UNIVERSITY OF CALIFORNIA
RIVERSIDE

Hardware Acceleration of Irregular Applications Using Event-Driven Execution

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Shafiur Rahman

December 2021

Dissertation Committee:

Professor Nael Abu-Ghazaleh, Co-Chairperson
Professor Rajiv Gupta, Co-Chairperson
Professor Walid Najjar
Professor Daniel Wong

The Dissertation of Shafiur Rahman is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

As I am close to the end of this journey, I look back and remember all the people who have inspired, supported, and guided me along the way. There are so many people I call family, friends, and mentors. I probably cannot adequately thank all of them here. But I am grateful that our paths crossed.

First and foremost, I must thank my committee members for their thoughtful advice and feedback, and their support over the years. I appreciate the time they have contributed into all the milestones in my journey.

This work would not have been possible without the support of my advisors. I am grateful to Professor Nael Abu-Ghazaleh for extending me the opportunity to begin my PhD here. He has been an exceptional mentor in both professional and personal aspects of my doctoral journey. He had great patience and insights to motivate and mold a young researcher. I am incredibly lucky to be working with him. I want to thank Professor Rajiv Gupta. He was always available to offer me guidance and ideas in every aspect of my research. His excitement for any new idea was contagious. His supervision was a significant contributor to the timely completion of this project. The steady guidance and continuous encouragement from my advisors have made me a better researcher and made this project achievable. There were no two scholars better suited to guide me to complete this project.

I want to thank all my colleagues in our group. You were the most remarkable part of my graduate life. Our time in and out of the lab will always be a cherished memory. I regret that we could not make the most out of our time on campus because of the pandemic. But I am hopeful that we will get chances to make up for it once the world becomes normal.

I am thankful to the people with whom I started my first research work during my undergraduate studies. Those days made me ready to undertake this far greater responsibility today. Thanks to my professors, Md Atiqur Rahman Ahad and Upal Mahbub, and my friend, Tonmoy Roy, for our collaboration. What I learned from them is invaluable.

A special thanks to my parents, who always supported all my life decisions and were with me in every step of my life. Thanks to everyone in my family – all the sacrifices I made, they made it too. I am thankful for all the support and comfort they have given me from miles away.

Finally, and very importantly, I would like to acknowledge my wife and partner, Farzana Kabir. You have been there through all the good times and all the struggles. We both shared the same journey at the same time. Thank you for being so understanding and being the person I can rely on. Your unwavering support and steady presence made it so much easier for me.

DEDICATION

To my mother.

ABSTRACT OF THE DISSERTATION

Hardware Acceleration of Irregular Applications Using Event-Driven Execution

by

Shafiur Rahman

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Professor Nael Abu-Ghazaleh, Co-Chairperson
Professor Rajiv Gupta, Co-Chairperson

The consistent growth of DRAM memory bandwidth and capacity has enabled the computation of increasingly larger workloads in high-performance computing. However, the memory latency improvement over time is nominal, which severely bottlenecks the performance of modern systems. Modern computers rely on the exploitation of data locality using large cache hierarchies to keep the throughput high. Additionally, architects employ multicore and multithreaded architectures to boost throughput by extracting parallelism. However, irregular applications do not enjoy the same performance gain from these techniques. The inadequate data locality in these applications renders the cache ineffective, and the dynamic data dependence causes sub-optimal parallel processing. Hence, researchers are motivated to look beyond conventional CPU and GPU platforms towards dedicated hardware accelerators for these applications.

In this dissertation, we present strategies and architectures for accelerating irregular applications using the event-driven execution technique. We examine three irregular applications – Parallel Discrete Event Simulation (PDES), Graph Processing, and Stream-

ing Graph Analytics – to study their limitations in conventional systems and solve these challenges using hardware primitives to build generalized accelerator frameworks. Our custom datapaths based on event-driven execution models promote parallelism and memory bandwidth utilization in these applications.

The first application, Parallel Discrete Event Simulation, is inherently event-driven. It demonstrates *ordered irregular parallelism*, characterized by strict ordering between tasks that constrains parallelism. First, we design an efficient hardware priority queue, which becomes the core component of our event-driven systems. Then, we build a highly scalable accelerator, PDES-A, that incorporates a decoupled datapath for robust transmission of events and masking long memory access latency.

The second application, graph processing, has partial ordering among the tasks and possesses similar dynamic data dependence. We convert the traditional iterative execution model into an event-driven execution model. We demonstrate that fine-grained control over the dynamic memory access patterns is achievable through strategic manipulation of events to maximize spatial locality and, hence, memory bandwidth utilization. Our acceleration framework, GraphPulse, uses this model to accelerate asynchronous graph processing.

Finally, we study a more complex variation of graph processing, streaming graph analytics, to illustrate the extensibility of the event-driven model. This application has multiple types of computation tasks and requires multi-phase execution. We employ incremental recomputation of streaming data to reduce redundant computation. We develop an incremental graph computation model suitable for the event-driven paradigm and subsequently develop JetStream to support streaming graphs using this algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For years, the growth in high-performance computation capacity was facilitated by the rapid increase in transistor density and clock speed in computer CPUs following the predictions of Moore's Law and Dennard Scaling [27]. However, the laws of physics finally appeared as a barrier to this exponential growth preventing the ability to shrink the transistors anymore than they already were. The end of Dennard Scaling restricts the ability of software to scale the performance using larger and faster processors due to the Dark Silicon effect. As a result, alternative strategies are required to support the rapidly growing computational demands of the modern days. This incentivizes the push towards custom hardware accelerators built for specific application domains. The accelerators can be orders of magnitude more efficient in terms of performance and power since the complex pipelines of modern CPUs can be streamlined and trimmed down. In addition, a large portion of the industry workloads is a small set of repetitive tasks that can benefit significantly from specialized accelerator units to execute them. The recent drive for integrating reconfigurable

accelerators in the cloud, as seen in the Microsoft Catapult Project [78] and Amazon F1 FPGA instances [45], recognizes the industry demand for application-specific accelerators.

This dissertation explores the advantages of using event-driven execution to accelerate irregular applications in hardware. An event-driven system records an update task as an event in a queue. It only performs the task when the event is scheduled. Such systems visit a state only when necessary, thus, reducing the workload of the system. Furthermore, the event-based model transforms the dynamic dependence of irregular applications into order among the events to preserve correctness. Then, synchronization complexity is reduced by controlling the flow of events in the system to enhance parallelism. Additionally, fine-grained control over the state access pattern is possible by manipulating the order of events processing. Thus, memory bandwidth can be optimized by scheduling the events to execute the tasks in a memory-access-friendly pattern. Event-driven execution, however, does not achieve expected performance in conventional shared-memory systems because the task processing throughput becomes dependent on the performance of an ordered queue structure containing the events. Such queues are challenging to implement with high performance in software, and event storage becomes the critical bottleneck of the system.

Nevertheless, it is possible to overcome this challenge in hardware accelerators with hardware primitives for robust event storage and transmission to achieve a high degree of parallelism. The hardware acceleration of these applications offers two primary advantages.

- *Low-latency and high-bandwidth on-chip communication:* Hardware platforms such as FPGAs support fast high-bandwidth on-chip communication, substantially alleviating the communication bottleneck limiting the performance of irregular applications [107].

- *Specialized high-bandwidth datapaths:* General-purpose 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. Moreover, specialized hardware allows 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 chip.

We approach the study of event-driven accelerator design from three perspectives. First, we investigate the **feasibility, profitability, and technical limitations** of building an event-driven system using hardware. Afterward, we analyze the **adaptability** of such systems to different application domains that are not typically event-driven. Finally, we explore the **expansibility** of the event-driven execution model for supporting complex applications with multiple types of tasks and multiple phases of computations. Our study confirms that event-driven execution can be a beneficial technique for accelerating diverse application classes in hardware.

We examine three irregular applications – Parallel Discrete Event Simulation (PDES), Asynchronous Graph Processing, and Streaming Graph Analytics – to investigate and demonstrate the methodologies for designing accelerators using the event-driven execution model. Parallel Discrete Event Simulation is an inherently event-driven application. We establish the sustainability of a hardware architecture for event-driven execution using Parallel Discrete Event Simulation as the target application. Graph Processing, on the other hand, traditionally uses the iterative execution method. We demonstrate the tech-

niques and considerations for the algorithmic conversion of an irregular application (Graph Processing) to execute in an event-driven approach and develop an architecture to support this execution model. Finally, we illustrate how the event-driven model can accommodate multi-phased computation and diverse tasks to support complex application domains by extending the execution model for streaming graph analysis.

In the remainder of this chapter, we describe the motivations for this dissertation and the limitations of existing technologies in Section 1.1. We continue to present the contributions of this dissertation in Section 1.2, Finally, we describe the organization of this dissertation in Section 1.3.

1.1 Motivation: Challenges in Irregular Applications

While computer architects quickly adopted multicore and multithreaded architectures to subvert the limitation in processor speed using parallel processing, this adds new challenges for scaling the performance of high-performance workloads. The increasing gap between the processing speed and memory bandwidth becomes the primary bottleneck in processing performance in these systems. Historically, the DRAM memory has enjoyed rapid growth in capacity, while the bandwidth increased at a much lower rate and the DRAM latency increased nominally. The high capacity of DRAM memory enables modern systems to undertake huge workloads, but the performance suffers because of long latency and limited bandwidth. If memory latency and bandwidth become insufficient to provide processors with enough instructions and data to continue computation, processors will effectively always be stalled waiting on memory. The trend of placing more and more cores

on-chip exacerbates the situation since each core enjoys a relatively narrower channel to shared memory resources. The problem is commonly referred to as *memory wall*, and it is particularly acute in highly parallel systems.

In modern systems, these bottlenecks are alleviated using speculative execution and large cache hierarchies. Many applications have little control flow divergence, predictable execution pattern, and high spatial or temporal locality receive good performance boost from these techniques. However, some applications, dubbed *Irregular Applications* have characteristics that prevent them from getting any significant improvements from these techniques. Irregular applications have pointer-based indirection in their data structures resulting in unpredictable memory access patterns and dynamic data-dependent control-flow. Many critical HPC applications such as graph algorithms, machine learning, and database operations fall in this category. These applications help derive actionable intelligence from highly connected data of massive size, and they depend on highly dynamic, multidimensional data structures such as graphs, trees, and grids.

In irregular applications, pointer-based memory indirections lead to data accesses unrelated in any temporal or spatial sense. Such a lack of locality limits the performance of modern processor architectures built on deep memory hierarchies and prefetching to mitigate latencies. Moreover, irregular data structure-based applications usually have limited arithmetic intensity, as most of the computation time is spent navigating the data structures. Thus, modern processors and accelerators designed to maximize floating-point operations per memory access are a poor fit for these applications. These applications also have large amounts of data parallelism as many data values can be explored in parallel.

However, the dynamic data dependence between tasks makes it difficult to exploit this parallelism. Moreover, they often require fine-grained synchronization as concurrent threads may compete to modify the same data elements. These irregular applications become the prime candidates for hardware acceleration. The dedicated architecture can re-purpose or discard the cache hierarchy to optimize memory access and use specialized hardware primitives to establish a fast and effective synchronization mechanism. As a result, significant parallelism and performance gain can be expected from the accelerator.

1.2 Contributions of the Dissertation

We have designed several computation models and accelerators for different applications during our research in the acceleration of irregular applications. These computation models demonstrate techniques for efficient event-driven execution and serve as pointers for adapting iterative applications into the event-driven model. We also built three accelerators demonstrating the performance potential of the event-driven method.

1.2.1 PDES-A: Parallel Discrete Event Simulation Accelerator

PDES-A is the first hardware accelerator for Parallel Discrete Event Simulation. It includes a hardware priority queue capable of fast insertion and retrieval. This pipelined priority queue allows PDES-A to support many processing cores and achieve high throughput. In addition, we relax synchronization by employing an optimistic execution with a rollback mechanism for recovery. PDES-A provides excellent scalability for PDES Model *Phold* for up to 64 concurrent event processors. Our baseline prototype outperforms a sim-

ilar simulation on a 12-core 3.5GHz Intel Core i7 CPU by 2.5x. Afterward, we describe re-designed PDES-A engines that relieve contention by partitioning all shared structures such as event and state queues into multiple substructures. It also incorporates a decoupled datapath that significantly relaxes synchronization and improves the scalability of the system. This optimized design results in another 25% improvement in throughput. The optimized accelerator outperforms a 12-core Intel Core i7 by 3.2x while consuming less than 15% of the power.

1.2.2 GraphPulse: an Accelerator for Graph Processing

Our graph processing accelerator, GraphPulse, is the first asynchronous event-driven graph processing accelerator. GraphPulse alleviates several performance challenges faced in traditional software graph processing while bringing the benefits of hardware acceleration to graph computations. GraphPulse's event-driven model decouples computations, ergo memory accesses, from communications. In this model, memory accesses are only necessary during vertex updates. As a result, one of the primary sources of random memory access is eliminated. Moreover, synchronization is simplified by having the accelerator serialize events destined to the same vertex; thus, the synchronization overhead of traditional graph processing is reduced. At any point in time, the events present in the system naturally correspond to the active computation. Thus the bookkeeping overhead of tracking the active subset of the graph is also masked.

Supporting this model in software is difficult due to the high overheads of generating, communicating, managing, and scheduling events. GraphPulse architecture supports events in the hardware queue and routes them using a fast on-chip communication network.

Thus, it essentially eliminates event-related overheads and making the model more efficient than software implementations. GraphPulse performs asynchronous graph processing, resulting in substantial speedups due to greater exploitation of parallelism and faster convergence. In addition to this, it introduces coalescing of events to control event population and allow transaction safety. We enhance the model with a prefetcher and smart scheduler to achieve high throughput. This design substantially outperforms software frameworks due to its efficient memory usage and bandwidth utilization. GraphPulse outperforms software frameworks on a 12-core Intel Xeon system by 28x with about 300x better energy efficiency.

1.2.3 JetStream: Accelerator for Streaming Graph Analytics

The last accelerator we built as part of this dissertation is JetStream, an accelerator for performing incremental evaluations of streaming graphs. Graphs are constantly evolving in real-world applications. Restarting a query from scratch on a mutated graph is wasteful because the changes usually modify only a small subset of the graph. Thus, much of the computation performed during reevaluation is redundant. Streaming graph systems support incremental update of query results to address this inefficiency, resulting in order of magnitudes speedups over cold-start recomputation. JetStream builds on GraphPulse and uses the same event-driven asynchronous processing model. It extends the event-driven graph computation model to support incremental computation for edge addition, deletion, and updates. JetStream proposes an event-driven approach to reset and recover after edge deletions for monotonic graphs. JetStream achieves 18x speedup over software graph processing framework, GraphBolt. It also performs 13x better than compared to a complete recomputation of the streaming graphs in GraphPulse.

1.3 Organization of the Dissertation

We begin the rest of the dissertation by reviewing important literature related to the applications considered in our work. Then, we describe the three accelerators that we built in the following chapters. In Chapter 3, we present the design of a Parallel Discrete Event Simulation accelerator, PDES-A. We explore the fundamental design considerations and a baseline architecture for the accelerator in this design. We analyze the limitations of the basic architectures that are resolved using a redesigned datapath in Chapter 4. This chapter describes the optimizations for PDES-A and our techniques for designing a decoupled datapath for the event-driven model.

Next, we present the limitations of conventional graph processing systems and the development of event-driven methods to alleviate them in Chapter 5. This chapter elaborates on the design considerations for event-driven execution and illustrates the transformation of an iterative algorithm to an event-based model. We describe the architecture of the GraphPulse accelerator in Chapter 6. This chapter also describes the evaluation of the accelerator against state-of-the-art software and hardware graph analytics frameworks.

The last part of this dissertation describes the design of JetStream, an accelerator for streaming graph analytics in Chapter 7. We present the challenges of incremental computation and how the event-driven model can be extended to perform incremental evaluation of streaming graphs. It is followed by the description of JetStream architecture that implements this extended model in Chapter 8.

Finally, we conclude with a brief summary of our research and potential future works in Chapter 9.

Chapter 2

Related Work

In this chapter, we present previous works relevant to this dissertation. We organize the relevant literature into three sections. The first section reviews literature related to the techniques and software frameworks for parallel discrete events simulation. Then, we describe recent works on software and hardware-based graph processing systems. Finally, we highlight crucial works in the field of streaming graph analytics.

2.1 Parallel Discrete Event Simulation

Discrete Events Simulation is heavily used in real-world simulations and performance evaluations where systems change in discrete states. A wide range of domains extending from military simulations and war-gaming [34, 42] to epidemiology and molecular biology [32, 114] uses discrete event simulation for studying vast simulation space, which is prohibitively expensive with traditional time-stepped simulation methods. PDES is an important tools for digital design simulation [63] industry as well.

The parallel Discrete Event Simulation community extensively uses Phold [34] models as a synthetic benchmark for profiling and evaluating PDES systems. The Phold model consists of a model where an event is generated for a random destination whenever a logical element receives an event. It is a communication-centric model to expose the communication robustness of the simulation system. There is another model [55] that utilizes real-world simulation profiles to generate more accurate and realistic synthetic PDES models that are more parameterizable and customizable.

In recent years, researchers have developed and analyzed PDES simulators on various parallel and distributed platforms as these platforms have continued to evolve. The widespread adoption of both shared and distributed memory cluster environments has motivated the development of PDES kernels optimized for these environments, such as GTW[25], ROSS[18] and WarpIV[95]. These systems can be conservative or optimistic simulators that employ a diverse range of techniques for checkpointing, reverse computation, implementations of GVT, and anti-messages. Apart from the works on new simulation frameworks, there are many individual works that experiment on optimizations for a particular system or architecture. For example, Jagtap et al. [49, 106] proposed and analyzed several optimizations for multi-threaded PDES simulators to improve scalability on shared-memory platforms such as Intel Core-i7 and such. One of the notable contributions of their work involves the conversion of an MPI-driven strategy to a shared-memory threaded strategy.

Other PDES implementations explore various upcoming architectures. For example, Jagtap et al. studies the performance of PDES [49] on the many-core Tilera architecture. Their result displays good scalability, and the architecture proves capable of sustaining high

throughput even under a heavy workload. A study of PDES in Intel Many Integrated Core (MIC) system [20] suggests that full utilization of the Knights Corner processor's vector units are required to outperform the Xeon host processors. A PDES system designed by Bauer et al. [9] replaces the checkpointing system in ROSS with reverse computation to achieve close to linear speedup, utilizing 128K cores in the IBM Blue Gene Supercomputer. Reverse computation tries to trace backwards the states using the opposite computation of the regular update operation. Thus, it can eliminate the overhead for checkpointing states and maintaining a global time if the computation can be rolled back using reverse computation. Barnes et al. [8] extend the previous work to analyze performance in the Sequoia BlueGene/Q supercomputer with 2 million cores. Their experiments show impressive throughput for PDES with an MPI-driven execution. Several researcher explore the use of GPGPUs to accelerated PDES [76, 97, 77, 19]. In contrast to these effort, very few works considered acceleration of PDES using non-conventional architectures such as FPGAs.

Other optimizations of PDES systems focus on improving the robustness and throughput of the priority queue of the PDES system. For example, the study by Gupta et al. [37] looks at the use of ladder queues for a lock-free event management structure that can substitute the more commonly used linked list-based event queue. This work assumes that the events in a small simulation time window have no dependence and can be issued without sorting in an optimistic simulation system. Optimization to the calendar queue is considerred in [68] to propose a lock-free non-blocking event pool woth $O(1)$ time complexity for both insertion and dequeuing. The event queue structure and its impact on PDES performance have been studied in the context of software implementations [82];

however, it is crucial to understand suitable queue organizations implemented in hardware. Prior work has studied hardware queue structures supporting different features. For example, hardware 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 [13] support $O(1)$ access time but are difficult and expensive to scale in a hardware implementation. QuickQ[80] uses multiple dual-ported RAM in a pipelined structure which provides easy scalability and supports constant-time access. However, the access time is proportional to the size of each stage of RAM. Therefore, configuring them to achieve a short access time necessitates many stages, which leads to high hardware complexity.

2.2 Graph Processing

The next application, graph processing, is employed in many domains, including social networks [96, 15, 22, 46, 30], web graphs [109], and brain networks [14], to uncover insights from high volumes of connected data. Iterative graph analytics require repeated passes over the graph until the algorithm converges. Since real-world graphs can be massive (e.g., YahooWeb has 1.4 billion vertices and 6.6 billion edges), these workloads are highly memory-intensive. Thus, there has been significant interest in developing scalable graph analytics systems. Some examples of graph processing systems include GraphLab [64], GraphX [36], PowerGraph [35], Galois [74], Giraph [3], GraphChi [60], and Ligra [90].

Maiter [116] is a graph analytics framework for delta-accumulative computation, which is the basis of the event-driven model. Maiter shows the computation model for a distributed systems where vertices communicate only the incremental change in their states, and graph updates incrementally accumulate in vertices to reach the final converged state.

Graph Accelerators

Template-based graph accelerators process hundreds of vertices in parallel to mask long memory latency [4]. They use hardware primitives for synchronization and hazard avoidance. Swarm [51] allows speculative execution to increase parallelism and support commit and rollback on a many-core architecture so that when tasks proceed speculatively, there is the option to revert with little overhead. This increases the number of concurrent tasks processed for ordered or tree-based algorithms. However, inherent memory access inefficiencies of graph algorithms still persist. Spatial Hints [52] uses application-level knowledge to tag tasks with specific identifiers for mapping them to processing elements which allows better locality and more efficient serialization, thus, addressing the inefficiencies of Swarm. Chronos [1] provides another hardware acceleration framework based on speculative execution. On the other hand, serialization becomes unnecessary after coalescing in GraphPulse since transaction safety is implicitly guaranteed by the execution model and architecture.

Ozdal [75] proposes a graph processing architecture in hardware that takes advantage of a hardware-based commit queue to provide transaction safety in asynchronous graphs. Graphicionado [38], a pipelined architecture, optimizes vertex-centric graph models using a fast temporary memory space. It improves locality using on-chip shadow memory for vertex property updates. Throughput increases due to reduced memory consistency over-

heads and higher memory bandwidth using the temporary storage. However, GraphPulse substantially outperforms Graphicionado due to the advantages of the event-driven model over conventional models. Graph processing systems for FPGAs include ForeGraph [23], Zhou et al. [119, 120] etc.

PIM-based solutions

These solutions lower memory access latency and increase performance. Tesseract [2] implements simplified general-purpose processing cores in the logic layer of a 3D stacked memory. Like distributed graph processing, it uses messages to access/update data. GraphPIM [73] replaces atomic operations in the processor with atomic operation capability in Hybrid Memory Cube (HMC) 2.0 to achieve low latency execution of atomic operations. Another solution explores robust partitioning and cross-communication to fit large graphs in 3D stacked memory [113]. There have been other recent works that focus on architectures for PIM-based graph processing, such as GraphP [112], and GraphQ [121]. Our approach has the potential to be advantageous on PIM platforms too, because memory accesses are simplified, and the complex scheduling and synchronization tasks are isolated to the logic layer.

Tolerating irregular memory accesses

The propagation blocking technique for PageRank [10] temporarily holds contributions in hashed bins in memory, merges contributions to the same vertex, and later replays them to maximize bandwidth utilization and cache reuse. However, it entails the overhead of maintaining bins. Zhou et al. [119, 120] store contributions temporarily to memory and

combines them using hardware when possible for the edge-centric model. Due to a large number of edges, there is a substantial increase in random memory writes to temporary bins and small combination windows limit combining. To optimize irregular memory accesses, IMP [110] uses a dynamic predictor for indirect accesses to drive prefetching. HATS [70] proposes a hardware-assisted traversal scheduler for locality-aware scheduling to increase data reuse. RAts [92] memory model exploits commutativity in atomic operations to reduce complexity. In contrast, GraphPulse coalesces the application's computations rather than memory operations and controls the scheduling and frequency of vertex accesses to improve memory access patterns.

Efficient vertex update handling

Since vertex updates are a crucial bottleneck in a graph-analytics application, some prior works focus on improving the locality and cost of scattering updates to the neighbors. Beamer et al. [10] uses *Propagation Blocking* to accumulate the vertex contributions in cache-resident bins instead of applying them immediately. Later, the contributions are combined and then applied, thus eliminating the need for locking and improving spatial locality. *Propagation Blocking* technique creates perfect spatial locality but fails to utilize potential temporal locality for vertices having many incoming updates since updates are binned and spilled to memory first. Other methods exploit commutative nature of the reduction (apply) operation seen in many graph algorithms to relax synchronization for atomic operations. Coup [111] extends the coherence protocol to apply commutative-updates to local private copies only and reduce all copies on reads. This reduces read and write traffic by taking advantage of the fact that commutative reduction can be unaware of

the destination value. PHI [71] also uses the commutativity property to coalesce updates in private cache and incorporates update batching to apply scatter updates in bulk while reducing the on-chip traffic further. Both PHI and Coup optimize memory updates, but have no fine grain control over the memory access pattern. Like PHI, GraphPulse utilizes commutative property to fully coalesce updates using the specialized on-chip queue. Furthermore, GraphPulse applies these updates at a dataflow-level abstraction to reorder and schedule updates for maximizing spatial locality and bandwidth use.

