NetGVT: Offloading Global Virtual Time Computation to Programmable Switches

Ricardo Parizotto
UFRGS - Brazil

Braulio Mello
UFFS - Brazil

Israat Haque
Dalhousie University - Canada

Alberto Schaeffer-Filho
UFRGS - Brazil

ABSTRACT

Distributed discrete-event simulation is an essential method for analyzing large-scale models, including weather forecast and network simulations. A distributed simulation often requires synchronizing state among the different parts of the model according to a global virtual time (GVT). However, existing approaches require multiple round-trip times to a server to compute a new GVT value. In this paper, we propose NetGVT, a system that computes GVT using programmable switches, thereby avoiding the round-trip latency of a server-based solution. In particular, our design is concerned with two main constraints of the switch programming model: the limited number of arithmetic and logic operations and the limited memory available on the device. We aggregate computations and unroll them across different pipeline stages in a hierarchical manner to address the former. Then, we adopt compression mechanisms to store a short representation of virtual clocks in the on-chip registers to tackle the memory limitations. We implemented a prototype of NetGVT and evaluated its performance with a synthetic lock-step simulation in a Tofino switch. Our results demonstrate that NetGVT outperforms techniques that do not rely on in-network computing by 40% in terms of distributed simulations completion time.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; • **Computing methodologies** → *Distributed simulation.*

SOSR '22, October 19–20, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9892-3/22/10…$15.00
https://doi.org/10.1145/3563647.3563648

KEYWORDS

In-network Compute, Global Virtual Time, Data Plane

ACM Reference Format:
Ricardo Parizotto, Braulio Mello, Israat Haque, and Alberto Schaeffer-Filho. 2022. NetGVT: Offloading Global Virtual Time Computation to Programmable Switches. In *The ACM SIGCOMM Symposium on SDN Research (SOSR) (SOSR '22), October 19–20, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3563647.3563648

1 INTRODUCTION

Large-scale distributed simulations play an essential role in scientific research and many other domains, including weather forecast [34], military training [15], large-scale system design simulation acceleration (e.g., VLSI layout [7, 14]), manufacturing and supply chains designs [36, 41], etc. Traditionally, different parts of a simulation model run on distributed servers or in a cluster and need to be periodically synchronized to exchange timing information and establish a *global virtual time* for consistent event processing [2]. Global virtual time (GVT) [19] is an important aspect of performance in these distributed simulations and works as a lower bound on the simulation time. GVT computation must be fast to avoid delaying event processing, which can negatively impact the simulation completion time. Consequently, the model outcome may delay warnings for natural but predictable events, e.g., hurricanes or other natural disasters.

Developers usually have two choices for synchronizing simulations using GVT: a centralized server-based mechanism to coordinate the computation or a decentralized synchronization algorithm for application processes to perform the computation on servers. Examples are the Chandy and Misra Null-message protocol [8], the Granted Time Window algorithm based on the Distributed Snapshot protocol proposed by Mattern [32] and the Time Warp Mechanism [19], which uses rollback to recover from causal consistency violations. However, these schemes impact the simulation completion time due to the RTT propagation delay and software stack traverse latency while computing a new GVT value. Server-only implementations may become a bottleneck for synchronization because of the processing, buffering, and

transmission operations [28, 37]. Even if GVT computation is pushed to a server placed in an optimal location in the network to minimize RTT, the delay will still be imposed because of the server software stack.

Recent advances in programmable forwarding devices enable us to rewrite the behavior of switches using software abstractions. Network programmability allows us to leverage switch hardware to process information at a rate that outperforms the servers [28] and with a shorter propagation delay. One of the most popular languages for switch programming is P4 [4], which allows modifying forwarding device behavior. This motivated many emerging applications ([9, 22, 39, 40]) that offload parts of the computation from servers to networks, achieving economies of scale and lower operating costs. We argue that this modern computing paradigm creates the need and opportunity to revisit synchronization algorithms for distributed simulations.

Different deployment scenarios can benefit from offloading GVT synchronization to the network. For example, a cloud provider can run an in-network GVT service for its tenants that need to run simulations. Alternatively, an enterprise can deploy switches near a server cluster incrementally to accelerate existing simulations. Devising an in-network GVT synchronization mechanism that relies on programmable switches will reduce the propagation delay compared to a server-based deployment. As such, it can speed up GVT synchronization and accelerate large-scale distributed simulations.

