PDES-A: a Parallel Discrete Event Simulation Accelerator for FPGAs

Shafiur Rahman
University of California
Riverside
mrahm008@ucr.edu

Nael Abu-Ghazaleh
University of California
Riverside
nael@cs.ucr.edu

Walid Najjar
University of California
Riverside
najjar@cs.ucr.edu

ABSTRACT

In this paper, we present initial 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 logic, 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 together to scale up the simulation. We evaluate the design and explore several design tradeoffs and optimizations. We show the accelerator can scale to 64 concurrent event processors relative to the performance of a single event processor.

Keywords

PDES, FPGA, accelerator, coprocessor, parallel simulation

1. INTRODUCTION

Discrete event simulation (DES) is an important application used in the design and evaluation of systems and phenomena where 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 this paper, we present an initial exploration of a general Parallel Discrete Event Simulation (PDES) accelerator implemented on an FPGA. In recent years, many researchers have developed and analyzed PDES simulators for 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 development of PDES kernels optimized for these environments such as GTW [7], ROSS [3] and WarpIV [29]. The recent emergence of multi-core and many-core processors has attracted considerable interest among the high-performance computing communities to explore PDES in these emerging platforms. Typically, these simulators [32, 10, 26] 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. Using similar insights, PDES has been shown to scale well on many-core architectures such as the Tilera Tile64, the Intel Xeon Phi (also known as Many Integrated Cores or MIC) as well as GPGPUs. Several researchers have explored the use of GPGPUs to accelerated PDES [19, 31, 20]. Similarly, Jagtap et al. explored the performance of PDES on the Tilera Tile64 [15], while Chen et al. studied its performance on the Intel Xeon Phi coprocessor [4, 34].

In contrast to these effort, relatively fewer works have considered acceleration of PDES using non-conventional architectures such as FPGAs, motivating our study. 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 promising area of future progress. Many types of accelerators have already been proposed for a large number of important applications such as deep learning [28] and graph processing [35]. Thus, the exploration of accelerator organization for PDES informs possible design of custom accelerators for important simulation applications.

An FPGA implementation of PDES offers several possible advantages.

- Fast 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 [33]. On the other hand, often the memory latency experienced by FPGAs is high (but the available bandwidth is also high), necessitating approaches to hide the memory access latency.
- Specialized, high-bandwidth datapaths: General pur-

SIGSIM-PADS '17, May 24–26, 2017, Singapore.

DOI: http://dx.doi.org/10.1145/3064911.3064930

pose processing provides high flexibility but at the price of high overhead and a fixed datapath. A specialized accelerator in contrast, can more efficiently implement a required task without 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 [11]. Moreover, an FPGA can allow high parallelism limited only by the number of processing units and the communication bandwidth available between them, as well as the memory bandwidth available to the FPGA chip.

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 [21]. OIP applications have inherent parallelism that is difficult to exploit in a traditional multiprocessor 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 [17]. Run-time overheads such as those of communication limit the scalability of PDES [12]; these overheads may be both lower and more easily maskable in the context of an FPGA[33]. For models where event process is computationally expensive, 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, for many reasons, FPGAs have exceptional energy properties compared to GPGPUs and many-cores.

We present our initial design of a PDES accelerator (PDES-A). We show that PDES-A can provide excellent scalability for Phold up to 64 concurrent event processors. Our initial prototype outperforms a similar simulation on a 12-core 3.5GHz Intel Core i7 CPU by 2.5x. We show that there remains several opportunities to optimize our design further. Moreover, we show that multiple PDES-A accelerators can fit within the same FPGA chip, allowing us to further scale the performance.

FPGA based accelerator development platforms have recently progressed rapidly to make FPGA based accelerators available to all programmers. Microsoft's Catapult [22] and the Convey Wolverine [5] are examples of recent systems that offer integrated FPGAs with programmability, tight integration, advanced communication and memory sharing with CPU in industry standard HPC clusters. After the acquisition of Altera last year, Intel has already started shipping versions of its Xeon processors with integrated FPGA support [2]. Modern memory technologies such as Micron's Hybrid Memory Cube [14] can offer up to 320GB/s effective bandwidth providing excellent bandwidth to applications requiring high memory demand such as PDES. However, specialized hardware support is required to take advantage of this bandwidth. The recent take over of Convey Computing by Micron paved the way to have Hybrid Memory Cube in FPGA based coprocessors. Potentially, an FPGA implementation can yield high performance and low-power PDES accelerators as well as inform the design of custom accelerators for PDES.

Our work is in the vein of prior studies that explored customized or programmable hardware support for PDES. Fujimoto et al. propose the Rollback chip, a special purpose processor to accelerate state saving and rollbacks in Time Warp[9]. In the area of logic simulation and computer simulation, the use of FPGA offers opportunities for performance since the design being simulated can simply be emulated on the FPGA [30]. Noronha and Abu-Ghazaleh explore the use of a programmable network interface card to accelerate GVT computation and direct message cancellation [18]. Similarly, Santoro and Quaglia use a programmable network interface card to accelerate checkpointing for optimistic simulation [27]. Our work differs in the emphasis on support of complete general optimistic PDES. Most similar to our work, Herbrodt et al. [13] explore FPGA implementation of a specific PDES model for molecular dynamics, but the design is specialized to this one application rather than supporting general simulation.

