Efficient Non-Blocking Event Management for Speculative Parallel Discrete Event Simulation

Andrea Piccione
andrea.piccione@huawei.com
Huawei Munich Research Center
Munich, Germany

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it
Tor Vergata University of Rome
Rome, Italy

ABSTRACT

Parallel Discrete Event Simulation (PDES) is a modelling technique that takes advantage of concurrent computing resources. However, its asynchronous nature can present challenges for efficient execution. This paper proposes a new non-blocking management system for handling messages and anti-messages in Time Warp simulations. This approach exploits the benefits of non-blocking algorithms to surpass the limitations of existing blocking mechanisms, resulting in more efficient and scalable simulations. Specifically, the approach relies on efficient atomic fetch-and-add operations provided by modern computer architectures for evaluating and updating the status of the event.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; **Simulation environments**; *Massively parallel and high-performance simulations*; *Simulation theory*.

KEYWORDS

Time Warp, Event Set, Non-blocking Algorithms, Synchronization

ACM Reference Format:
Andrea Piccione and Alessandro Pellegrini. 2024. Efficient Non-Blocking Event Management for Speculative Parallel Discrete Event Simulation. In *38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS '24), June 24–26, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3615979.3656053

1 INTRODUCTION

Parallel Discrete Event Simulation (PDES) is a modelling technique specifically designed to take advantage of concurrent computing resources. Despite its potential to speed up simulation, the asynchronous nature of PDES presents both difficulties and possibilities for efficient execution. Time Warp [6] is one of the many synchronisation methods available to manage this asynchronicity and has gained considerable attention.

Time Warp executes models speculatively, permitting logical processes (LPs) to optimistically advance their local simulation time without waiting for other concurrent processes to reach a synchronised state. This strategy offers the potential for increased parallelism, but also introduces the problem of *causality errors*, where events are processed out of their correct causality order. To resolve such issues, Time Warp incorporates the rollback mechanism to restore (locally) the simulation to a previous consistent state from which to restart execution. Key to this mechanism is the management of events and their counterparts, known as *anti-events*.

Their management is central to ensure both the *correctness* and *efficiency* of a Time Warp-based simulation, as it typically belongs to the critical path of the simulation main loop. In fact, poor event management may increase the rollback frequency as a secondary effect of increased clock skew between different LPs, negating the benefits of parallel execution and significantly increasing computational overhead. Previous works [16] have already observed this phenomenon, showing that optimising event management can result in up to 30% improvement in simulation efficiency and can avoid severe rollback explosions.

In this paper, we propose a simple yet effective strategy to efficiently manage messages and anti-messages in a Time-Warp simulation runtime environment. Our proposal is essentially a non-blocking algorithm that allows concurrent worker threads to advance the simulation and maintain a consistent view over event processing, also when event queues are shared between the different threads. Relying on non-blocking synchronisation can efficiently exploit the computing power of systems with a high degree of parallelism, like those often used for PDES, especially when dealing with traditional data structures used to support simulation [3].

Our non-blocking algorithm is based on a per-event finite-state machine that describes event "history" since their creation. Because the state machine is part of the event metadata, it can be managed by multiple worker threads, and it can also be transmitted over the network in distributed simulations, or it can be moved across different data structures locally, e.g. if future and past event sets are maintained in different data structures for performance reasons [11]. Since the state of the event can be represented using few bits, the state machine can be manipulated using efficient read-modify-write instructions. In this way, no explicit synchronisation is required, and the relevant aspects of the life of an event can be traced globally.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. Our non-blocking event management scheme is presented in Section 3. The results of our experimental evaluation are shown in Section 4.

2 RELATED WORK

The management of messages and antimessages in Time Warp simulations is a core operation, which has been extensively studied in the literature to improve the performance of simulations.