Dataflow architectures

GraphPulse bears some similarities to dataflow architectures in that computation flows with the data [98, 57, 24, 44]. The Horizon architecture supports lightweight context switching among a massive number of threads for tolerating memory latency. It heavily relies on the compiler [24] but dynamic parallelism in graph applications is not amenable to similar compiler techniques. The SAM machine [44], which is a hybrid dataflow/von Neumann architecture, employs a highspeed memory (register-cache) between memory and execution units is a better match for graph applications. However, neither of these architectures address issues in graph processing addressed by GraphPulse. Specifically, GraphPulse uses data-carrying events to eliminate random and wasteful memory accesses, coalescing strategy eliminates atomic operations and reduces storage for events, and event queues improve locality and enable prefetching.

2.3 Streaming Graph Processing

Taxonomy on streaming graph analytics is not yet well-established and several conventions are observed for streaming graph papers. Most works refer to a graph system where new edges additions or deletions appear over time as streaming graphs. However, these graphs are also referred to as *time-evolving* [47], *dynamic*, or *continuous* [28] in many literatures.

A number of streaming graph frameworks have been developed that are based on the BSP [100] model similar to a software framework like Pregel [65]. Of these frameworks, Kineograph [21], Tornado [89], Naiad [72], and Tripoline [53] are limited to growing graphs (i.e., deletion updates are not allowed). In contrast, Kickstarter [103], GraphBolt [67], and DZiG [66] support both addition and deletion of edges. Our work in this dissertation, JetStream, supports both the addition and deletion of edges.

Streaming graph frameworks employ various techniques for performing incremental computations. Some frameworks put the responsibility on the end-user to write methods for discovering inconsistent vertices after graph mutation is applied. GraphIn [88] and EvoGraph [87] use this technique. The user-defined method detects whenever inconsistency arises in a graph based on the batch of edge updates and schedules the updated vertices for recomputation. GraphBolt [67], and KickStarter [103] on the other hand, automatically detects the affected vertices using a dependence tracking technique built into their systems. There is a difference between the targeted class of graph algorithms between GraphBolt and KickStarter. KickStarter targets a class of algorithms that have monotonic properties and selects a single edge update to set the property of a vertex. GraphBolt, on the other hand,

proposes a dependency tracking technique that works on graph algorithms of iterative nature and accumulative update functions. GraphInc [16] uses a memoization-based technique to support incremental evaluation on iterative graphs. This technique attempts to preserve the states of all computations performed and attempts to reuse the preserved states to answer a graph query quickly. Other frameworks, such as KineoGraph [21] maintain a snapshot of the graph states at repeated intervals and recompute the differences in the snapshot to identify the part of the graph that needs recomputation. This strategy works even when vertex update computations are not minimalistic or deterministic.

There are also designs of graph representations to support high-throughput graph updates, such as Aspen [29], STINGER [31], and Version Traveller [54]. Other works handling changing graphs include GraphTau [48], Vora et al. [102], Chronos [40]. However, these works consider scenarios in which historical data is being analyzed, i.e., graph versions are available a priori. GraSU [108] provides the first FPGA-based high-throughput graph update library for dynamic graphs.

Chapter 3

PDES-A: Parallel Discrete Events Simulation Accelerator

In this section, we introduce a baseline design targeted to run a PDES system in an FPGA. The design focuses on the development of a high performance pipelined queue with constant time insertion and dequeue to facilitate maximum parallelism in the accelerator. It provides a simple datapath with all the components required for a PDES engine.

3.1 Parallel Discrete Event Simulation

A discrete event simulation (DES) models the behavior of a system that has discrete state changes. 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, wargaming/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 the simulation, allowing the simulation of larger, more detailed models and the consideration of more scenarios in a shorter amount of time [33].

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.

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 [50] 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 the GVT can be garbage collected, and events earlier than the GVT may be safely committed.

3.2 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, and 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 relatively clean and straightforward 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 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. For example, the Simian project[85] shows that a completely functional PDES engine can be implemented in less than 500 lines of python code. An FPGA-based PDES engine can leverage these properties 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 paper, 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 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 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.

3.2.1 Design Goals

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. We avoided any model specific tuning to retain generality of the accelerator.

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: it 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.

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.

3.2.2 General Overview

Figure 3.1 shows the major components of PDES-A and their interactions. The event queue contains a list of all the unprocessed events sorted in ascending order of their timestamp. Event processors receive event messages from the queue. After processing

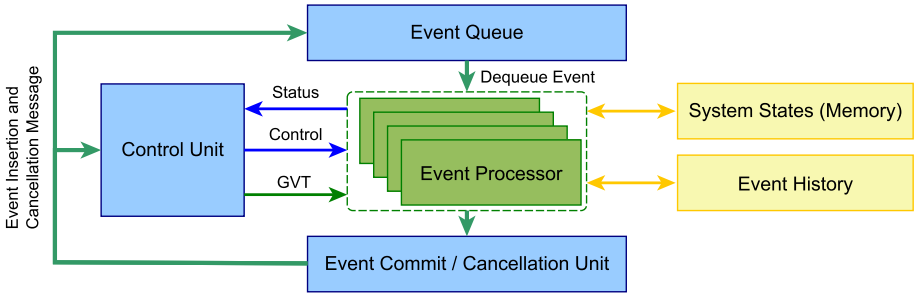


Figure 3.1: Block diagram of basic control and data flow in a PDES system

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 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.

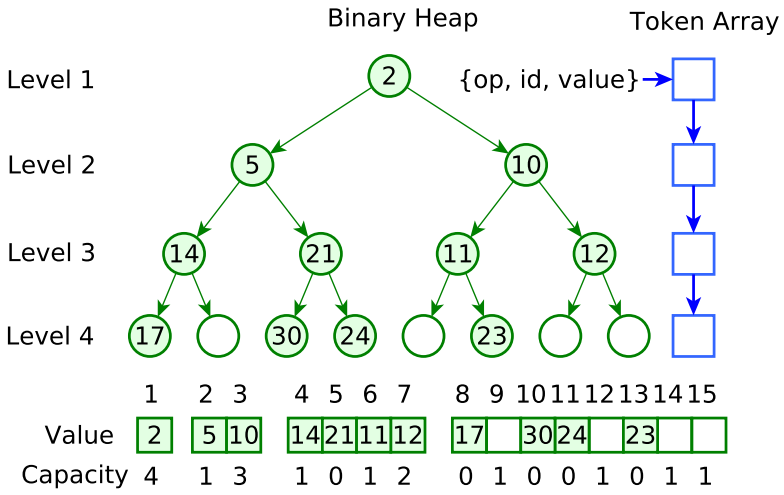


Figure 3.2: The P-heap data structure [11]

3.2.3 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.

We selected a *pipelined heap* (P-heap in short) [11] structure as the basic organization in our implementation, except for a few modifications described later. P-heap uses a pipelined binary heap to provide two cycles constant access time. 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 3.2). 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: 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 [11].

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

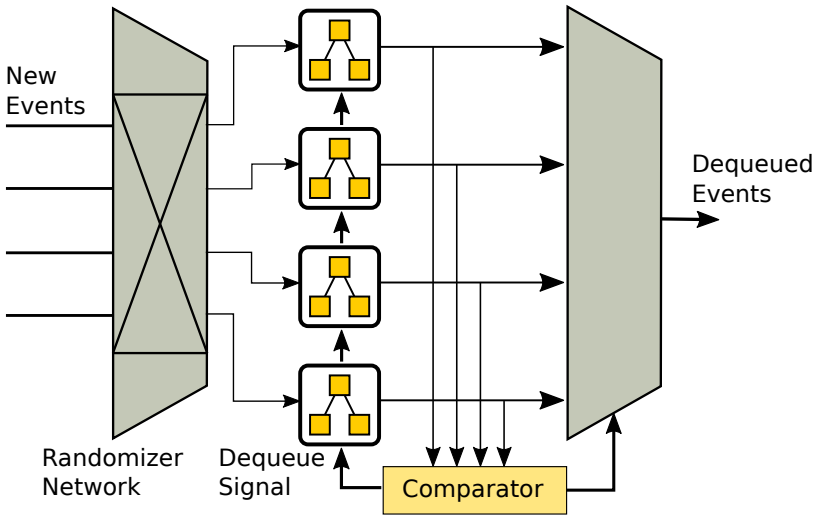


Figure 3.3: Multiple event issue priority queue

request in a single cycle. In an approach similar to [41], we can use a randomizer network to direct multiple requests to multiple available heap (Figure 3.3). There is a chance that two of the highest priority events may reside in the same heap and ordering violation will occur when a lower priority event from same LP is dequed from another queue during multiple dequeue. However, as the number of LPs and events grow, the probability that two events from the same LP are at different queue heads decreases. So, 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 our optimized architecture discussed in section 4.1. Other structures that sacrifice full ordering but admit higher parallelism such as Gupta and Wilsey's lock-free queue may also be explored [37].

The queue stores a key-value pair. Event time-stamp 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.

3.2.4 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) support events exchange with the event queue; and (4) 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 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.

3.2.5 Event scheduling and processing

Figure 3.4 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

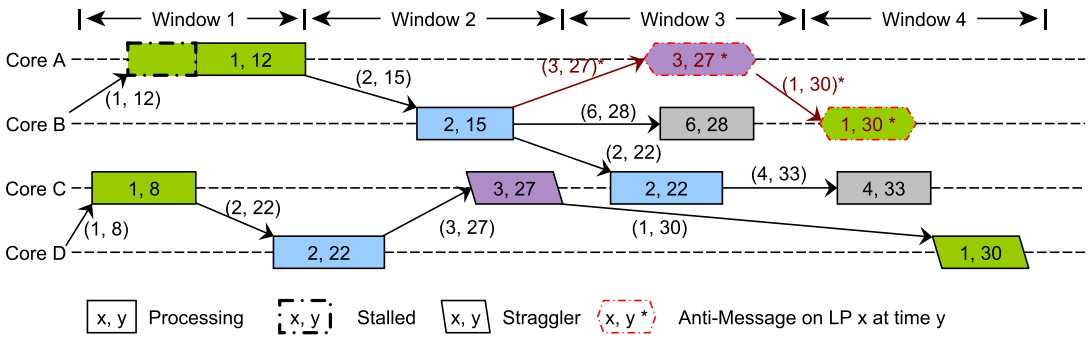


Figure 3.4: Simplified timeline representation showing scheduling of events in the system.

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 schedule 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 3.4 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 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 generate other anti-messages (1, 30*) to *chase* the erroneous message chain much like a regular events as shown in window 3 of Figure 3.4. 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.

3.2.6 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's 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 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 put a restriction on how large it can be. So, 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 the 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 history 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 doesn't add much overhead.

3.2.7 Rollback and Cancellation

The state restoring 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, processor receives a list of violating events in reverse order of execution. The programmer needs to define the way 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 events 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 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 3.2.5. Each event message, along with its payload, carries an unique identifier which is a tuple (processor ID, sequence number of active event) that serves as a 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 an unique sequence number from a counter in the processor. The unique identifier for the parent event is stored in its history entry along with rest of the event data. When the parent event is reversed, rollback process emits anti-messages carrying this identifier with timestamp and LP of the target events to be cancelled. The anti-message can be matched with their target event using the unique identifier and event timestamp.

3.3 Implementation Overview

We used a full RTL implementation on 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 as well as other measurements we collected to characterize the operation of the design.

For any values that we wanted to measure during run time, 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 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 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 handle overflow by spilling into the memory.

3.3.1 Design Language and Application Modeling

The baseline design was implemented using Verilog. However, during optimization, we reimplemented the system using Chisel[5], 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 to 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 from the end-users lesser understanding of the

framework. 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, and thus keep evolving rapidly throughout each experiment. In the standard HDL languages, small adjustments in one component tend to ripple through the whole design requiring lots of 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 put these comments as a note for the benefit of future researches.

Simulation 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 the 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 the 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[34] 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[17].

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.

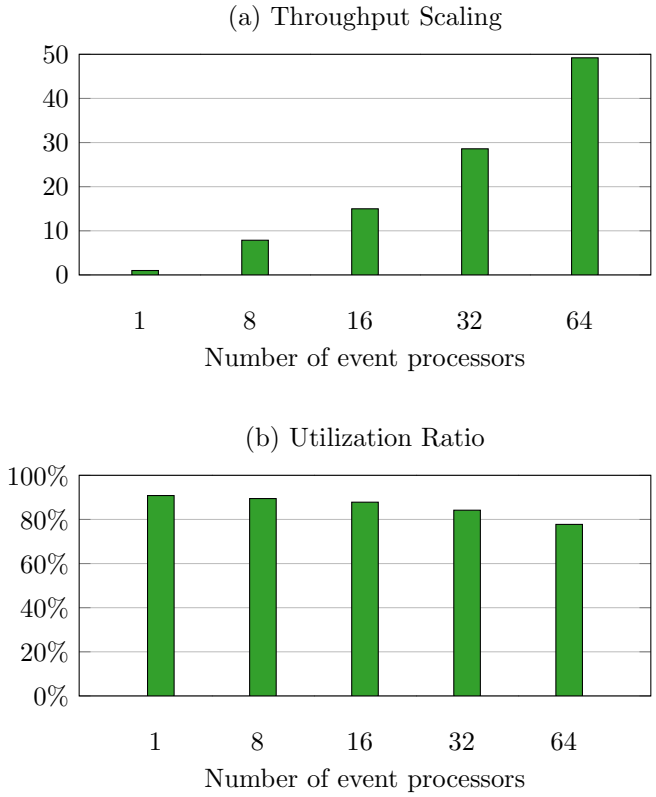


Figure 3.5: Effect of variation of number of cores on (a) throughput and (b) percentage of core utilization for 256 LP and 512 initial events in PDES-A.

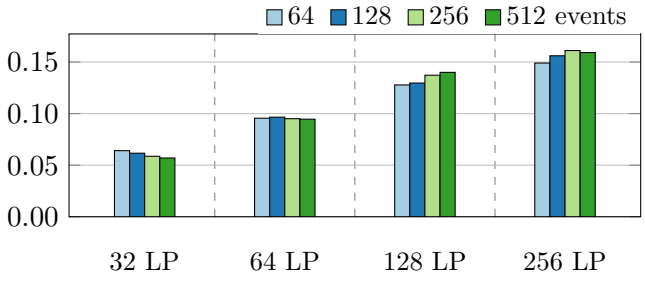
3.4 Performance Evaluation

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.

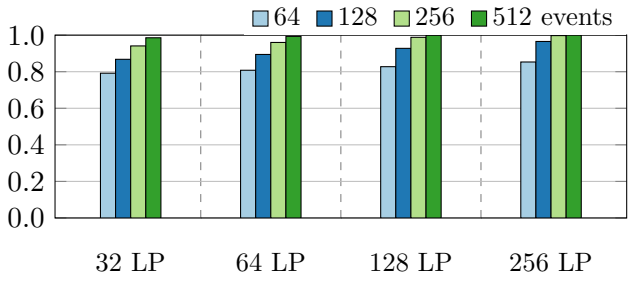
3.4.1 Performance and Scalability

In this first experiment, we scale the number of event processors from 1 to 64 while executing a Phold model. Figure 3.5-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 a 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 3.5-b shows the event processor utilization, that is, the portion of time the event processor is actively processing event and not stalling or waiting for resource. 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 3.6-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) Event Throughput



(b) Commit Efficiency

Figure 3.6: (a) Event processing throughput (events/cycle) and (b) Ratio of number of committed events to number of total processed events for different number of LPs and initial events on 64 event processors.

a large enough number of events increasing the event population further does not improve throughput measurably.

3.4.2 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 3.6-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 presence of sufficient number of events.

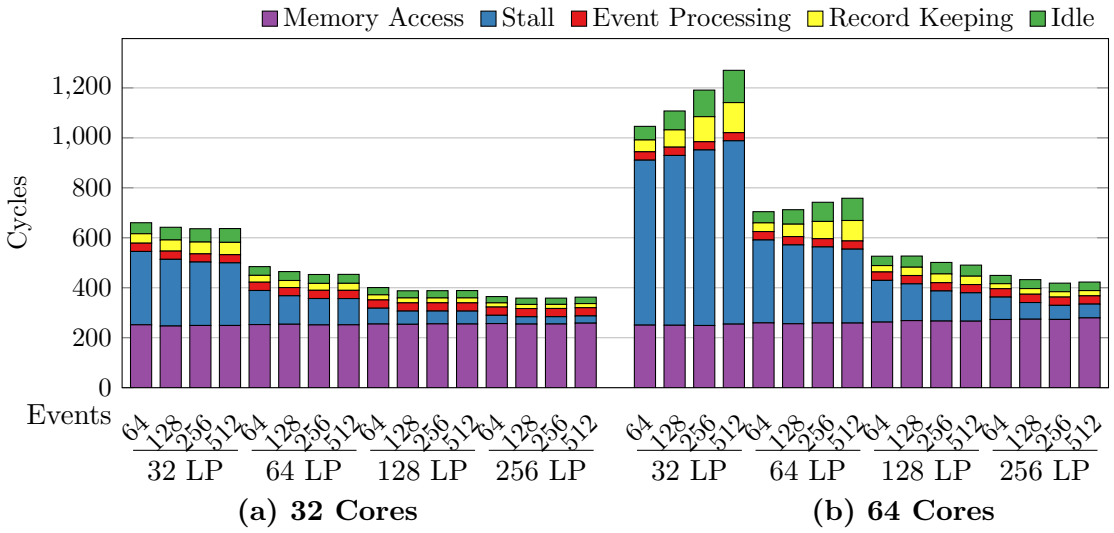


Figure 3.7: Breakdown of time spent by the event processors on different tasks to process an event using (a) 32 event processors and (b) 64 event processors with respects to different number of LPs and initial event counts.

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.

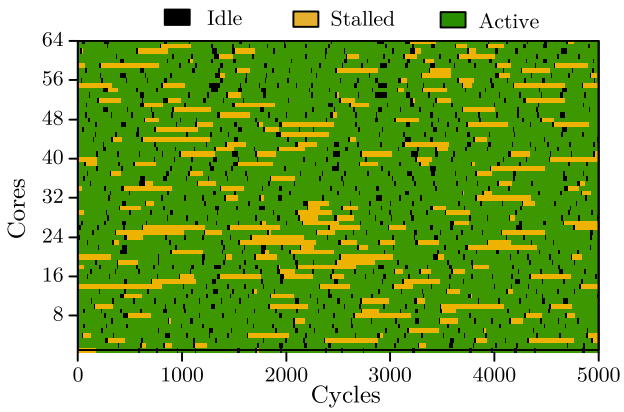


Figure 3.8: Timeline demonstrating different states of the cores for during a 5000 cycles frame of the simulation.

3.4.3 Breakdown of event processing time

Figure 3.7 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 the number of initial events is low. Conversely, the average number of cycles goes down as more events are issued to the system or the number of LP increases (which reduces the stalling probability). The reason for this behavior is apparent from the breakdown of the event cycles. We notice that about the same number of cycles are consumed for memory access regardless of the system's configuration 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 core holds an event for an LP that has another active event at the moment. A system of 64 LP has over 150 stall cycles on average with 64 processors. The stall times drop substantially as we increase the number of LPs and events. These dependencies result in many 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 LP is close to the number of event processors. As the number of LPs increases, the events are more distributed in terms of their associated LP and can be safely processed in parallel. Even if stalls are less frequent, each can take a long time to resolve.

Figure 3.8 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 color 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 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.

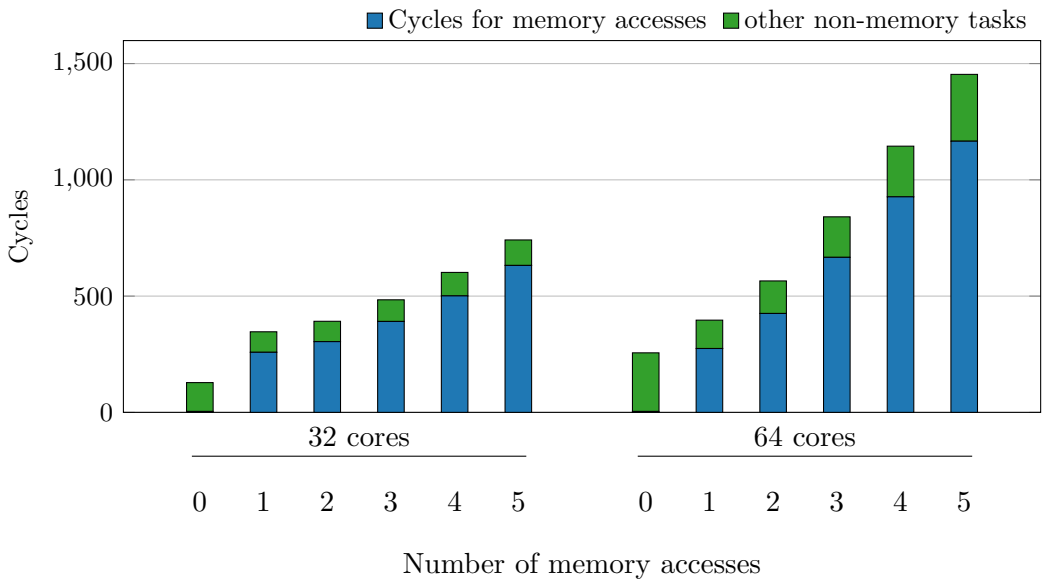


Figure 3.9: Effect of number/size of state memory access on event processing time

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.

3.4.4 Memory Access

Memory access latency is a dominant part of the time required to process an event. Figure 3.9 shows the effect of variation of state memory access pattern on average execution time. The number of 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 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. In an optimized event processing logic, the processor may continue operation with partially available state memory to overlap computation and communication time.

3.4.5 Effect of event processing time

Figure 3.10 shows the effect of event processing time on the system performance. Since a computation can be synthesized differently in a hardware (single-cycle combinational vs multi-cycle sequential) with different resource usage footprint, different processing time may be achieved for 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.

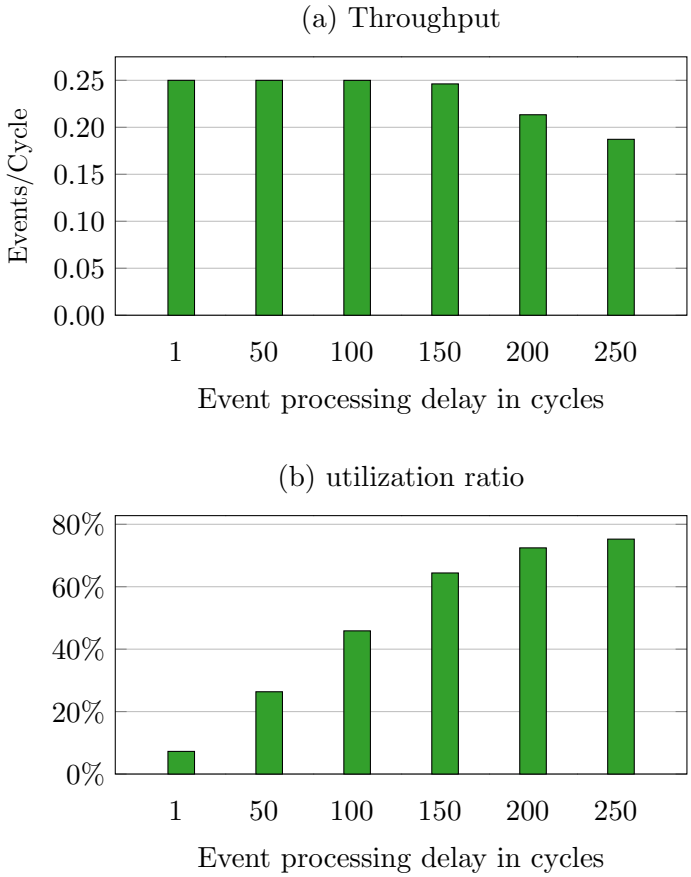


Figure 3.10: Effect of variation of processing delays (in cycles) on (a) throughput, (b) ratio of core utilization for 64 event processors with 256 LP and 512 initial events.

The results of this study are shown in Figure 3.10-a. A higher processing time represents computations for models that have computationally intensive event processing. When the processing time is small, changes in the processing time does 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 3.10-b), and increasing the event processing time start 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.

Chapter 4

Decoupled Datapath for PDES-A

From the performance analysis of the baseline PDES-A simulation engine presented in the previous section, we observed that a critical 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 4.2-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. For example, 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 cannot meet the demands of the event processors. Additional problems can be observed in Figure 3.10-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. Hence, because of contention, memory optimizations (the other major bottleneck) will not be rewarded with a proportional improvement in performance.

In this chapter, we describe a reorganization of the PDES-A datapath in order to alleviate these bottlenecks originating from contentions at different interfaces and naive scheduling.