The remainder of this paper is organized as follows. We use section 2 to present some background information related to PDES and introduce the Convey Wolverine FPGA system we use in our experiments. Section 3 introduces the design of our PDES-A accelerator and its various components. Section 4 overviews some implementation details and the verification of PDES-A. Section 5 presents a detailed performance evaluation of the design. In section 6 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 7 presents some 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 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 where 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 PDES, allowing the simulation of larger, more detailed models, and the consideration of more scenarios, in a shorter amount of time [8].

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 time stamp 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.

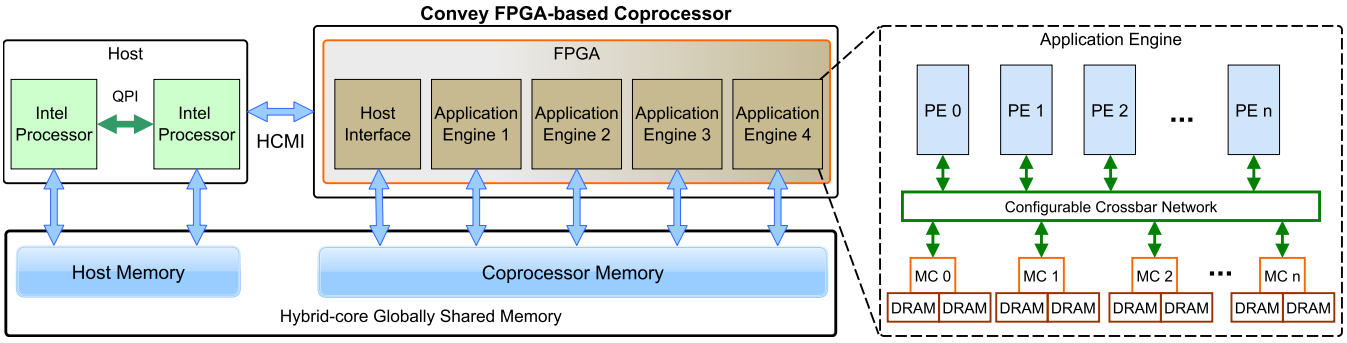

Figure 1: Overview of the Convey Hybrid-Core architecture

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; in other words, the model property determines a time window in which events cannot be generated due to the simulation time delay between processing an event and any future events it schedules. This synchronization imposes a overhead on the PEs to continue to advance. In contrast, optimistic simulation algorithms such as Time Warp [16] allow PEs to process events without synchronization. As a result, it is possible for an LP to receive a *straggler* event with a time stamp 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 the straggler event. The rollback may require the LP to cancel out any event messages it generated erroneously using *anti-messages*. This approach uses more memory for keeping checkpoint information, which need to be garbage collected when they are 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 GVT can be garbage collected, and events earlier than 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 standard FPGAs coupled with a standard x86 host interface to communicate with the host processor. The system also includes a standard Xeon based CPU integrated with the Convey Wolverine WX2000 coprocessor.

The Wolverine WX2000 integrates three major subsystems: Application Engine Hub (AEH), Application Engines (AEs), and the Memory Subsystem (MCs) [5]. 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 inside a Xilinx Virtex-7 XC7V2000T FPGAs. The AEs are connected to memory controllers via 10GB/s point-to-point network links. In an optimized implementation, up to 40GB/s of bandwidth is available to the system. The clock rate of the FPGAs is much lower than that of the CPU (150MHz), but they can implement many specialized functional units in parallel. When utilizing the memory bandwidth properly, the throughput can be many times that of a single processor. This is why the system is ideal for applications benefiting from high computation capability and large high-bandwidth memory. Each of the Application Engines are dedicated FPGAs that can be programmed with same or different application. 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 which allows any processing element to access any 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 8 DDR2 memory channels providing a high bandwidth, but also high-latency, connection between memory and application engines [6]. The memory subsystem provides simplified logical memory interface ports that connects to a crossbar network which in turn connects to the physical memory controller. Programmers can use the memory interface ports in any implementation.

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 virtual address. The memory subsystem implements the address translation, crossbar routing, and configuration circuits. Both the memory subsystem and the application engines hub are provided by the vendor and their implementation remains the same in all designs. Note that there is a substantial difference in the latency and bandwidth between accesses to local memory and accesses to remote (CPU) memory.

The architecture of the Convey system can present some advantages in the design of a PDES system. The event processing logic in PDES are simple for many models. Memory access latency and communication overhead usually prevent the system 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. Leveraging the standard x86 interface, multiple Convey servers can be interconnected, which opens up the possibilities to scale up the PDES system to a large cluster based implementation. And finally, the global shared memory architecture allows the host processor to easily initialize and observe the simulator.

3. PDES-A DESIGN OVERVIEW

In this section, we present an overview of the unit PDES accelerator (PDES-A). 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 scalability constraints invite a design where multiple interconnected PDES-A accelerators together work on a large simulation model, and exploiting the full available FPGA resources. In this paper, we explore and analyze only PDES-A, and not the full architecture consisting of many PDES-A accelerators. In the design of PDES-A, we are careful to modularize it to facilitate integration with other PDES-A accelerators or any PDES execution engines that are compatible with its event API.