In this paper, we propose NetGVT, a system for performing GVT computation using programmable data planes. NetGVT introduces primitives for distributed simulation synchronization that rely on emerging programmable switches to store, compute, and deliver GVT values to applications running on servers. Our design concerns two main challenges due to the switch programming model: the limited ALU operations available per stage and available memory on the device. To solve the former, we aggregate computations and unroll them across different pipeline stages when the computation exceeds the available ALU operations, judiciously resubmitting packets to iterate through them. Since naively resubmitting would impose additional overhead for packet processing, we do it hierarchically, requiring only a logarithmic set of resubmissions compared to the naive approach. To address the limited amount of SRAM available to store virtual times, we propose a compression mechanism capable of storing only short bit-vector representations of a virtual time but still ensuring correctness. Further, differently from existing in-network computing systems [18, 21, 22], NetGVT captures the essential notion of causality of logical clocks, which is required for GVT computation and distributed simulation synchronization.

Contributions. The main contributions of this paper can be summarized as follows:

- We design a GVT computation protocol with functionality offloaded to programmable data plane devices.
- We implement a prototype with the in-network component based on P4 for Tofino switches and a component based on Python for running on servers hosting distributed simulation nodes.
- We demonstrate that by offloading the GVT computation to programmable switches, we can improve up to 40% in the job completion time of a lock-step simulation.

2 BACKGROUND AND MOTIVATION

2.1 Distributed Simulations and GVT

A distributed simulation can be partitioned into a set of processes (P). Each process in P can simulate an entire simulation component [30]. For example, a component can be the physics of a hurricane or a gate from a circuit. During the simulation, the GVT is required to synchronize these components' events [19]. It is used to measure progress and define barriers to synchronization. The GVT is defined as the minimum value among all local virtual times (LVTs) and timestamps of all events in transit [19].

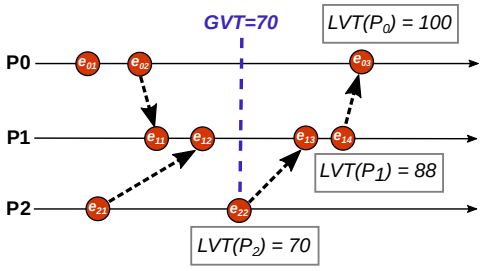

Figure 1: An example of GVT value in an event diagram.

Figure 1 presents, in an event diagram, a snapshot of three different processes, P_0, P_1, and P_2. An arrow represents a message exchange between two processes, and its endpoints consist of send and receive events. The GVT is then the minimum timestamp of the last event processed by all the processes. In this example, $GVT = LVT(P_2) = 70$, the timestamp of event e_{22}. Ideally, we want to have fast GVT computations so that the simulation will be subject to less wait time to order events.

The frequency at which GVT is computed depends on the synchronization algorithm implemented by the distributed simulator: it can be synchronous or asynchronous [23]. In a synchronous algorithm, events are performed in a lock-step manner [6]. Each event is processed only when its local

virtual time meets the GVT, and all the distributed components simulate the same virtual time. In an asynchronous algorithm, the processes can simulate the component events without a barrier in a lock-free manner [10]. However, this may violate causal consistency, thus requiring checkpoint-rollback mechanisms to restore the state of the processes and resume from a state where there is no causality violation. In the case of asynchronous simulations, GVT is used to define the virtual time when a process should conduct a checkpoint.

2.2 Programmable Data Planes

Data plane programmability allows network operators to define data plane functionality using software abstractions. The switch functionality is often expressed using domain-specific languages [4, 43] in a data plane model (e.g., TNA or v1model) [16]. The resulting code is then compiled into a packet processing architecture that supports the data plane model. An example architecture is the Protocol Independent Switch Architecture (PISA), which generalizes the Reconfigurable Match Table (RMT) [5] model.

In the PISA architecture, packets go through a packet parser, which instantiates user-defined protocols. After the parser processes a packet, it follows a pipeline of control flows and *match+action* tables. Match+action tables can be implemented on TCAM or SRAM, where actions are implemented using ALUs and run at line rate [42]. Finally, packet headers are emitted by a deparser. A packet can also be recirculated to the beginning of the pipeline or resubmitted to the parser, imposing additional overhead to packet processing. Recently, programmable data planes motivated the paradigm of in-network computing.

