A comparative study of pending event set implementations for PDEVS simulation

Romain Franceschini
University of Corsica
UMR SPE 6134 CNRS
UMS Stella Mare 3460
r.franceschini@univ-corse.fr

Paul-Antoine Bisgambiglia
University of Corsica
UMR SPE 6134 CNRS
UMS Stella Mare 3460
pa.bisgambiglia@univ-corse.fr

Paul Bisgambiglia
University of Corsica
UMR SPE 6134 CNRS
UMS Stella Mare 3460
bisgambi@univ-corse.fr

ABSTRACT

The choice of a particular event-list implementation can dramatically improve or reduce performance of a discrete event simulation (DES). For more than 40 years, several data structures had been proposed to address this problem. We present new empirical results using the parallel discrete event system specification (PDEVS) formalism and a DEVStone benchmark. Similar analyzes were previously conducted, the last one being published in 2007. This paper includes most recent proposals, particularly the LadderQueue [19], evaluated using the DEVS-Ruby simulator.

Author Keywords

Discrete event simulation, DEVS, PDEVS, DEVS-Ruby, Pending event set, PES, Future event set, DEVStone, benchmark, comparison, performance analysis

ACM Classification Keywords

I.6.7 SIMULATION AND MODELING: Simulation Support Systems — Environments; I.6.8 SIMULATION AND MODELING: Types of Simulation — Discrete event

INTRODUCTION

Discrete event systems (DES) represent many technological and engineering systems such as communication networks, computer networks, manufacturing systems, transportation systems, natural systems, and more. DES are driven by events. These events have to be scheduled and sorted by execution time. Large-scale models can be long to simulate because there are many events to manage. Among the acceleration methods (parallelization, etc.), we suggest to study data structures to hold scheduled events. Our interest focuses on DEVS (Discrete EVent system Specification) [22], which is a formalism for modeling systems based on the DES theory [5]. A Discrete Event System (DES) is a discrete-state, event driven dynamical system in which the state space is described by a discrete set, and states evolve in terms of asynchronous occurrence of discrete events over time. In DES theory, states, events and transition functions are defined by the following tuple: $M = \langle S, s_0, \lambda, \delta, \Sigma \rangle$ where: S is the set of states; Σ is the set of events containing detectable (i.e. an event is detectable if it produces a measurable change in the system output) and undetectable events, which are generally fired asynchronously; $\delta : S \times \Sigma \rightarrow S$ the state transition function; $\lambda : S \times \Sigma \rightarrow \Sigma_d$ the output function, where Σ_d and Σ_{ud} the set of detectable and undetectable events, respectively $\Sigma = \Sigma_d \cup \Sigma_{ud}$; and s_0 is the initial state vector. Many modeling approaches of DES have been proposed and developed, including the DEVS formalism.

The DEVS formalism allows composition of reusable models. It is an open, flexible formalism with a great capacity for extension. Recent studies have shown that DEVS may be considered as a multi-formalism because it allows encapsulation of other modeling formalisms. This capacity for opening and extension is very interesting, as the representation of the various entities which constitute a complex system can be accomplished by using the most appropriate methods. In DEVS, the number of models and the complexity induced by their couplings are an issue in terms of performance, particularly because of the events exchanged. Generally, the way events are handled in DES can dramatically improve or reduce simulation performance. This problem is commonly known as the pending event set problem (PES).

The pending event set (PES) problem is one of the oldest and most important problems in the field of discrete event simulation [10]. A great research effort was made during the last 40 years to find the most appropriate data structures to hold pending events, through comparative studies and new original data structures claiming amortized constant time complexity. Several interesting studies comparing priority queue data structures available were already conducted, the first being published in [20]. Other comparisons appeared as new algorithms were proposed [12, 11, 13, 15] and the most recent study [10] was for the first time realized with the PDEVS formalism in mind.

In this paper, we present new experimental performance results using two methods. An experiment model measuring performance of queue operations inspired by a PDEVS simulation and real PDEVS-based simulation using the DEVStone [21] benchmark and the DEVS-Ruby simulator [8]. We include most recent proposals, particularly the LadderQueue [19].

The remainder of this paper is structured as follows. Section 2 describes the pending event set problem and several priority queue data structures suitable for implementing the pending event set. Section 3 presents performance measurement techniques and tools used to generate experimental results. Conclusions are found in section 4.

TMS/DEVS 2015 April 12-15, 2015, Alexandria, VA

THE PENDING EVENT SET

In this section, we describe the pending event set problem in discrete event simulation and we give an overview of priority queues for implementing the pending event set.

Problematic

In discrete event simulations, a system is represented by one or several entities influencing at each other through messages, or scheduled events. The current simulation time of the system depends on those events. As the simulation advances, the clock is skipped to the timestamp of the next scheduled event to occur. The pending event set (PES) represents the set of all future events to be evaluated, some of these pending as a result of previously simulated events. Regardless of the point in time at which events are scheduled, they are executed in a strict order of precedence. The way the pending event set performs these insertion and deletion operations has a crucial impact on the overall simulation execution time, especially as systems become more complex.

