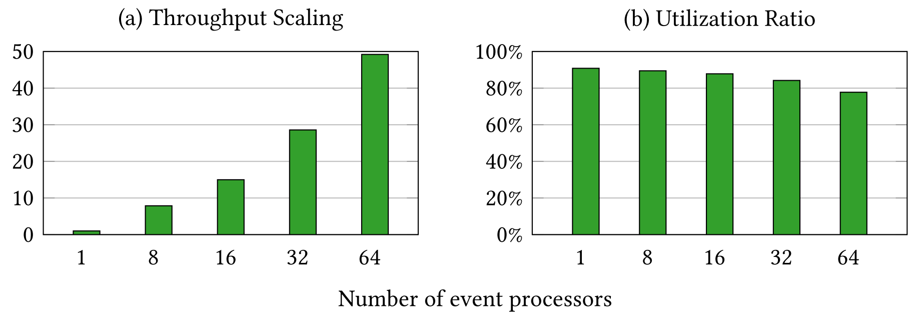

Fig. 6. Effect of variation of number of cores on (a) throughput and (b) percentage of core utilization for 256 LP and 512 initial events.

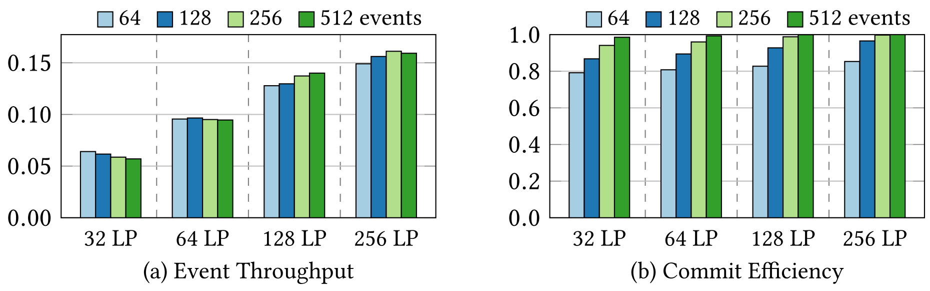

Fig. 7. (a) Event processing throughput (events/cycle) and (b) ratio of number of committed events to number of total processed events for different numbers of LPs and initial events on 64 event processors.

5.1 Performance and Scalability

In this first experiment, we scale the number of event processors from 1 to 64 while executing a Phold model. Figure 6(a) shows the scalability of the throughput normalized with respect to the throughput of a single event handler configuration. The scalability is almost linear up to 8 event handlers and continues to scale with the number of processors up to 64, where it reaches above 49x. As the number of cores increases, contention for the bandwidth of the different components in the simulation starts to increase, leading to very good but sublinear improvement in performance. Figure 6(b) shows the event processor utilization, that is, the portion of time that the event processor is actively processing an event and not stalling or waiting for resources. The utilization is generally high but starts dropping as we increase the number of event processors, reflecting that the additional contention is preventing the issuing of events to the handlers in time.

Figure 7(a) shows the throughput of the accelerator as a function of the number of LPs and the events population in the system for 64 event processors. The throughput increases significantly with the number of available LPs in the system. This is to be expected: as the events get distributed across a larger number of LPs, the probability of events being at the same LP and therefore blocking due to dependencies goes down. In our implementation, we stall all but one event when multiple cores are processing events belonging to the same LP to protect state memory consistency. Thus, having a higher number of LPs reduces the average number of stalled processors and increases utilization. In contrast, the event population in the system influences throughput to a lesser degree. Even though having a sufficient number of events is crucial to keeping the cores processing, once we have a large enough number of events, increasing the event population further does not improve throughput measurably.

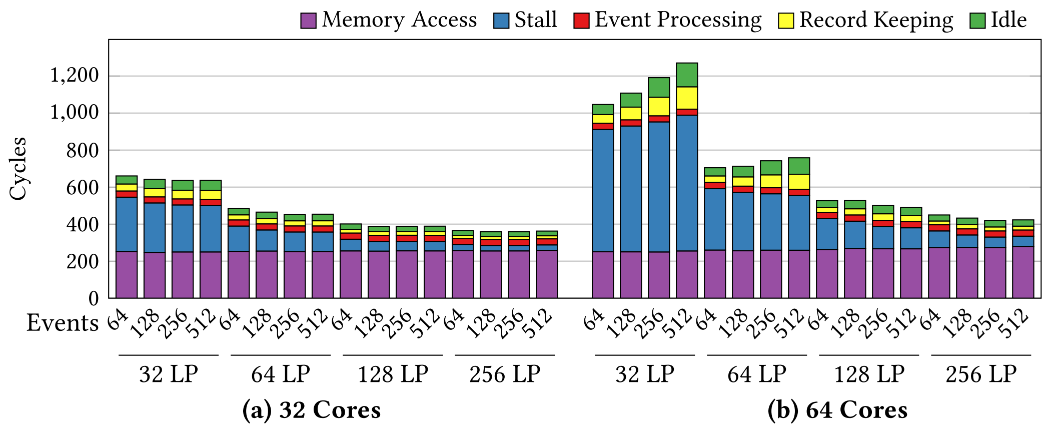

Fig. 8. Breakdown of time spent by the event processors on different tasks to process an event using (a) 32 event processors and (b) 64 event processors with respects to different number of LPs and initial event counts.