4.1 Datapath Optimization via structure partitioning

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 the 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 mechnism (e.g., crosspoint 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 number of processors are 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 3.2.3, 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 noticed that violating order between events associated to the same LP causes rollbacks which keeps 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 3.2.3) so that *insert* and *delete* operations become independent of each other. All event processors were given the ability to push events to any queue interfaces based on the target LP using a crossbar. Furthermore, the task of governing event issue was separated into a controller that translates event requests from available event handlers to event issue instructions for the cores. Based on the status of the event queues, this controller optimizes 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, complicating routing with other network structures. 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.

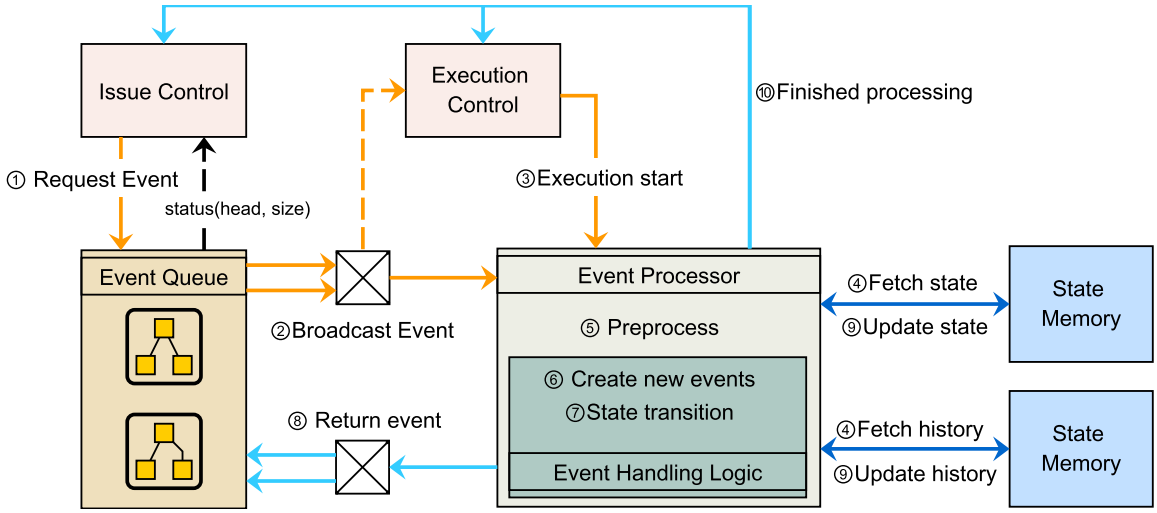


Figure 4.1: Overview of an event processing cycle

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, but in practice, since the design keeps the LPs within similar simulation time of each other, this effect is minimal.

4.1.1 Decoupled Event Processing Flow

Figure 4.1 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 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 keep a record of the LP-processor association. The controller checks its whether it is safe to process the event and signals the event processor in step 3 to start execution when it's not going to create any conflicts with other event processors.

Compute Phase

The event processors, upon receiving this signal, fetches state memory and processed events list in step 4. The event history is checked in the preprocessing step (step 5) to find if 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 step 6 and 7, the event handler performs rollback computation if necessary, computes new states based on the model, and create 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,

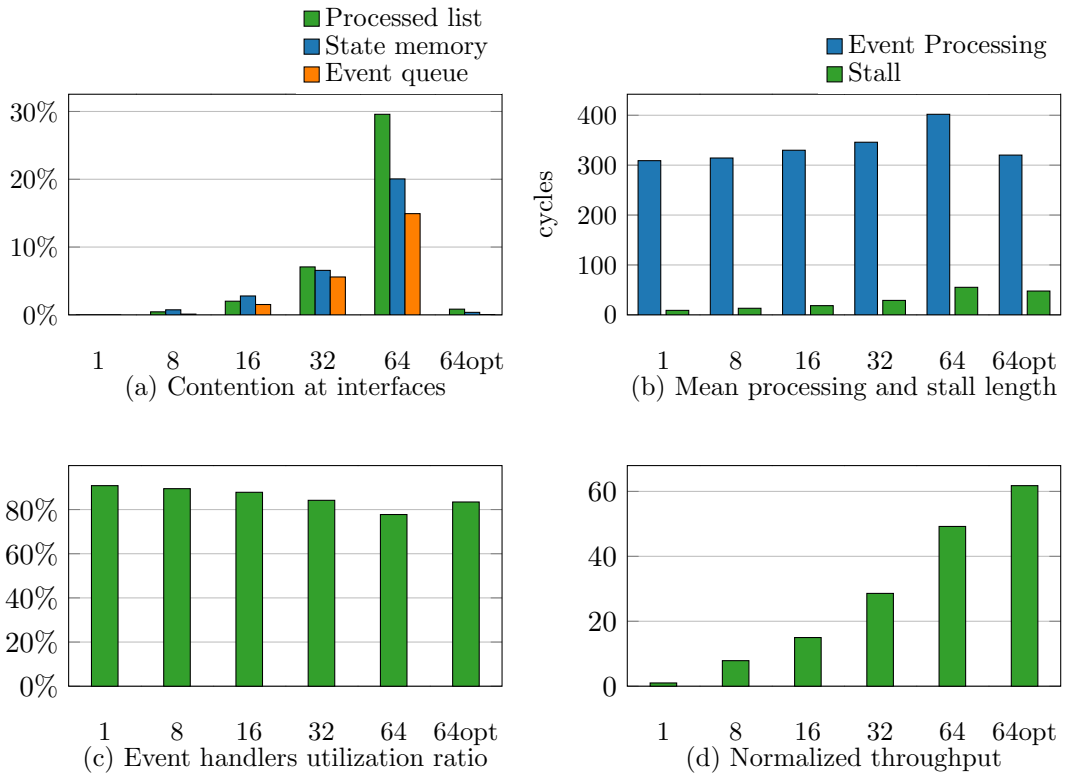


Figure 4.2: Effects of optimized dataflow and concurrent resource access on PDES-A accelerator performance compared to the baseline engine with different number of event handlers.

the updates to state memory and processed events list are written to the memory in step 9. At this point, the 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. Afterward, the processor prepares to receive a new event.

4.1.2 Operational Characteristics

Figure 4.2 shows different performance measures of the optimized design along with the baseline design at different point of scaling. We see in figure 4.2-a that the 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 4.2-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 4.2-c, which shows improvement in the fraction of time a processor remains active. Finally, all these effects combine into significant improvement in throughput.

This organization is highly throughput oriented and almost completely removes interdependence among the data flow paths. The only remaining source of divergence is straggler events. 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 4.2-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.

4.2 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[18].

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 requires floating point

numbers and vectors computations, long iterative operations, and traverses many conditional branches. A programmer can reduce these expensive tasks to only few cycles with hardware support. This will contribute to massive throughput gain in hardware compared to 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.

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 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 only 5% to allow marginal communication between cores.

Table 4.1 shows the parameters for both the systems used in comparison. Their performance reported in Table 4.2 include the numbers for both the baseline and optimized architecture. At this configuration, baseline PDES-A can process events 2.5x faster than a

Table 4.1: Summary of the configurations used for performance comparison of ROSS and PDES-A using Phold model

Parameters	ROSS	PDES-A
System		
Device	Intel Xeon E5-1650	Xilinx Virtex-7
	12 MB L2	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 Events	5%	100%

12-core CPU version of ROSS and 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 presence of multiple type of events. We achieve about 1.5x performance gain over ROSS. The gain drops compared to Phold model. LPs in this model send two-third of the events to self. Therefore, a the processors are often processing same LPs and had to stall more to avoid conflicts. This result highlights the need to implement new

Table 4.2: Comparative analysis of PDES simulation performance for Phold model 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

strategies to reduce stalls, such as interrupt based preemption, workload reassignment etc. We are exploring these optimizations for the next iteration of our design.

4.3 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 4.3. 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 which will remain constant when the simulator size scales. The core simulator logic utilizes 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.

Table 4.3: 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%

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 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 3.6-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 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 require proportional increase in memory for states and processed event history. The on-chip memory is a limited resource which 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 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[91].

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 in-

terconnected in a larger simulation environment. Designers may also also choose to develop 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.

Chapter 5

Event-Driven Execution Model for Graph Processing

Extracting massive parallelism is key to obtaining higher performance on large graphs. However, it is challenging to build an efficient parallel graph processing application from the ground up. Software graph processing frameworks solve this issue by providing simple primitives to the user for describing the algorithm specific operations and relying upon runtime system for complex data management and scheduling. Decoupling the application logic from low level management exposes opportunities to integrate many optimization techniques that are opaque to the application programmer. However, software frameworks do not fully address locality challenges originating from the irregular memory access patterns and synchronization requirements of graph applications. In this chapter, we discuss some relevant background on graph processing models and elaborate our design of an event-driven model for overcoming the limitations on existing graph processing techniques.

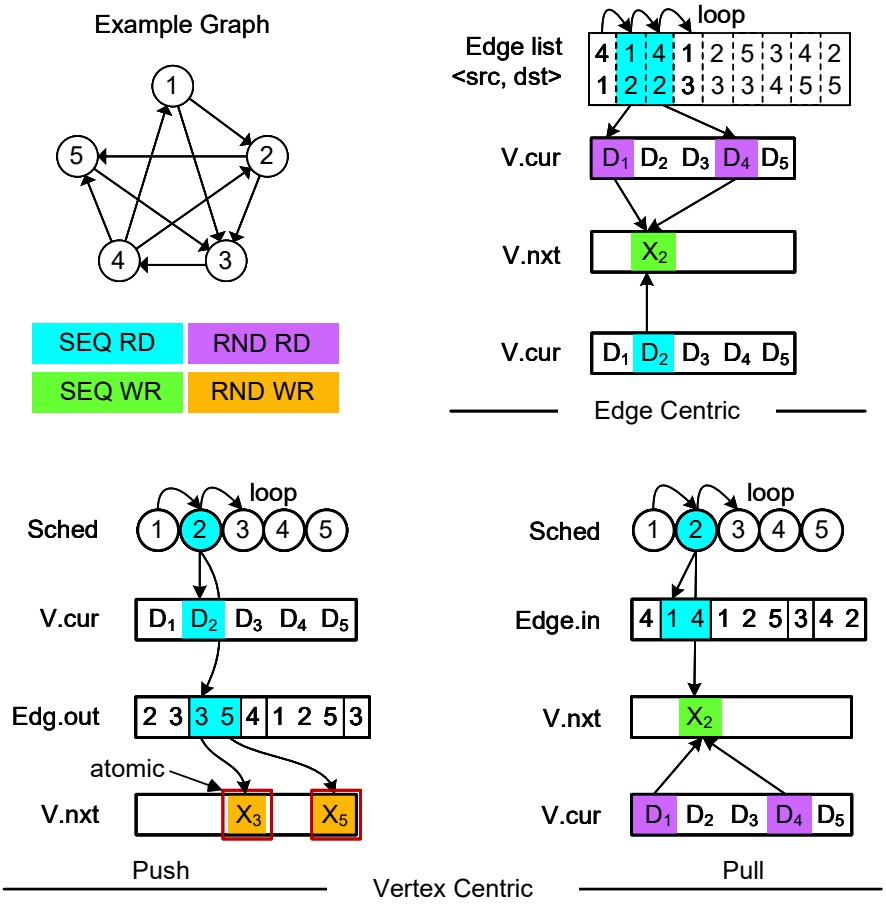


Figure 5.1: Data access patterns for conventional graph processing models: Edge Centric and Vertex Ordered (*Push* and *Pull* directions) processing paradigms.

5.1 Conventional Computation Models

Graph processing frameworks usually follow either *Vertex-Centric* or *Edge-Centric* paradigm for sequencing their computation. In these frameworks, graph memory contains three components: 1) a vertex property memory containing the vertex attributes; 2) a graph structure specifying the relationships, i.e. edges; and optionally 3) memory for intermediate states of computation in progress.

The scheduling determines the order in which the vertex or structural properties in memory are accessed. The memory access patterns for various approaches are shown in Figure 5.1. In the *vertex-centric* paradigm the vertex computation performed is designed from the perspective of a vertex, i.e. vertex property value is updated by a computation based upon property values of its neighbors [69]. Most vertex-centric computation models follow one of two approaches: *pull* or *push*. In the *pull* approach, each vertex reads the properties of all its incoming neighbors and updates its value. Thus, it involves random reads; many of which are redundant as the vertex values read may not have experienced any change and hence do not contribute any change to their outgoing neighbors. These redundant reads lead to poor utilization of memory bandwidth and wasted parallelism due to memory latency. On the other hand, *push* approach performs a vertex *read-modify-update* operation for each of its outgoing neighbor. These updates must be performed via atomic operations. Since graph processing algorithms suffer from poor locality resulting in frequent cache misses, atomic operations are very inefficient. For example, a Compare-And-Switch (CAS) operation on an Intel Haswell processor is more than 15 times slower when data is in RAM vs in L1 cache [86].

In an *edge-centric* model, the edges are sorted, typically in the order of their destination vertex ids, and streamed into the processor. The processors read both source and destination to perform the vertex update. This approach either suffers from redundant reads of inactive source vertices or locking overhead of destination vertices. The memory traffic for reading edges of a vertex v typically achieves high spatial locality since the edges are stored in consecutive locations. However, vertex accesses have poor spatial locality

as v can be connected to other vertices that are scattered in memory; there is a little chance of vertices being stored in consecutive memory locations. Thus, vertex traffic suffers significantly due to memory access latency being on the critical path. Additionally, since the graphs are large, the reuse distance of a cached memory is also large, i.e. temporal locality is non-existent. Thus, on-chip caches are mostly ineffective and compute resources are poorly utilized.

Without maintaining active sets, many vertices will be read unnecessarily as their values would not have changed in prior iterations. One could simply process all vertices in each iteration and forego the need for maintaining active sets, but this is extremely wasteful because the number of vertices that are active can vary greatly from iteration to iteration. To avoid processing of all vertices, software frameworks typically invest in tracking the *active set* of vertices. While this tracking eliminates redundant processing, it unfortunately incurs significant overhead for maintaining the active set in the form of bitvector or a list.

Efficient tracking of active set in hardware accelerators is difficult to achieve. The inherent simplicity of the vertex-ordered scheduling is lost due to scheduling and synchronization overheads in the hardware. Additionally, the efficacy of many performance-optimizing hardware primitives is reduced due to the irregularities introduced by active set scheduling.

5.2 Delta-based Accumulative Processing

GraphPulse targets graph algorithms that can be expressed as a delta-accumulative computation [116] – this includes many popular graph processing workloads [105, 104, 39,

116, 117]. In this model, updates aimed at a vertex by different incoming edges can be applied independently. A vertex whose value changes, conveys its *"change"* or *delta* to its outgoing neighbors. The neighbors update themselves upon receiving the *delta*, and propagate their own delta further. Thus, the computation is turned into a data flow computation that remains active as long as necessary until convergence. The continuous tracking of the active set is inherent to the data flow model. The updates are broken into two steps:

$$\begin{cases} v_j^k & = & v_j^{k-1} \oplus \Delta v_j^k \\ \Delta v_j^{k+1} & = & \sum_{i=1}^{n} \oplus g_{\langle i,j \rangle} \left(\Delta v_i^k \right) \end{cases} \tag{5.1}$$

v_j is the vertex state. Δv_j is the change to be applied to the vertex using algorithm specific operator '$\oplus$'. The two equations can be visualized as a sequence of recursive operations going back to the initial conditions v_j^0 and Δv_j^0 that are also specific to the algorithm under consideration. We highlight two key components in the equation: $g_{\langle i,j \rangle}$, the *propagate* function, which modifies and conveys the change(*delta)* in the vertex value to its neighbors; and '$\oplus$', the *reduce* function, that both reduces the propagated *deltas* to compute new *delta* and applies it to the current vertex state. To express an iterative graph algorithm in the incremental form, we make use of the following two properties:

Reordering Property. The *deltas* can be applied to the vertex state in any order. This reordering is allowed when the propagation function $g_{\langle i,j \rangle}$ is distributive over $\oplus$, i.e., $g(x \oplus y) = g(x) \oplus g(y)$; and $\oplus$ is both commutative and associative, i.e., $x \oplus y = y \oplus x$ and $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.

Simplification Property. Given an edge $i \rightarrow j$, the '$\oplus$' operation is constructed to incrementally update the vertex value v_j when there is a change in v_i. Therefore if v_i does

not change, it should have no impact on v_j. That is,

$$v_j^k \oplus g_{\langle i,j \rangle}\left(\Delta v_i^k\right) = v_j^k \text{ if } \Delta v_i^k = 0$$

This property is satisfied when $g_{\langle i,j \rangle}(\cdot)$ is constructed to emit an *Identity* value for the reduce operator $\oplus$ when $\Delta v_i = 0$.

A wide class of graph algorithms – PageRank, SSSP, Connected Components, Adsorption, and many Linear Equation Solvers – satisfy the above properties [116]. However, there are exceptions. For example, graph coloring cannot be expressed since the update is a function of all vertex values obtained along the incoming edges, i.e. they cannot be updated using a value obtained along a single edge. Delta-based update algorithms break the iteration abstraction, allowing asynchronous processing of vertices and thus substantially increasing available parallelism, removing the need for barrier synchronization at iteration boundaries (required by the *Bulk Synchronous Parallel* Model [100]), and providing opportunities for combining multiple delta updates. All these properties are exploited by GraphPulse to improve performance.

Limitations to the model

We assume that Reordering and Simplification preserve correctness; however, some graph algorithms do not satisfy this condition and thus cannot be expressed using our model. For example, Graph Coloring, K-Core, and MIS algorithms require vertex contribution across all incoming edges to update a vertex. This violates the Simplification Property since contributions from some neighboring vertices are needed even if their states were unchanged. If the algorithm requires contributions from neighbors that are multi-hop away

(e.g., Triangle Counting) or a normalization step after each iteration (e.g., Label Propagation), then they violate the Reordering Property because a particular order must be imposed upon the evaluation of the contributions through some edges. These algorithms cannot be implemented in GraphPulse. It should be noted that some algorithms that are not supported in their common iterative forms may have variations that may be suitable for event-driven implementations. For example, PageRank and Adsorption have incremental forms that are supported in GraphPulse and JetStream. As a rule of thumb, algorithms supported by this model *often* have the characteristic that a single edge can update a vertex, and the updates are monotonic.

5.3 Overview of Event-Driven Graph Processing

Before introducing the GraphPulse accelerator architecture, we overview important considerations in the event processing model (see Algorithm 1). This section also discusses the mapping of a delta-based graph computation to GraphPulse.

5.3.1 Event-Processing Considerations

Computation with Delta/Data Carrying Events

In the delta-based model, the only data that is passed between vertices are the *delta* messages. These messages (implemented as events) encode the computation and carry the input data needed by the computation as well, removing the need for expensive reads of the input set of a vertex computation. Moreover, vertex updates can be performed asynchronously; in other words, a vertex can be updated at any time with the *delta* it has

Algorithm 1 Event-Driven Execution Model for SSSP

$V[:] \leftarrow fill(\infty)$ ▷INITIALIZEVERTEX()

$Q \leftarrow insert(\{\langle root, 0\rangle\})$ ▷ INITIALEVENTS()

1: **procedure** COMPUTE($G(V, E)$, Q)
2: **while** Q is not *empty* **do**
3: $\langle i, \delta_i \rangle \leftarrow pop(Q)$
4: $temp \leftarrow V[i]$
5: $V[i] \leftarrow min(V[i], \delta_i)$ ▷ REDUCE(a, b)
6: **if** $V[i] \neq temp$ **then** ▷ Needs to propagate
7: **for each** $\langle u \rightarrow v, w \rangle \in E \mid u = i$ **do**
8: $\delta_v \leftarrow V[u] + w$ ▷ PROPAGATE(u, v, w)
9: $Q \leftarrow insert(\langle v, \delta_v \rangle)$
10: **end for**
11: **end if**
12: **end while**
13: **end procedure** ▷ Converged graph state in V

received so far. Based on these two properties, we develop an event-driven model to support delta-based graph computation. This approach completely decouples the communication and control tasks of the graph computation.

We define an *event* as a lightweight-message that carries a *delta* as its payload. Multiple events carrying *deltas* to the same vertex can be combined using a *reduce* operator specific to the application to reduce the event population and, subsequently, event storage and processing overheads. Execution of a vertex program can only be triggered by an event.

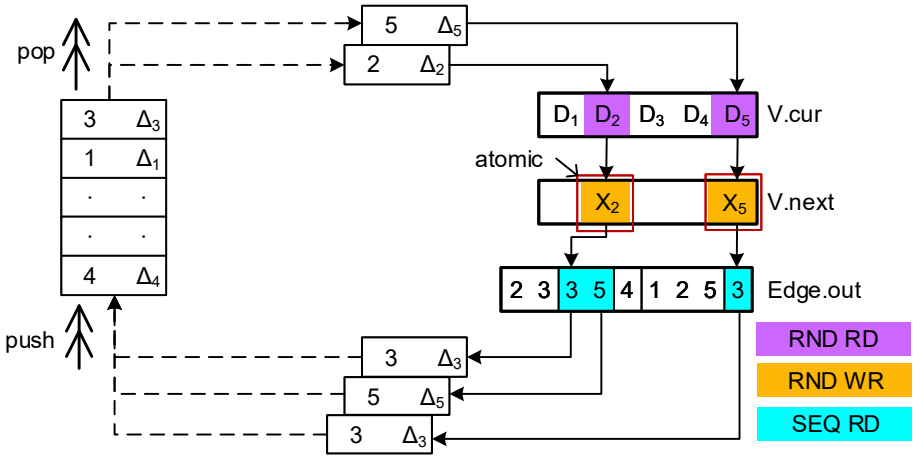


Figure 5.2: Data access pattern in event-driven approach with a FIFO event-queue.

A set of initial events is created at the beginning as part of the application initialization. When a processor receives an event for a vertex, it executes two tasks: 1) *update* of the vertex state using the *reduce* operation, and then 2) *generate* a new set of events using the *propagate* function described in Section 5.2. The newly generated events are collected in an event queue from which they are scheduled to other processors to start execution of new vertex programs.

Figure 5.2 shows a view of the computation model. At any time, the event queue has a set of pending events. The event at the head of the queue is issued to a processor which read-modify-writes the vertex value, then reads the corresponding adjacency list to prepare and insert new events into the event queues. The memory accesses are still in random order and require locking for parallel operation since two or more events to the same vertex may be issued from the queue; our optimized design mitigates both of these limitations.

Coalescing Inflight Events

As discussed in Section 5.2, the reordering property of the propagation parameter allows the architecture to combine multiple events to the same destination while still in the queue using the *reduce* function without affecting program correctness; we call this operation *event coalescing*. Event coalescing is critical for a practical asynchronous design because every event in the queue can cause the generation of new events for every outgoing neighbor or destination vertex, unless a termination condition is met. Consequently, for every event consumption, new events are produced and the number of events in the system will rapidly grow. For designing an event-driven processor with limited storage, we require the event coalescing capability to ensure control over the rate of event generation.

Implicit Atomic Updates

In parallel execution, processors may attempt to update the same vertex's state simultaneously, necessitating locking or atomic updates. In Graphpulse, all the vertex memory accesses are associated with an event, and an event only modifies a single vertex value. With the guarantee that, via coalescing, no more than one event is in-flight for any vertex, safety for atomic access is naturally ensured. Our implementation completely coalesces all events targeted to a vertex into one before it is scheduled preventing race conditions that can otherwise arise in presence of concurrent updates.

Isolating Control Tasks from Computation

All memory accesses to vertex and edge data are isolated to the algorithm specific task processing logic. The control tasks, which primarily consist of scheduling of vertex

operations, are naturally encapsulated using the *events* abstraction, and do not require any accesses to the graph data to schedule their computation. Coupled with the guarantee of memory consistency, this isolation makes the vertex scheduling logic extremely simple and the datapath highly independent and parallelizable. Also, the memory interfaces designed are simple and efficient since there are only simple memory accesses to the vertex properties. This model reduces memory accesses compared to the classical graph processing approaches including *Vertex-Centric Push/Pull* and the *Edge-Centric* paradigms.

Active Set Maintenance

The events resident in the queue encapsulate the entire active computation, which provides an alternative way to manage *active sets* using hardware structures. Vertices that are inactive will have no events updating them; and the set of unprocessed events indicate a set of vertices that are to be activated next. Most existing graph frameworks use bitmaps or vertex-lists to maintain an active set which entails significant management overhead. The event maintenance task is decoupled from the primary processing path in our model which results in greater parallelization opportunities. Efficient fine-grained control over the event-flow, thereby the scheduling of vertices, can be achieved via hardware support.