Typically, the future event set is represented as a priority queue data structure with two basic operations: *enqueue* and *dequeue*. The first inserts an event to the set and the latter remove and returns the event with the highest priority (the lower the timestamp is, the higher is the priority). Note that if several events scheduled at the same time, all are removed. But several discrete event simulation systems such as the DEVS formalism requires another important operation: *delete*. Indeed, if the execution of one event can involve the scheduling of any number of events, it also can lead to the cancellation of a queued pending event, hence the *delete* operation.

For better understanding, consider how PDEVS models are simulated. In PDEVS, models report the next timestamp at which their δ_{int} internal transition should be activated through the ta (time advance) function. But before executing the internal transition, the simulator will give a chance to the model to throw some output to its influencees by executing the λ function. When the influenced model receives input through the δ_{ext} external transition, it leads to a new state and maybe a new timestamp of activation and thus, a new call of the ta function. If a model was scheduled for the current simulation time t its the δ_{con} confluent transition that is activated (which default behavior is to activate δ_{int} and δ_{ext} sequentially) to give a chance to the model to handle the conflict. Executing a δ transition (δ_{int} or δ_{ext} or δ_{con}) function at time t will result in a change of state from S_t to S_{t+ta}. Because a δ_{ext} and a δ_{con} transition can update the timestamp of next activation of a model while it was already scheduled (at t or further in the future), the model need to be *adjusted* (or deleted and re-inserted) in the event set.

Among a wide range of general-purpose priority queue algorithms available, many recent publications proposed pending event set algorithms alternatives for discrete event simulation. The next subsection gives an overview of commonly used priority queues along with these new suggested algorithms.

Priority queue data structures

A priority queue differs from a regular queue in that each element is ordered by its associated priority. They are used in a wide range of applications such as bandwidth management, operating systems, data compression and discrete event simulation. There are a variety of ways to implement a priority queue, more or less naively. Here we present the non exhaustive list of priority queue data structures that we implemented in DEVS-Ruby [8]. Table shows operations complexity in big O notation for each structure. We specify expected and worst amortized performance if available or worst case performance in each column. Excepted for the most inefficient ones based on simple lists, they fall in two different groups: tree based or multi-list based algorithms. The first includes priority queues based on Binary Heap, Fibonacci Heap, Pairing Heap, Splay Trees [18] and the latter includes Calendar Queue [4], Lazy Queue [16], Ladder Queue [19]. Note that this study is focused on sequential pending event set algorithms and thus, do not consider parallel variants such as [7] or [9] which improves performance of their counterparts.

Structure	enqueue	dequeue	delete
MinimalList	O(1)	O(n)	O(n)
SortedList	O(n log(n)), O(n^2)	O(1)	O(log(n))
BinaryHeap	O(1), O(log(n))	O(log(n))	O(log(n)), O(n)
SplayTree	O(log(n))	O(1), O(log(n))	O(log(n)), O(n)
CalendarQueue	O(1), O(n)	O(1), O(n)	O(1), O(n)
LazyQueue	O(1), O(n)	O(1), O(n)	O(1), O(n)
LadderQueue	O(1), O(n)	O(1), O(n)	O(1), O(n)

Table 1. Worst case or expected and worst amortized performance of listed priority queues in Big O notation.

Simple priority queues

Sorted List

Elements in the sorted list are kept sorted. When *enqueue*, the element is appended is a sorted fashion. Depending on the sorting algorithm, this operation can be done in O(n log(n)). The *dequeue* operation is constant and the *delete* operation can be done in O(log(n)) using a binary search algorithm, since the list is always sorted.

Minimal List

The minimal list is a simple list where elements are kept unsorted. The element with the highest priority is cached. The *dequeue* operation has extremely bad performances (O(n) complexity) because the entire list has to be iterated in search of the new highest priority.

Tree oriented priority queues

Binary Heap

A binary heap is binary tree structure which satisfies the heap ordering property. The element with the highest priority is always stored at the root. It is a common implementation of a priority queue that use only a single list as an internal representation. Its performance offers a O(log(n)) complexity for all operations.

Splay Tree

The Splay Tree is a self-adjusting balanced binary search tree invented by D. Sleator and R. Tarjan [18] back in 1985. It is essentially a binary tree with particular access and update rules that allows recently accessed elements being quick to find. Common operations are combined with the *splay* operation, which consist in placing a targeted element to the root of the tree by re-arranging the tree while maintaining elements ordered. The shape of splay trees is not constrained, and varies based on what lookups are performed unlike other trees such as the AVL tree, which structure is constrained at all times so that the height of the tree never exceeds O(log n).