2.3 Motivation

In-network computing (INC) [3, 40] relies on programmable forwarding devices to offload computation into the network. We aim to reduce server [28] and network bottlenecks and increase the performance of GVT synchronization by leveraging programmable data planes and in-network computing to offload the virtual time synchronization to programmable switches. Our solution is an alternative to server-based synchronization protocols. Offloading the GVT computation to switches is beneficial for several reasons [44]. Firstly, performing the computation as early as possible decreases network traffic, reduces congestion, and cuts the number of network hops at least by half to complete a GVT computation. Secondly, because the GVT computation computes the minimum value of a set of events, simple arithmetic operations that run at a line rate with modern switch hardware can be easily implemented. Finally, since the computation on switches occurs at line rate, the switch can implement the GVT comparison for an increasingly high number of processes with no significant impact on processing delay. Thus, the proposed NetGVT can speed up distributed simulations running in data centers or clouds, which is common in the distributed simulation field [12].

3 NETGVT

This section presents NetGVT, a system for performing GVT computation using programmable data planes. The system is capable of intercepting event messages and encapsulating logical clocks in a custom protocol header, which is used by switches to compute the logical clock barriers.

3.1 Challenges

Performing computation in programmable switches has many advantages, including line rate processing and the reduction in propagation delay. However, there are fundamental challenges that need to be overcome to take advantage of the switch computation for virtual time synchronization.

Limited set of ALU operations. There are restrictions on which operations are allowed and how these operations are performed. For example, the number of ALUs at each pipeline stage is fixed. This creates a challenge if the program's layout requires more ALU operations than those available in the pipeline stage [17]. Additionally, P4 does not support loops, making it challenging to implement the GVT computation considering the constraints of the switch programming model. To overcome this challenge, NetGVT unrolls the GVT computation using resubmission. However, since using too many resubmissions can negatively impact throughput, we judiciously use it by dividing multiple logical clocks into hierarchical chunks and only traverse the essential chunks for the required computation.

Limited amount of memory. Within a processing pipeline, we use *stateful registers* to store logical clocks inside the switch. However, stateful registers consume SRAM space, which is limited. Such constraints limit the number of virtual clocks we can store in the switch at any point in time. To overcome this challenge, we propose a compression mechanism for virtual clocks that only needs to store an absolute difference between the virtual clock and an integer scalar. We periodically update the scalar to keep the memory demand low, ensuring that the absolute difference from process clocks to the scalar can fit in short bit vectors.

3.2 Overview

The architecture of NetGVT, presented in Figure 2, consists of multiple servers connected to a programmable switch and an SDN controller.

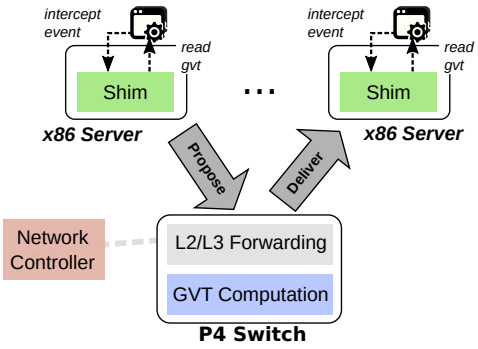

Figure 2: NetGVT architecture overview.

Programmable switch. The switch is the main component of NetGVT. Simulation processes exchange event messages, and the switch is responsible for intercepting packets and storing a compressed version of their local virtual time to compute a new GVT value. First, the switch intercepts and processes a new protocol header that encapsulates virtual clocks. Next, the comparisons required to determine the new GVT are unrolled across multiple pipeline stages, subject to ALUs constraints. Finally, the switch is responsible for sending the resulting computed value to all servers participating in the simulation using multicast primitives.

Shim layer. Existing distributed simulations often hard-code the synchronization protocol and how simulation nodes update their GVT. These protocols often operate over TCP, but they do not consider any computation occurring in the network. Instead, NetGVT provides a shim layer that resides in simulation nodes and is capable of intercepting event messages and encapsulating the local clocks within a custom protocol header to offload the GVT computation to switches.

SDN controller. Besides deploying simple forwarding rules, the NetGVT SDN controller is responsible for updating the switch when the set of servers in a cluster is modified (addition/removal) or when there are changes in the set of processes participating in a distributed simulation. When a simulation starts, the logical clock of each process is reset to zero. Finally, the controller is also responsible for configuring multicast rules for delivering GVT values to all the servers.

3.3 Handling event messages

Distributed simulation processes execute events according to a global virtual time. Consequently, processes must have ways to read the current GVT before executing an event. Additionally, after processing events, a process will have a different local virtual time, possibly leading to a new GVT value. Consequently, it is necessary to provide ways for the simulation process to encapsulate these values so that the NetGVT switch can intercept and process them.