5.2 Rollbacks and Simulation Efficiency

The efficiency of the simulation, measured as the ratio of the number of committed events to processed events, is an important indicator of the performance of optimistic PDES simulators. Figure 7(b) shows the efficiency of a 64-processor PDES-A as we vary the number of events and the number of LPs. For our Phold experiment, we observed that the fraction of events that are rolled back depends on the number of events in the system but is not strongly correlated to the number of LPs in the presence of a sufficient number of events. With a large population of initial events, we observe virtually no rollbacks since there are many events that are likely to be independent at any given point in the simulation. Newly scheduled events will tend to be in the future relative to currently existing events, reducing the potential for rollbacks. However, keeping all other parameters the same, reducing the number of initial events can cause simulation efficiency to drop to around 80% (reflecting around a 20x increase in the percentage of rolled-back events). For similar reasons, the number of rolled-back events decreases slightly with a greater number of LPs in the simulation. Most causality concerns arise when events associated with the same LP are processed in the wrong order. When events are more distributed, the number of LPs is higher, thus reducing the occurrence of stalled cores. However, this effect is relatively small.

5.3 Breakdown of Event-Processing Time

Figure 8 shows how average event-processing time varies with the number of LPs and initial events and breaks down the time taken for different tasks for systems with 32 and 64 processors. The primary source of delay in event processing is the large memory access latency on the Convey system. Another major delay is due to the processors stalling for potentially conflicting events. These two primary delays in the system dominate other overheads in the event processors, such as task-processing delays and event-history maintenance, which increase as we go from 32 to 64 cores.

The average event-processing time is highest when the number of LPs or number of initial events is low. The average number of cycles goes down as more events are issued to the system or the number of LPs is increased (which reduces the probability of a stall). The reason for this behavior is apparent when we consider the breakdown of the event cycles. We note that about the same number of cycles is consumed for memory access regardless of the configuration of the

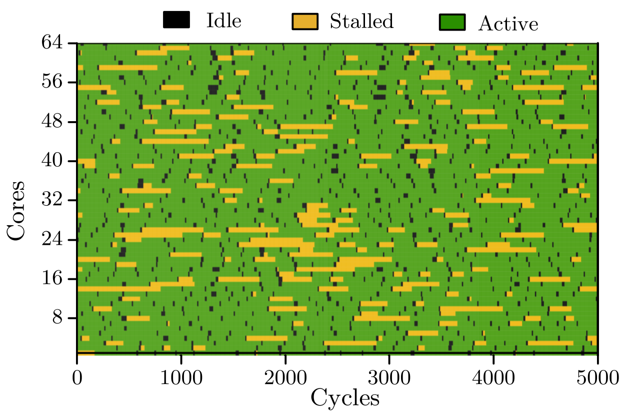

Fig. 9. Timeline demonstrating different states of the cores during a 5000-cycle frame of the simulation.

system because the memory bandwidth of the system is very large. However, the average stall time for the processors is significantly higher with fewer LPs and constitutes the major portion of the event-processing delay. For example, with 64 cores and 32 LPs, we can have no more than 32 cores active; any additional cores would hold an event for an LP that has another active event at the moment. A system of 64 LPs has over 150 stall cycles on average stall with 64 processors. The stall times drop substantially as we increase the number of LPs and events in the model. These dependencies result in a high number of stall cycles to prevent conflicts in LP-specific memory and event history. At the same time, a small number of LPs increases the chance of a causality violation. This effect is most severe when the number of LPs is close to the number of event processors. As the number of LPs is increased, the events are more distributed in terms of their associated LPs, and can be safely processed in parallel. Even if stalls are less frequent, each can take a long time to resolve.

Figure 9 helps visualize PDES-A's operation by showing how the processors are behaving over time for a simulation with 256 LPs and 512 events. The black marks show the cycles when the processors are idle before receiving a new event. Each yellow streak highlights the time a processor is stalled.

The memory access time remains mostly unaffected by the parameters in the system. The state memory is distributed in multiple banks of RAM and accesses depend on the LPs being processed. The appearance of different LPs in the event processor are not correlated in Phold and, therefore, poor locality results without any special hardware support. However, having a higher number of events may increase the probability of consecutive accesses to the same memory area and, therefore, occasionally decrease the memory access latency, reducing the average memory access time slightly.

We note that the actual event handler processing time is a minor component, less than 10%, of the overall event-processing time even in the best case. This observation motivates our future work to optimize PDES. In particular, the memory access time can be hidden behind event processing if we allow multiple applications to be handled concurrently by each handler: when one event accesses memory, others can continue execution. This and other optimization opportunities are a topic of our future research.

5.4 Memory Access

Memory access latency is a dominant part of the time required to process an event. Figure 10 shows the effect of variation of state memory access pattern on average execution time. The number of

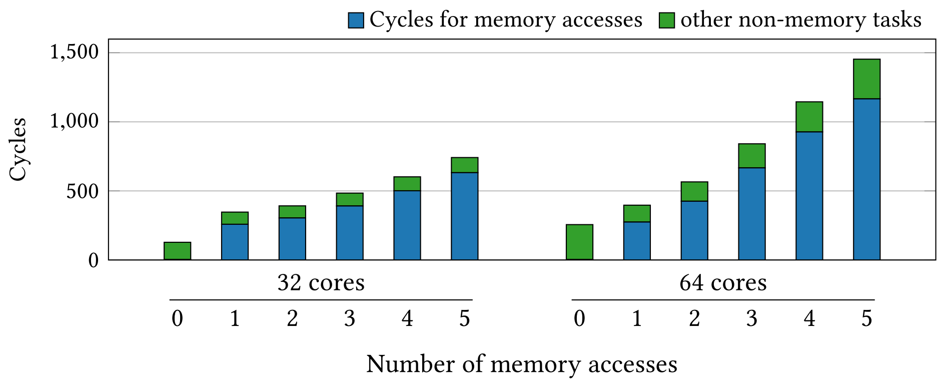

Fig. 10. Effect of number/size of state memory access on event-processing time.