The work in [1] discusses a protocol for speculative parallel discrete event simulation designed to manage events efficiently

SIGSIM PADS '24, June 24–26, 2024, Atlanta, GA, USA

without blocking, using local rollback mechanisms to handle inaccuracies. Although the high-level idea resembles what we are proposing in this paper, it focusses only on deterministic models, where potential message arrival times are predictable. Conversely, our state-machine-based approach adheres to the more general Time Warp protocol and can recover any model from possible out-of-order executions.

SPEEDES' approach to speculative parallel discrete event simulation [16] focusses on memory optimisation and the efficiency of event processing. Memory efficiency is enhanced by using free lists for memory management and the adoption of selective state changes to minimise unnecessary memory operations. Events are also augmented with information that describes the state changes performed by their execution. The Delta Exchange mechanism facilitates rapid rollback capabilities, similar in spirit to incremental checkpointing [10]. Additionally, event list management employs two distinct lists: one sorted and the other unsorted. This dual-list system aims to enhance performance by reducing the sorting overhead, with a reported superlinear speedup thanks to this approach. Similarly to our proposal, performance improvements in [16] related to event management can be based on differentiated tasks carried out to manage event reception and processing. However, we target operations completely different from [16].

Lazy cancellation and lazy re-evaluation [18] are techniques that try to reduce the impact of cascading rollbacks and event reprocessing, respectively. In both approaches, it is necessary to augment the data structure representing an event implementing explicit state machines, similarly to what we do. For example, in lazy cancellation [15], antimessages are sent out only if an output message reproduced after rollback is different from the previously-sent version. To avoid clock skew, special blocking events are sent upon a rollback, to temporarily mark previously sent output messages as blocked until the sender LP has decided whether or not they are correct. In lazy re-evaluation [8], previously-computed states are reused whenever possible. In particular, if the reprocessing of a straggler does not change the simulation trajectory, the execution can be jumped forward. To enhance performance, messages could be organised in trees and augmented with flags that describe whether they produced a state change. These metadata allow to avoid costly state comparisons. In addition, events could be used to maintain metadata describing hit/miss ratio of lazy cancellation [13, 14], enabling autonomic switching between the two.

3 NON-BLOCKING EVENT MANAGEMENT

In Time Warp, *positive events* (e^+) are coupled with their negative counterparts, called *anti-events* (e^-). The purpose of anti-events is to annihilate positive events that have been spread around the simulation, possibly having also already been processed. We consider an event $\hat{e} = (e^+, e^-)$ as the logical union[1] of a positive event e^+ and a negative event e^-. An event $\hat{e}$ conceptualises the idea that, in general, positive and negative events can be received in any order and that, if both positive and negative appear in the same node, $\hat{e}$ annihilates itself. The effect of this annihilation could be the execution of a rollback operation, which could in turn send out additional anti-events.

[1]This notation recalls the concept of zsets in [7].

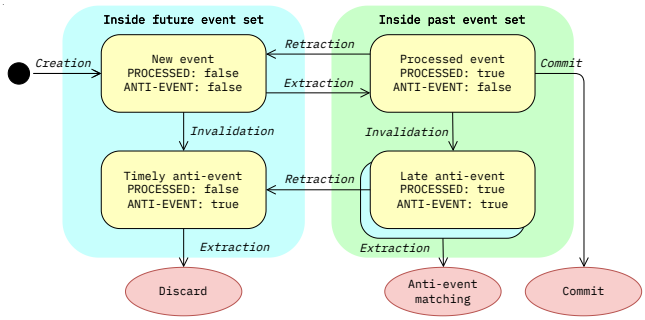

Figure 1: Non-blocking anti events

From a practical point of view, we can capture the lifecycle of an event $\hat{e}$ as a set of different stages, as follows:

- creation of the positive event e^+ by the source LP and insertion into the future event set of the destination LP;
- extraction, processing by a destination LP, and insertion into the past event set.
- removal from the past event set as part of a rollback operation initiated by the receipt of a straggler message $e_s^+ \neq e^+$, which causes the reinsertion of e^+ into the future event set;
- annihilation, leading to its removal from either the future event set (if still in transit) or the past event set, with a rollback operation if necessary.
- commitment, if the event falls beyond the global virtual time's commitment horizon.

