Performance analysis of a parallel PDEVS simulator handling both conservative and optimistic protocols

Ben Cardoen† **Stijn Manhaeve**† **Tim Tuijn**†
{firstname.lastname}@student.uantwerpen.be

Yentl Van Tendeloo† **Kurt Vanmechelen**†
Hans Vangheluwe†‡ **Jan Broeckhove**†
{firstname.lastname}@uantwerpen.be

† University of Antwerp, Belgium
‡ McGill University, Canada

ABSTRACT

With the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform the simulation within reasonable time bounds. The built-in parallelism of Parallel DEVS is often insufficient to tackle this problem on its own. Several synchronization algorithms have been proposed, each with a specific kind of simulation model in mind. Due to the significant differences between these algorithms, current Parallel DEVS simulation tools restrict themself to only one such algorithm. In this paper, we present a Parallel DEVS simulator, grafted on C++11, which offers both conservative and optimistic simulation. We evaluate the performance gain that can be obtained by choosing the most appropriate synchronization protocol. Our implementation is compared to adevs using hardware-level profiling on a spectrum of benchmarks.

1. INTRODUCTION

DEVS [18] is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. In fact, it can serve as a simulation "assembly language" to which models in other formalisms can be mapped [17]. A number of tools have been constructed by academia and industry that allow the modelling and simulation of DEVS models.

With the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform the simulation within reasonable time bounds. And while Parallel DEVS [4] was introduced to increase parallelism, this is often still insufficient. Several synchronization algorithms from the discrete event simulation community [8] have been applied to DEVS simulation in particular. While several parallel DEVS simulation kernels exist, they often restrict themselves to a single synchronization protocol. This is a logical choice, as these synchronization protocols have nearly no commonalities. But the exact reason for different synchronization protocols, is that their distinct nature makes them very applicable to specific situations. As such, current DEVS simulation kernels always allow parallel simulation, but good performance can only be expected from some models.

Users that wish to simulate a set of distinct models, with different synchronization protocols, are therefore out of luck: Either they accept the lower performance for some of the models, or they use two distinct simulation kernels. Neither of these options is acceptable for the simulation of complex models, as the performance can become arbitrary worse, and the simulation kernels can diverge significantly.

In this paper, we introduce DEVS-Ex-Machina ("dxex"), our simulation tool which offers multiple synchronization protocols: no synchronization (sequential execution), conservative synchronization, or optimistic synchronization. The selected synchronization protocol is transparant to the simulated model. Users should merely determine, at the start of simulation, which protocol they wish to use. Our tool is implemented in C++11, to increase both performance and portability across different platforms.

A simple model is used to prove that the ideal synchronization algorithm is indeed dependent on model details. In our case, a single parameter of the model can have a significant impact on simulation performance, and especially on which is the ideal synchronization protocol. In order to show that this flexibility does not counter our performance, we compare our tool to Adevs, currently one of the fastest DEVS simulation tools available [16, 6].

Dxex, as well as all of our benchmark models, results, and profiling results, can be found online at `https://bitbucket.org/bcardoen/devs-ex-machina`.

We introduce the necessary background for this paper in Section 2. Section 3 elaborates on our features and design. In Section 4, we evaluate performance of our tool by comparing its different synchronization protocols. Related work is discussed in Section 5. Section 6 concludes the paper and gives future work.

2. BACKGROUND

This section briefly introduces two distinct synchronization protocols, as used by dxex. Furthermore, we make note of several new features of C++11.

SpringSim-TMS/DEVS 2016 April 3-6, Pasadena, CA, USA

2.1 Conservative Synchronization

The first synchronization protocol that we will introduce, is *conservative synchronization* [8]. In conservative synchronization, a node is allowed to progress in simulated time, independent of all other nodes, up to the point where it can guarantee that no causality errors can happen. When this point in time is reached, the node has to block until it is allowed to progress any further. In practice, this means that all nodes need to be aware of the current simulation time of all other nodes, and the time it takes an event to propagate (called *lookahead*). Several algorithms are defined in the literature to implement this behaviour. An overview is given in [8].

Deadlock can occur when a dependency cycle occurs and the amount of exchanged messages is low. Multiple algorithms are defined to handle this situation, such as deadlock avoidance and deadlock recovery.

The main advantage of conservative synchronization is that it has a low overhead if high parallelism exists between nodes. Each node can simulate in parallel, while sporadically notifying other nodes that they can progress even further. The disadvantage, however, is that the amount of parallelism is explicitly limited by the size of the lookahead. If a node can influence another (almost) instantaneously, no matter how rarely it occurs, the amount of parallelism is severely reduced. It is also up to the user to define the size of the lookahead, depending on how the model is known to behave. Slight changes in model behaviour can cause significant changes to the lookahead, and can therefore also have a significant influence on simulation performance.