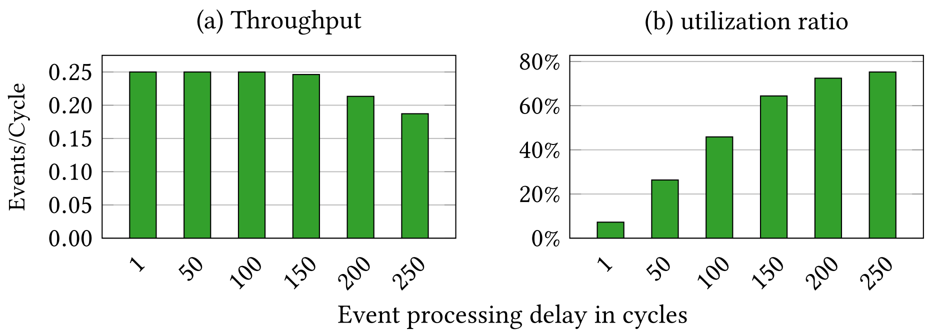

Fig. 11. Effect of variation of processing delays (in cycles) on (a) throughput and (b) ratio of core utilization for 64 event processors with 256 LP and 512 initial events.

memory accesses can also be thought of as the size of state memory read and updated during each event processing. The leftmost column in the plot shows the execution time without any memory access, which is small compared to the execution time with memory accesses. About 300 cycles are added for the first memory access. Each additional memory access adds about 50 cycles to the execution time. The changes in the average execution time are almost completely the result of the changes in memory access latency. It is apparent that the memory access latency does not scale linearly with the number or size of memory requested. Even if stalls are less frequent, each can take a long time to resolve. Thus, we believe that the memory system can issue multiple independent memory operations concurrently, leading to overlap in their access time. We have made the memory accessed by any event a contiguous region in the memory address space, which may also lead to DRAM side row-buffer hits and/or request coalescing at the memory controller. In an optimized event-processing logic, the processor may continue operation with partially available state memory to overlap computation and communication time.

5.5 Effect of Event-Processing Time

Figure 11 shows the effect of event-processing time on system performance. Since a computation can be synthesized differently in hardware (single-cycle combinational vs. multi-cycle sequential) with different resource usage footprints, a different processing time may be achieved for the same model. We use this experiment to analyze how different processing time affects performance to serve as a guideline for RTL design of the model. Since memory access latency is a major source that is currently not being hidden (and, therefore, adds a constant time to event processing), we configure a model that does not access memory in this experiment. We also allow the event-processing time to be artificially adjusted. The results of this study are shown in Figure 11(a). A higher processing time represents computations for models that have computationally intensive event processing. When the processing time is low, changes in the processing time do not reflect much in the system throughput since the system overheads lead to low utilization of the event-handling cores causing throttled speed. When processing time is higher, the utilization rises (Figure 11(b)), and increasing the event-processing time starts to lower the throughput. Thus, reducing processing time can improve performance, but up to a certain degree. Throughput gain becomes negligible for reduction of processing time beyond 150 cycles.

6 DATAPATH OPTIMIZATION VIA STRUCTURAL PARTITIONING

From the performance analysis of the baseline PDES-A simulation engine presented in the previous section, we observed that a key source of inefficiency is the overhead that results from multiple event processors contending for access to one of the shared processor components, such as the event list. This contention both hurts the performance of the system and presents a major barrier to future optimizations. In particular, Figure 13(a) shows that the contention at the interfaces of the processed events list, state memory, and event queue grows quickly as the number of event processors increases. With 64 event processors, through 30% of the total execution time, at least one processor is waiting for the processed events list to become available. The same happens for the event queue through 15% of total simulation time. We discover that going to a larger scale does not result in additional performance as expected since the shared structures are not able to meet the demands of the event processors. Additional problems can be observed in Figure 11(a), where we see that the throughput of the system remains flat at 0.25 events per cycle even when the event processing time is made very small. This indicates that these shared structures form a significant bottleneck at this scale, and memory optimizations (the other major bottleneck) will not be rewarded by a proportional improvement in performance due to contention.

Reducing wait times due to contention can result in substantial improvement in performance because contention delay creates an implicit positive feedback loop that can amplify the effect: waiting for resources increases event-processing time, which, in turn, causes longer stall for other conflicting events if present. Moreover, delay in processing events increases the probability that the resulting event will be a straggler event, which consequently creates more rollbacks, anti-messages, and more entries in processed event history, again increasing contention.

One approach for alleviating this problem is to increase the number of ports available for each resource. However, handling simultaneous requests requires an arbitration mechanism (e.g., cross-point switches) to allow event processors to access any of the available resources; such structures introduce significant hardware complexity. Moreover, while this approach is conceptually simple, implementation becomes difficult because of the increased complexity of synchronization in the presence of multiple communication paths. In contrast, the baseline design needs only simple synchronization since critical events were already being serialized as exchanges happen at one interface at a time. This serialization did not obstruct parallelization when the number of processors was smaller and concurrent accesses to any particular resource were rare. However, as the number of event processors increases, we observe sublinear throughput scaling as contention grows.