Multi-list oriented priority queues

Calendar Queue

Suggested in 1988 by Brown [4], the Calendar Queue was not the first multi-list structure to appear. The idea of dividing the queue elements appeared with the Two List [3], which holds elements with highest priority into a short sorted list and keeps elements with lower priority into an unsorted list. The Calendar Queue was inspired by the way humans solves the problem of ordering a future event set with desk calendars. It is essentially composed of an array of buckets, each bucket containing an ordered list of events that fall within a particular time gap. The time interval associated to each bucket represent a day whereas the array of total buckets represent the time interval of a year. Figure 1 shows its structure. During *enqueue*, an index indicating which bucket the event belongs to is computed. To *dequeue* an event, the bucket from which the last event was dequeued is checked to find the next event with the highest priority for the current year. If the bucket is empty or full of events scheduled for future years, next buckets are checked until the highest priority is found. If no more event belongs to the current year, the next year starts and the next event is re-searched. The calendar queue try to keep a reasonable size for its buckets. When needed, that is if the queue size exceeds an upper threshold or falls below a lower threshold, a resize operation is triggered. The resize operation double or halve the number of days and re-order all elements. This operation has high performance costs (O(n) complexity), but is amortized over time. The main disadvantage of this queue is its sensitivity to skewness and peaks in priority distribution.

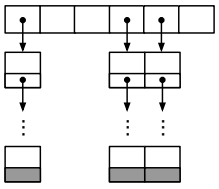

Figure 1. Representation of the calendar queue structure.

Ladder Queue

The Ladder Queue [19] is the most recent, especially designed priority queue structure for managing the pending event set that achieves O(1) amortized performance. Its name was inspired from its multi-list structure recalling a ladder and its rungs. The structure divide elements into three parts:

1. *Bottom*, a list which is kept sorted and holds all events scheduled to a near future.
2. *Ladder*, the middle layer which consists in several *rungs* of buckets where each bucket contains an unsorted list. Events in this tier of the queue are considered scheduled to a far future. They are not sorted but grouped with each other based on the time-gap.
3. *Top*, a simple unsorted list which serves as a buffer for events scheduled into a very far future.

It strength comes from two ideas: (1) the event sorting process is deferred until absolutely necessary and (2), it adapts its structure to event distribution. Events are sorted only when they are close to being dequeued, when transferred to *Bottom*. This means that most events are simply either appended to the *Top* structure or appended into buckets of the *Ladder* structure without sorting. The number of rungs in the *Ladder* tier varies depending on event distribution, each rung having a different granularity of inter-event time-gap. This way, the queue is much less sensitive to skewness or peaks in the priority distribution. Figure 2 shows a representation of its structure.

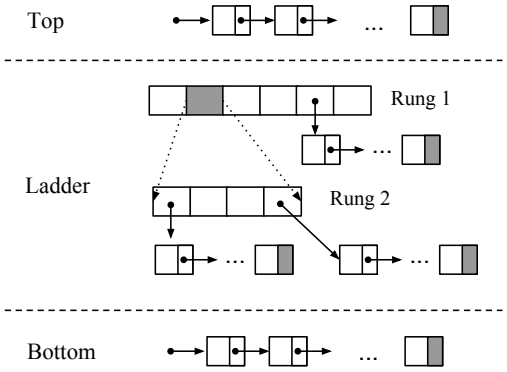

Figure 2. Representation of the ladder queue structure.

Another similar structure can be found in the literature, named the **Lazy Queue** [16]. It is also organized as three tiers, holding events in a similar fashion but the middle tier is simpler and algorithms are different. [15] showed that it performs similarly or worse (depending on the distribution) than the Calendar Queue, so we will not include it in our performance analysis.

COMPARISON

In this section we present results of the performance measurements of each PES implementation. A first subsection details our methodology and test environment while a second subsection present the actual results.

Methodology

To study a priority queue in the context of the PDEVS formalism, it is important to find a method which covers as closely as possible all practical aspects of a discrete event simulation. If we refer to the most used methods in the literature for similar comparative studies we count several approaches:

1. the *hold model* introduced in 1975 [20] and completed by [11] take a basic approach. Each event processed leads to

only one event being scheduled. With an initially randomly populated event queue, an event is dequeued followed by an enqueue of a new event until reach a given number of steps.

2. the *up/down* model suggested by [16] takes a different approach. Events are enqueued until reach a given queue size and then dequeued until the queue is empty.

3. a variant of the hold model imitating a PDEVS simulation, proposed by [10] consist in dequeuing all events scheduled for the same time at once instead of dequeuing a unique event. Then, an equal number of new events will be enqueued. Finally, a random number (between 0 and 50) of events in the queue are *adjusted* (their priority is updated to a new one). These operations are repeated until a given number of steps.