Each of these stages may be considered as an atomic operation and implemented as such, in the sense that no transient states are ever observed; for example, an event extracted by a processing thread would be processed and inserted into the past event set without any visible interruption.

As a result, at any given moment, an *active event*, i.e. an event that has not yet reached a terminal state such as annihilation or commitment, can either be in the past or the future event set; at the same time, it can either be valid or invalidated. The logical superposition of these four different states gives rise to the finite-state machine depicted in Figure 1, which can be represented using two flags packed in the same word, which we call the *processed flag* and the *anti-event flag*, respectively.

This representation enables the use of efficient atomic *fetch-and-add* operations offered by modern computer architectures to evaluate and update the status of the event. Fetch-and-add is a Read-Modify-Write operation, typically based on single machine instructions, that performs two fundamental actions atomically: it reads the current value from a memory location and then adds a given value to it, subsequently storing the result back in the same memory location. The atomicity of this operation ensures that no other operations can intervene between the fetch and the add, thus preventing race conditions.

When an event is created, it is not subject to concurrent access by other threads, so both flags can be safely initialised to false. Then the events can be regularly inserted in the future event set of the destination LP. The way to correctly carry out this latter operation depends on the data structure being used for the set and on whether

it is private (e.g., per LP) or shared (e.g., per thread). Conversely, when a thread extracts an event from the future event set, it atomically retrieves the two flags of the message while marking the event as processed[2]. In this way, the thread acquires ownership of the event. The need for an atomic acquire operation lies in the possibility that multiple threads can compete to acquire the event to schedule an LP, for example, in scenarios where there is a loose binding between threads and LPs (see, e.g., [5]).

More importantly, in our proposal, we consider the possibility that the same event is placed in both future and past event sets *at the same time*. The reason for this strategy is that some housekeeping operations, which are executed on the critical path of the simulation, may be too costly if executed all at once; therefore, it makes sense to spread different parts of the operation across different worker threads—this approach recalls the grounding principles of flat combining [4]. One worker thread will therefore perform part of the housekeeping operation, e.g. updating the state machine of the event and moving it to the future event set, while another one will finalise the operation, e.g. ordering the event in the future event set to prepare for the next event extraction.

Therefore, there are three possible scenarios to handle, based on the value of the flags that is atomically fetched (i.e., the old value before the fetch-and-add is completed):

- if the processed flag is false and the anti-event flag is false, then the event is a fresh one, so it is processed normally;
- if the processed flag is false and the anti-event flag is true, the event has been marked as inconsistent by another thread by sending the negative counterpart e^- that was received before processing e^+, so it is safely deallocated;
- if the processed flag is true and the anti-event flag is true, we incurred in an inconsistent event that this LP already processed, so it is handled using the standard Time Warp logic, namely searching for the matching event in the past event set and performing a rollback.

Similarly, when a thread has to mark an event as invalid, it atomically retrieves the two flags of the message while marking the event as inconsistent. An already invalidated message cannot be again invalidated, so there are only two possible scenarios:

- if the processed flag is false and the event is still in the future event set, no further operations are needed: when a processing thread will extract it, it will directly discard it;
- if the processed flag is true and the event has been already processed, it is reinserted into the future event set so that it will be processed as a regular anti-event when a processing thread extracts it.

In addition, when a thread has to rollback a logical process to a previous consistent state, it will remove from the past event set the events that occurred between the current LP state and the received straggler/anti-event. Some of these events may still be valid, while others may have been marked invalid by another concurrent rollback operation. To efficiently distinguish the two conditions, the two flags of the message can be atomically retrieved while unmarking the event as processed. The processed flag had already been set after the extraction, so here are the two possible scenarios to deal with:

- if the anti-event flag is false, the event is still valid and is placed back into the future event set;
- if the anti-event flag is true, the event has been marked as invalid, but the invalidating thread has already put it back in the future event set, so deallocation is not necessary—indeed, it would be incorrect, because there is another reference to the event in the future event set.