Instead of increasing the number of ports, we elected to re-architect the communication scheme in a way that simplifies the synchronization requirements, making the synchronization less tightly coupled. Our future vision for the framework also motivated this change: we plan to integrate multiple PDES-A engines on the same FPGA or even across multiple FPGAs to increase the throughput of the system. In such a setting, the system must be able to manage a high volume of remote events.

A centralized synchronization scheme would result in very high contention at event queues, making fast event exchange nearly impossible.

As discussed in Section 3.3, it is possible to drain events from the *event queue* without maintaining strict order because the chance of violating causality constraints is small among events near the top of the heap. When causality is violated, we have the rollback mechanism to fall back on and recover. Therefore, we built the event queue out of multiple smaller queues and created an interface for each of them to be accessed independently. From our simulations, we have observed that violating the order between events associated to the same LP causes rollbacks that keep cascading in the absence of order. Therefore, we map events for each LP to one queue to make sure that they remain ordered with respect to each other. We do not maintain order between different queues, and an event for LP A may be processed ahead of another at LP B even if it has a later timestamp when the LPs are mapped to different event queues.

We have added a *replace* capability to the queues (described in Section 3.3) so that *insert* and *delete* operations become independent of each other. All event processors were given the ability to push events to any of the queue interfaces based on the target LP using a crossbar. The task of governing event issue was separated into a controller that translates an event request from available event handlers to event issue instructions for the cores. This controller can, based on status of the event queues, optimize the event issue task to achieve minimum rollback and maximum utilization. The events issued from the queues are delivered to the recipient event handler using a broadcast network. A broadcast network is simpler than a more precise event delivery mechanism since the number of cores can be large, which complicates routing if a different network structure is used. The broadcast mechanism can be hierarchical and easily alterable to fit any system configuration efficiently. Thus, decoupling the task of insertion and removal of events makes it possible to separate the control and dataflow paths for the queues and reduces interdependence between the two tasks.

To adapt the *state memory* and *processed events history* for servicing multiple requests while maintaining memory consistency semantics, we partitioned the state space and events history with respect to LPs in a fashion similar to the event queue. This design ensures consistency without special mechanisms when considering the fact that the design guarantees that no read will be performed to data associated with an LP while another event processor is in the process of updating it. Similar to the event queue, requests are sent through a crossbar network and responses are broadcast to the core.

Essentially, this design creates different partitions for each LP while preserving their ability to share event processors. It also spreads contention to two stages: the crossbar followed by the partitioned queues, resulting in a higher throughput multi-stage network. As a result, the effective bandwidth of each of the shared structures is multiplied by the number of partitions since each of them can operate on an event independently. This relaxation in synchronization comes at the price of relaxing strict (sequential) event processing across partitions, which can result in additional rollbacks. However, in practice, since the design keeps the LPs within similar simulation time of each other, this effect is minimal.

6.1 Decoupled Event-Processing Flow

Figure 12 shows the event-processing flow in the optimized design. An event goes through three different phases throughout its life cycle.

Issue Phase: At any given moment, a list of idle event processors is available to the issue controller. Whenever any processor is idle, the issue control logic requests events from the event queue on behalf of the idle event processors in step 1. In step 2, the event at the head of the queue

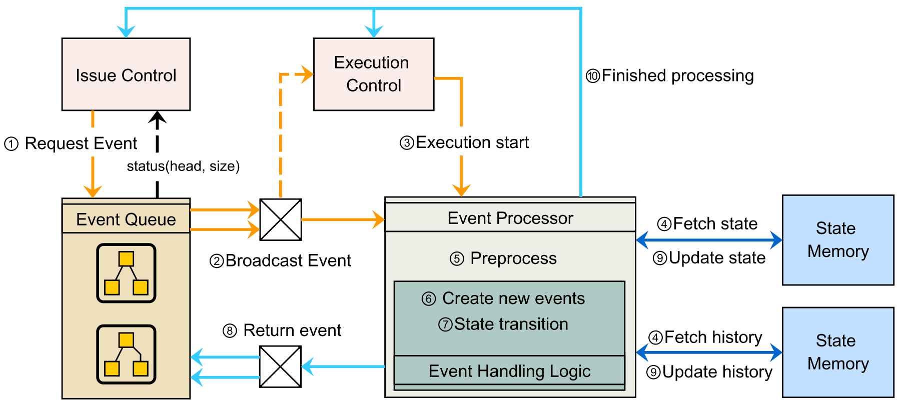

Fig. 12. Overview of an event-processing cycle.

is broadcast to a bus connected to all processors. The targeted event processor picks up the event from the broadcast bus. An execution controller always monitors the event broadcast activity and keeps a record of the LP-processor association. The controller checks whether it is safe to process the event and signals the event processor in step 3 to start execution when it is not going to create any conflicts with other event processors.

Compute Phase: The event processor, upon receiving this signal, fetches state memory and the processed events list in step 4. The event history is checked in the preprocessing step (step 5) to determine whether any rollback or cancellation is necessary. At the same time, the processor cleans up the stale history entries from the event history list. The event data, state memory, and event types are then delivered to the model-specific event-handling logic provided by the user. Depending on the event type, in steps 6 and 7, the event handler performs rollback computation if necessary, computes new states based on the model, and creates the next set of events along with any anti-messages generated due to rollback.