With the hold model, the size of the queue remains the same during the complete test. Conversely, the up/down model grows and shrinks at each step of the test. Theoretically, the static nature of the hold model is not an issue for PDEVS simulation as there is always one event per model. But one model can be scheduled to occur at infinity, and as some structures such as the calendar queue or the ladder queue are disturbed by an infinite priority, we don't add them to the PES. Furthermore, if the model structure is static in PDEVS, some extensions (such as [2]) allows a dynamic structure. Consequently, the queue size can change in a PDEVS simulation depending on the implementation. This does not mean the up/down model is ideal either because as we said in section , PDEVS requires a PES that support *adjusting* an event and both the hold model and the up/down model are not considering it. The latter approach, however, does and is the closest to a real PDEVS based simulation as Himmelspach [10] concludes. From the three methods presented, we use this one, along with an original new one, based on DEVStone.

DEVStone

DEVStone [21] is a generic benchmark used to study the performance of DEVS and PDEVS based simulators. It is able to generate automatically a suite of models with varied structure and behavior. It is designed to evaluate efficiency of several (P)DEVS simulation engines and different software versions of the same simulator. We propose to present the results obtained with DEVStone to compare the performances of all pending event set. We measure elapsed wall clock time in seconds for a simulation involving a DEVStone suite of models. Events traverse all models of the generated structure with a fixed *depth* (number of nested coupled models in the hierarchy) of 2, *HI* coupling type (a type of models interconnections defined in [21]) and δ transitions times set to 0 seconds. The varying parameter is the *width* (number of components in each coupled model).

Distribution

Among the variables that can affect the performance of a priority queue, the initial and increment priority distribution is a crucial one. We had to adapt DEVStone so it could support different distributions. Given as a parameter, the distribution is used to compute the return value of the time advance (ta) function of all generated atomic models. Table 2 lists the different distributions we used in our performance analysis and that are commonly used in the literature to compare priority queues with the expression used to compute the random value. The differences perceptible between two distributions shape are important because a particular skewness or peak can expose a weakness in a priority queue implementation.

Distribution	**Expression**
Constant	1
Uniform (0, 1)	$rand$
Uniform (0.9, 1.1)	$0.9 + 0.2 * rand$
Exponential	$-ln(rand)$
Triangular (0, 1.5)	$1.5 * sqrt(rand)$
Neg Triangular (0, 1000)	$1000 * (1 - sqrt(1 - rand)$
Bimodal	$0.95238 * rand + rand < 0.1$? 9.5238 : 0
Camel (2, 0.8, 0.2)	cf. [16]

Table 2. Priority increment distributions.

Experimental framework

The environment used to run our benchmarks is based on an Intel(R) Core(TM) i5-3360M CPU @ 2.80GHz (3MB L2 cache), 16 GB (2 x DDR3 - 1600 MHz) of RAM, a Toshiba MK5061GS hard drive, running on Ubuntu 14.04 (64bit). Regarding software, the DEVS-Ruby simulator in its 0.6 version running on the official Ruby 2.2.0 VM. DEVS-Ruby [8] is a library developed at University of Corsica that allows formal specifications of classical DEVS and parallel DEVS models.

For our performance analysis, we ran two different benchmarks. In the first, we run a DEVStone simulation for each priority queue listed in section and for each of the eight distribution. We measure wall clock time in seconds with a varying width and a fixed depth of 2, HI coupling type and δ transition times set to 0 seconds. The second benchmark is based on the method suggested by [10].

Performance analysis

DEVStone experiment

Unsurprisingly, the results of the two most naive implementations (Figures 3 and 4) are far behind other tree and multi-list based queues. The sorted list forms the performance lower bound. We should emphasize that it could perform better if we used a more stable sorting algorithm. Our implementation use a quick sort algorithm, which is known to have a worst case performance of O(n^2) especially when the elements are almost sorted, but it could never exceed O(n log(n)) a performance. The minimal list have slight better results than the sorted list but remains extremely inefficient. It worth noted that concerning the constant distribution, which represent a discrete time simulation (which is a special case of DES), the minimal list does not offer the worst results.

At first sight, other queues (Figures 5, 6, 8 and 7) seems to be on an equal footing although the advantage goes to the multi-list queues family. The binary heap (Figure 6) is slightly better than the splay tree (Figure 5). Between the ladder queue (Figure 7) and the calendar queue (Figure 5), the ladder queue is fastest. If our DEVStone benchmark offers a

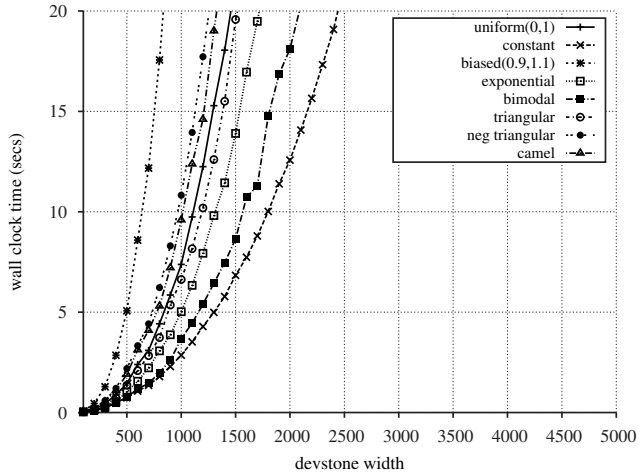

Figure 3. Wall clock time of a DEVStone simulation using the Minimal List based PES.