In an FPGA implementation, event processing, communication, synchronization, and memory access operations occur in a way different from how these operations occur on 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

The goal of the design is to provide a general PDES accelerator, rather than an accelerator for a specific model or class of models. We expect that knowledge of the model can be exploited to fine tune the performance of PDES-A, but we did not pursue such opportunities. To support this generality, PDES-A provides a modular framework where various components can be adjusted independently to attain the most effective data path 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 assumption about event execution time. We decided to implement an 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.

3.2 General Overview

The overall design of PDES-A is shown in Figure 2. The simulator is organized into four major components: (1) Event Queue, which stores the pending events; (2) Event Processors: custom datapaths for processing the event types in the model; (3) System State Memory: holds relevant system state, including checkpointing information; (4) and the Controller: which coordinates all aspects of operation. The first three components correspond to the same functionality in traditional PDES engines in any discrete event simulator, and the last one oversees the event processors to ensure correct parallel operation and communication. We will look into how these components fit together in a PDES system first, and then into their implementation.

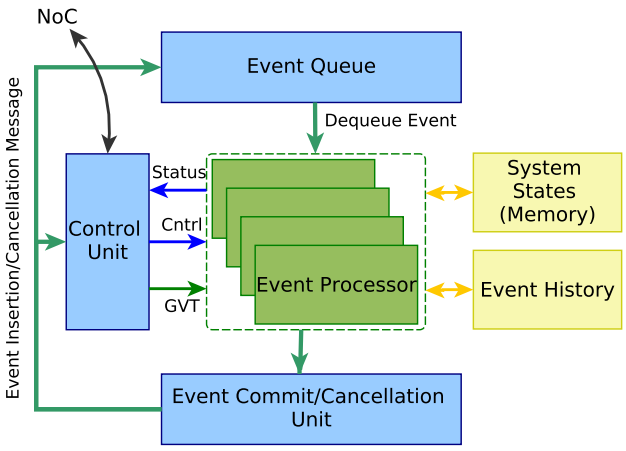

Figure 2: PDES-A overall system organization

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. Please note that the architecture can be modified to support conservative simulation while preserving most of its structure because of the decision to support general message passing. A conservative version of PDES-A requires changes to message types and the controller (different dispatch logic, and replacing GVT with synchronization), while eliminating the checkpointing information; we did not build a conservative version of PDES-A.

Figure 2 shows the major components of PDES-A and their interactions. The event queue contains a sorted list of all the unprocessed events. 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 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

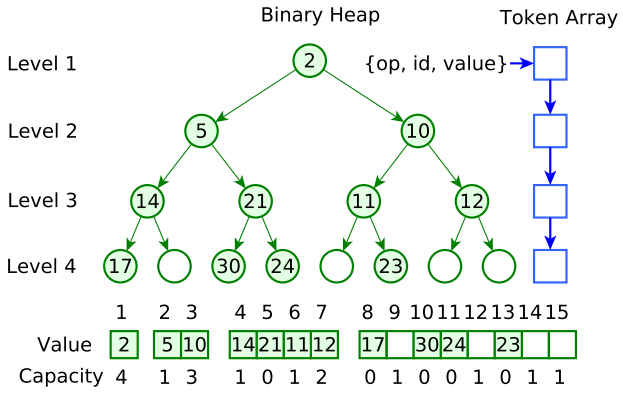

Figure 3: The P-heap data structure[23]

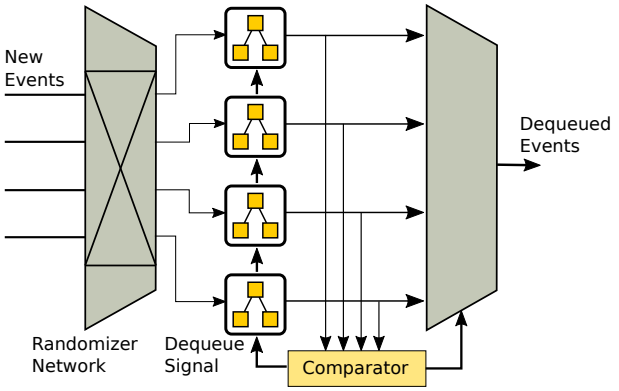

Figure 4: Multiple event issue priority queue

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 make early cancellation possible for straggler events that have not been processed yet and still reside in the queue.

The event queue structure and its impact on PDES performance has been studied in the context of software implementations [25]; however, it's 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 requires $O(log(n))$ time for enqueue and dequeue operations. Other options have other drawbacks; for example, Calendar Queues [1] support $O(1)$ access time but are difficult and expensive to scale in a hardware implementation. QuickQ [24] uses multiple dual-ported RAM in a pipelined structure which provides easy scalability and to support constant time access. However, the access time is proportional to the size of each stage of RAM. Configuring these stages 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 in short) structure as the basic organization in our implementation [23], except for a few modifications which we describe later. P-heap uses a pipelined binary heap to provide two cycles constant access time (to initiate an enqueue or dequeue operation), 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 sub-tree rooted at the node (Figure 3). The capacity values are used by *insert* operations to find the path in the heap that it should percolate through. 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: the smaller value replaces the target node value and the 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 of its child nodes. The same operation continues to recurse 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 [23].