Apply Phase: When the event-handling logic returns, the event processor does the necessary clean up by pushing the new event to the appropriate queues in step 8. At the same time, the updates to state memory and the processed events list are written to the memory in step 9. At this point, event processing is complete and the event processor notifies the execution controller and issue controller in step 10 so that their internal records can be updated. Then, the processor prepares to receive a new event.

6.2 Operational Characteristics

Figure 13 shows different performance measures of the optimized design along with the baseline design at different point of scaling. We see in Figure 13(a) that contention at the interfaces was almost completely eliminated. The effect of this is apparent in the plot of cycles required on average for event processing shown in Figure 13(b). The average event-processing time decreases to the same level as it was for 16 event processors in parallel. The time spent stalling to avoid conflicts also reduces as a result of faster event-processing time. Consequently, the system achieves better utilization ratio, as can be seen in Figure 13(c), which shows improvement in the fraction of time that a processor remains active. Finally, all of these effects combine into significant improvement in throughput.

This organization is highly throughput oriented and almost completely removes interdependence among the dataflow paths. The only remaining source of divergence is straggler events.

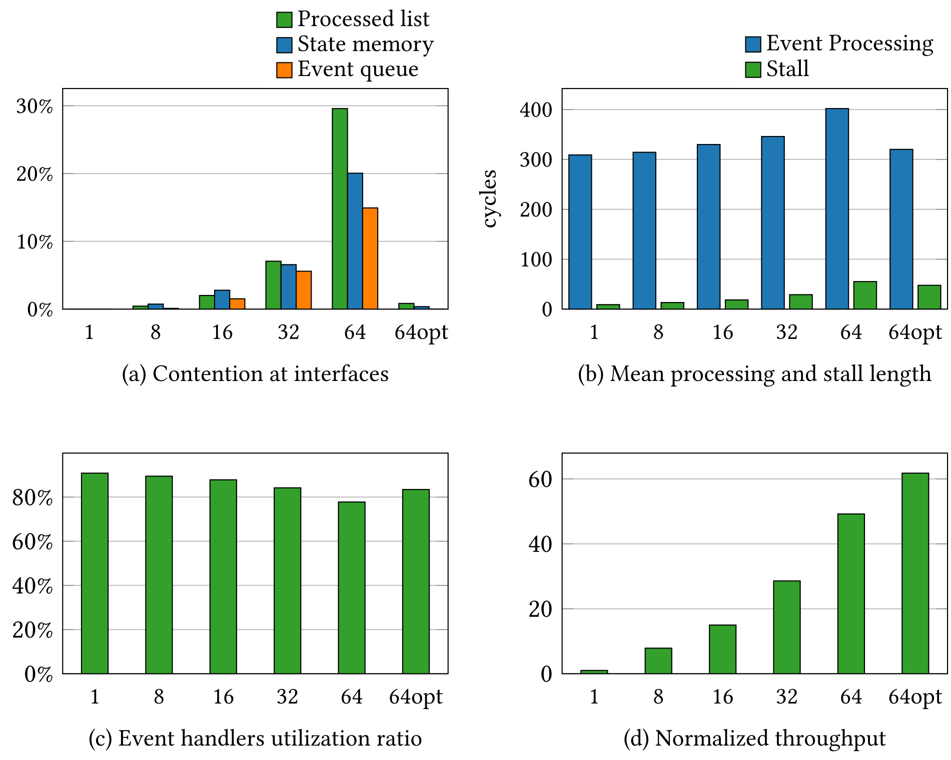

Fig. 13. Effects of optimized dataflow and concurrent resource access on accelerator performance compared with the baseline engine with different numbers of event handlers.

This organization restores the throughput to almost linear scaling. With 64 cores, this organization achieves approximately 62 times the throughput of a single core (Figure 13(d)), where the baseline design shows throughput dwindling to only 49x. We expect the design to be scalable to a higher number of event processors given the reduction in contention.

7 COMPARISON WITH ROSS

To provide an idea of the performance of PDES-A relative to a CPU-based PDES simulator, we compared the performance of PDES-A with MPI-based PDES simulator ROSS [6].

We urge the readers to note that a simple comparison between a software framework and hardware cannot be taken as a serious benchmark. In a realistic application, major performance gain for the hardware accelerator will come from the superiority of hardware primitives. For example, many mathematical and scientific libraries require floating point numbers and vector computations, long iterative operations, and traverse many conditional branches. A programmer can reduce these expensive tasks to only a few cycles with hardware support. This will contribute to massive throughput gain in hardware compared with a conventional CPU.

The purpose of our comparison with ROSS is to establish that the base framework has comparable performance with software. It also helps us estimate the relative complexity of the models from their software evaluation and use that knowledge in hardware analysis. For this reason, Phold is a good choice of benchmark because it models the underlying operations without being burdened by application-specific logic.