Initialization and Termination

After loading a graph, we bootstrap the computation as follows. We define an *Identity* parameter that, when passed to the *reduce* operator with another value, leaves the latter unchanged (e.g., 0 is *identity* for the *sum()* operation). We set the vertex memory to the identity parameter for the graph. The initial events, that are set with the initial target

Table 5.1: Functions for mapping algorithm to GraphPulse

	propagate(δ)	reduce	$V_{j,init}$	$\Delta V_{j,init}$
PR-Delta	$\alpha \cdot E_{i,j} \cdot \delta$/`N(src)`	+	0	$1-\alpha$
Adsorption	$\alpha_i \cdot E_{i,j} \cdot \delta$	+	0	$\beta_j \cdot I_j$
SSSP	$E_{i,j} + \delta$	min	∞	0 (j=r); ∞
BFS	0	min	∞	0 (j=r); ∞
Conn. Comp.	δ	max	-1	j

value of the vertices, populate the event queue. The first event of a vertex is guaranteed to trigger a change and then propagate it to other vertices to bootstrap the computation. The event population eventually declines as the computation converges; an update event may not generate new events if the update magnitude is below a threshold (e.g., in PageRank). Eventually, the event queue becomes empty without replenishment and the application terminates when there is no more event to schedule. We also provide the ability to specify a global termination condition for better control with some applications (see Section 6.1.2).

5.3.2 Application Mapping

To implement a delta-based accumulative graph computation for compatible applications with our model shown in Algorithm 1, the user must define the few functions described next. Table 5.1 shows the reduction and propagation functions, and the initialization values for five graph applications.

`Reduce` function expresses the *reduction* operation that accumulates incoming neighbors' contributions to a vertex, and coalesces event *deltas* in queue. It takes a delta value and current vertex state to update $state = state \oplus delta$.

`Propagate` function expresses the source vertex's *propagation* function ($g(x)$) that generates contributions for the outgoing neighbors. It uses the change in state to produce outgoing *delta*, $\Delta_{out} = g_{\langle E.src,E.dst \rangle}(\Delta V)$.

`Initialization` function defines the initial vertex states to an *identity* value for the reduction operator. Also, the initial event delta is set such that `Reduce(Identity, delta)` results in the intended initial state of the target vertex.

`Terminate` function defines a local *boolean* termination condition in the framework that checks for changes in the vertex state. Propagation for an event stops when the local termination condition is valid and the vertex state is unmodified. The program stops if all events have terminated locally.

Application Programming Interface

Due to the simple programming abstraction, user effort is modest. The user can define program logic in HDL using custom functional modules and pipeline latency, or use some common functional modules in GraphPulse (e.g., *Min, Max, Sum*). The user also creates the array of initial events and vertex states, which are written to the accelerator memory and registers by the host CPU.

Chapter 6

GraphPulse: an Asynchronous Graph Processing Accelerator

GraphPulse is an event-based asynchronous graph processing accelerator that leverages the decoupled nature of event-driven execution. The event processing datapath exposes the computational parallelism and exploits available memory bandwidth and hardware resources. The accelerator takes advantage of low-latency on-chip memory and customizable communication paths to limit the event management and scheduling overheads. The following insights from Section 5.3.1 guide the datapath design:

1. Vertex property reads and updates are isolated and independent, eliminating the need for atomic operations. When sufficient events are available for processing, the throughput is only limited by the memory bandwidth.
2. To sustain many parallel vertex operations, it should be possible to insert, dequeue, and schedule events with high throughput.

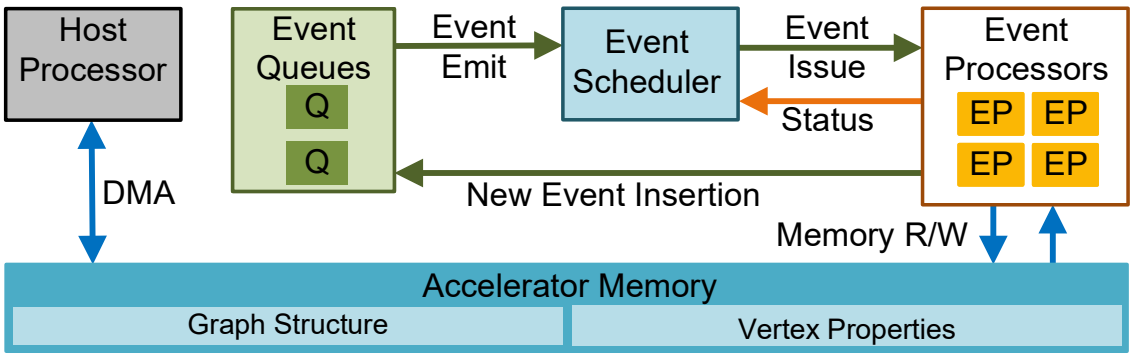


Figure 6.1: Overview of GraphPulse Design

3. Since no explicit scheduling is needed, the number of parallel vertex processing tasks can be easily scaled to process increasingly larger graphs.

We next describe a baseline implementation guided by these considerations, and then describe the optimizations we incorporate to improve its performance.

6.1 GraphPulse Architecture

Figure 6.1 overviews the architecture of GraphPulse. The primary components of the accelerator are Event Queues, the Event Scheduler, Event Processors, the System Memory, as well as the on-chip network interconnecting them. The event processors directly access the memory using an efficient high-throughput memory crossbar interface. For scalability our goal is to leverage the bandwidth to support high degree of memory parallelism and simultaneously present many parallel requests to memory. Because events are the unit of computation, we aim to fit all active events in on-chip memory to avoid having to spill and fetch events. However, for larger graphs, this is not possible, and we use a partitioning

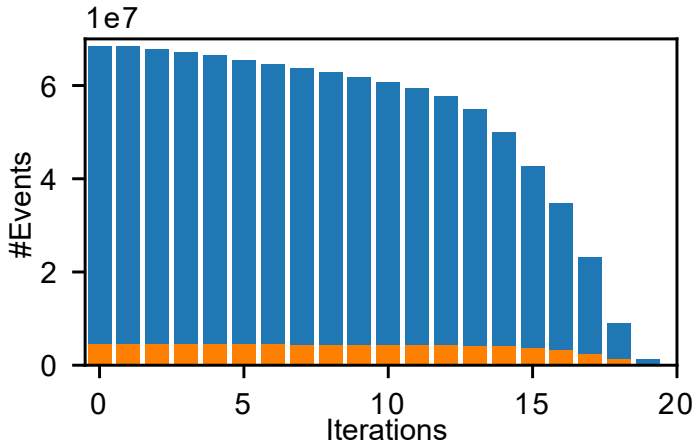


Figure 6.2: Total events produced (blue) and remaining after coalescing (orange) with the event-driven execution model in GraphPulse.

approach to support them (see Section 6.1.6). We consider a configuration with 256 event processors for our baseline.

6.1.1 Event Management

Event Queue stores the events representing the active vertex set of the graph. Events are stored as a tuple of destination vertex ID and payload (delta). Schedulers drain events from the queue in sequence giving them to the processors, while newly generated events are fed back to the queue. Since events are generated for all edges, the volume of events grows rapidly, which represents an obstacle for efficient processing. Moreover, due to the asynchronous processing, multiple activations of a vertex can coexist that then generate multiple set of events over the vertex edges. Figure 6.2 shows the total number of events produced during each iteration and the number of events remaining after coalescing for PageRank running on the LiveJournal social network graph [6] (∼5M nodes, ∼69M edges). We see that over 90% of the events are eliminated via coalescing multiple events destined

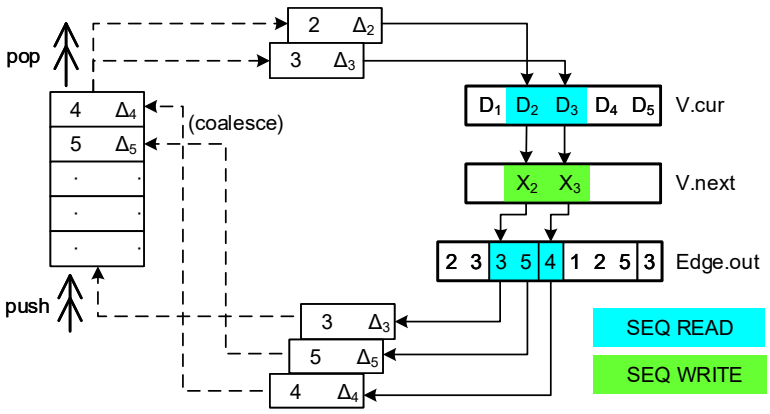


Figure 6.3: Data access pattern in event-driven approach with coalescing & sorting.

to the same vertex. Dramatic reduction in the number of events also reduces the numbers of computations and memory accesses. The queue is modeled as a group of collector bins dedicated to a subset of vertices to simplify and scale event management. We set up the mapping of vertices to bins such that a block of vertices close in memory map to the same bin. Thus, when events from a bin are scheduled, the set of vertices activated over a short period of time are closely placed in memory and thus the memory accesses exhibit high spatial locality. The ordering approach transforms the inefficient random read/writes into sequential read/writes as shown in Figure 6.3.

Events are deposited in the bin and the coalescer iterates over the events and applies the Reduce methods over matching events. Following coalescing passes, only a small fraction of unique events remain. However, buffering uncoalesced events significantly increases pressure on the event queues, increases congestion and requires large memory. Therefore to address this limitation, in Section 6.1.3, we present an in-place coalescing queue that combines events during event insertion.

6.1.2 Event Scheduling and Termination

The event scheduler dequeues a batch of events in parallel from the collectors. It arbitrates and forwards new events to the idle processors via the interconnection network. Scheduler drains events from one bin at a time, and iterates over all bins in a round-robin manner (other application-informed policies are possible). We call one complete pass over all bins a *round*. The scheduler allocates events to any idle processor through an arbiter network. The processing cycle for an event begins with the event scheduler dequeueing an event from the output buffer of the event queue when it detects an idle processor. The event is sent via the on-chip routing network to the target processor. Upon receiving an event, the processor starts the vertex program that can cause memory reads and updates of the vertex state. After processing the event, the processor produces all output events to propagate update of its state to directly impacted vertices along outgoing edges. The events produced are sent to the event queues mapped to the impacted vertices.

Global Termination Condition

The scheduler maintains an accumulator to store the local progress from the processors after they perform updates. The default behavior is to terminate when no events remain. However, for applications that can propagate events indefinitely, an optional termination condition provides a way to stop the execution based on a user defined condition such as a convergence threshold. For example, PageRank terminates when the sum of changes in score of all vertices are lower than a threshold. Here, processors pass the deltas as local progress updates to the scheduler where they are summed. A pass over the queue means

all active vertices are accessed once, and the global accumulator represents global progress, which can be used for termination condition.

6.1.3 In-Place Coalescing Queue

To avoid rapid growth of event population, we explore an in-place coalescing queue that combines events during insertion, compressing the storage of events destined to the same vertex. If no matching event exists, the event is inserted normally. Conversely, if an event exists, the deltas are simply combined based on the application's reduction function.

We use multiple bins inside the queue, with each bin structured like a direct-mapped cache (Figure 6.4). The bins are split into rows and columns, and only one vertex ID maps to a bin-row-column tuple so that there is no collision. Vertex ID isn't stored since the events are direct mapped. Vertices are mapped in column-bin-row order so that clusters in the graph are likely to spread over multiple bins. The number of rows is based on the on-chip RAM block granularity (usually 4096) and multiple memory blocks are operated side-by-side to get a wider read/write interface that can hold power-of-two number of columns. Each bin consists of a *Simple Dual-Ported* RAM block (with one read and one write port).

Event Insertion and Coalescing

Each bin can accept one new event per cycle, but the insertion has multi-cycle latency. Specifically, insertion units are pipelined so that a bin can accept multiple events in consecutive cycles. In the first cycle during the insertion of an event, the event in its mapped block (if one exists) is read using the read port. In the next cycle, the incoming

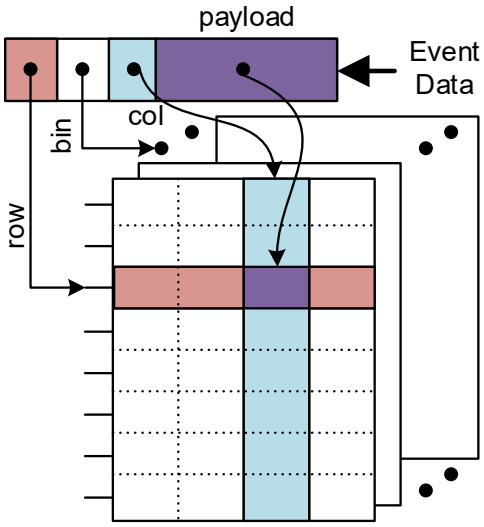


Figure 6.4: An event is direct-mapped to a cell in a queue bin. Bits in the destination vertex Id is used to find cell mapping.

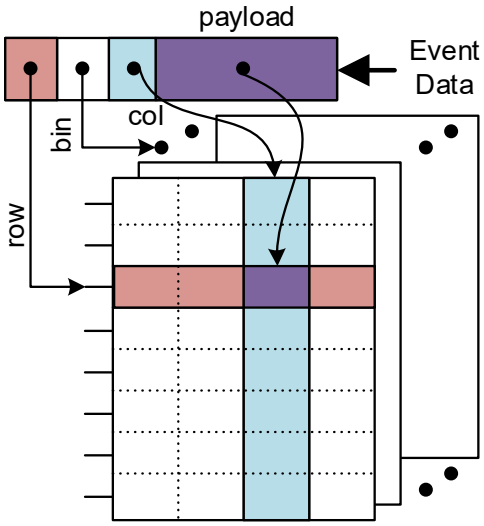


Figure 6.5: In-place coalescing of events and retrieval in the direct mapped event storage (for PageRank).

event is pushed into a combiner function pipeline along with the existing event (FPA unit in Figure 6.5). We use four stages in the pipeline as most common operators can be designed to have less than 4 cycles latency (e.g., 3 cycles for floating point addition) while maintaining desirable clock speed. After the operation is finished, the combined event is sent to the write port. During these 4 cycles, other event insertions can be initiated for another row, since the read-write ports are independent and the coalescer is pipelined. When insertions contend for the same row, the later events are stalled until the first event is written.

Event Retrieval

Events are removed from one bin at a time in a round-robin fashion. When it's time to schedule events from a bin, a full row is read in each cycle and the events are placed in an output buffer. We prefer wide rows so that many events can be read in one cycle. Insertion to the same bin is stalled in the cycles in which a removal operation is active. Often towards the beginning or the end of an application, the queue is sparsely occupied. It might waste many cycles sweeping over empty rows in these situations. We mark the row occupancy using a bit-vector for each bin. A priority encoder gives fast look-up capability of occupied rows during sweeping the queue.

Due to coalescing at insertion, only one event exists for a vertex in the queue. As removal is done by sweeping in one direction in the bins, we can issue only one event for a vertex in a given round. After a round is complete, the scheduler waits until all the cores are idle before rolling over to the first bin again. This guarantees that race conditions cannot occur without the need for atomic operations or per vertex synchronization.

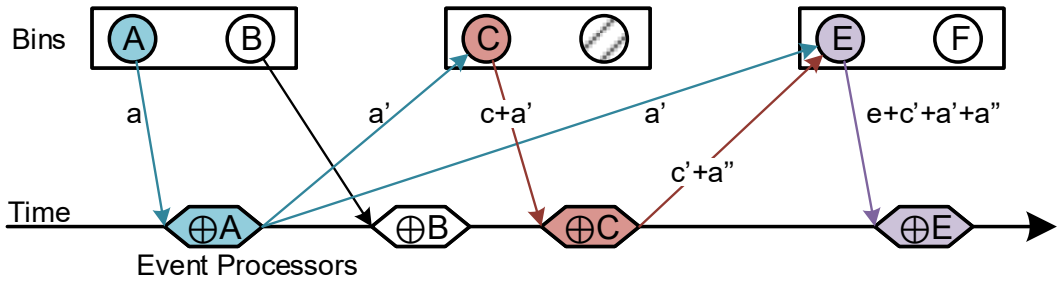


Figure 6.6: Look-ahead: Vertex contributions are compounded across iterations in the event-driven model.

Another advantage of event coalescing is its ability to combine the effect of propagation across multiple iterations, which is a virtue of the asynchronous graph processing model. Figure 6.6 shows an example: the delta from processing event A in bin 1 is sent to vertex C mapped to bin 2, where another event for vertex C already exists. Due to coalescing, vertex C will pick up the contribution that otherwise would have been processed in the next iteration. Similarly event E will compound the effect of A two iterations earlier than usual. We call this effect *lookahead*. In Figure 6.2, we showed that a significant fraction of the events are eliminated by coalescing. Figure 6.7 shows the degree of lookahead contained in these coalesced events for each round in a 256-bin event queue during PageRank-Delta running on the LiveJournal graph. Because of coalescing and asynchronous execution, an event quickly compounds the effects of hundreds of previous iterations of events in a single round. Note that, the contributions from many vertices should quickly stop propagating in uncoalesced model because of damping in PageRank, but they carry on here after being compounded with a bigger valued event. Coalescing exploits temporal locality for the graph, while binning promotes spatial locality, without requiring large caches.

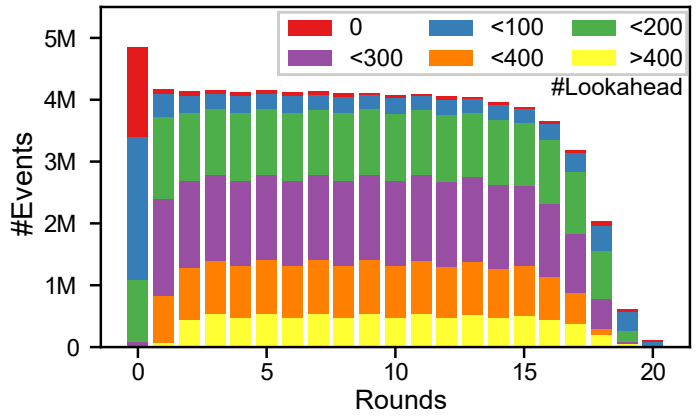


Figure 6.7: Degree of lookahead in events processed in each round.

The event-driven approach is prohibitively expensive to implement in software due to the high overhead for generation, management, queueing, sorting, coalescing and scheduling of events using message passing in software. However, since these primitives are directly implemented by the accelerator in hardware, the overheads are essentially mitigated.

6.1.4 Event Processors and Routing Network

The event processors are independent, parallel, and simple state machines. The processors are connected to the scheduler using a broadcast network to enable delivery of events from any bin to any available processor. A memory bus connects the event processors to the main memory for reading graph properties. The graph is stored in a Compressed Sparse Row format in memory. The state machine starts after receiving a new event from the scheduler. It reads the vertex property from memory, computes update from the received event using the `reduce()` function, and writes update to the memory in the subsequent steps. It resolves local termination check, and starts reading from *EdgeTable* if

it is not terminated. Then, it uses `propagate()` function to compute new delta using the neighbor ID. It pushes the new events to a broadcast channel which connects to the event queues where they are picked up. After finishing its tasks, the processor generates a local progress update (also defined by the application) that is passed to the scheduler along with the processor's status message for global progress checking. In our evaluation, we assumed that event processing logic is specified via a Hardware Description Language, resulting in specialized processors for the application. However, the function encapsulation provides a clean interface to build customizable event processors or use a minimalistic CPU for the event processors.

The baseline GraphPulse configuration consists of 256 processors on a system connected to 4 DRAM memory controllers and coalescing event queues. The scheduler-to-processor interconnect for the baseline design is a multi-staged arbiter network. The processor-to-queue network is a 16x16 crossbar with 16 processors multiplexed into one crossbar port. The complexity of the network is minimized by a number of characteristics of the design: (1) we only need unidirectional dataflow through the network; (2) the datapath communication can tolerate delays arising due to conflicts enabling us to use multi-stage networks and to share ports among multiple processing elements; and (3) our events are fixed in size so that we do not face complexity of variable size messages. We note that the optimizations in Section 5 allow us to retain performance with a much smaller number of cores which further reduces interconnect complexity.

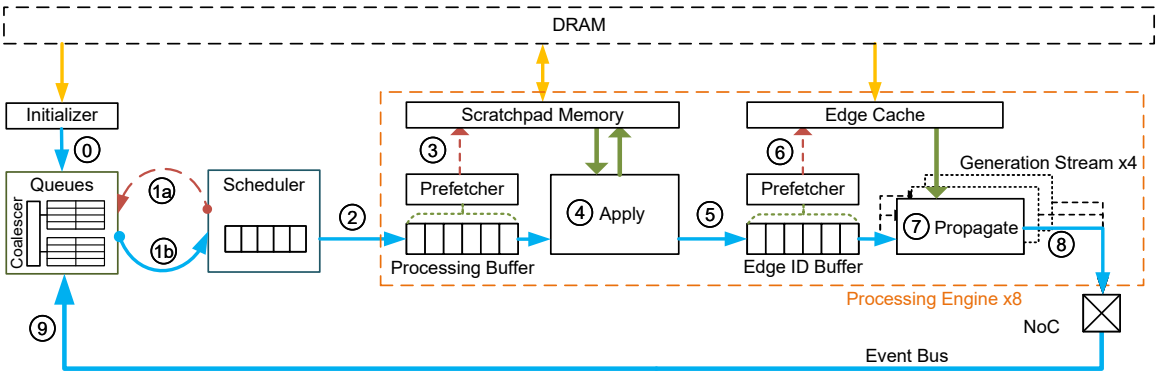


Figure 6.8: Detailed GraphPulse datapath. Blue arrows show data flow, red arrows indicate control signals, green and yellow arrows represent on-chip and off-chip memory transfers respectively.

6.1.5 GraphPulse Execution Flow

Fig. 6.8 shows the steps and direction of the dataflow during the life-cycle of an event in GraphPulse. Execution starts with some events already loaded in the queue, and continues until the queue is empty.

Initialization. We assume that the accelerator starts with the host processor writing the graph structure, initial vertex states, and a list of initial events corresponding to the application to the main memory. Then, during step ⓪, the *Initializer* module reads and inserts the initial events into the queue to make the system ready for processing.

Event Issue. In step ①a, the scheduler requests events from the queue, and the queue emits events (if any) in batches in ①b. The steps in ① execute in a continuous loop. The scheduler holds the events in a buffer and passes them to the processing buffer in ② where they are staged for execution.

Vertex Update. While the events wait in the queue, the prefetcher scans the vertex id, computes the memory addresses, and prefetches all vertex properties (typically located in

the same memory page) to the scratchpad memory in ③. The *Apply module* takes the event at the head of the buffer, reads vertex states and edge pointers from the scratchpad, and applies the update to the event in ④. After writing back the updated value to memory via the scratchpad, ⟨*update value, edge pointer, number of edges*⟩ for a vertex is pushed to the *Edge Buffer* in ⑤ to generate the outgoing events only if the vertex requires propagation (i.e., its state has been updated).

Event Generation. During step ⑥, the prefetcher computes the edge address range to be read, and fetches all needed edges (typically within a single memory page) to the cache. Each generation stream takes the head of the buffer and loops over all the edges for the vertex to generate new events in ⑦. The events are pushed to an event bus through an on-chip routing network in ⑧. In step ⑨ the event queue continuously scans the event bus to pick up and insert the events in corresponding bins. This processing cycle repeats until the queue is empty; this marks the end of evaluation where the initial graph has been updated to the converged state.

6.1.6 Scaling to Larger Graphs

GraphPulse uses the on-chip memory to store the events in the coalescer queue. Each vertex is mapped to an entry in the coalescer, which puts a limit on the size of the active portion of the graph to be less than the maximum number of vertices serviced by the coalescer. For large graphs, the on-chip memory of the accelerator will, in general, not be big enough to hold all vertices. The inherent asynchronous and distributed data-flow pattern of GraphPulse model allows it to correctly process a portion of the graph at a time.

Thus, to handle large graphs, we partition the graph into multiple slices such that each slice completely fits into the on-chip. Each slice is processed independently and the events produced from one slice are communicated to other slices. This can be achieved using two different strategies: a) on-chip memory can be shared by different slices sequentially over time while the inter-slice events are temporarily stored in off-chip memory; and b) multiple accelerator chips can house all slices while an interconnection network streams inter-slice events in real-time. We use the first option to illustrate GraphPulse scalability.

We assume that the graph is partitioned offline into slices that each fits on the accelerator [56, 93, 94]. Most graph frameworks employ either a *vertex-cut* or *edge-cut* strategy in partitioning graphs. Since our model is dependent on the number of vertices, we limit the maximum number of vertices in each slice while minimizing edges that cross slice boundaries. We relabel the vertices to make them contiguous within each slice. When a slice is active, the outbound events to other slices are stored in off-chip memory. These events are streamed in later when the target slice is swapped in and activated. Partitioning necessarily gives rise to increased off-chip memory accesses and bandwidth demand. However, the events do not require any particular order for storing and retrieval. We buffer the events that are outbound to each slice to fill a DRAM page with burst-write. When a slice is marked for *swap-out*, the bins are drained to the buffer and the new active slice's events are read in from memory. Both the read and write accesses to the off-chip memory is very fast since they are sequential and can be done in bursts. The bins in the queues have their independent pipelined insertion units that can insert the *swapped-in* events in parallel without delay. Event coalescing occurs during insertion of events into the bins. Normal

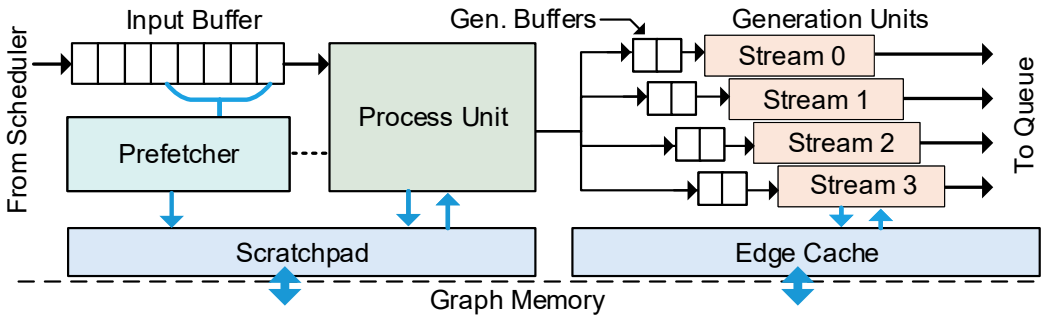


Figure 6.9: Optimization of event processing and generation in GraphPulse.

operation can proceed as soon as the first bin is swapped-in, allowing the swap-in/swap-out process to be pipelined, and masking the *switch-over* latency.