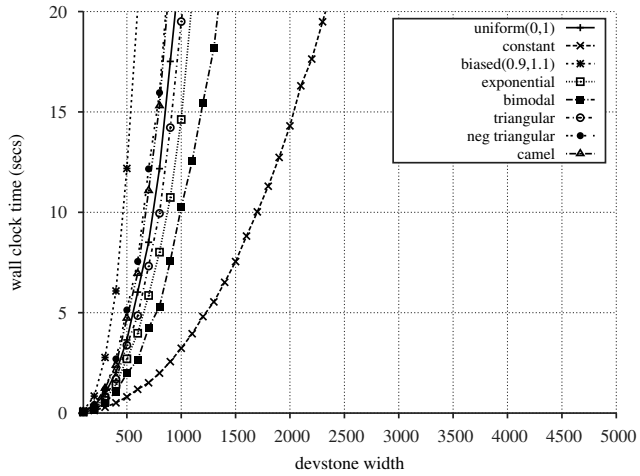

Figure 4. Wall clock time of a DEVStone simulation using the Sorted List based PES.

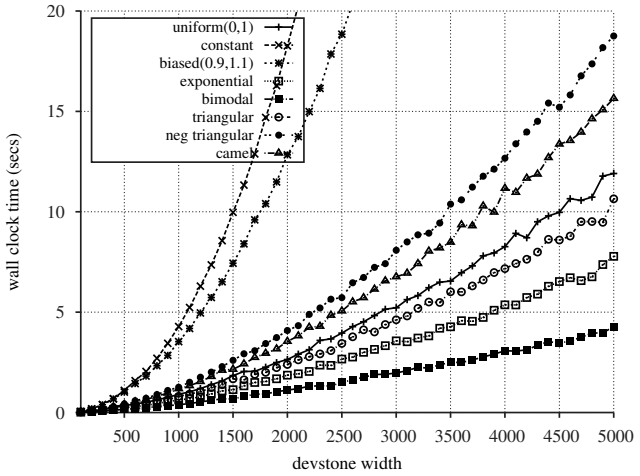

Figure 5. Wall clock time of a DEVStone simulation using the Splay Tree based PES.

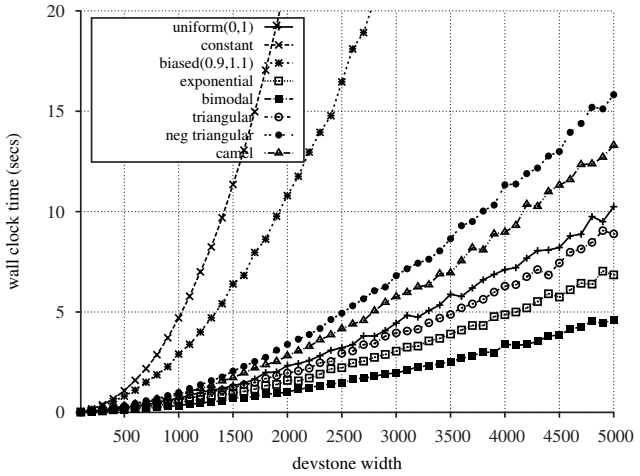

Figure 6. Wall clock time of a DEVStone simulation using the Binary Heap based PES.

good overview of pending event set performances since it is based on a plain and compliant PDEVS simulator, the results are not offering major indications concerning the behavior of each scheduler.

PDEVS model experiment

The second benchmark, however, reveals much more differences, first between multi-list (Figures 11 and 12) and tree based (Figures 9 and 10) priority queues. The splay tree has more performance costs with small queue sizes compared to the binary heap. Globally performances remains on the same scale but the splay tree is more sensitive to variations in the distribution.

Which is more surprising on the multi-list queues side is that on this benchmark, the calendar queue has lower access times than the ladder queue unlike the results of our first benchmark. We explain this because the initialization time is not measured in the original PDEVS benchmark method. By initialization time, we mean the time took to allocate, initialize the PES and to enqueue all events. Yet, this measure is important because the calendar queue adapt its internal structure also during enqueue operations while the ladder queue does it mostly over dequeue operations. Two new figures (13 and 14) are showing ladder queue and calendar queue results for the PDEVS benchmark but with init time taken into account. This clearly expose superiority of the ladder queue. It is faster and more stable than the calendar queue, which performances are depending on priority distribution.

CONCLUSIONS

This work present a qualitative study based on the performance of several data structures suitable for the implementation of a pending event set. This issue is essential in the context of event-driven simulation. We realized two different experiments found in the literature to compare our priority queue implementations within our (P)DEVS compliant simulator DEVS-Ruby. Our results shows that the Ladder Queue is the fastest and most reliable implementation.