Intercepting packets. The shim layer must intercept event messages from distributed simulation processes. After intercepting a message, the shim extracts the virtual time of the process and encapsulates this information using a custom packet header as shown in Figure 3. The custom header added by the NetGVT shim layer is composed of the following attributes:

- **`Type`**: denotes an operation, whether the packet is *proposing* or *delivering* a new value, or *starting* a new execution.
- **`Process ID (Pid)`**: is the unique identifier of a process and specifies the *process* that created a message.
- **`Value`**: stores a value to be exchanged between servers and the network device. It can be a GVT or an LVT, depending on the context of the operation.

These header fields are modified by the switch and by shim layer instances during event message transmission. Finally, to maintain a common timing reference among the distributed processes, the shim layer intercepts packets arriving from switches and stores the new GVT.

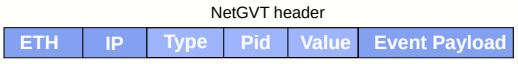

Figure 3: NetGVT protocol packet format.

Packet losses. Although packet losses are uncommon in a cluster environment, NetGVT employs a mechanism for dealing with this loss. Because switches can not create new packets, we delegate most tasks for dealing with packet losses to the shim layers running on servers. The shim uses timeouts and re-transmissions for detecting and recovering from packet losses. The shim associates a timer to each proposal. If the timer triggers a timeout, the system assumes a packet was lost and retransmits it to the switch.

3.4 Data Plane Layout

Figure 4 presents the layout of the NetGVT switch data plane. Upon receiving a packet, the switch checks whether it is an event or standard forwarding packet. Standard forwarding packets are forwarded normally to an output port, thus enabling our system to be incrementally deployable. This also enables us to deploy NetGVT into existing distributed simulators without requiring us to change how event message exchange occurs.

Updating virtual times. In the case of event packets, the switch updates the respective LVT in a `LVTList` according to the process identification. The `LVTList` can be implemented either using *match+action* tables or as an array of on-chip registers. The former would require interaction with the control plane to update the list's contents, imposing a control

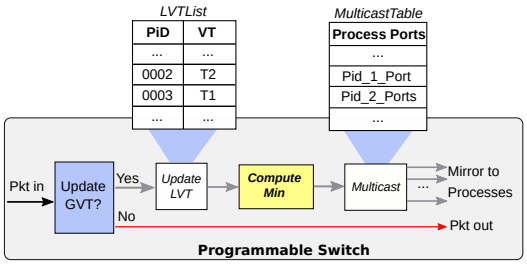

Figure 4: The layout of NetGVT switch data plane.

plane control-loop as part of the GVT computation. Thus, we use the on-chip registers[1]; hence, we can easily update the virtual time of each process without any interaction with the control plane. However, registers consume precious SRAM memory. We propose a compression mechanism that stores only the absolute difference between the LVT and a *scalar*, in order to keep SRAM memory usage low.

By periodically updating the scalar to a value close to the processes LVT (e.g., the last GVT value), we ensure that the absolute difference is a small integer that can fit in a small bit vector. For example, in a scenario where the simulation process time is $t = 100,000$ and the scalar is $s = 99,070$, NetGVT only needs to store the difference $d = 930$, which can be represented with a bit vector of size 12, instead of regular 32-bit integers.

Note that instead of updating all LVTs during a scalar update, the NetGVT switch is able to detect when an LVT update exceeds its scalar range. Specifically, the switch defines a *shadow scalar*, based on the current GVT, which corresponds to the scalar that is going to succeed the current one. The switch then identifies any LVT that needs a new scalar and uses the shadow one[2]. The remaining LVTs still keep using the current scalar. Over time all LVTs will exceed the current scalar range and converge to the shadow one, which will then become the current scalar. Keeping the notion of a shadow in addition to the current scalar avoids having to update all LVTs at once in situations when the scalar needed to change, at the cost of only two extra registers.

Computing the global virtual time. After updating the LVT of the process, the switch will start computing the GVT by iterating through other virtual times and calculating the minimum value among all the LVTs. Figure 5 illustrates how our design maps the GVT computation into the ASIC. Given that the P4 language does not support loops, we unroll GVT calculation as a set of *if-else* statements that iterate through the list of local virtual times. However, having a large number of processes leads to a large number of local virtual times. Consequently, the chain of *if-else* statements may not fit in the pipeline width because it would require more ALU operations than what is available in a pipeline. To address this challenge, our design partitions the *if-else* chain into different *chunks*, where each chunk is responsible for a portion of virtual times. Assuming that n virtual times are partitioned into chunks of the pipeline $width$ size, a naive approach to computing the GVT is to use resubmissions to iterate through all the chunks. This would enable a correct computation by iterating through the list of virtual times, even if the list is large. However, this approach still requires $r = n/width = |chunks| = O(n)$ resubmissions, imposing additional overhead for packet processing. Thus, reducing the number of resubmissions is necessary to get the best throughput from the switch.