P-heap can be efficiently implemented in hardware on an FPGA. Every stage requires *a Dual Ported RAM*, which is a memory element having one write port and two read port. Depending on the size of that stage, it can be synthesized with registers, distributed RAM, or block RAM elements to maximize resource utilization. An arbitrary number of stages can be added (limited by block RAM resource availability) as the performance is not hurt by the number of stages in the heap due to pipelining, 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 relaxation 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 that used by Herbordt et al.[13], we can use a randomizer network to direct multiple requests to multiple available heap (Figure 4). There is a chance that two of the highest priority events may reside in the same heap and ordering violation will occur at the queue during multiple dequeue. However, when the number of LPs and PEs are large, such occurrence which is handled by the rollback mechanism would be rare resulting in a net performance gain. Although we have a version of this queue implemented, we report our results without using it. Other structures that sacrifice full order-

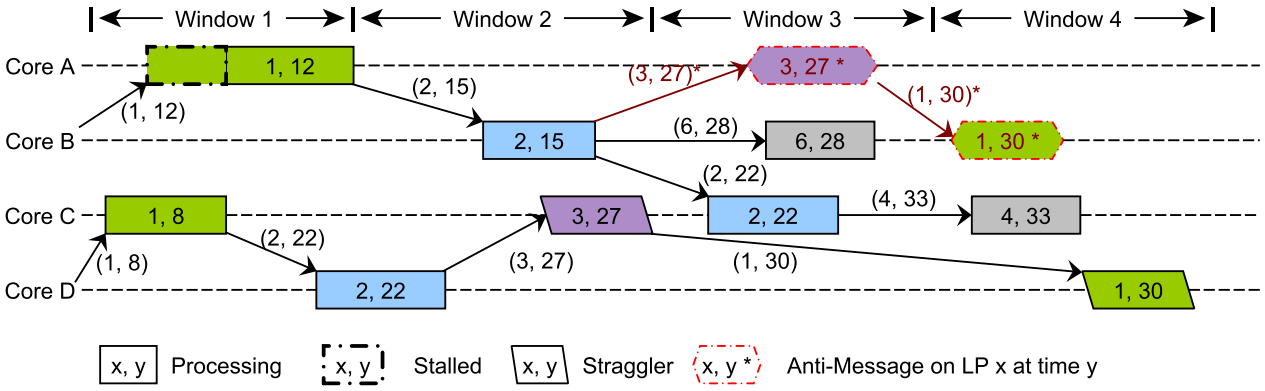

Figure 5: Simplified timeline representation showing scheduling of events in the system.

ing but admit higher parallelism such as Gupta and Wilsey's lock-free queue may also be explored [10].

The queue stores a key-value pair. We use 64 bit entries with event time-stamp acting as the key. The value contains the id of the target LP and a payload message. In cases where the payload message is too large, we store a pointer to a payload message in memory. For the Phold model we use in our evaluation, all messages fit in the default value field.

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 which 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 pre-configured interface. The event processor passes event message and relevant data to the core logic by populating FIFO buffers. Once the events are processed, the core logic uses output buffers to store any 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.

The model we use in our evaluation (Phold) has only one type of event handler, simplifying the mapping of events to handlers. However, in models where multiple event handlers exist, interesting design decisions arise about whether to specialize the event processors to each event type, or to create more general, but perhaps less efficient event handling engines. If a reasonable approximation of the distribution of the task frequency in knows, the numbers of different kind of event processors may be tuned to the requirement of the model to maximize resource utilization. It is also possible to create a mix of specialized handlers for common events, and more general handlers to handle rare events. We will consider such issues in our future work.

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; on the figure, the event is represented by a tuple (x, y) where x is the LP number and y is the simulation time. Event (1, 8) is scheduled to core C. A second event (1, 12) belonging to the same LP is scheduled to core A while event (1, 8) is still being processed. Because of the dependency, core A is stalled by the controller unit until the first event completes. At the completion of an event, the controller allows the earliest timestamp among the waiting (stalled) events for that core to proceed as shown in window 1. Each event may generate one or more new events when it terminates. These events 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*). When discovering this causality error, the erroneously processed events need to be rolled back to restore causality. Window 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 event history table 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 (anti-message (3, 27*) in this example) for all events generated by the straggler event, and new event (2, 22) that reschedules the cancelled event.

The anti-messages may get processed before or after its 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 necessary anti-messages (1, 30*) to *chase* the erroneous message chain (i.e., cascading rollbacks) much like a regular events 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 discov-

ery of the anti-message in the history and no new event is generated as shown in window 4.

4. IMPLEMENTATION OVERVIEW

We used a full RTL implementation on Convey WX-2000 accelerator for prototyping the simulator. The current prototype fits in one of the four available Virtex-7 XC7v2000T FPGAs (Figure 13). The event history table and queue were implemented in the BRAM memory available in the FPGAs. The on-board 32GB DDR3 memory was used for state memory implementation, although very little memory was necessary for the Phold model 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 communicated through the host interface to report results as well as other measurements we collected to characterize the operation of the design. For any values that we want to measure during run time, we instrument the design with hardware counters that keep track of these events. We complemented these results with others such as queue and core occupancy that we obtained from functional simulator of the RTL implementation using Modelsim.