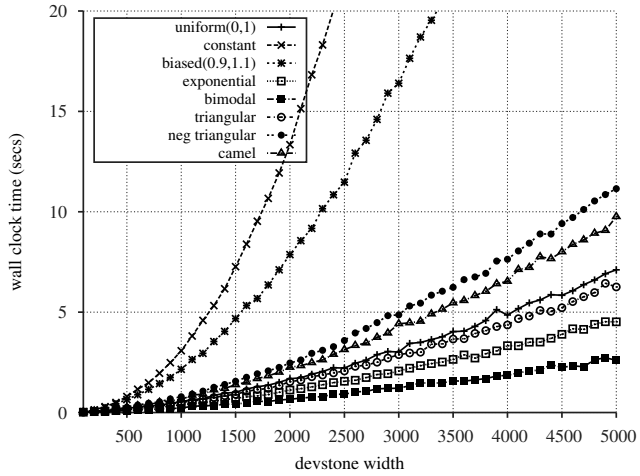

Figure 7. Wall clock time of a DEVStone simulation using the Ladder Queue based PES.

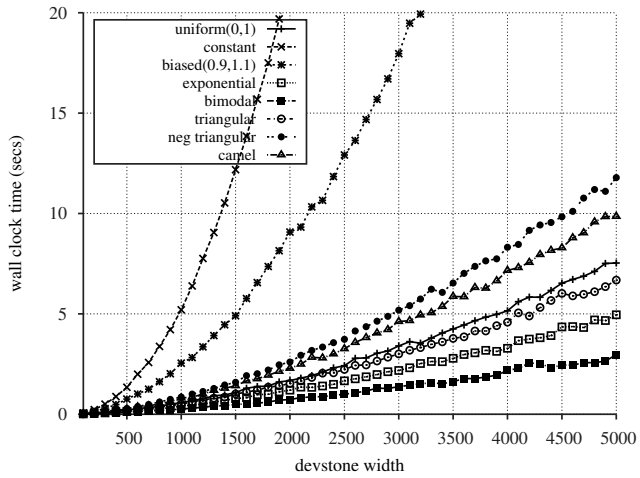

Figure 8. Wall clock time of a DEVStone simulation using the Calendar Queue based PES.

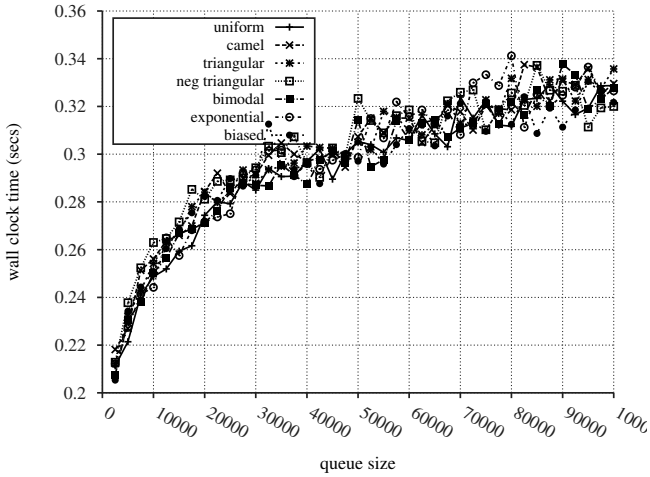

Figure 9. Runtime results of the Splay Tree based on the PDEVS benchmark method.

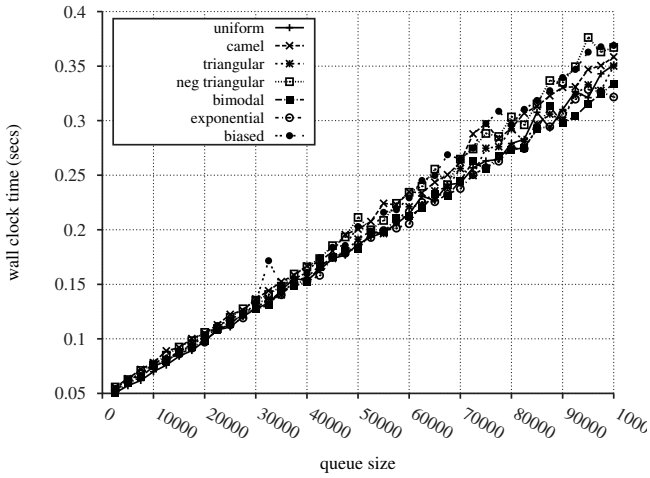

Figure 10. Runtime results of the Binary Heap based on the PDEVS benchmark method.

As a future work, it would be interesting to complete this study with additional data structures such as the Lazy Queue [16], or the P-Tree [1] along with variants of the considered data structures such as calendar queue [14, 17] and ladder queue variants [9, 7] which improves performance of their original counterparts. Although this study is strictly focused on the pending event set problem in sequential simulations, there is a set of works focusing on parallel discrete event simulation and on parallel priority queue algorithms [6, 15] which should be addressed in future works.