When the event overcomes the commitment horizon, i.e. its associated timestamp falls behind the value of the Global Virtual Time (GVT), it can be safely deallocated. The commit operation does not need to handle concurrency, because for a correctly computed GVT, no thread will ever mark a committed event as invalid.

4 EXPERIMENTAL ASSESSMENT

We have implemented our proposal in the ROme OpTimistic Simulator (ROOT-Sim) [9]. We carried out our experimental evaluation on a server equipped with an Intel Xeon 2699v4 CPU with 128 GB of memory. Each data point is the average of 15 runs. As the simulation model used to benchmark our proposal, we have used the traditional PHOld [2] benchmark and a variant, named PHold Memory, originally proposed in [17], which simulates a set of LPs that interact through events distributed randomly on the set of LPs. Events' timestamps are drawn according to an exponential distribution, thus implementing Poisson processes. The variant we have used entails reading/writing to memory buffers of variable size kept in the LP's simulation state. Buffers are also transferred between LPs, as payloads of exchanged events. Both models have been run with 10,000 LPs.

In Figure 2, we provide the results when running the PHold Memory benchmark. In our configuration, we have used a dynamic array for the past event set, and we have experimented with two different data structures to represent the future event set, namely a heap and a skip list [12].

Beyond the non-blocking management presented in this paper, we have also used two different scenarios for comparison: in one scenario, the worker thread immediately discards annihilated events (we refer to this configuration as eager deletion in the plots), while in the other the deletion of the annihilated event is performed lazily, only when the event is extracted again (lazy deletion in the plots).

We note that eager deletion from the skiplist is an operation with logarithmic asymptotic cost, while, for the binary heap, the same operation has a worse linear bound.

From the results, we observe that lazy deletion provides the worst performance in the simulation. This is true for both skip-list and heap-based configurations. Indeed, in this model configuration, as indicated by the efficiency plots, we have a sizeable number of generated anti-events, which, for the lazy deletion strategy, tends to saturate the future event set.

Eager deletion, by preemptively removing annihilated events from the future event set, keeps the size of the future event set under control, avoiding the aforementioned phenomenon.

[2]The event is marked as processed *before* its processing is completed in wall-clock time. This is in accordance with the event being an atomic unit of execution in Time Warp.