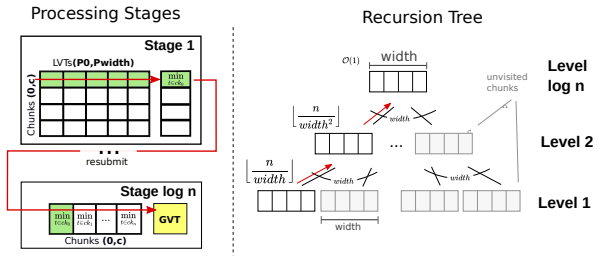

Figure 5: Mapping GVT computation to programmable ASIC.

Judiciously using resubmissions. The insight to reducing the number of necessary resubmissions is that we can memorize the minimum value of each chunk computed by previous events and only iterate through these memorized values to compute the GVT. This avoids iterating through all the virtual clocks by just looking at each chunk's local minimum value. We generalize our design to scenarios in which iterating through the chunks' minimum also requires more ALU operations than what is available in a single pipeline. We hierarchically create chunks of minimum values of lower-level chunks and perform this process recursively. The process of creating chunks will occur until the number of operations required to compute the local minimum does not exceed the amount of ALU operations available in the pipeline.

Complexity analysis. As we can observe in Figure 5, the process of mapping these operations to the pipeline can be seen as a recursion tree, with the branching factor of $width$. Each node has at most $width$ operations, which can run at a line rate. In contrast, the next levels have more $width$ operations associated with each of the $width$ chunks, and so on. The tree has height $\log_{width}(n)$, and at the leaves is our

[1]LVT slots are indexed by *metadata* loaded into a *match+action* table that maps a PID to the correct register slot.
[2]Since these LVTs are on a different scale, we invalidate them by setting up a bit in a bitmask. This does not affect the GVT computation accuracy because it only occurs to processes that clearly are not the new GVT.

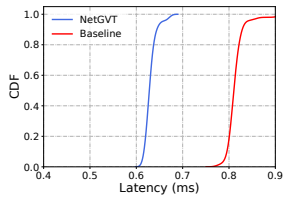

Figure 6: Latency

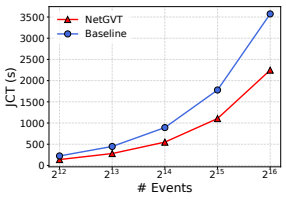

Figure 7: JCT

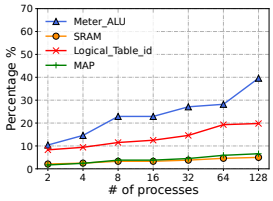

Figure 8: % Resources

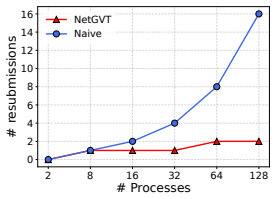

Figure 9: # Resubmissions

base case, which has as the default $\frac{n}{width}$ leaf chunks. Since sometimes $\frac{n}{width}$ can not fit in a single pipeline, we perform the hierarchical process by dividing the entry size by the pipeline $width$ at each recursion level. In level i, we remove $\lfloor n/width^{i-1} \rfloor - 1$ chunks from the list of chunks the switch needs to visit using resubmission. Computing a new GVT value will require starting from a leaf node and resubmitting until reaching the root chunk. In the worst-case scenario, the amount of resubmissions is $O(\log n)$ instead of $O(n)$ from the naive approach. At that point, NetGVT will be able to compute the current GVT value.

Delivering the GVT. After computing and saving the new GVT value, the switch inserts this value in the packet header and sends the packet to match the `MulticastTable`. `MulticastTable` will create a copy of the packet with the updated GVT for each shim instance and change the output ports. Finally, all packets are forwarded to their destination.

4 EXPERIMENTAL RESULTS

In this section, we present the experimental results. Our experiments focus on answering two main questions: (i) How does the performance of NetGVT compare to a server-only solution; and (ii) How does NetGVT scale with different workloads.

Experimental setup. We implemented the switch logic as a P4-16 [4] program based on the `TNA` model in approximately 600 lines of code. The controller is a python program (~40 lines of code). The P4 compiler generates an API that the controller uses to modify the multicast groups and addresses according to the mechanism described earlier. We developed the shim layer in Python using Scapy (~120 lines of code). The source code is available in [1]. The experiments were conducted in a testbed with two servers connected by a Wedge 100BF-32X 32-port programmable switch with a 3.2 Tbps Tofino ASIC. Each server is an Intel(R) Xeon(R) Silver 4210R CPU @ 2.4 GHz, with 10 cores and 32 GB memory.