If the data structure algorithm is of main importance for the pending event set, implementation details can also make a difference, especially for a simulator implemented in a managed environment such as Ruby. We are currently working on C-based extensions for the DEVS-Ruby simulator where we are re-implementing the data structures in C to expose them to the Ruby VM. This allows us to get back control on the memory and to avoid the garbage collector. Moreover it will allow to study the effect of the Ruby programming language on the results and in the meantime, to report lower level dynamics data to the study, such as the locality in memory accesses or CPU cache-misses by the different data structures.

Acknowledgements

The present work was supported in part by the French Ministry of Research, the Corsican Region and the CNRS.

REFERENCES

1. Asdre, K., and Nikolopoulos, S. D. P-tree structures and event horizon: Efficient event-set implementations. In *Proceedings of the 37th Conference on Winter Simulation*, WSC '05, Winter Simulation Conference (Orlando, Florida, 2005), 2700–2709.

2. Barros, F. J. Dynamic structure discrete event system specification: A new formalism for dynamic structure modeling and simulation. In *Proceedings of the 27th Conference on Winter Simulation*, WSC '95, IEEE Computer Society (Washington, DC, USA, 1995), 781–785.

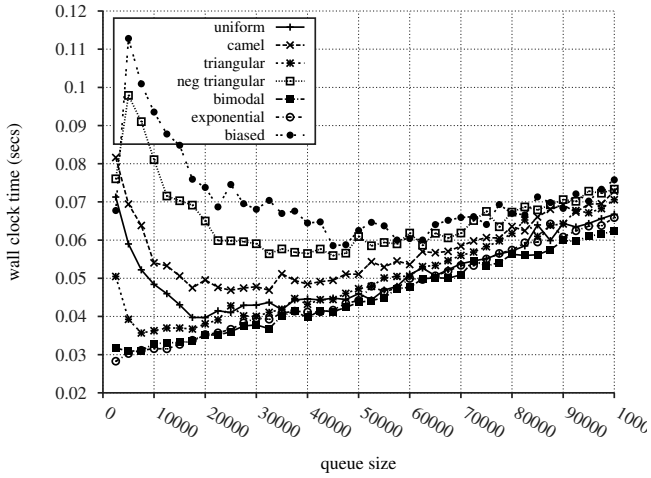

Figure 11. Runtime results of the Ladder Queue based on the PDEVS benchmark method.

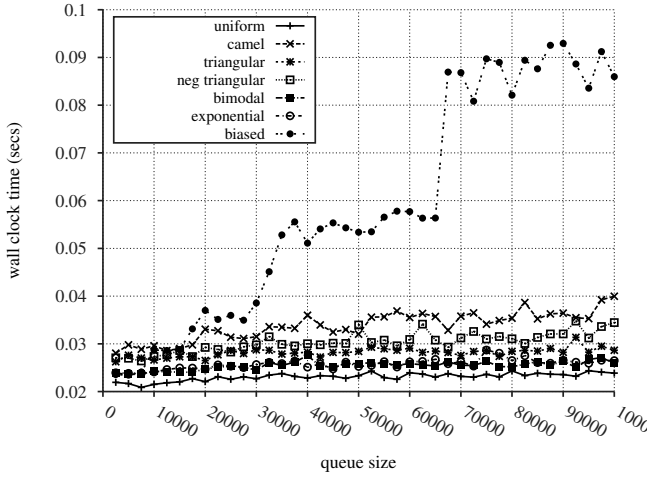

Figure 12. Runtime results of the Calendar Queue based on the PDEVS benchmark method.

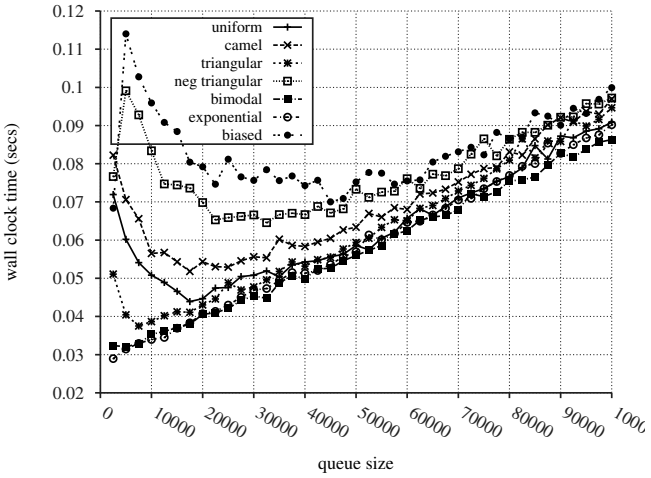

Figure 13. Runtime results of the Ladder Queue based on the PDEVS benchmark method with PES init time.