Our goal in this paper 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 the Convey, the memory system provides high bandwidth but also high latency (a few hundred cycles). This latency could dominate event execution time for fine-grained models where event handling is simple. To emulate event-processing overhead we let each event increment a counter to a value picked randomly between 10 and 75 cycles. The model generates memory accesses by reading from the memory when the event starts and writing back to it again when it ends.

Since our design is modular, we can scale the number of event processors. 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 number of LPs and event distribution gives us insight about 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 even be made to dynamically grow.

Design Validation: Verification of hardware design is complex since it is difficult to peek into the hardware as it executes. 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, crossbar 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 that are not model specific. 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.

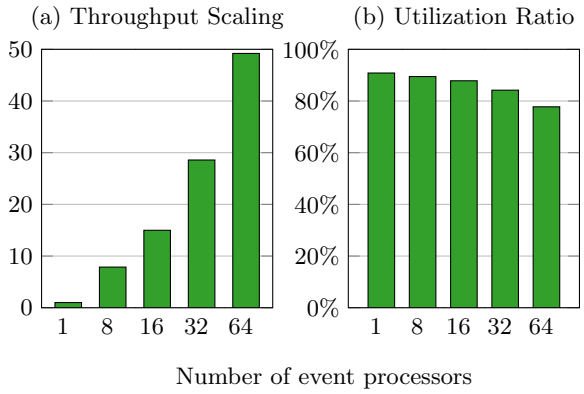

Figure 6: Effect of variation of number of cores on (a) throughput and (b) percentage of core utilization for 256 LP and 512 initial events.

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 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.

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 to the throughput of a configuration with a single event handler. 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 a little bit 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 sub-linear improvement in performance. Figure 6-b shows the event processor occupancy, which is generally high, but starts dropping as we increase the number of event processors reflecting that the additional contention is preventing the issue of the events to the handlers in time.

Figure 7 shows the throughput of the accelerator as a function of the number of LPs and the density of events in the system for 64 event processors. The throughput increases significantly with the number of available LPs in the system. As the events get distributed across a larger number of LPs, the probability of events belonging to 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 density in the system influences throughput to a lesser degree. Even though having a sufficient number of events is important to keeping the cores processing, once we have a large enough number of events increasing the event population further does not improve throughput appreciably.

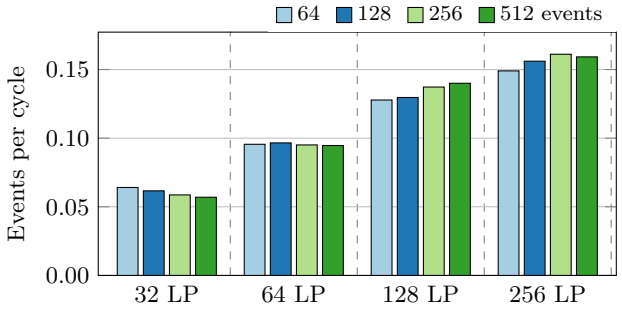

Figure 7: Event processing throughput using 64 event processors for different number of initial events and LPs.

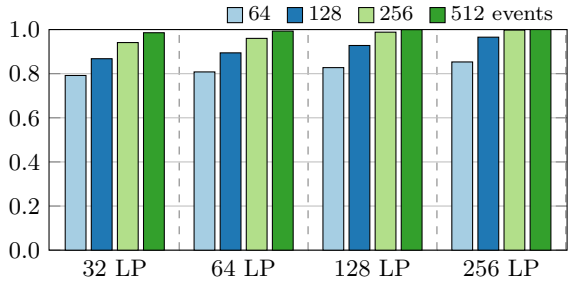

Figure 8: Ratio of number of committed events to total processed events using 64 event processors.

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 8 shows the efficiency of a 64-processor PDES-A as we vary the number of events and the number of LPs. 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. 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 same, reducing the number of initial events can cause the simulation efficiency to drop to around 80% (reflecting around 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 same LP are processed in the wrong order. When events are more distributed when 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 9 shows how the average event processing time varies with the number of LPs and the initial number of events, along with the breakdown of 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 other major delay source is the delay of processors stalling to wait for potentially conflicting events. These two factors are the primary delays in the system and dominate other overheads in the event processors such as task logic delays and maintaining event history, which also 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 are 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 notice that about the same number of cycles are consumed for memory access regardless of the configuration of the 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, 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. The probability that an event will become a straggler goes up with a smaller number of LPs 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 distributed to more LP, and can be safely processed in parallel.

Figure 10 shows a visualization fo the PDES-A's core operation by showing how the processors are behaving over time for a simulation with 256 LPs and 512 events. The black color shows the cycles when the processors are idle before receiving a new event. Yellow streaks represt the times a processor is stalled. Since an event processor has to stall until all other events associated with the same LP finish, the stall time can sometimes be long if more than two events belonging to the same LP are dispatched. Fortunately, scenarios like this are rare when there are sufficiently large number of LPs and for models that achieve uniform event distribution over the LPs.