6.2 GraphPulse Optimizations

In this section, we discuss optimizations and extensions to the baseline GraphPulse. Analyzing the performance of the event execution, we discovered that the event processing time was dominated by two overheads: (1) memory accesses to obtain the output vertices needed to identify the targets of the generated events; and (2) the sequential cost to generate the outgoing events. In this section, we introduce two optimizations to alleviate these overheads.

6.2.1 Vertex Property Prefetching

Graph processing applications have notoriously low computation to memory latency ratio. We implement a prefetching scheme to prevent starvation and idling of the event processors. An input buffer is added to the processors and a small scratchpad mem-

ory sits between the processor and the graph memory to prefetch and store vertex properties for the events waiting in the input buffer as shown in Figure 6.9. Prefetching is possible since we know the vertices mapped to each coalescing queue, and we are able to accurately prefetch their outgoing set while they are scheduled for execution. We map the events in the queue such that a block of vertices that are adjacent in graph memory remains adjacent in the queue. The events in a block are *swept* and streamed together to the same input buffer. The predictor inspects a window from the buffer to prefetch only the required data in *cache-line-size* granularity. A carefully sized block (128 in this work) will cause prefetch of all required addresses from a DRAM page together, allowing higher bandwidth utilization than possible via caching alone. Since processors no longer manage data themselves and the memory latency is separated from their critical path, we employ fewer processors (8 in the experiments) to process only the vertices with data available for processing.

We include a small caching buffer with the edge memory reader to enhance the throughput. Prefetching the outgoing edges makes it possible to streamline the generation of events without experiencing expensive memory accesses during event generation. This substantially reduces the event processing time and enhances the event processing throughput. A simple N-block prefetching (N=4) scheme is used for edge memory reads. Since the degree of a vertex are known during the processing phase, we pass this information to the generation unit encoded in the vertex data as a hint for the edge prefetcher to set the limit of prefetching (N) to avoid unnecessary memory traffic for low degree vertices.

6.2.2 Efficient Event Generation

The memory traffic requirement for edge data compared to vertex properties is very high for most graphs: edge data is typically orders of magnitude larger for most graphs. After an event is processed, update events are generated to its outgoing edge set. We observed that this step is expensive and frequently stalls the event processors limiting processing throughput. The data per edge is small (4 bytes in most of our graphs and applications). This makes reading and generation of events for multiple edges in the same cycle essential for saturating memory bandwidth. Since the data dependence between the processing and event generation phase is unidirectional, we decouple the processor into two units: Processing and Generation (see Figure 6.9). We increase the event generation throughput by connecting multiple of these generation streams to the same processing unit. A group of streams in one generation unit share the same cache but multiple ports in the event delivery crossbar. Each generation stream is assigned one vertex from the processing unit when idle. Thus, we use parallelism to match the event generation bandwidth to the event processing bandwidth enabling the processing units to work at or near capacity.

6.3 Experimental Evaluation

Next we evaluate GraphPulse along a number of dimensions: performance, memory bandwidth requirements, hardware complexity, and power consumption. First we describe our experimental methodology.

6.3.1 Experimental Methodology

System Modeling

We use a cycle accurate microarhitectural simulator based on Structural Simulation Toolkit [81] to model the primary components, the memory controller, and interconnection network. The event processor models are designed as state machines with conservative estimation for latency of the computation units. The memory backend is modeled with DRAMSim2 [83] for realistic memory access characteristics. The coalescing engine was modeled as a 4 stage pipelined floating point unit in RTL. The interconnection network is simulated with input and output queue to ensure congestion does not create a bottleneck.

Comparison Baselines

We compare the performance of GraphPulse with a software framework, Ligra [90]. We chose Ligra as the software baseline because, along with Galois [74], it is the highest performing generalized software framework for shared-memory machines [118]. Moreover, Ligra has an efficient shared memory implementation of one of the most robust technique for active set management and versatile scheduling depending on the active set, which is at the core of our work. We considered frameworks that support delta-accumulative processing but those were all targeted for distributed environments and performed much slower than Ligra. We measure the software performance on a 12-core Intel Xeon CPU. The relevant configurations for both systems are given in Table 6.1.

In addition, we compare the performance with a hardware accelerator Graphicionado [38], a state of the art hardware-accelerator for graph processing that uses the Bulk

Table 6.1: Device configurations for software framework evaluation and GraphPulse with optimizations.

	Software Framework	GraphPulse
Compute Unit	12× Intel Xeon Cores @3.50GHz	8× GraphPulse Processor @ 1GHz
On-chip memory	12MB L2 Cache	64MB eDRAM @22nm 1GHz, 0.8ns latency
Off-chip Bandwidth	4× DDR3 17GB/s Channel	4× DDR3 17GB/s Channel

Synchronous execution model. Since the implementation of Graphicionado is not publicly available, we modeled Graphicionado to the best of our ability with the optimizations (parallel streams, prefetching, data partitioning) proposed by the authors. We also gave zero-cost for active vertex management and unlimited on-chip memory to Graphicionado to simplify implementation, making our speedup vs. Graphicionado conservative. We provision Graphicionado with a memory subsystem that is identical to that of GraphPulse.

Workloads

We use five real world graph datasets – Google Web graph, Facebook social network, LiveJournal social network, Wikipedia link graph, and Twitter follower network in our evaluations obtained from the Network Repository [84] and SNAP network datasets [61] (see Table 6.2). We evaluate five graph algorithms – PageRank (PR), Adsorption(AD), Single Source Shortest Path (SSSP), Breadth-first Search (BFS) and Connected Components

Table 6.2: Graph workloads used in the evaluations of GraphPulse.

Graph	Nodes	Edges	Description
Web-Google(WG) [62]	0.87M	5.10M	Google Web Graph
Facebook(FB) [99]	3.01M	47.33M	Facebook Social Net.
Wikipedia(Wk) [26]	3.56M	45.03M	Wikipedia Page Links
LiveJournal(LJ) [6]	4.84M	68.99M	LiveJournal Social Net.
Twitter(TW) [59]	41.65M	1.46B	Twitter Follower Graph

(CC) on each of these graphs. We use the contribution based PageRank implementation (commonly referred to as PageRankDelta), which is a delta-accumulative version of PageRank. PageRankDelta execution was faster than the conventional PageRank in the Ligra software framework and Graphicionado for our graph workloads, and therefore we use it for our baselines as well. Ligra does not provide a native Adsorption implementation. We created randomly weighted edges for the graphs and normalized the inbound weights for each vertex. PageRank-Delta model was modified to consider edge weights and propagate based on the functions provided in Table 5.1 for Adsorption. Twitter is large and does not fit within the accelerator memory; thus we split it into *three slices* with one slice active at a time using the methodology from Section 6.1.6.

6.3.2 Performance and Characteristics

Overall Performance

Figure 6.10 shows the performance of the GraphPulse architecture in comparison to the Ligra software framework. We observe an average speedup of 28× (10× to 74×) for GraphPulse over Ligra across the different benchmarks and applications. The speedup mainly comes from hardware acceleration, memory friendly access pattern, and the on-the-fly event coalescing capability. BFS, SSSP and CC have similar traversal algorithms. However, BFS and SSSP performance suffers because fewer vertices are active at a time and vertices are reactivated in different rounds of the computation in contrast to CC where the full graph is active for the majority of the computation. The Twitter graph achieves comparable speedup to the other graphs, despite the fact that it incurs the overhead of switching between active slices. Our intuition is that software frameworks incur more overhead for large power law graphs for a computation like PageRank where vertices are visited repeatedly; these overheads are not incurred by GraphPulse as communication is mostly on chip.

Comparing GraphPulse performance to Graphicionado [38], we found that, on average, GraphPulse is about 6.2× faster. The Figure also shows the performance of both the baseline and the optimized version of GraphPulse (with prefetching and parallel event generation); we see that the two optimizations dramatically improve performance.

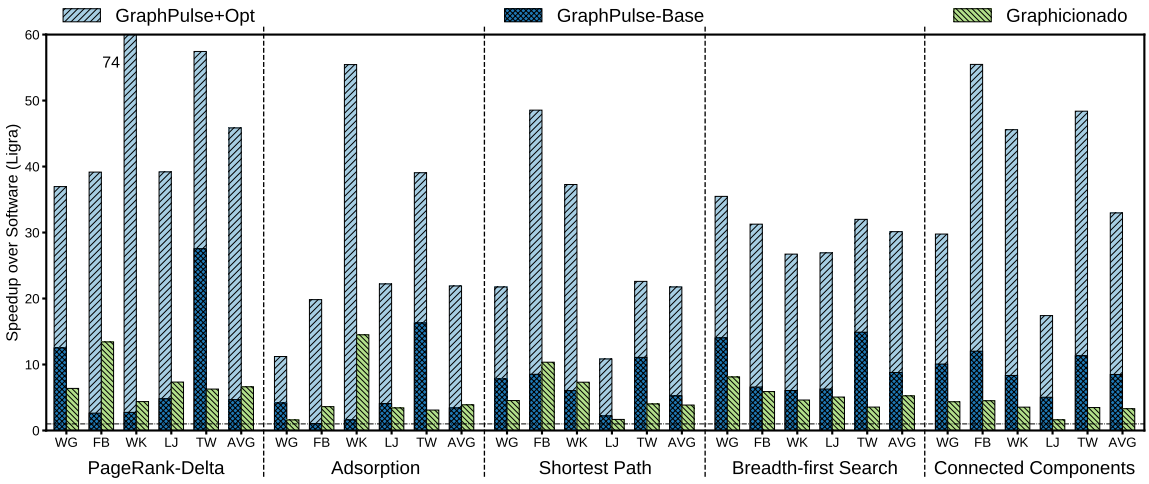


Figure 6.10: Performance comparison between GraphPulse (with and without optimizations), Graphicionado [38], and Ligra [90] frameworks; all normalized with respect to the Ligra software framework. Twitter required partitioning from Section 6.1.6.

Memory Bandwidth and Locality

GraphPulse implements a number of optimizations to promote spatial locality and utilize DRAM burst transfer speed whenever possible. Figure 6.11 shows the total number of off-chip memory accesses required by GraphPulse normalized to Graphicionado. Even compared to the efficient data access of Graphicionado, GraphPulse requires 54% less off-chip traffic on average. GraphPulse's processing model is memory friendly with events carrying the input data to the computation. Coalescing and *lookahead* also contribute heavily to reduce data traffic by combining computations and memory accesses and stabilizing many nodes earlier. The effect is particularly apparent in CC, where many vertices gets stabilized with the very first event. Finally, Figure 6.12 shows that in GraphPulse very large fraction of data brought via off-chip accesses is utilized by the computation supporting its ability to reduce random memory accesses.

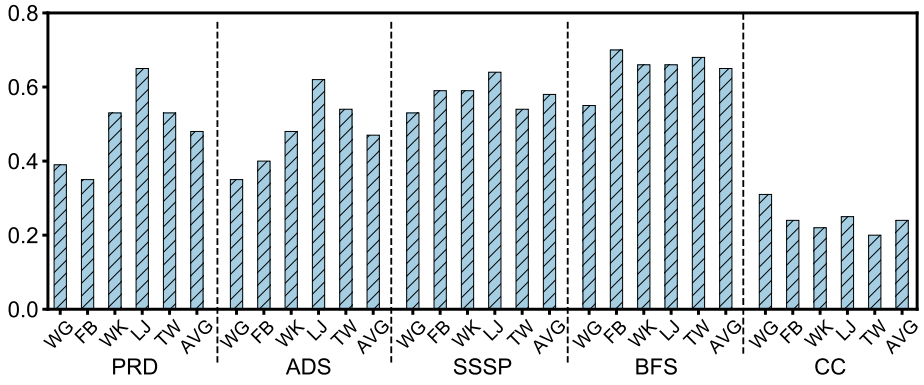


Figure 6.11: Total off-chip memory accesses of GraphPulse normalized to Graphicionado.

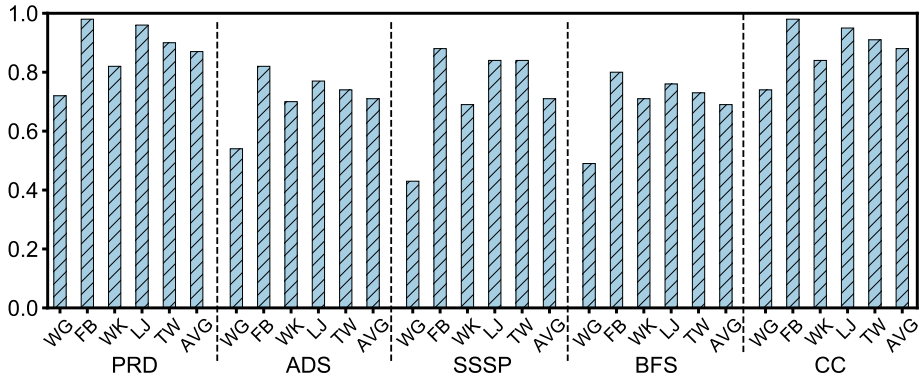


Figure 6.12: Fraction of off-chip data transmissions that resulted in useful computations in GraphPulse.

Event Execution Profile

The average life-cycle of an event is highly dependent on the graph structure and the algorithm. Figure 6.13 shows a breakdown of average time spent in different stages of the processing path for an event. Individual vertex memory reads have long memory latency. But due to locality aware scheduling and prefetching in the input buffer, latencies for the accesses are masked and the average latency for the vertex memory reads become only few cycles. This indicates the efficiency of the prefetcher. The process stage takes only few

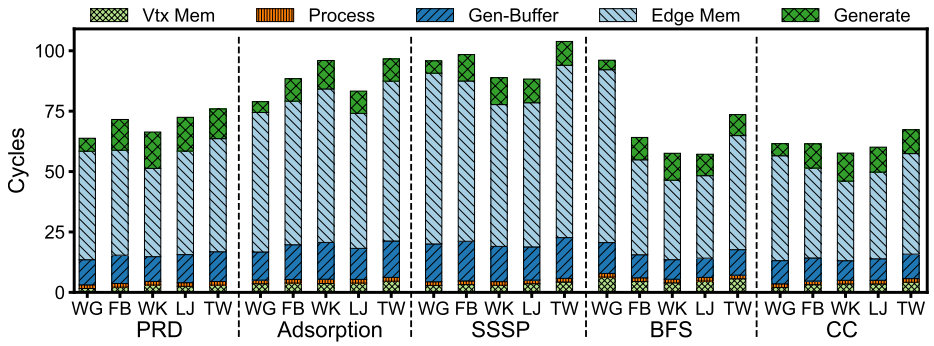


Figure 6.13: Cycles spent by an event in each execution stage, shown chronologically from bottom to top.

cycles too because of pipelining and brevity of typical *apply* tasks. The *Gen Buffer* stage shows the time spent in the input buffer of generation streams after an event is processed and waiting for generation units to be available. The time spent on edge memory access appears to be high, but this is due to the large number of edges that need to be read for event generation in power-law graphs. Figure 6.14 shows the fractions of time the processors and generators spend accessing memory, processing and stalling. It is noticeable that event generation units (right-side bar) spend close to 80% of the cycles reading edge memory. This includes the latency to both read edges from cache and fetch from main memory. We observed that the generation units saturate the memory bandwidth with prefetching and high off-chip utilization. The processors (left hand bars) stall for about 70% of the cycles waiting for generators to become available. We observed that this can be reduced to less than 40% by doubling the ratio of generation streams at the trade-off of increased routing complexity.

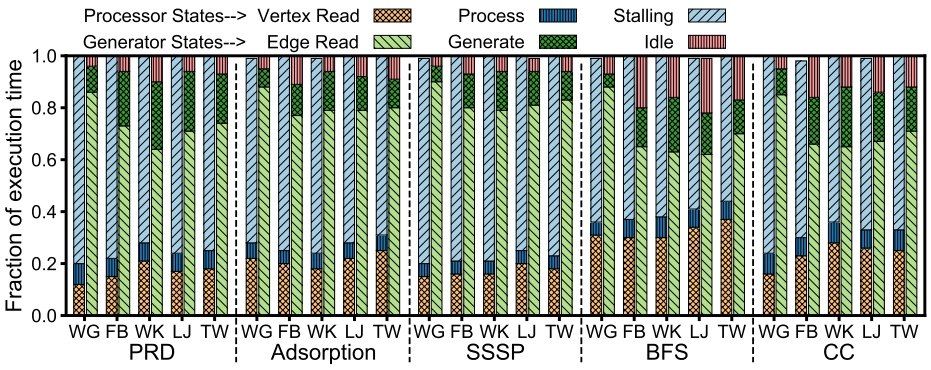


Figure 6.14: Time breakdown for the GraphPulse processors (left-bar) and generation units (right-bar).

6.3.3 Hardware Cost and Power Analysis

The coalescing event queue consumes the most power since it closely resembles a cache in design and operation with the addition of a coalescer pipeline. We model the queue using 64MB on-chip memory composed of 64 smaller bins operating independently. 8 scratchpads with 1KB capacity are placed alongside 8 processing cores (with 8×4 generation streams). We use CACTI7 [7] for analysis of power and area for both memory elements. The dynamic memory access is estimated conservatively from simulation trace. The total energy for the whole event queue memory is $\sim$9 Watts when modeled in a 22nm ITRS-HP SRAM logic. Although we use identical systems, GraphPulse accesses 60% less memory than Graphicionado; we did not include DRAM power.

The event collection network is a 16×16 crossbar attached to a network of arbiters allowing groups of Generation Streams to share a port. We modeled a complete RTL design containing the communication network, coalescer engine, and event processors using Chisel and synthesized the model. We assumed that the coalescing pipeline and event processors require floating point units, which results in worst case complexity and power consumption

Table 6.3: Power and area of the GraphPulse accelerator components

	#	Power(mW)			Area(mm^2)
		Static	Dynamic	Total	
Queue	64	116	22.2	8825	190
Scratchpad	8	0.35	1.1	11.6	0.21
Network		51.3	3.4	54.7	3.10
Processing Logic		-	-	1.30	0.44

estimates (recall that the coalescing logic is application dependent). The area of the circuit stands at $3.5mm^2$ with a 28nm technology (excluding the on chip memory) and comfortably meets the timing constraint for 1GHz clock. Power estimates show that custom computation modules and the communication network consumes less than 60mW. A breakdown of the power consumption of our evaluated design is presented in Table 6.3. GraphPulse is 280× more energy efficient than Ligra due to the low power from the customized processing and faster overall execution time.

Chapter 7

Incremental Recomputation of Streaming Graphs

Most graph frameworks optimize the performance of a given query against a fixed graph. However, in many real-world applications, we are faced with the *streaming graph* scenario where the graph is constantly changing as new entities are created, old entities are removed, and new interactions take place over time. A stream of updates in the form of edge/vertex additions/deletions is typically applied to the graph in batches for efficiency. As the graph evolves, a straightforward approach is to restart the query from scratch after applying a batch of graph updates. However, the number of vertices or edges modified in a batch is typically exceedingly small relative to the size of the graph. Thus, as the changes only modify a small subset of the graph for many queries, much of the computation performed during reevaluation is redundant.

To address this inefficiency, streaming graph systems support incremental update of query results following changes to the graph, resulting in order of magnitudes speedups over restarting the query. Examples of such software systems include Kineograph [21], Tornado [89], and Naiad [72] that can handle only growing graphs (i.e., no deletions are allowed). By far, the problem of incrementally supporting deletions is more challenging, and only KickStarter [103], Graphbolt [67], and DZig [66] support it. We propose an algorithm for incremental recomputation of a streaming graph and design an accelerator, JetStream for running this algorithm in hardware.

The addition of edges is straightforward in the event-driven model; the added edge simply creates a new event. In contrast, edge deletion is substantially more difficult for most algorithms because it is often impossible to determine whether an update should propagate. We support deletions in two phases: (1) incrementally transforming query results for the previous version of the graph into a *recoverable state* for the updated graph, and (2) bringing the results to convergence again. Although GraphBolt and KickStarter also proceed in two phases, they rely on the Bulk Synchronous Processing (BSP), model which cannot work in JetStream's asynchronous model. Therefore, we develop new event-based algorithms where both phases execute in a fully asynchronous fashion. JetStream serves both the class of accumulative algorithms supported by GraphBolt and monotonic algorithms supported by KickStarter.

In this chapter, we first illustrate the challenges and techniques for streaming graph analytics, and then describe our event-driven algorithm for JetStream.

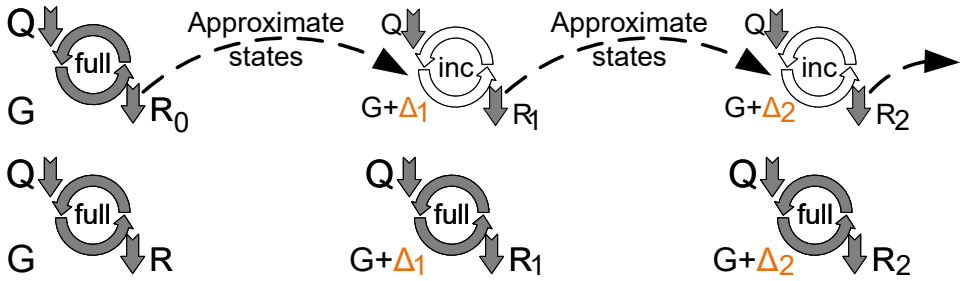


Figure 7.1: Query evaluation on a streaming graph using an incremental algorithm (top) and static algorithm (bottom).

7.1 Streaming Graph Analytics

A query evaluation over a streaming graph, as shown in Fig. 7.1, has two distinct characteristics. First, it supports streaming updates: new graph updates also arrive as the query is being evaluated. These updates are collected in a batch (e.g., Δ_1 or Δ_2 in Fig. 7.1) and are applied only after the query evaluation is complete and its results reported. *Graph updates consist of edge additions and deletions.* A vertex addition can be modeled by addition of the first edge to/from the vertex while modification of an edge weight is modeled by its deletion followed by an addition of an edge with the same weight. Second, query reevaluation leverages the existing state computed before the updates: after a batch of updates has been applied, the query evaluation is resumed incrementally to obtain the query results for the updated graph. In an algorithm (or accelerator) that supports streaming operation, the reevaluation is performed as an incremental update of the previous query result computed on the original graph, shown as *approximate states* in Fig. 7.1, to avoid wasteful redundant computations. As updates continue to arrive, the incremental computation is performed repeatedly. JetStream improves upon most prior software streaming algorithms, which only support streaming edge additions, by allowing

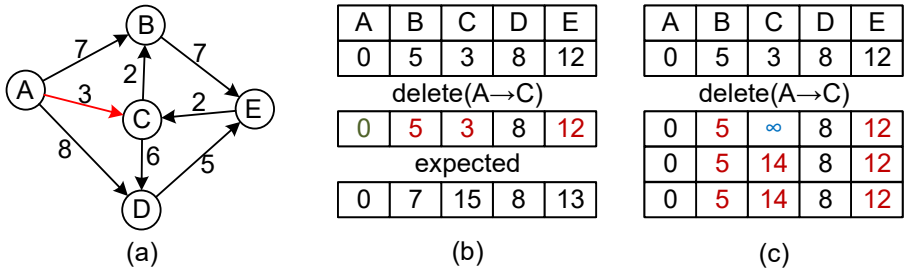


Figure 7.2: Using intermediate and initial values leads to incorrect results for SSSP: (a) an example graph; (b) uses previous state to recompute; (c) resets impacted vertex.

edge deletions. It also improves on most software frameworks by supporting concurrent processing of multiple updates, gaining efficiencies from combining some of their overheads.

7.1.1 Incremental Query Evaluation

Incremental reevaluation uses the result of the prior query to find an intermediate approximation, which becomes the initial state for computing the query result on the updated graph. Using the previous result for an approximation can lead to faster convergence than using a random initial state for the updated graph. Intuitively, for many query types, only a small fraction of vertices are affected by graph changes since batch sizes are typically tiny compared to the size of the graph (thousands of edges in graphs with billions of edges). Thus, a complete restart of the graph computation ends up doing substantial redundant work. Of course, we need to have an effective algorithm for identifying which vertices require recomputation for doing incremental updates.