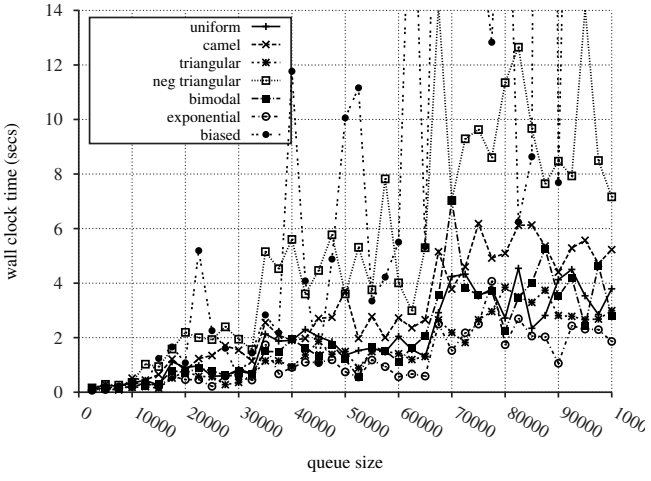

Figure 14. Runtime results of the Calendar Queue based on the PDEVS benchmark method with PES init time.

3. Blackstone, Jr., J. H., Hogg, G. L., and Phillips, D. T. A two-list synchronization procedure for discrete event simulation. *Commun. ACM 24*, 12 (Dec. 1981), 825–829.

4. Brown, R. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM 31*, 10 (Oct. 1988), 1220–1227.

5. Cassandrass, C. G., and Lafortune, S. *Introduction to Discrete Event Systems*, springer ed. 2nd ed. 2008. Kluwer Academic Publishers, 1999.

6. Dahl, J., Chetlur, M., and Wilsey, P. A. Event List Management in Distributed Simulation. In *Euro-Par 2001 Parallel Processing*, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds., Lecture Notes in Computer Science, Springer Berlin Heidelberg (2001), 466–475.

7. Dickman, T., Gupta, S., and Wilsey, P. A. Event Pool Structures for PDES on Many-core Beowulf Clusters. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, ACM (New York, NY, USA, 2013), 103–114.

8. Franceschini, R., Bisgambiglia, P.-A., Bisgambiglia, P., and Hill, D. R. C. DEVS-Ruby: a Domain Specific Language for DEVS Modeling and Simulation (WIP). In *DEVS 14: Proceedings of the Symposium on Theory of M&S*, SCS International (Apr. 2014), 393–398.

9. Gupta, S., and Wilsey, P. A. Lock-free Pending Event Set Management in Time Warp. In *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, ACM (New York, NY, USA, 2014), 15–26.

10. Himmelspach, J., and Uhrmacher, A. M. The event queue problem and PDevs. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, SpringSim '07, Society for Computer Simulation International (San Diego, CA, USA, 2007), 257–264.

11. Jones, D. W. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29*, 4 (Apr. 1986), 300–311.

12. McCormack, W. M., and Sargent, R. G. Analysis of future event set algorithms for discrete event simulation. *Commun. ACM 24*, 12 (Dec. 1981), 801–812.

13. Nikolopoulos, S. D., and MacLeod, R. An experimental analysis of event set algorithms for discrete event simulation. *Microprocessing and Microprogramming 36*, 2 (Mar. 1993), 71–81.

14. Oh, S., and Ahn, J. Dynamic calendar queue. In *Simulation Symposium, 1999. Proceedings. 32nd Annual* (1999), 20–25.

15. Ronngren, R., and Ayani, R. A comparative study of parallel and sequential priority queue algorithms. *ACM Trans. Model. Comput. Simul. 7*, 2 (Apr. 1997), 157–209.

16. Ronngren, R., Riboe, J., and Ayani, R. Lazy queue: An efficient implementation of the pending-event set. In *Proceedings of the 24th Annual Symposium on Simulation*, ANSS '91, IEEE Computer Society Press (Los Alamitos, CA, USA, 1991), 194–204.

17. Santoro, T., and Quaglia, F. A low-overhead constant-time LTF scheduler for optimistic simulation systems. In *2010 IEEE Symposium on Computers and Communications (ISCC)* (June 2010), 948–953.

18. Sleator, D. D., and Tarjan, R. E. Self-adjusting binary search trees. *J. ACM 32*, 3 (July 1985), 652–686.

19. Tang, W. T., Goh, R. S. M., and Thng, I. L.-J. Ladder Queue: An O(1) Priority Queue Structure for Large-scale Discrete Event Simulation. *ACM Trans. Model. Comput. Simul. 15*, 3 (July 2005), 175–204.

20. Vaucher, J. G., and Duval, P. A comparison of simulation event list algorithms. *Commun. ACM 18*, 4 (Apr. 1975), 223–230.

21. Wainer, G., Glinsky, E., and Gutierrez-Alcaraz, M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION 87*, 7 (July 2011), 555–580.

22. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation, Second Edition*. 2000.