The memory access time remains mostly unaffected by the parameters in the system. The state memory is distributed in multiple memory banks 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, the higher number of events may increase the probability of repeating accesses in the same memory area and therefore occasionally decrease the memory access time as these accesses are coalesced by the memory controller or cached by the DRAM row buffers. This effect reduces the average memory access delay slightly.

We note that the actual event handler processing time is a minor component of the event execution time consuming less than 10% of the overall event processing time even in the worst case. We believe that this observation motivates our future work to optimize PDES. In particular, the mem-

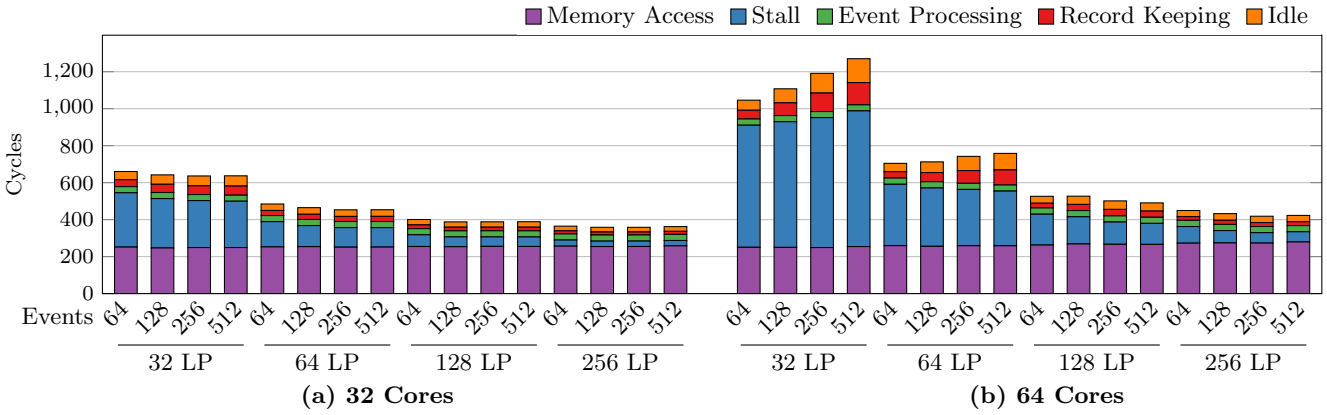

Figure 9: Breakdown of time spent by the event processors on different tasks to process an event using (a) 32 event processors and (b) 64 event processors with respects to different number of LPs and initial event counts.

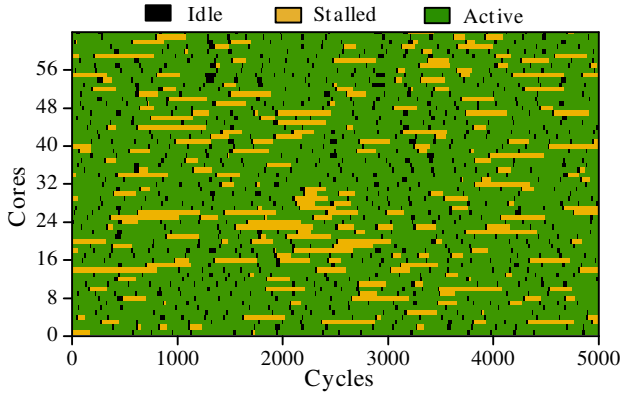

Figure 10: Timeline demonstrating different states of the cores for during a 5000 cycles frame of the simulation.

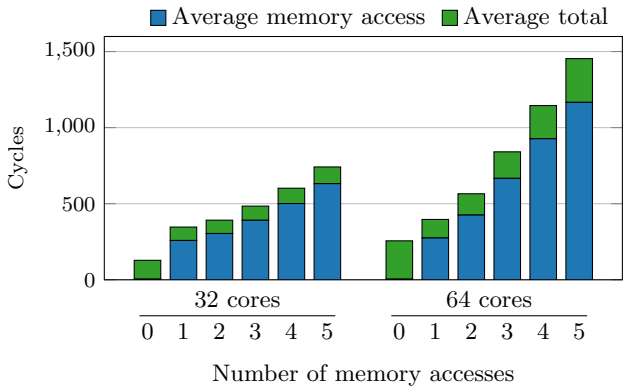

Figure 11: Effect of number/size of state memory access on event processing time

ory access time can be hidden behind event processing if we allow multiple event executions 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 11 shows the effect of variation of the memory access pattern on average execution time. The number of memory accesses can also be thought of as the size of the state memory read and updated in the course of 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 needed 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 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 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.