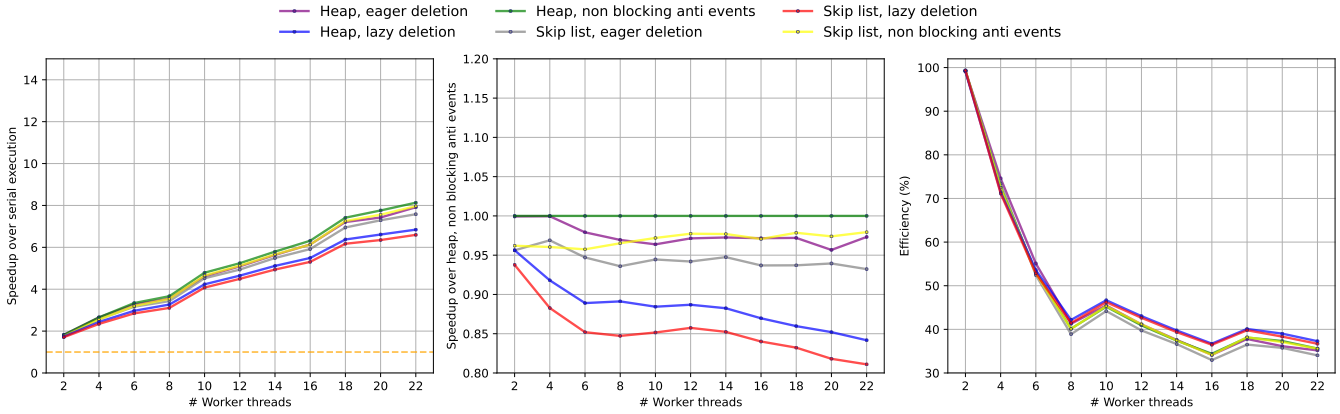

Figure 2: Results for PHold memory

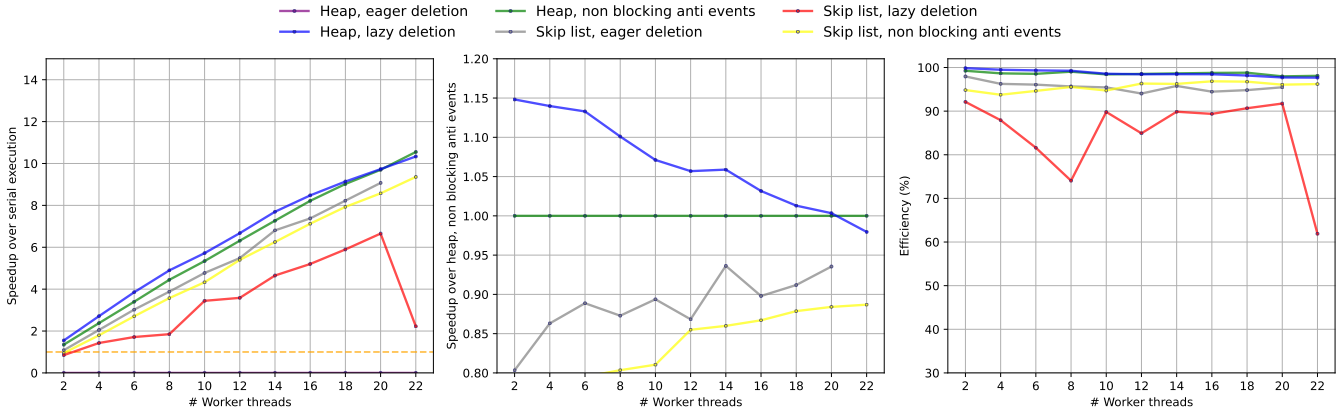

Figure 3: Results for PHold

When relying on a heap to maintain the future event set, the performance is generally higher than in the case of a skip list. Nevertheless, if the skip list is coupled with our non-blocking management, it outperforms the heap, also when eager deletion is employed.

Our proposal can contain the size of the future event set like the eager deletion, but without incurring the cost of explicit removal, similarly to the lazy approach. The impact of our proposal is larger for hard-to-parallelise models, such as the one used in this evaluation; with 22 threads, the maximum speedup achieved is ~ 8.

Lazy deletion achieves greater efficiency because the higher cost of future event set management is effectively *throttling* the execution of events. In friendlier scenarios, like the one shown in Figure 3, heap lazy deletion is competitive, outperforming our proposal by 15% with 2 threads. The gap narrows down with higher thread counts, because anti-messages increase in number and, as mentioned, in these contended scenarios lazy deletion does worse.

On the other hand, skip-list lazy deletion does not provide good performance. We speculate this is due to lazy deletion increasing the size of the future event set. Apparently, the skip list reacts to this size increase with a larger variance in queue management costs, resulting in a higher rollback probability, as shown by the efficiency.

Using vanilla PHold, heap eager deletion fails to provide any acceleration compared to serial execution. Specifically, our experiments timed out with durations exceeding 5 times that of serial execution. Other missing data points in the plot are a result of this time-out choice. Skip-list eager deletion, because of better asymptotic bounds, is still able to scale reasonably.

Overall, our proposal behaves competitively, providing stable performance in both proposed evaluation scenarios. In general, we observe that anti-events management strategies have a difficult to predict effect on final performance.

5 CONCLUSIONS

In this paper, we have presented a simple yet efficient way to handle the lifecycle of events in a Time Warp simulator. By relying on atomic fetch-and-add instructions, multiple worker threads can query and update a finite state machine that describes the current evolution of an event $\hat{e}$ that can travel back and forth across different queues representing the future and the past event set. The results have shown that the simple introduction of non-blocking management of events can produce a non-minimal speedup over other common strategies.