Motivation and Basic Operation

Monotonic algorithms often produce incorrect results in the presence of deleted edges. We consider the example of an edge deletion(A→C) in the graph of Fig. 7.2(a) for Shortest Path algorithm. Since the vertices only update when they receive a shorter path value than their current state, the graph never reaches the expected result using the previous result as shown in Fig. 7.2(b). We call this approximation *unrecoverable* because the computation cannot recover to the correct result after being set into an incorrect state by the edge deletion. If we reset the target of deletion to its initial value as shown in Fig. 7.2(c), it still never reaches the correct result because other vertices (B, D, E) previously influenced by it are also in incorrect states.

Fig. 7.3 shows the progress of a query evaluation through different phases. First, a graph is *initialized* to an *initial state*. As computation progresses, the graph moves through several *intermediate states* to reach a *final state* when the algorithm terminates. Here, the final state is the correct *converged state* (static), and all intermediate states (including the initial state) are *recoverable states* because the graph can reach the correct state from there. A *recoverable approximation* is equivalent to one of these *recoverable states* from which the graph is guaranteed to converge correctly. After applying the graph mutations, the challenge in incremental graph computation is to find a recoverable approximation based on the previous converged states. For this example, all the vertices possibly influenced through the deleted edge in the initial evaluation is identified and reset in the *recovery phase* to arrive at a recoverable approximation for the reevaluation. Incremental recomputation on this approximation in the *reevalauation phase* leads to the correct result.

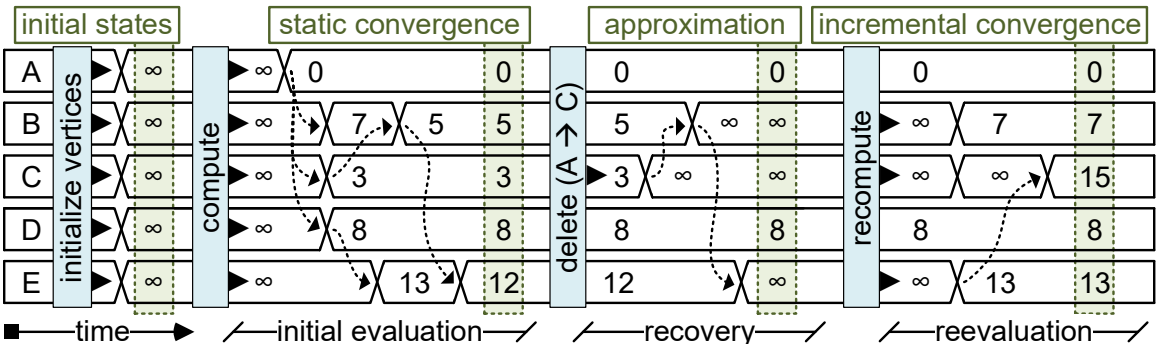


Figure 7.3: Conceptual timeline showing vertex values over time through initial evaluation, recovery, and reevaluation phases for SSSP on the example graph in Figure. 7.2.

Recovery Algorithms

A simple way to find the set of vertices affected by a deleted edge is to iteratively propagate a *tag* downstream from the target vertex of the deleted edge as in GraphIn [88]. Note that if a vertex is not affected by an update, the propagation is not forwarded again. The set of vertices tagged this way definitively contains all possibly impacted vertices. The tagged vertices can then be reset to the initial value to acquire a recoverable approximation for a monotonic convergence. When the query is reevaluated, the reset vertices converge to correct states based on the mutated graph. An example for obtaining a recoverable approximation using tag propagation in the recovery phase is shown in Fig. 7.3.

JetStream develops event-driven adaptations of vertex tagging and dependence tracking so that they can be used to extend the GraphPulse architecture to support incremental computation over a streaming graph. Monotonically converging algorithms where vertex value computation is a *selection* task – such as ShortestPath, ConnectedComponents, WidestPath, and BFS – benefit from this approach. Graphs with accumulative

update functions – such as PageRank and Adsorption – uses a simpler recovery technique in the event-driven approach. Here, the impact of a deleted edge is negated by sending the total contribution through that edge with negative polarity. This makes the event-driven approach highly suited for the incremental computation of these algorithms.

7.2 JetStream Design Overview

We present the design of our event-based streaming accelerator and its underlying algorithms in this section. First, we describe the event-driven execution model that GraphPulse [79] is based on. Then, we formalize the problems of building a streaming accelerator over a static one and describe the JetStream model that solves these problems.

7.2.1 Event-based Processing in GraphPulse

JetStream extends GraphPulse to support streaming graphs [79]. GraphPulse employs event-driven execution to eliminate overheads of shared-memory frameworks (e.g., poor temporal and spatial locality, atomic memory accesses, and synchronization). The event-driven execution is based on the delta-accumulative incremental computation (DAIC) [116] model. In this model, contributions coming over different edges (termed *delta*) can be applied independently and without any fixed order to compute the vertex state. The model has two primary components – *i)* a REDUCE task used to compute vertex state from incoming *deltas* and previous vertex state; and *ii)* a PROPAGATE task used to compute the *delta* over each outgoing edge. In the event-driven model, lightweight messages called *events* carry the *deltas* to their respective destination vertices. A vertex recomputes its state only

if it receives an event (*delta*) and generates a new event only when its state changes from the incoming event.

GraphPulse presents a complete execution model to run an iterative graph algorithm using the event-based approach. Algorithm 1 shows the event-driven execution model and how the SSSP application is mapped to the model. The user defines a REDUCE() method (line 5) expressing the *reduction* of incoming contribution and vertex state. A PROPAGATE() function (line 8) is defined for finding the delta over an outgoing edge and creating new events. INITIALVERTEX() and INITIALEVENTS() methods are defined to initialize the vertex states, V, and the initial set of events (Q) before the execution starts. The initial vertex values are set to an IDENTITY value for the REDUCE() function, so that a vertex's first *reduction* operation with an events is bound to change its state and propagate. With the processing of the initial events, vertex states get updated towards convergence, and new events are generated and inserted to Q. For each event in Q, the vertex update task is triggered. When a vertex reaches convergence, its state does not change from incoming events, preventing new event propagation (line 6). Eventually, Q becomes empty when all vertices reach convergence terminating the application.

Proper execution and termination of the event-driven model depend on two properties of the graph algorithms. First, the Reordering Property requires that incoming contributions over the incoming edges can be applied to a vertex in any order and independently of each other. Second, the Simplification Property requires that vertex that does not change state should not impact other vertices, i.e., it should not propagate, and other vertices should not require its contribution for computing their states. Many important graph al-

gorithms such as SSSP, SSWP, BFS, Connected Components, Incremental PageRank, and many Linear Equation Solvers satisfy these properties. These algorithms are supported in GraphPulse. JetStream too supports all these algorithms supported in GraphPulse without any change to the application model.

7.2.2 Streaming Graph Computation Objective

GraphPulse computes the final converged state of a static graph. We want to find the new converged state of the graph using JetStream after some mutation is applied to the graph structure. To formally describe the objective of JetStream, we consider a graph $G^0 = (V, E^0)$ being initialized to a set of values $I_G = \langle \forall j : i_j = \textsc{Identity} \rangle$ and converging to $C_G^0 = \langle c_0, c_1, ..., c_{n-1} \rangle$ for its final state. The IDENTITY parameter is application-specific for the graph algorithm; it is the *initial value* of the vertices and the non-dominant value for the REDUCE() operation (Algorithm 1). For streaming algorithms, we need to compute a new convergence state C_G^1 for the mutated graph, $G^1(V, E^1)$, using a recoverable approximation A_G based on C_G^0. The approximation $A_G = \langle a_0, a_1, ..., a_{n-1} \rangle$ is *recoverable* if convergence can be reached for algorithm S starting from this approximation (Section 7.1.1). For the *selection-type* algorithms, The vertex states progress from the initial value (IDENTITY) to the direction of convergence monotonically. A *more progressed* value dominates the REDUCE operation. In a valid approximation, all elements in A_G must be *less progressed than* or *equal to* the corresponding elements in the eventual converged state, C_G^1. An approximation, $A = \langle \forall i, a_i = \textsc{Identity} \rangle$, set to the initial value is a valid recoverable approximation but an inefficient one since it is equivalent to computing the graph from the beginning. Hence, finding a good approximation is critical for performance. Our proposed approaches in

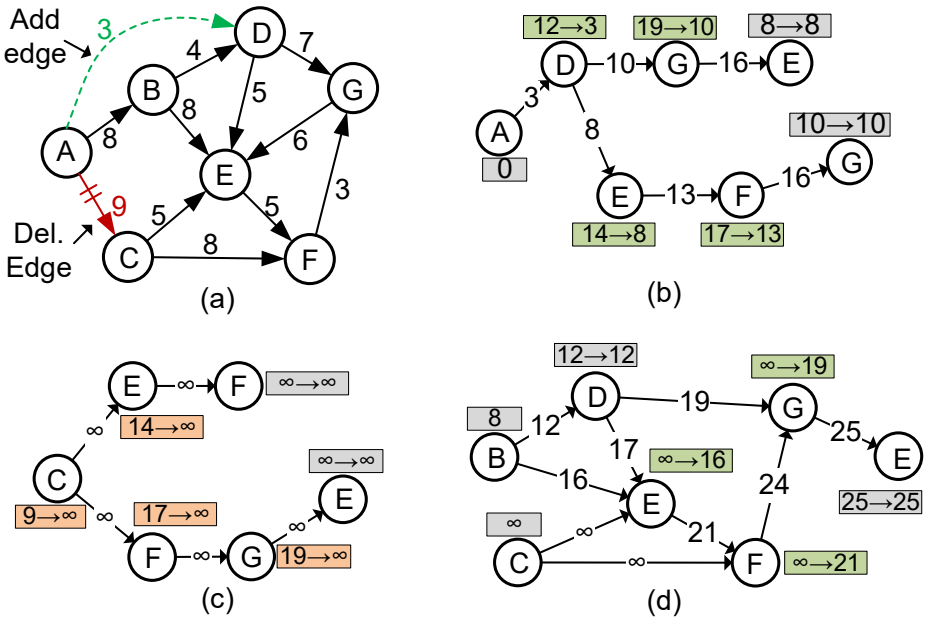


Figure 7.4: Propagation of events during processing of streaming edges in SSSP. (a) An example graph. (b) Propagation and updates from the insertion of edge $A \rightarrow D$ in the graph. (c) Propagation of deletes and resetting impacted vertices due to the deletion of edge $A \rightarrow C$ in the graph. (d) Recovery of approximate state after request events are processed.

JetStream accomplish this by expressing the graph mutation as events and restoring the mutated graph to a recoverable approximation for subsequent processing using the event-driven model.

7.2.3 Event Representation of Graph Mutation

Any modification to the graph structure is expressed using an event in JetStream. We assume that the modifications are batched, consistent with prior works on streaming graphs. A batch will be queued as events that are released once the ongoing processing iteration is complete. This choice to separate the update phase from the processing phase eliminates the need for resolving race conditions between old and new values as the com-

putation proceeds. Each modified edge is expressed as an event from the source to the destination of the edge. The payload (*delta*) carried by the event is generated by reading the previous converged state of the source vertex (which is *approximate* with respect to the mutated graph) and computing the propagation value based on this state and the edge attribute. This event represents the effect of the modified edge with respect to the previous graph structure. Events are queued and held until all the modified edges have generated a corresponding event. At this point, the new graph structure is active, and the events are processed from the queue. We demonstrate the processing of edge insertion and deletion events next.

Edge Insertions

Edge insertions are supported naturally by the event-driven model. The inserted edge did not exist in the previous graph and had no effect that needs to be reverted. An update along an edge can be applied to a vertex at any time in the asynchronous model. Hence, an update coming along a newly-inserted edge is conceptually similar to an update along an existing edge that was delayed; it has the same effect and gets processed in the same way. JetStream computes an update using the old converged state of the source vertex and the weight of the inserted edge, and queues it as an event for the destination vertex along with regular events (Algorithm 2). Fig. 7.4(b) shows how an edge insertion triggers a chain of updates. As the new edge ($A \rightarrow D$) contributes to vertex D, the vertex gets updated and propagates further with more events ($D \rightarrow G$). Propagation ultimately stops due to monotonicity when the event arrives at a more progressed receiver via ($G \rightarrow E$). If the state of the source vertex A itself is not stable, subsequent updates to the vertex will

Algorithm 2 Converting edge-insertions to events

```
1: procedure ProcessInserts(G(V, E), Q, A⟨u → v, w⟩)
2:     for each ⟨u → v, w⟩ ∈ A do                ▷ A = list of added edges
3:         δ_v ← V[u] + w                        ▷ Propagate(a, b)
4:         Q ← insert(⟨v, δ_v⟩)
5:     end for
6: end procedure
```

be propagated using the mutated graph along the new edges and send the correct values downstream eventually. Hence, a graph always remains in a correct or recoverable state after edge insertions.

Edge Deletions

Edge deletions are not supported by most streaming systems (the exceptions being Kickstarter and GraphBolt). JetStream supports deletions as in KickStarter while overcoming some of its performance limitations when handling a batch of deletions. Specifically, JetStream queues edge deletions as events in the same way as insertions. However, edge deletion is more complicated since the deleted edge's contribution to the previous converged state must be reversed. For algorithms with *accumulative* updates, reverting the effect of deleted edges is simpler. A vertex propagates an update downstream for all the updates it receives and accumulates. As a result, we can infer the combined value of all updates it sent along an edge during the previous evaluation by looking at its accumulated state and using the Propagate function. Sending the inverse of its previous converged state, transformed by the Propagate function, negates the cumulative effect of all updates over this edge.

Algorithm 3 Converting deletions to events for PageRank (accumulatative algorithm).

1: **procedure** ProcessDeleteCumulative($G(V, E), Q, D\langle u \to v, w\rangle$)

2: **for each** $\langle u \to v, w\rangle \in D$ **do** ▷ D = list of deleted edges

3: $\delta_v \leftarrow -1 \times V[u] \times (1 - \alpha)/deg(u)$ ▷ Propagate(a, b)

4: $Q \leftarrow insert(\langle v, \delta_v\rangle)$

5: **end for**

6: **end procedure**

Further propagation downstream of negative events from the receiver vertices leads to the rollback of all contributions from this edge and puts the graph in a recoverable state. We create negative events for the deleted edges as shown in Algorithm 3 to initiate recovery.

For algorithms having *selective* updates, it is more difficult to identify which edges contributed to a vertex. The destination vertex of a deleted edge is reset to its initial value so that it can be updated later in the reevaluation phase. We queue events with a *delete flag* as shown in Algorithm 4. A vertex, upon receiving an event with a delete flag, will reset itself. This change in the state goes against the direction of monotonicity. Therefore, when this vertex propagates its updates to its neighbors, the update events will be discarded by the receivers in the Reduce function since they already have a more progressed state. However, this more progressed state may have resulted from the contribution of the deleted edge. Hence, the graph stays in an incorrect state if these vertices are not corrected. To solve this problem, we devise an event-driven edge deletion algorithm that identifies the potentially affected vertices and efficiently resets them to acquire a recoverable approximation following the technique described next.

Algorithm 4 Converting deletions to events for SSSP (selective algorithm).

```
1: procedure ProcessDeletesSelective(G(V, E), Q, D⟨u → v, w⟩)
2:     for each ⟨u → v, w⟩ ∈ L do
3:         Q ← insert(⟨v, 0⟩)
4:     end for
5: end procedure
```

7.2.4 Impacted Vertex Detection and Recovery

To handle an edge deletion correctly, the vertices impacted by a deletion must be identified, and their states reset to a recoverable value. Impacted vertices are identified by propagating a delete tag to all outgoing neighbors of an impacted vertex and tagging them as impacted in a manner similar to KickStarter [103]. When a deletion event first arrives at a vertex, we set the vertex state to the initial *Identity* value (*tag* it) as shown for vertex C in Figure 7.4(c). Hence, textittagged vertices can react to updates from future events. Delete events are propagated along each outgoing edge. A delete event cycling back to an already tagged vertex (e.g., $G \rightarrow E$) will not propagate. Multiple delete events queued for the same vertex can be *coalesced* since tagging a vertex once is sufficient. When a vertex is reset, the vertex Id is added to a list. Hence, the set of vertices tagged this way contains all vertices whose states could have been potentially influenced by the deleted edge. The process is shown in Algorithm 5. The list is used to revisit these vertices to recompute their approximate states as described next.

A new recoverable approximation for the impacted vertices must be found in case the query cannot progress to some impacted vertices. For example, in Fig. 7.4(a), a SSSP

query running from A cannot reach E because vertices B and D are already in a correct state, and will not propagate new events along $B \rightarrow E$ and $D \rightarrow E$ after edge deletion. KickStarter solves this problem by reading all neighbors states again to reestablish an approximate state for an impacted vertex. Unfortunately, this approach generates many memory reads with a random access pattern. Many of the vertices are also read by multiple deleted vertices creating opportunities for data reuse. Instead of reading the states of the neighboring vertices directly, we create a *request* event to request updates from the neighbors. The *request* event has a *request-flag bit* set and the payload set to Identity in order to avoid affecting any other events and vertices. When a vertex detects the *request-flag*, it must propagate to its neighbors, even if it does not update itself. The request events are coalesced, hence, combining the reads for each vertex. Also, when they pass through the queue, the events are sorted by their destination vertex ID so that a sequential memory access pattern occurs when they are processed. Upon receiving the response to the *request* event, the impacted vertex will reestablish an approximate state closer to convergence based on its neighbors' approximate states.

A second inefficiency persists in other approaches because computing an approximate state from neighbors' approximate states is often wasteful since these approximate states may change again during query evaluation. To address this problem, we exploit the asynchronous nature of the model – we can delay the vertex reads or recomputation until after the effect of the initial events and inserted edges are applied. We overlap the execution of request events with query events and edge insertions, so the vertex updates move the vertex closer to the final converged states.

Algorithm 5 Recovering approximations of vertices impacted by deletions for SSSP.

```
1: procedure ResetImpacted(G(V, E), Q)
2:     X ← ∅                                          ▷ List of impacted vertices
3:     while Q is not empty do
4:         (i, δ_i) ← pop(Q)
5:         if V[i] ≠ Identity then
6:             V[i] ← Identity                        ▷ Tag vertex
7:             X ← X ∪ {i}
8:             for each (u → v, w) ∈ E | u = i do
9:                 Q ← insert(⟨v, 0⟩)                 ▷ Propagate delete
10:            end for
11:        end if
12:    end while
13: end procedure

14: procedure Reapproximate(G(V, E), Q, X)
15:    for each i ∈ X do                              ▷ Create events with request flag(ρ)
16:        for each (u → v, w) ∈ E | v = i do
17:            Q ← insert(⟨u, Identity, ρ⟩)
18:        end for
19:    end for
20: end procedure
```

After the delete phase is over, JetStream revisits each vertex in the list of impacted vertices and sends *request events* along each incoming edge of a vertex at the beginning of the processing phase. If the impacted vertices are on the path of a propagating query, their states update to the correct states since their approximate state (Identity) can be updated by all contributions. If the vertex does not belong to the query propagation path, the responses to request events set them to the correct state. Thus, a graph always remains in a correct state after deletion is processed in this technique. The pseudocode for processing deletes is shown in Algorithm 5.

7.2.5 Recomputaion of the Mutated Graph

JetStream execution process uses the original computation technique of GraphPulse to recompute the graph after setting up the approximate state and populating the event queue with appropriate events as described above. Because the recovery after delete is handled differently in the two different types of algorithms (accumulative vs. monotonic), the processing phases are scheduled differently for them. We discuss both of them next.

Algorithms with Selective Update

After receiving a batch of edge updates, we first process the deleted edges and insert deletion events in the queue to reset the target vertices. In the next phase, the events are allowed to execute on the previous version of the graph; all potentially impacted vertices are reset to their initial value. Afterward, events with *request-flags* are queued for all the neighbors of the impacted vertices. We process the inserted edges at this point to

Algorithm 6 Overall processing flow for SSSP.

1: **procedure** ProcessStream($G(V,E),\ Q,\ A\langle u \to v, w\rangle,\ D\langle u \to v, w\rangle$)

2: ProcessDeleteSelective($G(V,E),\ Q,\ D\langle u \to v, w\rangle$)

3: $X \leftarrow$ ResetImpacted($G(V,E),\ Q$) ▷ Queue is empty

▷ Delete phase ends

4: Reapproximate($G(V,E),\ Q,\ X$)

5: ProcessInsertions($G(V,E),\ Q,\ A\langle u \to v, w\rangle$)

▷ Switch to new graph structure

6: Compute($G(V,E),\ Q$)

7: **end procedure** ▷ V holds correct result

Algorithm 7 Overall processing flow for PageRank.

1: **procedure** ProcessStream($G(V,E),\ Q,\ A\langle u \to v, w\rangle,\ D\langle u \to v, w\rangle$)

2: ProcessDeleteCumulative($G(V,E),\ Q,\ D\langle u \to v, w\rangle$)

▷ Switch to intermediate graph structure

3: Compute($G(V,E),\ Q$) ▷ Q empty : Delete phase ends

4: ProcessInsertions($G(V,E),\ Q,\ A\langle u \to v, w\rangle$)

▷ Switch to new graph structure

5: Compute($G(V,E),\ Q$)

6: **end procedure** ▷ V holds correct result

create and queue the events for them. The insertion events can *coalesce* with the *request* events existing in the event-queue simply by setting their *request-flag* bit. The graph is then switched to the new version, and the events in the queue are allowed to process in the typical computation flow of GraphPulse. *The only difference is that whenever any vertex receives an event with a request flag, it propagates its state to all its outgoing neighbors even if it does not change its state.* These responses to the reapproximation request allow the impacted vertices to set their new state using the states of their neighbors. At the end of this phase, when the queue is empty, the graph arrives at a correct state, and the process of reevaluation concludes. The process is shown in Algorithm 6.

Algorithms with Accumulative Update

These algorithms do not need reset since a deleted edge can be negated with a regular event with negative polarity. After creating events for the deleted edges, we load an intermediate version of the graph without the deleted edges to break any cyclic path in the graph. Algorithms that propagate updates based on degree, such as PageRank, undergo changes in the weight of all edges when an edge is added or deleted. To handle this, we first delete all outgoing edges of the vertex having an edge added or deleted, turning it into a complete sink for the intermediate version of the graph. In the example of Fig. 7.5(a), any cyclic propagation through vertex B is stopped by deleting edges to D and E too. All outgoing edges of vertex B are added to the batch of deleted edges (Fig. 7.5(b)). We next process these deleted edges to populate the event queue with negative events. Next, a computation phase on this intermediate graph effectively removes all contributions of vertex B from the graph. Creating the intermediate graph is not expensive since it can be

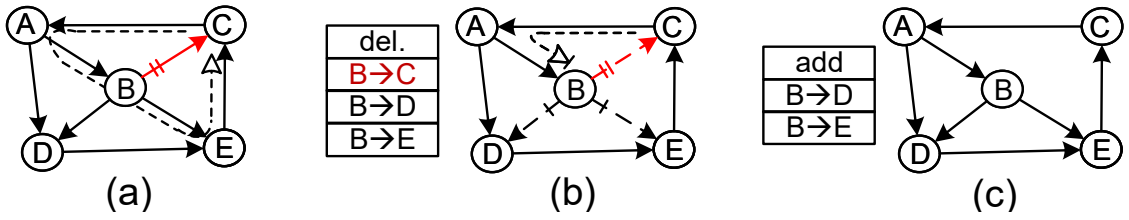


Figure 7.5: Showing an edge deletion for accumulative algorithms: (a) initial graph with B→C to be deleted; (b) intermediate representation; (c) mutated graph.

achieved simply by adjusting the pointers to the edge list to skip the deleted vertices. We then add back all the edges of vertex B (except the actually deleted edge B→C) to the batch of inserted edges so that it resembles the new graph structure (Fig. 7.5(c)). This batch of edge additions is processed to create events in the queue. When the compute phase is rerun on the new version of the graph, the result is correct for the mutated graph. The steps in this model are shown in Algorithm 7. We note that the manipulation of the edge addition or deletion batch only affects the preparation of the streaming batch; the actual vertex computation remains the same as GraphPulse.

Chapter 8

JetStream: a Streaming Graph Processing Accelerator

JetStream builds on GraphPulse, which uses an event-driven asynchronous processing model, with reported speedups of up to 6× relative to BSP-based accelerator (Graphicionado [38]). The event-driven model naturally supports asynchronous graph processing with faster convergence via greater parallelism, reduced work, and elimination of synchronization at iteration boundaries. In addition to its state-of-the-art performance, we chose GraphPulse because it maps incremental update operations to a series of events naturally within the existing architecture. JetStream supports all algorithms compatible with delta-accumulative computation [116], as is the case in GraphPulse.