5.5 Effect of event processing granularity

Figure 12 shows the effect of processing time granularity on the system performance. Since memory access latency is a major source of delay 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 to be configured to a controllable delay by controlling the number of iterations the event handler increments a counter. The results of this study are shown in Figure 12-a. As the granularity increases, we simulate models that have increasingly computationally intensive event processing. Initially, the additional processing per event does not affect the system throughput since the system overheads lead to low utilization of the event handling cores when the granularity is small. As the utilization rises (Figure 12-b), additional increases in the event gran-

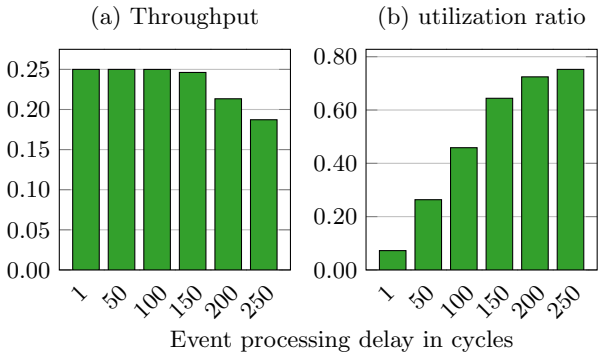

Figure 12: Effect of variation of processing delays (in cycles) on (a) throughput, (b) ratio of core utilization for 64 event processors with 256 LP and 512 initial events.

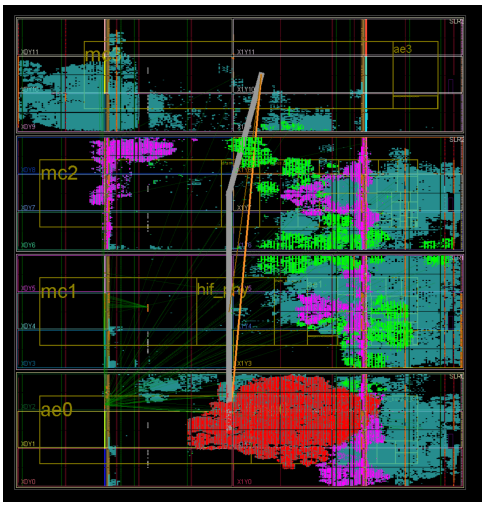

Figure 13: Implementation of the synthesized design on a Virtex-7 XC7V2000T FPGA. Red highlight marks the core simulator, green shows crossbar network, and memory interface logic is highlighted purple.

ularity start to lower the average rate of committed events per cycle since each event is more computationally demanding. Throughput does not improve after reaching 150 cycles event processing time.

5.6 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[3]. 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 simulation. The FPGA implementation has the advantage of having access to customized data paths to provide functions such as a single cycle hardware implemented Pseudo Random Number Generator (PNRG), which would require significantly longer time to implement on the CPU in software. On the other hand, the CPU can handle irregular tasks well,

Table 1: Comparative analysis of PDES simulation of Phold model ROSS and PDES-A

Parameters	ROSS	PDES-A
System		
Device	Intel Xeon E5-1650 12 MB L2	Xilinx Virtex-7 XC7V2000T
Frequency	3.50GHz	150MHz
Memory	32 GB	32 GB
Simulation		
PE	72 (12 cores×6 KP)	64
LP	252	256
Event Density	504	512
Remote Event	5%	100%
Performance		
Events/second	9.2 million	23.85 million
Efficiency	80%	∼100%
Power	130 Watt	<25 Watt

and execute multiple instructions per cycle at a much higher clock rates.

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 remote events are generated and handled in ROSS. In our system, all cores are connected to a shared set of LPs, so 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 5% to allow marginal communication between cores.

Table 1 shows the parameters for both the systems and their performance. At this configuration, PDES-A can processes events 2.5x faster than a 12-core CPU version of ROSS. When the remote percentage drops to 0% (all events are generated to local LPs), the PDES-A advantage drops to 2x that of ROSS. At higher remote percentages, the advantage increases, up to 10x at 100% remote messages. We believe that as we continue to optimize PDES-A this advantage will be even larger. Moreover, as we see in the next section, there is room on the device to integrate multiple PDES-A cores, further improving the performance.

6. FPGA RESOURCE UTILIZATION AND SCALING ESTIMATES

In this section, we first present an analysis of the area requirements and resource utilization of PDES-A. The FPGA resources utilization by the cores is presented in table 2. A picture of the layout of the design with a single PDES-A core is shown in Figure 13. The overall system takes over about 20% of the available LUTs in the FPGA. The larger portion of this is consumed by the memory interface and other static coprocessor circuitry which will remain constant when the simulator size scales. The core simulator logic utilizes 3.3% of the device logics. Each individual Phold event processor contributes to less than 0.03% resource usage. Register usage is less than 2% in the simulator. We can reasonably expect to replicate the simulation cluster more than 16 times in

Table 2: FPGA resource utilization

Component	LUT (1221600)		FF (2443200)		BRAM (1203)	
	Utilization	% Util.	Utilization	% Util.	Utilization	% Util.
Simulator	40412	3.31%	46715	1.91%	4	0.33%
Event Processor (1x)	391	0.03%	393	0.02%	0	0%
Controller	3610	0.30%	5557	0.23%	0	0%
Event Queue	6795	0.56%	5278	0.22%	0	0%
Memory Interface	143945	11.78%	132857	5.44%	206	17.12%
Crossbar Network	22051	1.81%	38713	1.58%	0	0%
Overall	236673	19.37%	261567	10.71%	223.5	18.58%

an FPGA, even when a more complex PDES model is considered and networking overheads are taken into account. Thus, there is significant potential to improve the performance of PDES-A as we use more of the available FPGA real-estate.

Finally, an inherent advantage of FPGAs is their low power usage. The estimated power of PDES-A was less than 25 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 significant boost in PDES simulation performance.