REFERENCES

[1] Phillip M Dickens and Paul F Reynolds, Jr. 1990. SRADS with Local Rollback. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation, San Diego, CA, USA, 161–164.

[2] Richard M Fujimoto. 1990. Performance of Time Warp Under Synthetic Workloads. In *Distributed Simulation (PADS '90)*, David Nicol (Ed.). Society for Computer Simulation International, San Diego, CA, USA, 23–28.

[3] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. *SIGPLAN notices* 50, 8 (Dec. 2015), 1–10. https://doi.org/10.1145/2858788.2688501

[4] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM, New York, NY, USA, 355. https://doi.org/10.1145/1810479.1810540

[5] Mauro Ianni, Romolo Marotta, Davide Cingolani, Alessandro Pellegrini, and Francesco Quaglia. 2018. The Ultimate Share-Everything PDES System. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '18)*. ACM, New York, NY, USA, 73–84. https://doi.org/10.1145/3200921.3200931

[6] David R Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404–425. https://doi.org/10.1145/3916.3988

[7] David R Jefferson and Peter D Barnes. 2022. Virtual Time III, Part 1: Unified Virtual Time Synchronization for Parallel Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation* 32, 4 (Sept. 2022), 1–29. https://doi.org/10.1145/3505248

[8] A C Palaniswamy, S Aji, and P A Wilsey. 2003. An efficient implementation of lazy reevaluation. In *Proceedings of the 25th Annual Simulation Symposium*. IEEE Comput. Soc. Press, Washington, DC, USA, 140–146. https://doi.org/10.1109/simsym.1992.227568

[9] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The ROme OpTimistic Simulator: Core Internals and Programming Model. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS)*. ICST, Brussels, Belgium, 96–98. https://doi.org/10.4108/icst.simutools.2011.245551

[10] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2015. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems* 26 (2015), 1560–1569. https://doi.org/10.1109/TPDS.2014.2323967

[11] Andrea Piccione. 2022. Comparing Different Event Set Management Strategies in Speculative PDES. In *Proceedings of the 2022 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS '22)*. ACM, New York, NY, USA, 55–56. https://doi.org/10.1145/3518997.3534993

[12] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33 (1990), 668–676. https://doi.org/10.1145/78973.78977

[13] Raghunandan Rajan, Radharamanan Radhakrishnan, and Philip A Wilsey. 1999. Dynamic Cancellation: Selecting Time Warp cancellation strategies at runtime. *VLSI design* 9, 3 (Jan. 1999), 237–251. https://doi.org/10.1155/1999/61087

[14] R Rajan and P A Wilsey. 2002. Dynamically switching between lazy and aggressive cancellation in a Time Warp parallel simulator. In *Proceedings of the 28th Annual Symposium on Simulation (SIMSYM '02)*. IEEE Comput. Soc. Press, Washington, DC, USA, 22–30. https://doi.org/10.1109/simsym.1995.393597

[15] Hussam M Soliman Ramadan. 2008. Throttled lazy cancellation in time Warp parallel simulation. *Simulation* 84, 2-3 (Feb. 2008), 149–160. https://doi.org/10.1177/0037549708090864

[16] Jeffrey S Steinman. 1991. SPEEDES: A Unified Approach to Parallel Simulation. In *Advances in Parallel and Distributed Simulation (PADS '91)*, Vijay K Madisetti, David Nicol, and Richard M Fujimoto (Eds.). Society for Computer Simulation, San Diego, CA, USA, 1111–1115.

[17] Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia. 2009. Benchmarking Memory Management Capabilities within ROOT-Sim. In *Proceedings of the 13th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. IEEE, Piscataway, NJ, USA, 33–40. https://doi.org/10.1109/DS-RT.2009.15

[18] Darrin West. 1988. *Optimising Time Warp: Lazy Rollback and Lazy Reevaluation*. Ph. D. Dissertation. University of Calgary, Calgary, AB, Canada.