Workload. We run a microbenchmark that instantiates one process at each server of our testbed. For the microbenchmark, we configured the processes to repeatedly send proposals that update the GVT value and used `tcpdump` to measure the latency. We compared our system to a server-only solution implemented with the same functionality as NetGVT. We configured the server-only solution in one of the servers, but instead of using the NetGVT switch, we used a simple L2 switch to forward event messages to the central server.

Computation time. We measured the latency to propose, compute and deliver a new GVT value using both NetGVT and the server-only solution. Figure 6 presents the CDF for the latency with a sample of 1,000 GVT updates. This experiment demonstrates that our solution consistently outperforms the server solution. We observed that the latency using NetGVT is not higher than 0.65ms for 50% of the proposals. Conversely, the server solution takes about 0.85s. This happens because NetGVT cuts a network hop avoiding a round trip to a server solution.

Job completion time. To understand the impact of the GVT computation time in a distributed simulation, we implemented a lock-step (synchronous) simulation that executes different amounts of events. In this simulation, the processes keep a local virtual time (LVT) and only run the next event when the LVT is less or equal to the current GVT; otherwise, the process is locked. We measured the time to complete the simulation and presented it in Figure 7. We can see that for 4096 events, the server solution performs similarly to NetGVT. However, as the number of events increases, we observe that NetGVT can complete simulations considerably faster than the server-only solution. This happens because events that the GVT delays will start earlier when using NetGVT. For example, instead of a process waiting 0.8 ms for a new GVT value to start processing a new event, it will wait instead for about 0.6 ms. As the number of events to be processed in a simulation increases, this early start makes a considerable difference. As part of ongoing work, we are studying the integration of NetGVT with existing simulators, such as NS-3[3] and NEST[4].

Resource utilization. We evaluated resource consumption of NetGVT on top of a simple L3/L4 forwarding switch and configured NetGVT for an increasing number of processes. Because we observed that the majority of simulations NetGVT aims to deal with often require around 2-128 processes [10, 35, 38, 45], we assumed this setup in our analysis. In order to handle a total of 128 processes, we compare LVTs of up to 8 processes (8 is the pipeline width) at each pipeline stage and need up to 3 levels of chunks, incurring in a total

[3]https://www.nsnam.org/docs/models/html/distributed.html
[4]https://www.nest-simulator.org/

of 2 resubmissions for updating the GVT. Differently from the previous experiments that run on the testbed, we used Intel P4 Insight for measuring resource consumption as the number of processes increases (Figure 8). For 128 processes, the virtual time storage increases the consumption of SRAM and MAP RAM up to less than 10% of the memory available in the switch. ALU consumption is close to 40%, and the logical table ID to 20%.

Resubmission analysis. As mentioned earlier, NetGVT may use resubmissions to enable the GVT computation. We note that resubmission adds around $0.65 - 0.75$ ns to latency [46]. Figure 9 shows the number of resubmissions required by the naive approach compared to NetGVT. For 128 processes, NetGVT requires only two resubmissions; resubmitting adds only $2 \times \pm 0.7$ ns to the latency of the scenario using only two processes. Instead, the naive approach would need to traverse all the 128 LVTs without optimizations, requiring up to 16 resubmissions (considering a pipeline width of 8). Thus minimizing the number of resubmissions is necessary for the overall computation performance.

5 RELATED WORK

Synchronization protocols. Fujimoto et al. [11] proposed a GVT protocol for shared-memory processors. Their protocol requires a round of communication for computing GVT since there is no need for message exchange between processors. Mattern [33] presented an algorithm for GVT computation for distributed simulations. This algorithm uses a variant of a distributed snapshot to compute the GVT value. However, none of these works consider network programmability. The most similar to our work is [37], which migrates the GVT computation to a programmable NIC. Although using the NIC avoids the overhead of the server software stack, this approach still requires an entire RTT to estimate the GVT. Differently, our work supports the GVT computation using programmable data planes, thus avoiding the overhead and latency of an end-to-end communication.