JetStream is an asynchronous graph processing accelerator leveraging the event-driven execution model to operate on streaming graphs. The decoupled nature of event-driven execution allows the accelerator to extract abundant parallelism for the computation

flow and utilize memory bandwidth efficiently. A significant performance boost comes from the efficient utilization of low-latency on-chip memory resources for the transient short-lived communication data. In addition, specialized communication paths and scheduling primitives allow the accelerator to operate with very little overhead for control and synchronization. JetStream extends the datapath of GraphPulse, to accommodate the model described in Section 7.2. JetStream adds new modules for reading and processing streaming data, as well as re-implements the coalescing queue, and vertex update and propagation logic to account for the new types of events.

The accelerator is designed to work alongside a host as an ASIC/FPGA-based co-processor with dedicated DRAM memory and independently addressable memory space. The host processor allocates and initializes the graph and the initial events in the accelerator memory as defined by the programmer via a provided API. The accelerator performs the graph computation independently based on configurations received from the host. It alerts the host when computation finishes so that the graph state can be read back.

This chapter describes the architectural components of the GraphPulse core and highlights the extensions for JetStream. JetStream retains the GraphPulse datapath and adds a *Stream Reader* module for creating events from streaming data as described in section 7.2.3. It extends the vertex update module with a vertex reset logic, a scheduler with multiple policies, and coalescer logic incorporating *delete event* coalescing described in Section 7.2.4. A detailed view of the JetStream datapath is shown in Fig. 8.1, where the shaded components indicate modules added to or extended from GraphPulse. JetStream's architectural changes do not disrupt the regular computation on static graphs. As a result,

JetStream can perform both the initial non-incremental evaluation (like GraphPulse) and streaming evaluation efficiently. We describe the complete execution flow of JetStream later in this section. Furthermore, JetStream derives its functional module from the same programming API defined for GraphPulse; so minimal additional user effort is necessary to program JetStream. In the remainder of this section, we describe the primary GraphPulse components and how JetStream extends them.

8.1 Event Management

All computations are expressed as contributions along edges and propagated using events in the event-driven model. Events are lightweight messages that trigger vertex computation at the destination vertex. GraphPulse events are tuples containing a target vertex Id and a payload. The payload contains the vertex contribution along the edge. In JetStream, event payloads also contain some flags indicating special tasks (e.g., request flag mentioned in Algorithm 5). We describe optimizations in Section 8.7 that add extra data to the event payload in JetStream.

The event queue is the storage for active events in the system representing the set of active vertices. GraphPulse employs a fast on-chip queue capable of in-place coalescing. The queue contains multiple bins. Each bin is structured into a grid of rows and columns, and only one vertex is mapped into each cell by vertex index. The bins behave similar to a direct-mapped cache. During event insertion, if another event already exists in its mapped cell in the queue, the events are combined with the Reduce operation (coalescing). Thus, only one event for a vertex can exist in the queue at any time.

The queue is capable of fast parallel insertion of events received on the input bus. The bins are implemented on *Simple Dual-Ported* on-chip memory where one row can be read and written in each cycle. Furthermore, each bin is equipped with a coalescer pipeline that can insert one event every cycle even though coalescing may have multi-cycle latency. During insertion, the coalescer reads the existing event (if any) in the mapped block on the first cycle. Then, the existing event is *reduced* with the new events in the following cycles and written back.

Events are emitted in batches for processing. Since GraphPulse supported algorithms allow reordering of edge contributions, events can be emitted in any order. GraphPulse reads one full row of events at a time from a bin and puts it into a *drain buffer*. Events are drained from one bin at a time in a round-robin fashion. The vertices are mapped in such a way that a group of vertices whose states reside in the same DRAM page is also mapped in the same row in the queue. Thus, processing the events in one row of the queue within a short period provides a high spatial locality for the graph memory.

JetStream leverages the same queue architecture as GraphPulse. The coalescer pipelines are extended to combine delete events as well during the *recovery phase*. Two delete events can be merged since they do not carry any data. Additionally, fewer vertices can be mapped to the queue (for the same on-chip memory size) since the event payload in JetStream is bigger than GraphPulse. Hence, JetStream uses smaller-sized graph partitions than GraphPulse.

8.2 Event Scheduler

The GraphPulse event scheduler dequeues events from the queue and puts them in a buffer. It keeps track of processor occupancy, and arbitrates events to the processors with the least workload. It issues the events in the same *queue row* to the same processor for enhancing spatial locality. The scheduler also tracks the progress of the processing engines and the occupancy of the queue. When all the bins have been drained once, we say that a *round* is completed. The scheduler waits for the processors to idle before starting a new round. Since only one event for a vertex can exist at the time of emitting event, there cannot be more than one event scheduled for the same vertex in one round; this eliminates the need for atomic operations and simplifies memory access and synchronization. When the scheduler detects that the queue is empty and all processors have completed their assigned workload, it indicates the end of the computation phase and terminates the application.

In JetStream, the scheduler is extended to run the execution in multiple phases. When a streaming batch is ready, the scheduler starts processing for the recovery phase that precedes the regular computation phase. The recovery phase starts with populating the queue with *delete events* from graph mutation. Then it proceeds like a regular computation phase and ends when there is no *delete event* remaining in the queue. At the end of this phase, the graph is in a recoverable approximation state. Finally, the scheduler triggers the creation of *addition events* from added edges and runs the a regular computation phase (reevaluation) to obtain the final graph state.

8.3 Event Processing Engine

GraphPulse event processors are independent, parallel, and simple state machines. They continuously process events that are placed in their input FIFO buffers by the scheduler. The processors compute the vertex states using the user-defined REDUCE() method and apply the updates to the vertex memory. Since the processors receive events that are closely located in the memory in one batch, they can prefetch the vertex properties for these events. Each processor is equipped with an on-chip scratchpad prefetcher that can prefetch vertex data for all the events in the processing buffer. The prefetcher scans the buffer and reads the off-chip memory in such a way that vertex properties residing in the same DRAM memory page are read in a group, thus increasing memory access efficiency. The processors read and write vertex data through the scratchpad memory. The scratchpads can access any memory channel through an efficient memory bus.

When vertex states change, the processors pass the updates to one of their event generation streams. The generation streams read the edges and compute the contributions using the PROPAGATE() method to pass along the edges. Event generation streams also read the edge data through an edge cache connected to an off-chip memory bus. Since edge lists are contiguous in memory, the prefetcher requests the next memory block smartly based on the edge pointers and the number of edges in the *Edge ID Buffer*. The generation streams are connected to the queue using a crossbar. 32 generators of 8 processing engines share the input ports of the 16×16 crossbar, and the queue bins share the output ports.

JetStream utilizes the same event processor system during its regular computation phase. The apply logic is extended with a *reset logic* that sets a vertex to IDENTITY

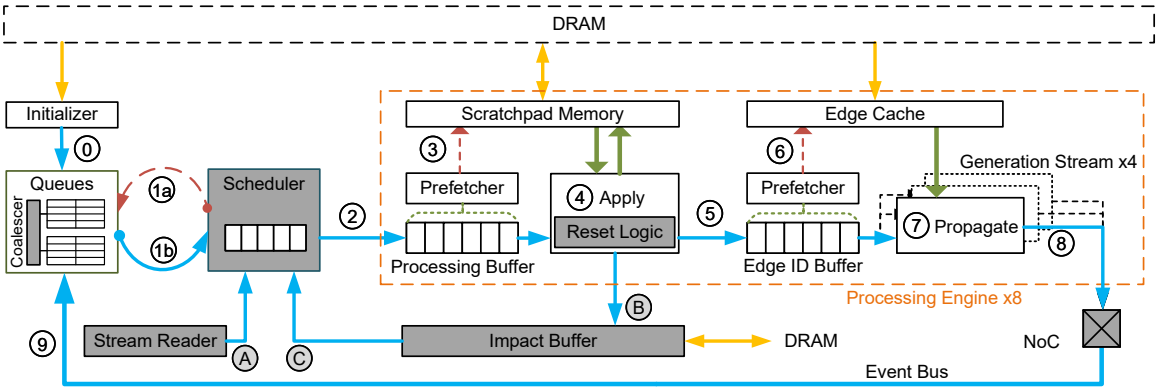


Figure 8.1: Detailed JetStream datapath. Blue arrows show data flow, red arrows indicate control signals, green and yellow arrows represent on-chip and off-chip memory transfers respectively. Shaded modules are new or modified in JetStream.

(Algorithm 5, line 6) when it receives a valid *delete event* during the approximation phase. It also, writes the vertex index to the *Impact Buffer* if a vertex resets its state from a *delete event*. Additionally, the processing buffer is increased in width to accommodate larger event size for JetStream.

8.4 Stream Processing Modules

JetStream adds a *Stream Reader* module that reads the lists of deleted and inserted edges from the main memory and schedules them to the processing engines during the approximation phase. The Stream Reader module and the other new modules that are added for the JetStream datapath are shown with shaded boxes in Fig. 8.1. The list of deleted edges is read first as $\langle source, destination, weight \rangle$ and events are created from these edges according to Section 7.2.3. Next, these events are used to find the sets of impacted

vertices during the recovery phase. Finally, the list of added edges are read, and events are created after the approximation is complete.

The *Impact Buffer* stores the indices of the vertices impacted by the deleted edges during the recovery phase. The *Apply unit* sends the index of an impacted vertex to the *Impact Buffer* module that writes to a list in its internal buffer. The list is written from the buffer to the main memory in batches. The *Impact Buffer* module also reads back the list and creates *request events* for the impacted vertices as described in Section 7.2.5. Since the reads and writes are done in bulk, they are sequential accesses and can be done with low overhead.

8.5 JetStream Execution Flow

Fig. 8.1 shows the steps and direction of the dataflow during the life-cycle of an event. The dataflow differs for the initial (static) and incremental evaluation. JetStream inherits the regular computation phase described in section 6.1.5 from GraphPulse and uses it for the initial static evaluation. The incremental evaluation is added in JetStream and it is required for fast evaluation of streaming graphs.

Delete Setup and Preparation. Edge additions are directly supported as regular events since they do not affect the monotonicity of the algorithm; we focus on the more difficult deletion support. The *Stream Reader* reads the deleted edges first in Ⓐ and passes them to the processing engines through the scheduler (②). Reusing steps ③ - ⑤, the vertex state for the source vertex is read (but not updated) and the ⟨*vertex state, destination, edge weight*⟩ is passed to the generation unit. Step ⑦ is used to find the propagated value, and

create a delete event for the destination vertex that is forwarded to the queue using ⑧, ⑨. Note that the computing elements of ④ and ⑦ are not necessary for the basic model. But they are used during the optimizations described in Section 8.7.

Delete Propagation. After all the delete events are queued, a normal computation cycle (steps ⓪-⑨) is executed until there remains no delete events in the queue. The Apply unit and Propagation unit use the logic defined in Algorithm 5, line 6 and 9. The Apply unit also writes the Id of a deleted vertex in step Ⓑ to the *Impact Buffer* during step ④.

Finalizing Approximation. After the delete propagation step concludes, we reschedule the vertices from the *Impact Buffer* (step Ⓒ) and reuse steps ②-⑨ once to create request events for their incoming edges. In this phase, step ④ reads the incoming edge pointers from the memory (in contrast to the outgoing edge pointer as in other phases). Following this, the *Stream Reader* reads the inserted edges, and creates insertion events using ②-⑨ the same way as deleted events. This completes the approximation phase. At this point, the regular computation phase (⓪-⑨) can execute again to evaluate the modified graph. As further streaming updates are received, the engine keeps finding recoverable approximation and rerun computation phase keep processing streaming data.

8.6 Graph Representation and Partition

GraphPulse stores the graph structure in a *Compressed Sparse Row* (CSR) format and the vertex states in simple contiguous arrays. JetStream assumes the same CSR graph storage format. However, different from GraphPulse, JetStream requires access to the incoming edges for each vertex, which are stored in another CSR structure. Since the host

processor maintains the graph structure, we leave the task of maintaining the evolving edge list to a suitable software graph versioning framework. In the simplest case, we assume the host writes a new CSR for the mutated graph version to the accelerator memory and swaps the pointer to the CSR after each batch iteration. Thus, JetStream can start using the new version of the graph. In practice, any graph versioning storage, such as Version Traveler [54] or GraphOne [58], can be used. JetStream can interface with any framework that allows a CSR abstraction to access the internal evolving graph structure, and only the *address translation logic* needs to be extended for interfacing.

The hardware queue can accommodate events for a limited number of vertices. So large graphs are partitioned into slices using a minimum edge-cut strategy to avoid overwhelming the queue. GraphPulse processes one slice of the graph at a time in a round-robin manner and temporarily stores the cross-partition events to the off-chip memory. After one *round* over a slice, it is swapped out by writing the pending events to the off-chip memory. Then, a new slice is activated; its events are read back from memory and inserted into the queue. We keep the same partitioning and swapping technique of GraphPulse, as JetStream extensions are not dependent on graph structure. Note that the partitions may not remain optimal as the graph continues to evolve. To reduce the fraction of edge-cuts, we can periodically re-partition the graphs or deploy dynamic graph partitioning tools [43, 101] without affecting the JetStream workflow.

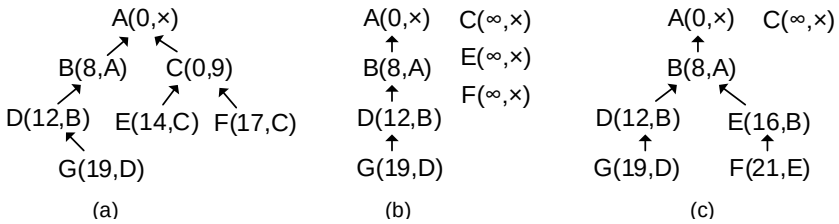


Figure 8.2: Dependency tree for the example in Figure 7.4: (a) before deletion; (b) after reset; (c) after reevaluation for the deleted edge A→C.

8.7 Optimizations

We have described a system that uses a tagging approach during edge deletion (Section 7.2). Next, we describe extensions to the delete propagation algorithms to capture a smaller set of impacted vertices.

8.7.1 Value Aware Propagation (VAP)

A fundamental property of monotonic algorithms is that the updates propagated from a vertex along its outgoing edges are always *less progressed* (closer to IDENTITY) than the vertex itself. For example, in a Shortest Path (SSSP) algorithm, all the distances transmitted via edges are longer than the vertex's distance from the root. In typical selection-based algorithms, a vertex selects only the incoming edge with the most *progressed* update to set its state. VAP exploits the observation that any source vertex that propagates an update that is *less progressed* than the destination's state, can not be the contributor to its state. Thus, when a vertex is impacted, VAP avoids resetting any neighbor that is more progressed than the resulting contribution from the impacted source.

Implementing VAP requires changes to the event propagation and update logic. The JetStream engine already uses a PROPAGATE logic to compute the value of the events

generated along outgoing edges. This same logic is used to compute the propagated value along the deleted edge during the creation of delete events. Upon receiving this event, a receiver vertex compares the event payload to its current state. If the received value is *less progressed* than the receiver, it can be safely discarded. Otherwise, the vertex resets itself to the initial value and propagates the updates along its edges using its previous state. The delete events with value can be coalesced in the queue using the same REDUCE() function as the one for regular events. Only the most progressed event will remain, and if that does not impact the destination vertex, the delete event is not propagated. This substantially reduces the number of impacted vertices in the system for applications with distinct edge weights and vertex states

8.7.2 Dependency Aware Propagation (DAP)

Comparing values in applications where clustering vertices settle to the same value is futile. For example, a BFS algorithm sets all nodes to the same value, and VAP cannot exclude any vertex based on value. For such algorithms, we exploit another observation that the vertex states depend on the contribution of only one incoming edge for each vertex. The first contribution that sets a vertex state to the final value is the one that the vertex depends on. Subsequent contributions carrying the same update value cannot affect the vertex. Therefore, deletes propagated along these edges can be safely discarded. The approximate state is recoverable as long as the first contributing vertex remains stable. We adapt the notion of *Dependency Tree* introduced in KickStarter [103] to the event-based model for these kinds of applications.

Formalization

We capture the flow of *useful* contributions across the graph to identify dependency. We use the notion of a `Leads-To` relationship ($\Mapsto$) that represents the effect of a vertex on the transition of a neighbor's state. Specifically, $A \Mapsto B$ if the state of B transitions from the contribution of A. In a cyclic path $A \to B \to C \to A$ with a BFS query, if $A \Mapsto B$ and $B \Mapsto C$, then $C \not\Mapsto A$ because A would have already reached the final state and would not transition from the contribution (*futile*) from C. Discarding all delete propagation $u \to v$ where $u \not\Mapsto v$ still produces a recoverable approximation. We can represent the `Leads-To` relationship in the form of a *tree*. Note that multiple valid versions of the dependency tree may exist depending on the order in which events are processed.

Implementation

We add a dependency field to the vertex state to record the source of the first event that updates it to a stable value. We also add a field to the event payload that carries the Id of the source of that event. When an event updates a vertex, the vertex changes its dependency field to match the source of this event.

While coalescing two events in the queue during regular computation, we retain the source of the event that is *dominant* in the REDUCE function. We disable coalescing during the *recovery phase* because the source information of a delete event will be lost after coalescing. We extend the queue with an overflow buffer that stores the extra events when multiple events are received for the same vertex. The overflow buffer writes to the off-chip memory in blocks when full and reads back in blocks when issuing events. These off-chip

accesses have low overhead as the number of delete events is far smaller than the events in a regular computation.

During event processing, a vertex only resets itself and propagates the delete if the dependency field matches the source ID of the delete event. Other delete events are discarded, greatly pruning the set of impacted vertices. Fig. 8.2 shows the vertex states and dependency trees during different stages of the incremental evaluation for the example graph of Fig. 7.4.

Overheads

This approach changes the data structure requiring more memory for vertex states and on-chip events compared to VAP. However, the dataflow architecture and the control sequence remain intact. Only the vertex update logic and event coalescing logic need to be modified. Not coalescing events during recovery raises the concern of transaction safety if multiple events are issued to processors concurrently. This is not an issue. Because in this approach, only one event matching the dependency field can reset a vertex, and thus only one vertex process will write back to memory.

8.8 Evaluation

JetStream is implemented on a cycle-accurate microarchitectural simulator based on the Structural Simulation Toolkit (SST) [81]. The off-chip memory is modeled with DRAMSim2 [83]. We use a detailed bus communication, scratchpad, and cache memory model built within SST to evaluate communication and memory access characteristics. The

Table 8.1: Experimental configurations for JetStream.

	Software Framework	JetStream
Compute Unit	36× Intel Core i9 @3GHz	8× JetStream Processor @ 1GHz
On-chip memory	24MB L2 Cache	64MB eDRAM @22nm 1GHz, 0.8ns latency
Off-chip Bandwidth	4× DDR4 19GB/s Channel	4× DDR3 17GB/s Channel

event processing and memory system configuration of the modeled framework is shown in Table 8.1. For large workloads unable to fit in the on-chip memory, we followed the same partitioning technique as GraphPulse. We used PulP [93] for edge-cut-based slicing of the graphs.

8.8.1 Experimental Setup

Our comparison is focused on showing both the advantage stemming from algorithmic support and hardware acceleration. First, we show the benefit of the incremental reevaluation by comparing the performance with *"cold-start"* computation of GraphPulse, where the whole graph is processed from initial states after each batch of updates. We used the same hardware configuration for GraphPulse and JetStream. Then, we compare the performance and characteristics with two software frameworks to show the benefit of

Table 8.2: Input graphs used in the experiments for JetStream.

Graph	Nodes	Edges	Description
Wikipedia(Wk) [26]	3.56M	45.03M	Wikipedia Page Links
Facebook(FB) [99]	3.01M	47.33M	Facebook Social Network
LiveJournal(LJ) [6]	4.84M	68.99M	LiveJournal Social Network
UK-2002(UK) [12]	18.5M	298M	.uk Domain Web Crawl
Twitter(TW) [59]	41.65M	1.46B	Twitter Follower Graph

accelerating a streaming graph analytics engine. We compare with GraphBolt [67] for accumulative algorithms and KickStarter [103] for monotonic algorithms with selective updates. The system configuration for software benchmarks is shown in Table 8.1.

Workloads

To demonstrate the performance of realistic workloads, we select five real-world graph datasets (see Table 8.2). Among these workloads, Wikipedia and UK-2002 domains graphs represent narrow graphs with long paths, and Facebook, Livejournal, and Twitter graphs represent large, highly connected networks. We run 6 graph algorithms on these datasets for our evaluation. ShortestPath (SSSP), WidestPath (SSWP), Breadth-First Search (BFS) and Connected Components (CC) are the representative applications for selection based update functions. Incremental PageRank and Adsorption are evaluated to show the performance of accumulative algorithms. We note that, for our optimization technique with the embedding of dependency information in events (DAP), the event size

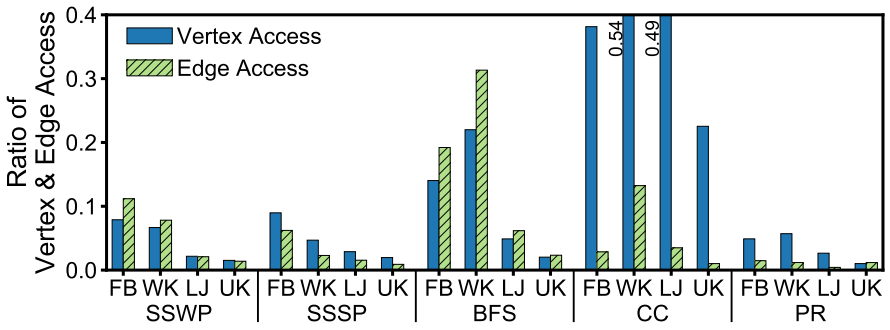


Figure 8.3: Number of vertex and edge accesses in JetStream normalized to GraphPulse.

is bigger than GraphPulse and thus requires a smaller graph slice to fit in the memory. We run 6 slices on Twitter and 3 slices on UK-domain graph for the selective algorithms in JetStream compared to 3 and 2 slices respectively for GraphPulse.

8.8.2 Performance and Characteristics

Overall Performance

Table 8.3 shows the execution time of JetStream with different workloads for batches of 100K edge updates. Each batch contains 70% insertions and 30% deletions of edges. The table also shows the speedup over GraphPulse (GP), KickStarter (KS), and GraphBolt (GB) for comparative workloads. GraphPulse demonstrates the cost of complete recomputation of the graph in an accelerator. JetStream takes 3 to 74 times less than GraphPulse (13× on average) to reevaluate a graph. This advantage primarily comes from heavily reduced vertex computation and edge communication required in JetStream. Fig. 8.3 shows that JetStream limits the number of vertex accesses to less than 54% and as low as 3% of what GraphPulse would require with less than 30% events generated.

Table 8.3: Execution time (in ms) per query on JetStream and speedup over full evaluation in GraphPulse(GP), and incremental evaluation in KickStarter(KS) and GraphBolt(GB).

		WK	FB	LJ	UK	TW	GMean
SSWP	Jet	1.63	1.21	4.17	3.87	22.55	
	GP	10.4×	9.3×	16.7×	66.7×	43.2×	21.6×
	KS	12.4×	13.1×	8.4×	24.2×	5.2×	11.1×
SSSP	Jet	4.76	4.31	5.36	6.23	15.17	
	GP	9.4×	9.95×	13.3×	73.4×	35.5×	20.1×
	KS	21.8×	8.7×	6.5×	25.6×	11.2×	12.9×
BFS	Jet	2.74	1.24	1.61	8.12	17.75	
	GP	3.10×	5.35×	7.80×	8.18×	15.1×	6.9×
	KS	30.1×	8.31×	11.7×	11.5×	5.57×	11.3×
CC	Jet	1.64	1.44	2.59	5.07	11.73	
	GP	12.9×	13.2×	12.4×	21.4×	23.4×	16×
	KS	7.62×	8.60×	5.25×	9.38×	8.51×	7.72×
PageRank	Jet	5.17	4.29	6.62	6.99	169	
	GP	12.8×	19.5×	19.9×	56.6×	9.70×	19.4×
	GB	143×	231×	180×	402×	51.6×	165×
Adsorption	Jet	4.19	5.27	9.84	12.10	65.30	
	GP	5.78×	3.90×	5.08×	5.95×	9.41×	5.77×
	GB	12.7×	14.4×	15.9×	12.8×	38.6×	17.1×

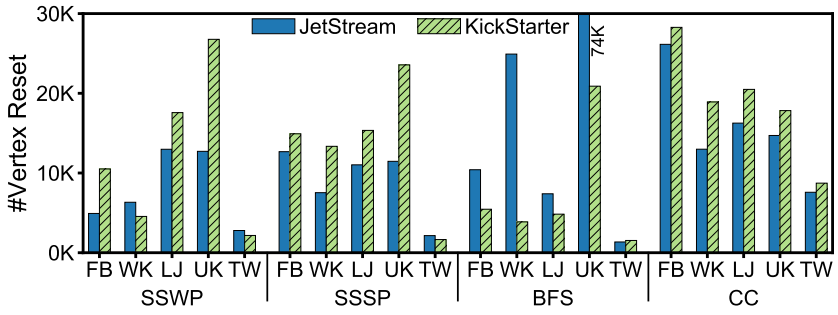


Figure 8.4: Number of vertices reset by 30K edge deletions.