Table 1. Summary of Configuration Used for Performance Comparison of ROSS and PDES-A Using Phold

Parameters	ROSS	PDES-A
System		
Device	Intel Xeon E5-1650 12MB L2	Xilinx Virtex-7 XC7V2000T
Frequency	3.50GHz	150MHz
Memory	32GB	32GB
Simulation		
PE	72 (12 cores× 6 KP)	64
LP	252	256
Event Density	504	512
Remote Events	5%	100%

Table 2. Comparative Analysis of PDES Simulation Performance of Phold and Airport Models on ROSS and PDES-A

Performance	Phold			Airport	
	ROSS	Basic PDES-A	Opt. PDES-A	ROSS	Opt. PDES-A
Events/second	9.2 mil	23.85 mil	29.98 mil	5.7 mil	8.7 mil
Commit Efficiency	80%	~100%	~100%	83%	100%
Power Estimate	130 Watt	~17.8 Watt	~18.5 Watt	130 Watt	~18.5 Watt

Although the modeling flow for the two environments is quite different, we configured ROSS to run the Phold model with similar parameters to the PDES-A model. We changed the Phold model in ROSS to resemble our system by replacing the exponential timestamp distribution with a uniform distribution. We set the number of processing elements, LPs, and number of events to match our system closely. One particular difference is in the way that remote events are generated and handled in ROSS. In our system, all cores are connected to a shared set of LPs; thus, there is no difference between local and remote events. In ROSS, remote events have to suffer the extra overhead of message passing in MPI, although MPI uses shared memory on a single machine. We set the remote event threshold in ROSS to only 5% to allow marginal communication between cores.

Table 1 shows the parameters for both systems used in comparison. Their performance reported in Table 2 includes the numbers for both the baseline and optimized architecture. At this configuration, baseline PDES-A can process events 2.5x faster than a 12-core CPU version of ROSS; after optimization, the advantage grows to 3.2x. When the remote event percentage in ROSS is higher, ROSS performance suffers and the PDES-A advantage increases, gaining up to 15x for 100% remote messages. We believe that as we continue to optimize PDES-A, this advantage will be even larger.

We also run the *Airport* model in the partitioned version of the accelerator to compare performance in the presence of multiple types of events. We achieve about 1.5x performance gain over ROSS. The gain drops compared with the Phold model. LPs in this model send two-thirds of the events to self. Therefore, the processors are often processing the same LPs and have to stall more to avoid conflicts. This result highlights the need to implement new strategies to reduce stalls, such as interrupt-based preemption and workload reassignment. We are exploring these optimizations for the next iteration of our design.

Table 3. FPGA Resource Utilization for Optimized Datapath PDES-A

Component	LUT (1221600)		FF (2443200)		BRAM (1203)	
	Used	% Util.	Used	% Util.	Used	% Util.
Simulator	74670	6.11%	56115	2.30%	8	0.67%
Event Processor (each)	367	0.03%	211	0.01%	0	0%
Controller	3610	0.30%	5557	0.23%	0	0%
Event Queue (each)	4488	0.37%	1402	0.06%	0	0%
Memory Interface	116799	9.56%	4748	0.19%	222	18.45%
Crossbar Network	15757	1.29%	28192	1.15%	32	2.66%
Overall	300695	24.61%	320728	13.13%	271	22.53%

8 RESOURCE UTILIZATION ANALYSIS AND SCALING ESTIMATES

In this section, we first present an analysis of the area/utilization requirements of PDES-A. The FPGA resources utilization by the cores is presented in Table 3. The overall system takes over about 25% of the available LUTs in the FPGA. The larger portion of this is consumed by the memory interface and other static coprocessor circuitry that will remain constant when the simulator size scales. The core simulator logic uses 6.11% of the device logics. Each individual Phold event processor contributes to less than 0.03% resource usage. Register usage is less than 3% in the simulator. We can reasonably expect to replicate the simulation cluster more than 10 times in an FPGA, even when a more complex PDES model is considered and networking overheads are taken into account. This would put 640 cores in the coprocessor. The simulator offers good raw computing potential if it can be scaled up to this extent.

Finally, an inherent advantage of FPGAs is their low power usage. The estimated power of PDES-A was less than 18 Watts in contrast to the rated 130 Watts TDP of the Intel Xeon CPU. We believe that this result shows that PDES-A holds promise to uncover a significant boost in PDES simulation performance.

FPGA designs are limited by a lot of engineering constraints. Even when scaling shows a promising trend of performance increase, sometimes it is undesirable to simply increase the design size: since the design has to be physically synthesized using limited resources of the chosen FPGA, routing complexity puts a limit on how large a tightly coupled module can be.

In our design, we observe that without sufficient event saturation and higher number of LPs, the amount of time spent in stall and the possibility of causality violation becomes prohibitively large, reducing commit efficiency, and can slow the system down (Figure 7(a)). Increasing the number of events is relatively simple. Either the queue sizes need to be increased to accommodate more events or queues should be spilled into the main memory in case of overflow. The first approach requires a linear increase in on-chip memory usage; the second chokes the system if it occurs repeatedly. Increasing the number of LPs also requires proportional increase in memory for states and processed event history. The on-chip memory is a limited resource that is scattered throughout the chip. One cannot simply use as many of them as needed in a compact design because the routing complexity will prevent synthesis at optimum frequency.

However, the routing complexity becomes most prominent when scaling the number of event processors because they need to be connected to the same interfaces and synchronization mechanisms. The custom logic has to be physically synthesized alongside the common logic. However, the design is then highly likely to fail timing constraints. A good engineer with enough perseverance can probably make any configuration work by using hierarchies and buffers, but this contradicts our design objective of making a portable framework with minimal user input. Proper partitioning of the design is crucial in ensuring that the design should work after customizations [36].