In-network computing. Research efforts have proposed several solutions related to in-network computing, such as in-network concurrency control [20, 31, 47]. Coordination services [9, 18] offload the Paxos consensus protocol to the network hardware in order to minimize exchanges with servers. HovercRaft [27] uses programmable switches to collect quorum and accelerate communication. Others run vertical Paxos between switches to build a reliable storage [21] or between servers to tolerate failures of network applications running on switches [26]. However, the purpose of consensus is to make sure the same value is delivered to all participants without any calculation in the switch to decide which value should be agreed upon. Instead, NetGVT performs the GVT calculation before returning it to servers. Further, differently from these works, NetGVT captures the notion of causality which is required for GVT computation.

Our work is also aligned with recent efforts that leverage the benefits of in-network computing to accelerate the processing of scientific workloads for high-performance computing. Kim et al. [25] presented NSinC, an architecture that provides a closed control-loop for in-network acceleration of scientific workloads and simulations. However, NSinC does not focus on synchronization but instead enables telemetry over scientific data using programmable data planes.

A related area of research is moving physical clock synchronization to data plane devices. HUYGENS [13] improves the synchronization of datacenter servers by moving it to the NIC. Also, DTP [29] improves the synchronization by moving it to the physical network layer. Further, DPTP [24] leverages high-resolution clocks available in programmable switching ASICs to respond to physical synchronization queries entirely in the data plane. Although keeping a reference to a real clock in the data plane improves accuracy, the clock skew of physical clocks would make it impossible to define precisely if an event happens before another. Instead, virtual clock synchronization ensures causal consistency, preserving the Lamport *happens-before* relation. We propose ways to offload virtual time synchronization using logical clocks to programmable switches to speed up distributed simulations.

6 CONCLUSIONS

Network programmability provides the opportunity for us to make distributed simulations faster by offloading virtual time synchronization functionality into the network hardware. In this paper, we proposed NetGVT, a system that offloads GVT synchronization into programmable switches. We presented our design and an evaluation of our prototype, and showed the scalability of our solution. Our results demonstrate that it is possible to offload the GVT computation to programmable switches, promoting reduced time to complete a simulation compared to techniques that do not rely on network programmability. As part of our future work, we intend to investigate how the placement of NetGVT switches in the topology impacts the overall GVT computation time to reduce even further the communication latency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Marios Kogias. We also thank Jennifer Rexford for her helpful input which significantly improved the quality of this paper. This work was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq (grant #311276/2021-0), FAPERGS (grant #19/2551-0001645-0) and FAPESP (grant #2020/05152-7 - PROFISSA, and grant #15/24494-8 - BigCloud).

REFERENCES

[1] 2022. NetGVT. https://github.com/RicardoParizotto/p4app-NetGVT.

[2] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient speculative parallelism for accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1247–1262.

[3] Theophilus A. Benson. 2019. In-Network Compute: Considered Armed and Dangerous. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. ACM, New York, NY, USA, 216–224. https://doi.org/10.1145/3317550.3321436

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[6] Joachim Breitner and Chris Smith. 2017. Lock-step simulation is child's play (experience report). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 1–15.

[7] Alisson V Brito, Harald Bucher, Helder Oliveira, Luis Feliphe S Costa, Oliver Sander, Elmar UK Melcher, and Juergen Becker. 2015. A distributed simulation platform using HLA for complex embedded systems design. In *2015 IEEE/ACM 19th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, 195–202.

[8] K. Mani Chandy and Jayadev Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* 5 (1979), 440–452.

[9] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking* (2020).

[10] Ali Eker, Barry Williams, Kenneth Chiu, and Dmitry Ponomarev. 2019. Controlled asynchronous GVT: accelerating parallel discrete event simulation on many-core clusters. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[11] Richard M. Fujimoto and Maria Hybinette. 1997. Computing Global Virtual Time in Shared-Memory Multiprocessors. *ACM Trans. Model. Comput. Simul.* (1997), 22 pages. https://doi.org/10.1145/268403.268404

[12] Richard M Fujimoto, Asad Waqar Malik, A Park, et al. 2010. Parallel and distributed simulation in the cloud. *SCS M&S Magazine* 3 (2010), 1–10.

[13] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. 2018. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 81–94.

[14] Elsa Gonsiorowski, Christopher Carothers, and Carl Tropper. 2012. Modeling large scale circuits using massively parallel discrete-event simulation. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. IEEE, 127–133.

[15] Jo Erskine Hannay and Tom van den Berg. 2017. The NATO MSG-136 reference architecture for M&S as a service. In *Proc. NATO Modelling and Simulation Group Symp. on M&S Technologies and Standards for Enabling Alliance Interoperability and Pervasive M&S Applications (STO-MP-MSG-149)*.