Similar speedup in comparison to KickStarter and GraphBolt shows that the event-driven model is effective across incremental techniques. We observe up to 30× speedup over KickStarter and 400× over GraphBolt. JetStream is 18× faster on average than both. JetStream's asynchronous model performs better on the narrow but long graphs (UK, WK) than the synchronous software frameworks.

Approximation Effectiveness

JetStream adopts a technique similar to KickStarter for trimming the set of vertices. KickStarter employs value-aware and dependency graph (with levels) based trimming to limit recomputations. The source-based dependency-aware propagation technique in JetStream often finds smaller set of impacted vertices. Fig. 8.4 shows the number of vertices reset in JetStream and KickStarter for the same 30K batch of deletions.

Memory access efficiency

The ability to prefetch and utilize memory effectively is one of the major source of speed up in GraphPulse. The caches use 64-bytes lines which may not all be accessed. We

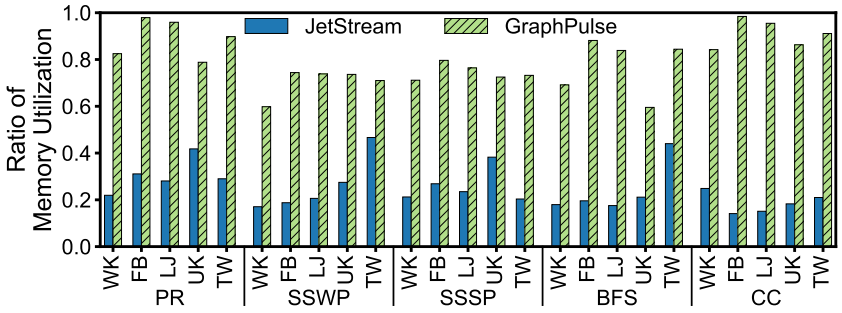


Figure 8.5: Utilization of off-chip memory transfers in JetStream.

show in Fig. 8.5 the ratio of bytes read into the computation engine from cache/prefetcher to bytes read from memory into caches to demonstrate how efficiently the off-chip data transfers were utilized as an indication of spatial locality. JetStream uses the same memory prefetching and edge cache structures already built into the GraphPulse datapath. Since the active tasks (*events*) in JetStream are fewer and sparse in JetStream, it cannot harvest spatial locality as well as GraphPulse. As a result, the memory access utilization ratio is less than one-third of GraphPulse. However, having fewer computational tasks still makes JetStream significantly faster during incremental computation. Optimizing the memory access efficiency of JetStream is a potential avenue for future improvements.

Effects of Optimizations

We show the effects of the optimizations in terms of speedup over full recomputation in GraphPulse in Fig. 8.6. The baseline JetStream model is conceptually simple. However, without a mechanism to restrict tagging to only the affected vertices, it tags too many vertices in the graph, often leading to work comparable to full recomputation for most applications. VAP performs sufficiently well for SSSP and SSWP, but fails to provide

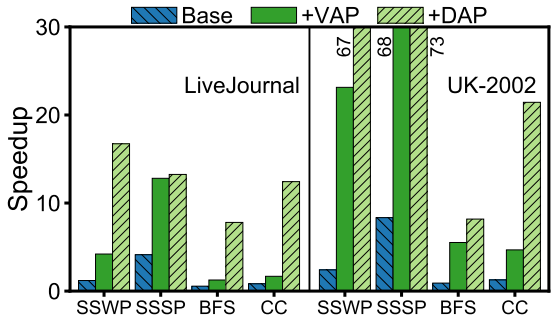


Figure 8.6: Speedup over GraphPulse for Baseline JetStream, VAP and DAP optimizations.

a noticeable advantage for BFS and CC. The latter two applications have many vertices set to the same value, making the VAP optimization ineffective. DAP alleviates this problem and works well for all applications. However, VAP has the advantage over DAP in that it does not expand the event size to include source information.

Sensitivity to Batch Size

In Fig. 8.7, we have shown how the performance of the engine varies with different batch sizes. Taking a 100K batch size as the baseline, we showed the speed up for different batches for PageRank and SSSP running on LiveJournal graph. The speedup is based on JetStream's runtime for 100K batch size. JetStream speeds up significantly as the batch size gets smaller because it has little overhead for incremental data maintenance. JetStream can handle computations very fast for smaller batches where the number of changes or computations is low. JetStream's speedup grows orders of magnitude faster that KickStarter. This time is only the processing time, and the end-to-end performance may have other overheads to receive and batch the updates.

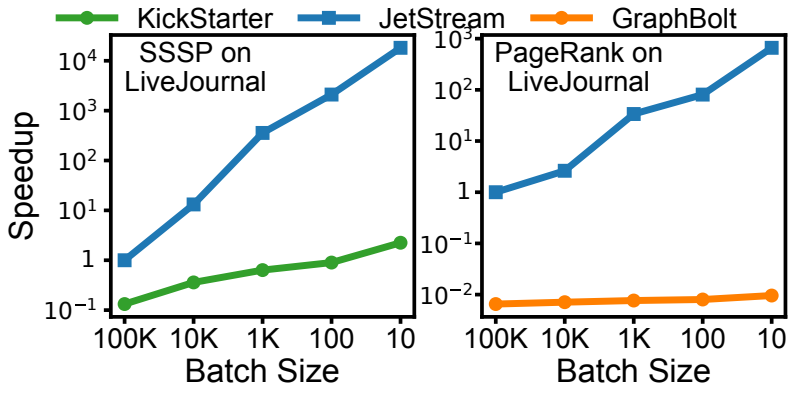


Figure 8.7: Sensitivity to batch size. Run-time shown as speedup over JetStream with 100K batch.

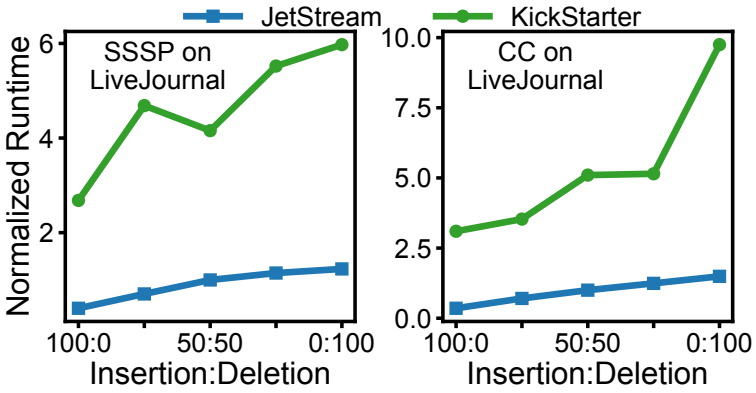


Figure 8.8: Run-time sensitivity to batch composition. Run-time is normalized to 50:50 composition on JetStream.

Sensitivity to Batch Composition

Edge deletions require more processing than edge additions in JetStream. An approximation phase is required to revert the effects of a deleted edge on the graph, which may propagate to many vertices for some critical edges. All the impacted vertices need to be reprocessed in the recomputation phase. Edge addition resembles regular events during the recomputation phase, and their effects are usually localized. Fig. 8.8 shows the effect of the composition of a batch on the run-time for SSSP and CC. Note that the run-times are

normalized to JetStream's run-time for a 50-50 batch. An insertion-only batch converges 3 to 4 times faster on average than a deletion-only batch of the same size. Run-time increases as the ratio of deleted edges increases. KickStarter, too, demonstrates faster convergence with fewer deletions, but there is no concrete dependence of the run time on the ratio of deletions. KickStarter attempts to approximate the value of an impacted vertex before propagating the tag. JetStream attempts to minimize tagging using DAP optimization but only approximates after all tags are propagated. On the other hand, for PageRank and Adsorption in JetStream, the addition or deletion of one edge also mutates the other edges (weight) for a vertex, and both types of updates are handled similarly. Therefore, such algorithms are not noticeably affected by batch composition.

8.8.3 Hardware Cost and Power Analysis

We model JetStream using the same configuration as GraphPulse: 64MB on-chip memory for queue, and 8 processing pipelines with 2KB scratchpad and 1KB edge-cache on each. We use CACTI 7 [7] for power and area estimate for all memory elements. The queue memory is modeled in 22nm ITRS-HP SRAM logic. The biggest component of the communication network is a 16x16 NoC between the event generation streams and the queues. Each port of the NoC is shared by several generator or queue ports.A breakdown of the total power and area estimate for the accelerator is shown in Table 8.4. The number in parenthesis is the increase over similarly configured GraphPulse. The overall increase in area and power is around 3% and 1% respectively.

JetStream reuses most architectural components of GraphPulse, including the event queue, prefetcher, and cache. Memory elements have the same physical size but

Table 8.4: Power and area of the JetStream accelerator components

	#	Power(mW)			Area(mm^2)
		Static	Dynamic	Total	
Queue	64	117 (+1%)	20.7 (-6%)	8815 (∼0%)	192 (+1%)
Scratchpad	8	0.35 (∼0%)	1.2 (+6%)	12.1 (+4%)	0.21 (∼0%)
Network		91 (+78%)	5.4 (+58%)	97 (+77%)	5.7 (+84%)
Proc. Logic		-	-	1.8 (+40%)	0.7 (+51%)
Total		-	-	8926 (+1%)	199 (+3%)

contain fewer events due to the larger event size. As a result, there are some resource overheads due to larger buffers and interconnects. However, the dynamic energy is lower because JetStream processes fewer vertices propagating events (many vertices are already converged). Overhead from the buffers and communication buses also increases due to the larger event size. Floating point units account for the bulk of the processing and coalescing logic and remain the same in input size. Thus, the extra processing logic for JetStream adds only a small power and area overhead. The processing time in JetStream is shorter, making JetStream ∼13 times more energy-efficient than full recomputation with GraphPulse. The total area of JetStream is about $200mm^2$ with a 28nm technology.

Chapter 9

Conclusions and Future Work

This dissertation presents and analyzes our methodology for building accelerators for irregular applications using event-driven techniques. We have studied the limitations of irregular applications and their current software implementations to identify bottlenecks and overheads. To develop efficient hardware implementations for these applications, we found that we need to shift focus away from the execution techniques optimized for conventional CPUs and look for specialized methods that can take advantage of the features and capacities of hardware platforms such as FPGAs and ASIC. An event-driven system stands out as the optimal candidate for implementation in hardware. Many distributed implementations of different irregular applications follow a message-passing approach, which is not replicated in a shared memory system because the overhead for maintaining and managing the messages can be overwhelming and become the bottleneck of these systems. However, the complexities of the message storage can be relegated to dedicated hardware components, making these systems very simple to parallelize and optimize in a shared memory system.

Therefore, we studied event-driven systems focusing on designing suitable hardware implementation and developing methodologies for converting traditional execution methods to event-driven execution models.

In this course of our work, we designed and analyzed a PDES accelerator on an FPGA. PDES-A is designed to support 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 significant opportunities to improve further 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.

We also presented GraphPulse, an event-based asynchronous graph processing accelerator. We showed how the event abstraction naturally expresses asynchronous graph computations and optimizes memory access patterns. It also simplifies computation scheduling and tracking, and eliminates the overhead for synchronization or atomic operations. As a result, GraphPulse achieves an average of 28x improvement in performance over Ligra running on a 12 core CPU implementation and an average of 6.2x performance improvement over Graphicionado.

Our third accelerator, JetStream, is the first hardware accelerator for streaming graphs. JetStream extends GraphPulse to reuse intermediate states to avoid a complete cold-start recomputation on the updated graph. JetStream supports edge additions and

deletions for both monotonic and accumulative algorithms. It achieves an average speedup of 13x over hardware accelerator GraphPulse and 18x over software frameworks at baseline batch sizes. This advantage increases substantially for small batch sizes.

The accelerators share a similar datapath but differ in implementing the event queues and the execution model for the applications they support. A standard limitation of the accelerators is their dependence on the size of the event queue for the largest workload they can handle at a time. The way to subvert this limitation is through partitioning approaches. We show how a large graph can be partitioned for GraphPulse and JetStream so that the event queue can fit within the available on-chip memory. However, there are many different possible partitioning approaches that can result in different performance characteristics for the application. Analysis of different partition approaches and their profitability with event-driven systems is a future area of study for our research.

There is vast potential for future extensions and impacts for the event-driven computation model. We can look into the possibilities from two perspectives. First, within the applications that we have demonstrated, there are many possible areas that can result in significant performance improvement. Task scheduling is decided by the order of events in these applications. There are opportunities to gain algorithmic or computational benefits by studying different event-scheduling policies. For instance, events can be manipulated in GraphPulse to impart fine-grained control over the vertex access pattern or the convergence speed. Prior works on graph processing demonstrate good performance from well-crafted edge update prioritization policy [115]. Similar benefits may be obtainable in GraphPulse by exploiting the event scheduling policy.

The second approach is to broaden the application domains supported by the event-driven acceleration. Since event-driven representation closely resembles message-passing-based distributed systems, many existing distributed computation models can be adapted to the event-driven execution model, as demonstrated by our development of the graph execution model. Especially the domains working on relational data, such as Graph Mining and Graph Neural Network, may be susceptible to an event-driven implementation with relative ease. We consider this an area of possible future explorations.

The analysis we have shown on the three accelerators in this dissertation establishes our hypothesis that significant performance gain can be achievable for irregular applications using accelerators based on event-driven execution. The dissertation also details the methodology for developing such architectures and converting existing algorithms to event-driven paradigm. Overall, we consider event-driven systems to have great potential as the basis for accelerating irregular applications.

Bibliography

[1] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. *SIGARCH Computer Architecture News*, 43(3), June 2015.

[3] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 11, 2011.

[4] A. Ayupov, S. Yesil, M. M. Ozdal, T. Kim, S. Burns, and O. Ozturk. A Template-Based Design Methodology for Graph-Parallel Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):420–430, Feb 2018.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.

[6] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proc. International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pages 44–54, 2006.

[7] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.*, 14(2), June 2017.

[8] Peter D. Barnes, Christopher D. Carothers, David R. Jefferson, and Justin M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, page 327–336, New York, NY, USA, 2013. Association for Computing Machinery.

[9] David W. Bauer Jr., Christopher D. Carothers, and Akintayo Holder. Scalable time warp on blue gene supercomputers. In *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 35–44, 2009.

[10] S. Beamer, K. Asanović, and D. Patterson. Reducing Pagerank Communication via Propagation Blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, May 2017.

[11] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 2, pages 538–547 vol.2, Tel Aviv, Israel, 2000. IEEE.

[12] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[13] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM*, 31(10):1220–1227, October 1988.

[14] E. Bullmore and O.Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. In *Nature Reviews Neuroscience, 10(3)*, pages 186–198, 2009.

[15] P. Burnap, O. F. Rana, N. Avis, M. Williams, W. Housley, A. Edwards, J. Morgan, , and L. Sloan. Detecting tension in online communities with computational Twitter analysis. In *Technological Forecasting and Social Change, 95*, pages 96–108, 2015.

[16] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. Facilitating real-time graph mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, page 1–8, New York, NY, USA, 2012. Association for Computing Machinery.

[17] Christopher D. Carothers. ROSS-Models. `https://github.com/carothersc/ROSS-Models`, 2018.

[18] Christopher D. Carothers, David Bauer, and Shawn 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.

[19] Guillaume Chapuis, Stephan Eidenbenz, Nandakishore Santhi, and Eun Jung Park. Simian Integrated Framework for Parallel Discrete Event Simulation on GPUs. In *Proceedings of the 2015 Winter Simulation Conference*, WSC '15, pages 1127–1138, Piscataway, NJ, USA, 2015. IEEE Press.

[20] Huilong Chen, Yiping Yao, Wenjie Tang, Dong Meng, Feng Zhu, Yuewen Fu, and Yiping Yao. 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 2015, pages 41–49, Piscataway, NJ, USA, 2015. IEEE Press.

[21] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. 7th ACM european conference on Computer Systems*, pages 85–98, 2012.

[22] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. of the VLDB Endowment*, 8(12):1804–1815, 2015.

[23] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. ForeGraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture. In *Proc. International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 217–226, 2017.

[24] J. M. Daper. Compiling on Horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 51–52, 1988.

[25] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria 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.

[26] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Mathematical Software*, 38(1):1:1–1:25, December 2011.

[27] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[28] Philip Dexter, Yu David Liu, and Kenneth Chiu. Formal foundations of continuous graph processing, 2020.

[29] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proc. 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.

[30] M. De Domenico, A. Lima, P. Mougel, and M. Musolesi. The anatomy of a scientific rumor. In *Scientific Reports, http://dx.doi.org/10.1038/srep02980*, 2013.

[31] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *Proc. IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[32] R Ewald, C Maus, A Rolfs, and A Uhrmacher. Discrete event modelling and simulation in systems biology. *Journal of Simulation*, 1(2):81–96, 2007.

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

[34] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[35] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[36] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[37] Sounak Gupta and Philip 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*, SIGSIM PADS '14, pages 15–26, New York, NY, USA, 2014. ACM.

[38] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proc. 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[39] Minyang Han and Khuzaima Daudjee. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *PVLDB*, 8(9):950–961, 2015.

[40] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proc. European Conference on Computer Systems (Eurosys)*, 2014.

[41] M. C. Herbordt, F. Kosie, and J. Model. 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*, pages 248–257, Palo Alto, CA, USA, April 2008. IEEE.

[42] R.R. Hill, J.O. Miller, and G.A. McIntyre. Applications of discrete event simulation modeling to military problems. In *Proceeding of the 2001 Winter Simulation Conference (Cat. No.01CH37304)*, volume 1, pages 780–788 vol.1, 2001.

[43] Jiewen Huang and Daniel J. Abadi. Leopard: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *Proc. VLDB Endowment*, 9(7), March 2016.

[44] H.H.J. Hum and G.R. Gao. Efficient support of concurrent threads in a hybrid dataflow/von Neumann architecture. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, IEEE IPDPS '91, pages 190–193, 1991.

[45] Amazon Web Services, Inc. Amazon EC2 F1 Instances, 2018.

[46] H. Isah, P. Trundle, , and D. Neagu. Social media analysis for product safety using text mining and sentiment analysis. In *14th UK Workshop on Computational Intelligence (UKCI)*, 2014.

[47] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. In *Proc. Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[49] Deepak Jagtap, Ketan Bahulkar, Dmitry Ponomarev, and Nael 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.

[50] David R. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.

[51] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 228–241, Dec 2015.

[52] Mark C. Jeffrey, Suvinay Subramanian, Maleen Abeydeera, Joel Emer, and Daniel Sanchez. Data-Centric Execution of Speculative Parallel Programs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[53] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. Tripoline: generalized incremental graph processing via graph triangle inequality. In *Proc. Sixteenth European Conference on Computer Systems*, pages 17–32, 2021.

[54] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 523–536, USA, 2016. USENIX Association.

[55] Sean Kane, Sounak Gupta, and Philip A. Wilsey. Analyzing simulation model profile data to assist synthetic model generation. In *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–10, 2019.

[56] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[57] J. T. Kuehn and B. J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 28–34, 1988.

[58] Pradeep Kumar and H. Howie Huang. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[59] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proc. 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[60] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi : Large-scale graph computation on just a PC. In *USENIX OSDI*, pages 31–46, 2012.

[61] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. `http://snap.stanford.edu/data`, June 2014.

[62] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[63] Lijun Li and Carl Tropper. A design-driven partitioning algorithm for distributed verilog simulation. In *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, pages 211–218. IEEE, 2007.

[64] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[65] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[66] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proc. Sixteenth European Conference on Computer Systems*, page 83–98, 2021.

[67] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proc. Fourteenth European Conference on Computer Systems*, pages 1–16, 2019.

[68] Romolo Marotta, Mauro Ianni, Alessandro Pellegrini, and Francesco Quaglia. A lock-free o(1) event pool and its application to share-everything pdes platforms. In *2016 IEEE/ACM 20th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 53–60, 2016.

[69] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, October 2015.

[70] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *Proc. International Symposium on Microarchitecture (MICRO)*, 2018.

[71] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. PHI: Architectural Support for Synchronization- and Bandwidth-Efficient Commutative Scatter Updates. In *Proc. International Symposium on Microarchitecture (Micro)*, page 1009–1022, 2019.

[72] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proc. Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[73] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, Feb 2017.

[74] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, page 456–471, 2013.

[75] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proc. ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 166–177, June 2016.

[76] Hyungwook Park and Paul A. Fishwick. A GPU-Based Application Framework Supporting Fast Discrete-Event Simulation. *Simulation*, 86(10):613–628, October 2010.

[77] Kalyan S. Perumalla. 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, pages 74–81, Washington, DC, USA, 2006. IEEE Computer Society.

[78] 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. 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.

[79] S. Rahman, N. Abu-Ghazaleh, and R. Gupta. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proc. 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 908–921, 2020.

[80] Joseph Rios. An efficient FPGA priority queue implementation with application to the routing problem. Technical report, UC Santa Cruz, Santa Cruz, CA, USA, 2007.

[81] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011.

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

[83] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[84] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[85] N. Santhi, S. Eidenbenz, and J. Liu. The Simian concept: Parallel Discrete Event Simulation with interpreted languages and just-in-time compilation. In *2015 Winter Simulation Conference (WSC)*, pages 3013–3024, Huntington Beach, CA, USA, December 2015. IEEE.

[86] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 445–456, October 2015.

[87] D. Sengupta and Shuaiwen Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *ISC*, 2017.

[88] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. GraphIn: An Online High Performance Incremental Graph Processing Framework. In *Proc. European Conference on Parallel Processing (EuroPar)*, page 319–333, 2016.

[89] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proc. International Conference on Management of Data*, pages 417–430, 2016.

[90] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proc. SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146, 2013.

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

[92] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing Away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems. In *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, page 161–174, Jun 2017. Citation Key: RAts.

[93] G. M. Slota, K. Madduri, and S. Rajamanickam. PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks. In *IEEE International Conference on Big Data (Big Data)*, pages 481–490, 2014.

[94] George M. Slota, Sivasankaran Rajamanickam, Karen Devine, and Kamesh Madduri. Partitioning Trillion-Edge Graphs in Minutes. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 646–655. IEEE, May 2017.

[95] Jeffrey 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.

[96] L. Takac and M. Zabovsky. Data analysis in public social networks. In *Proceedings of the International Scientific Conference and International Workshop Present Day Trends of Innovations*, 2012.

[97] Wenjie Tang and Yiping Yao. A GPU-based Discrete Event Simulation Kernel. *Simulation*, 89(11):1335–1354, November 2013.

[98] M. R. Thistle and B. J. Smith. A Processor Architecture for Horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 35–41, 1988.

[99] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of Facebook networks. *Phys. A*, 391(16), Aug 2012.

[100] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[101] Luis M. Vaquero, Felix Cuadrado, Dionysios Logothetis, and Claudio Martella. Adaptive Partitioning for Large-Scale Dynamic Graphs. In *Proc. International Conference on Distributed Computing Systems*, pages 144–153, 2014.

[102] Keval Vora, Rajiv Gupta, and Guoqing Xu. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):1–27, 2016.

[103] Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proc. 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 237–251, 2017.

[104] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, New York, NY, USA, 2014. ACM.

[105] Keval Vora, Guoqing Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 507–522, Denver, CO, 2016. USENIX Association.

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

[107] Jingjing Wang, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 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, pages 93–95, Washington, DC, USA, 2012. IEEE Computer Society.

[108] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. GraSU: A Fast Graph Update Library for FPGA-Based Dynamic Graph Processing. In *Proc. International Symposium on Field-Programmable Gate Arrays*, page 149–159, 2021.

[109] Yahoo! Webscope Program. `http://webscope.sandbox.yahoo.com/`. Accessed: 2021-09-22.

[110] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.

[111] G. Zhang, W. Horn, and D. Sanchez. Exploiting commutativity to reduce the cost of updates to shared data in cache-coherent systems. In *Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 13–25, Dec 2015.

[112] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 544–557, 2018.

[113] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication of PIM-based Graph Processing with Efficient Data Partitioning. In *Proc. IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

[114] Xiange Zhang. Application of discrete event simulation in health care: a systematic review. *BMC health services research*, 18(1):1–11, 2018.

[115] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[116] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Maiter: An Asynchronous Graph Processing Framework for Delta-based Accumulative Iterative Computation. *CoRR*, abs/1710.05785, 2017.

[117] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping. *IEEE Trans. Knowl. Data Eng.*, 30(5):895–907, 2018.

[118] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018.

[119] S. Zhou, C. Chelmis, and V. K. Prasanna. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *Proc. IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110, May 2016.

[120] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. An FPGA Framework for Edge-centric Graph Processing. In *Proc. 15th ACM International Conference on Computing Frontiers*, pages 69–77, 2018.

[121] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. GraphQ: Scalable PIM-Based Graph Processing. In *Proc. International Symposium on Microarchitecture (Micro)*, page 712–725, 2019.