We found that the optimal size for a PDES-A engine is 64 event processors. At this scale, the engine remains tightly coupled with sufficient parallelism while remaining capable of synthesizing any model-specific logic. It should be noted that this observation is purely empirical in nature and the number should increase for different generations of more powerful FPGAs. Each PDES-A engine would be equivalent to a design partition interconnected in a larger simulation environment. Designers may also choose to develop an Application-Specific IC (ASIC) version of their model using PDES-A as base, in which case we expect PDES-A to continue to scale to reach either the limit on parallelism within the model or the memory bandwidth of the chip.

9 CONCLUDING REMARKS

In this article, we presented and analyzed the design of a PDES accelerator on an FPGA. PDES-A is designed to allow supporting arbitrary PDES models, although we studied our initial design only with Phold. The design shows excellent scalability up to 64 concurrent event handlers, outperforming a 12-core CPU PDES simulator by 3.2x for this model. We identified major opportunities to further improve the performance of PDES-A targeted around hiding the very high memory latency on the system. We also analyzed the resource utilization of PDES-A: we believe that we can fit more than 10 PDES-A processors with 64 event-processing cores on the same FPGA chip, further improving performance at a fraction of the power consumed by CPUs.

Our future work spans at least three different directions. First, we will continue to optimize PDES-A to reduce the impact of memory access time and resource contention. Next our goal is to study a full chip (or even multi-chip) design consisting of multiple PDES-A accelerators working on larger models. Finally, we hope to provide programming environments that allow rapid prototyping of PDES-A cores specialized to different simulation models.

REFERENCES

[1] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*. ACM, New York, NY, 1216–1225. DOI: https://doi.org/10.1145/2228360.2228584

[2] R. Bhagwan and B. Lin. 2000. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings of IEEE INFOCOM 2000. Conference on Computer Communications. 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 2. IEEE, Tel Aviv, Israel, 538–547. DOI: https://doi.org/10.1109/INFCOM.2000.832227

[3] R. Brown. 1988. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM* 31, 10 (Oct. 1988), 1220–1227. DOI: https://doi.org/10.1145/63039.63045

[4] J. Burt. 2016. Intel Begins Shipping Xeon Chips With FPGA Accelerators. Retrieved February 2017 from http://www.eweek.com/servers/intel-begins-shipping-xeon-chips-with-fpga-accelerators.html.

[5] Christopher D. Carothers. 2018. ROSS-Models. Retrieved January 31, 2019 from https://github.com/carothersc/ROSS-Models.

[6] Christopher D. Carothers, David Bauer, and Shawn Pearce. 2000. ROSS: A high-performance, low memory, modular time warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS'00)*. IEEE Computer Society, Washington, DC, 53–60. http://dl.acm.org/citation.cfm?id=336146.336157

[7] Guillaume Chapuis, Stephan Eidenbenz, Nandakishore Santhi, and Eun Jung Park. 2015. Simian integrated framework for parallel discrete event simulation on GPUs. In *Proceedings of the 2015 Winter Simulation Conference (WSC'15)*. IEEE Press, Piscataway, NJ, 1127–1138. http://dl.acm.org/citation.cfm?id=2888619.2888742

[8] Huilong Chen, Yiping Yao, Wenjie Tang, Dong Meng, Feng Zhu, Yuewen Fu, and Yiping Yao. 2015. Can MIC find its place in the field of PDES? An early performance evaluation of PDES simulator on Intel many integrated cores coprocessor. In *Proceedings of the 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT'15)*. IEEE Press, Piscataway, NJ, 41–49. DOI: https://doi.org/10.1109/DS-RT.2015.23

[9] Convey Computers Corporation. 2013. *The Convey WX Series* (conv-13-045.5 ed.). https://www.micron.com/-/media/client/global/documents/products/product-flyer/conv13045,-d-,5-wolverine_r1b.pdf.

[10] Convey Computers Corporation. 2014. Convey Wolverine® Application Accelerators Architectural Overview (CONV-14-049.1 ed.). https://www.micron.com/-/media/client/global/documents/products/white-paper/wp_conv14049,-d-,1_wolverine_arch_overview.pdf.

[11] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. 1994. GTW: A time warp system for shared memory multiprocessors. In *Proceedings of the 26th Conference on Winter Simulation (WSC'94)*. Society for Computer Simulation International, San Diego, CA, 1332–1339. http://dl.acm.org/citation.cfm?id=193201.194885

[12] Richard Fujimoto. 2015. Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference (WSC'15)*. IEEE Press, Piscataway, NJ, 45–59. http://dl.acm.org/citation.cfm?id=2888619.2888624

[13] Richard M. Fujimoto. 1999. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY.

[14] Richard M. Fujimoto, Jya-Jang Tsai, and Ganesh C. Gopalakrishnan. 1992. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Trans. Comput.* 41, 1 (Jan. 1992), 68–82. DOI: https://doi.org/10.1109/12.123382

[15] Sounak Gupta and Philip A. Wilsey. 2014. Lock-free pending event set management in time warp. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS'14)*. ACM, New York, NY, 15–26. DOI: https://doi.org/10.1145/2601381.2601393

[16] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 37–47. DOI: https://doi.org/10.1145/1815961.1815968

[17] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, NY, 3–12. DOI: https://doi.org/10.1145/1941553.1941557

[18] M. C. Herbordt, F. Kosie, and J. Model. 2008. An efficient O(1) priority queue for large FPGA-based discrete event simulations of molecular dynamics. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Palo Alto, CA, 248–257. DOI: https://doi.org/10.1109/FCCM.2008.49

[19] Hybrid Memory Cube Consortium. 2014. *Hybrid Memory Cube Specification 2.1* (2.1 ed.). http://hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.1_20151105.pdf.