[16] Frederik Hauser, Marco Häberle, Daniel Merling, Steffen Lindner, Vladimir Gurevich, Florian Zeiger, Reinhard Frank, and Michael Menth. 2021. A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. *arXiv preprint arXiv:2101.10632* (2021).

[17] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. 2022. Modular Switch Programming Under Resource Constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 193–207.

[18] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 425–438.

[19] David R Jefferson. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1985).

[20] Theo Jepsen, Leandro Pacheco de Sousa, Masoud Moshref, Fernando Pedone, and Robert Soulé. 2018. Infinite Resources for Optimistic Concurrency Control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing (NetCompute '18)*.

[21] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. Netchain: Scale-free sub-rtt coordination. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 35–49.

[22] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM.

[23] Edimar Roque Martello Junior, Acacia Terra, Ricardo Parizotto, and Braulio Mello. 2020. Closing the Gap Between Lookahead and Checkpointing to Provide Hybrid Synchronization. In *Anais do XLVII Seminário Integrado de Software e Hardware*. SBC, 104–115.

[24] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. 2019. Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs. In *Proceedings of the 2019 ACM Symposium on SDN Research* (San Jose, CA, USA) *(SOSR '19)*. Association for Computing Machinery, New York, NY, USA, 8–20. https://doi.org/10.1145/3314148.3314353

[25] Daehyeok Kim, Ankush Jain, Zaoxing Liu, George Amvrosiadis, Damian Hazen, Bradley Settlemyer, and Vyas Sekar. 2020. Unleashing In-network Computing on Scientific Workloads. *arXiv preprint arXiv:2009.02457* (2020).

[26] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. 2021. RedPlane: enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 223–244.

[27] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3342195.3387545

[28] Carson Kuzniar, Miguel Neves, Vladimir Gurevich, and Israat Haque. 2022. IoT Device Fingerprinting on Commodity Switches. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*. 1–9. https://doi.org/10.1109/NOMS54207.2022.9789865

[29] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 454–467.

[30] Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: end-to-end network system evaluation with modular simulation. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 380–396.

[31] Jialin Li, Ellis Michael, and Dan RK Ports. 2017. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[32] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, Cosnard M. et al. (Ed.).

[33] F. Mattern. 1993. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *J. Parallel and Distrib. Comput.* 18, 4 (1993), 423–434. https://doi.org/10.1006/jpdc.1993.1075

[34] Paul Mazzurana. 2021. *Weather Research and Forecasting Model Workload Evaluation on IBM Cloud.* https://www.ibm.com/cloud/blog/weather-research-and-forecasting-model-workload-evaluation-on-ibm-cloud

[35] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. 2017. dist-gem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 153–162.

[36] Eiji Morinaga, Eiji Arai, and Hidefumi Wakamatsu. 2012. A Basic Study on Highly Distributed Production Scheduling. In *IFIP International Conference on Advances in Production Management Systems*. Springer, 638–645.

[37] Ranjit Noronha and Nael B Abu-Ghazaleh. 2002. Using programmable NICs for time-warp optimization. In *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 8–pp.

[38] Joshua Pelkey and George Riley. 2011. Distributed simulation with MPI in ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*. 410–414.

[39] Marcelo Pizzutti and Alberto E Schaeffer-Filho. 2019. Adaptive Multipath Routing based on Hybrid Data and Control Plane Operation. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*.

[40] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI)*.

[41] Juan L Sarli, Horacio P Leone, and Ma De los Milagros Gutiérrez. 2016. Ontology-based semantic model of supply chains for modeling and simulation in distributed environment. In *2016 Winter Simulation Conference (WSC)*. IEEE, 1182–1193.

[42] Anirudh Sivaraman, Mihai Budiu, Alvin Cheung, Changhoon Kim, Steve Licking, George Varghese, Hari Balakrishnan, Mohammad Alizadeh, and Nick McKeown. 2015. Packet transactions: A programming model for data-plane algorithms at hardware speed. *CoRR, vol. abs/1512.05023* (2015).

[43] Haoyu Song. 2013. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (Hong Kong, China) *(HotSDN '13)*. Association for Computing Machinery, New York, NY, USA, 127–132. https://doi.org/10.1145/2491185.2491190

[44] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[45] Barry Williams, Ali Eker, Kenneth Chiu, and Dmitry Ponomarev. 2021. High-performance pdes on manycore clusters. In *Proceedings of the 2021 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 153–164.

[46] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated service chaining on a single switch ASIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 141–149.

[47] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, Centralized Lock Management Using Programmable Switches. In *SIGCOMM*.