7. CONCLUDING REMARKS

In this paper, 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 2.5x 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 up to 16 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.

Acknowledgements

This material is based upon work supported by the Air Force Office of Scientific Research (AFOSR) under Award No. FA9550-15-1-0384 and a DURIP award FA9550-15-1-0376.

8. REFERENCES

[1] R. Brown. Calendar queues: A fast 0 (1) priority queue implementation for the simulation event set problem. 31(10):1220–1227.

[2] J. Burt. Intel begins shipping xeon chips with fpga accelerators, June 2016. Downloaded Feb. 2017 from eWeek: http://www.eweek.com/servers/intel-begins-shipping-xeon-chips-with-fpga-accelerators.html.

[3] C. D. Carothers, D. Bauer, and S. Pearce. Ross: A high-performance, low memory, modular time warp system. In *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*, PADS '00, pages 53–60, Washington, DC, USA, 2000. IEEE Computer Society.

[4] H. Chen, Y. Yao, W. Tang, D. Meng, F. Zhu, and Y. Fu. Can mic find its place in the field of pdes? an early performance evaluation of pdes simulator on intel many integrated cores coprocessor. In *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 41–49, Oct 2015.

[5] Convey Computers^{TM} Corporation. *The Convey WX Series*, conv-13-045.5 edition, 2013.

[6] Convey Computers^{TM} Corporation. *Convey PDK2 Reference Manual*, 2.0 edition, jul 2014.

[7] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. Gtw: A time warp system for shared memory multiprocessors. In *Proceedings of the 26th Conference on Winter Simulation*, WSC '94, pages 1332–1339, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[8] R. Fujimoto. Parallel and distributed simulation. In *Proceedings of the 2015 Winter Simulation Conference*, WSC '15, pages 45–59, Piscataway, NJ, USA, 2015. IEEE Press.

[9] R. M. Fujimoto, J.-J. Tsai, and G. C. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for time warp. *IEEE Transactions on Computers*, 41(1):68–82, 1992.

[10] S. Gupta and P. A. Wilsey. Lock-free pending event set management in time warp. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 15–26, 2014.

[11] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 37–47, 2010.

[12] M. A. Hassaan, M. Burtscher, and K. Pingali. 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, pages 3–12, New York, NY, USA, 2011. ACM.

[13] M. C. Herbordt, F. Kosie, and J. Model. An Efficient O(1) Priority Queue for Large FPGA-Based Discrete Event Simulations of Molecular Dynamics. pages 248–257. IEEE.

[14] Hybrid Memory Cube Consortium. *Hybrid Memory Cube Specification 2.1*, 2.1 edition, 2014.

[15] D. Jagtap, K. Bahulkar, D. Ponomarev, and N. Abu-Ghazaleh. 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, pages 53–62, Washington, DC, USA, 2012. IEEE Computer Society.

[16] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[17] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 228–241, New York, NY, USA, 2015. ACM.

[18] R. Noronha and N. B. Abu-Ghazaleh. Early cancellation: an active nic optimization for time-warp. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 43–50. IEEE Computer Society, 2002.

[19] H. Park and P. A. Fishwick. A gpu-based application framework supporting fast discrete-event simulation. *Simulation*, 86(10):613–628, Oct. 2010.

[20] K. S. Perumalla. Discrete-event execution alternatives on general purpose graphical processing units (gpgpus). In *20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, pages 74–81, 2006.

[21] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[22] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[23] B. L. Ranjita Bhagwan. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *In Proceedings of Infocom 2000*. IEEE Communications Society.

[24] J. Rios. An efficient FPGA priority queue implementation with application to the routing problem. Technical Report UCSC-CRL-07-01, University of California, Santa Cruz, 2007. Downloaded March 2017 from https://www.soe.ucsc.edu/research/technical-reports/UCSC-CRL-07-01.

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

[26] A. Santoro and F. Quaglia. Multiprogrammed non-blocking checkpoints in support of optimistic simulation on myrinet clusters. *Journal of Systems Architecture*, 53(9):659 – 676, 2007.

[27] A. Santoro and F. Quaglia. Multiprogrammed non-blocking checkpoints in support of optimistic simulation on myrinet clusters. *Journal of Systems Architecture*, 53(9):659–676, 2007.

[28] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[29] J. S. Steinman. The warpiv simulation kernel. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS '05, pages 161–170, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic. Ramp gold: an fpga-based architecture simulator for multiprocessors. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 463–468. IEEE, 2010.

[31] W. Tang and Y. Yao. A gpu-based discrete event simulation kernel. *Simulation*, 89(11):1335–1354, Nov. 2013.

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

[33] J. Wang, D. Ponomarev, and N. Abu-Ghazaleh. Performance analysis of a multithreaded pdes simulator on multicore clusters. In *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, pages 93–95, July 2012.

[34] B. Williams, D. Ponomarev, N. Abu-Ghazaleh, and P. Wilsey. Performance characterization of parallel discrete event simulation on knights landing processor. In *Proc. ACM SIGSIM International Conference on Principles of Advanced Discrete Simulation*, 2017.

[35] S. Zhou, C. Chelmis, and V. K. Prasanna. High-throughput and energy-efficient graph processing on fpga. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110, May 2016.