[20] Amazon Web Services, Inc. 2018. Amazon EC2 F1 Instances. Retrieved January 31, 2019 from https://aws.amazon.com/ec2/instance-types/f1/.

[21] Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2012. Characterizing and understanding PDES behavior on Tilera architecture. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*. IEEE Computer Society, Washington, DC, 53–62. DOI: https://doi.org/10.1109/PADS.2012.10

[22] David R. Jefferson. 1985. Virtual time. *ACM Trans. Program. Lang. Syst.* 7, 3 (July 1985), 404–425. DOI: https://doi.org/10.1145/3916.3988

[23] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*. ACM, New York, NY, 228–241. DOI: https://doi.org/10.1145/2830772.2830777

[24] Ranjit Noronha and Nael B. Abu-Ghazaleh. 2002. Early cancellation: An active NIC optimization for time-warp. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS'02)*. IEEE Computer Society, Washington, DC, 43–50. http://dl.acm.org/citation.cfm?id=564062.564070

[25] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 5–14. DOI: https://doi.org/10.1145/3020078.3021740

[26] Hyungwook Park and Paul A. Fishwick. 2010. A GPU-based application framework supporting fast discrete-event simulation. *Simulation* 86, 10 (Oct. 2010), 613–628. DOI: https://doi.org/10.1177/0037549709340781

[27] Alessandro Pellegrini and Francesco Quaglia. 2014. Transparent multi-core speculative parallelization of DES models with event and cross-state dependencies. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS'14)*. ACM, New York, NY, 105–116. DOI: https://doi.org/10.1145/2601381.2601398

[28] Kalyan S. Perumalla. 2006. Discrete-event execution alternatives on general purpose graphical processing units (GPGPUs). In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*. IEEE Computer Society, Washington, DC, 74–81. DOI: https://doi.org/10.1109/PADS.2006.15

[29] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of parallelism in algorithms. *SIGPLAN Not.* 46, 6 (June 2011), 12–25. DOI: https://doi.org/10.1145/1993316.1993501

[30] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*. IEEE Press, Piscataway, NJ, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[31] Shafiur Rahman, Nael Abu-Ghazaleh, and Walid Najjar. 2017. PDES-A: A parallel discrete event simulation accelerator for FPGAs. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'17)*. ACM, New York, NY, 133–144. DOI: https://doi.org/10.1145/3064911.3064930

[32] Joseph Rios. 2007. *An Efficient FPGA Priority Queue Implementation with Application to the Routing Problem*. UC Santa Cruz Technical Report. University of California, Santa Cruz, Santa Cruz, CA. https://www.soe.ucsc.edu/research/technical-reports/UCSC-CRL-07-01

[33] Robert Rönngren and Rassul Ayani. 1997. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation* 7, 2 (1997), 157–209.

[34] N. Santhi, S. Eidenbenz, and J. Liu. 2015. The Simian concept: Parallel discrete event simulation with interpreted languages and just-in-time compilation. In *2015 Winter Simulation Conference (WSC'15)*. IEEE, Huntington Beach, CA, 3013–3024. DOI: https://doi.org/10.1109/WSC.2015.7408405

[35] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Press, Piscataway, NJ, Article 17, 12 pages. http://dl.acm.org/citation.cfm?id=3195638.3195659

[36] Philip Andrew Simpson. 2015. *FPGA Design*. Springer International Publishing, Cham. http://link.springer.com/10.1007/978-3-319-17924-7 DOI: 10.1007/978-3-319-17924-7

[37] Jeffrey S. Steinman. 2005. The WarpIV simulation kernel. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*. IEEE Computer Society, Washington, DC, 161–170. DOI: https://doi.org/10.1109/PADS.2005.32

[38] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. 2010. RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In *Proceedings of the 47th Design Automation Conference (DAC'10)*. ACM, New York, NY, 463–468. DOI: https://doi.org/10.1145/1837274.1837390

[39] Wenjie Tang and Yiping Yao. 2013. A GPU-based discrete event simulation kernel. *Simulation* 89, 11 (Nov. 2013), 1335–1354. DOI: https://doi.org/10.1177/0037549713508839

[40] Jingjing Wang, Deepak Jagtap, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2014. Parallel discrete event simulation for multi-core systems: Analysis and optimization. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2014), 1574–1584.

[41] Jingjing Wang, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2012. Performance analysis of a multithreaded PDES simulator on multicore clusters. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*. IEEE Computer Society, Washington, DC, 93–95. DOI: https://doi.org/10.1109/PADS.2012.33

[42] Barry Williams, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Philip Wilsey. 2017. Performance characterization of parallel discrete event simulation on Knights Landing processor. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'17)*. ACM, New York, NY, 121–132. DOI: https://doi.org/10.1145/3064911.3064929

[43] S. Zhou, C. Chelmis, and V. K. Prasanna. 2016. High-throughput and energy-efficient graph processing on FPGA. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'16)*. IEEE, Washington, DC, 103–110. DOI: https://doi.org/10.1109/FCCM.2016.35

Received December 2017; revised September 2018; accepted December 2018