2.2 Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [11]. Whereas conservative synchronization would prevent causality errors at all costs, optimistic synchronization will allow them to happen, but correct them afterwards. Each node is allowed to progress in simulated time as much as possible, without taking note of the state of any other node. When an event arrives at a node, which is already further in simulated time, the node will have to roll back its state to right before the event would normally have to be processed. As the simulation time is now rolled back to before the event is processed, the event can simply be processed as if no causality error ever occured.

Rolling back the simulation time requires the node to store past model states, such that they can be restored later on. Furthermore, all incoming and outgoing events need to be stored as well. Incoming events need to be passed to the models again, when the correct simulation time has again been reached, and outgoing events need to be cancelled, as potentially a different series of output events would normally have been generated. Cancelling events, however, can cause further rollbacks, as the receiving node might also have to roll back its state. In practice, a single causality error could have significant repercussions on the complete simulation.

Further changes are required for unrecoverable operations, such as I/O (*e.g.*, tracing, writing to file, printing output) and memory management. Lightweight algorithms are still required to determine the lower bound of all simulation times, through the computation of a *Global Virtual Time* (GVT).

The main advantage is that performance is not limited by a small lookahead, caused by a very infrequent event. If an (almost) instantaneous event occurs rarely, performance will only be impacted if it occurs, and not at every simulation step. The main disadvantage is unpredictable performance and arbitrary cost of rollbacks due to the propagation of causality errors. If rollbacks occur frequently, simulation quickly becomes slow, as the overhead of the recovery mechanisms becomes significant. Apart from overhead in CPU time, a significant memory overhead is present: all intermediate states qre stored up to a point where it can be considered *irreversible*.

While optimistic synchronization does not explicitly depends on the lookahead, simulation performance is still bound by the lookahead implicitly.

2.3 C++11 Parallelism Features

Apart from various other additions, C++11 adds a wide range of portable synchronization primitives to the Standard Library. In earlier versions of the language standard, one had to resort to non-portable C implementations. Furthermore, C++11 is the first version of the standard that explicitly defines a multi-threaded abstract machine memory model. Most modern compilers support the full standard, allowing the kernels to be portable by default on any standard compliant platform.

Simulation values that need to be shared between nodes, can now be coded portable and efficiently through the use of atomic fields. This avoids latencies caused by the exchange of synchronization messages. For an in-depth study, we refer to [5].

3. FEATURES

Historically, dxex is based on PythonPDEVS [16]. While Python is a good language to create software prototypes, its performance has proven to be insufficient to compete with other simulation kernels [14]. Dxex implements only a subset of PythonPDEVS, but makes some of its own additions. The core simulation algorithm and optimizations, however, are highly similar.

While we will not make a detailed comparison with PythonPDEVS here, dxex also supports direct connection [3], Dynamic Structure DEVS [1], termination conditions, and a modular tracing framework [16]. But whereas PythonPDEVS only supports optimistic synchronization, dxex support multiple synchronization protocols. This is in line with the design principles of PythonPDEVS: offer users the option to pass information on how to efficiently simulate the model. In our case, it now becomes possible to pass the simulation kernel the "hint" as to which synchronization protocol would be ideal for this model. Furthermore, the implementation in C++11 allows many more (static) optimizations, which were plainly impossible when using an interpreted language.

Note that, since there is no universal DEVS model standard, dxex models are incompatible with PythonPDEVS and vice

versa. This is due to dxex models being grafted on C++11, whereas PythonPDEVS models are grafted on Python.

In the remainder of this section, we will elaborate on our prominent new feature: support for multiple synchronization protocols within the same simulation tool.

3.1 Synchronization protocols

We have previously shown that different synchronization protocols exist, with each of them being optimized for a specific kind of model. As no single synchronization protocol is ideal, a general purpose simulation tool should support multiple situations. Currently, however, most parallel simulation tools chose only a single synchronization protocol due to the inherent differences between these approaches. An uninformed choice on which to implement is insufficient, as performance is likely to be bad. We therefore argue that a real general purpose simulation tool should support sequential, conservative, and optimistic synchronization.

Each of them is applicable in specific model configurations. Conservative synchronization is ideal when high lookahead exists between different nodes, and barely any blocking is necessary. Optimistic synchronization is ideal when lookahead is unpredictable, or there are rare (almost) instantaneous events. Finally, sequential simulation is still required for models where parallelism is bad, causing significant overhead.

Sequential

Our sequential simulation algorithm is very similar to the one found in PythonPDEVS, including most of its optimizations. However, the simulation algorithm is designed to be overloaded by different synchronization protocol implementations. This way, the DEVS simulation algorithm is implemented once, but parts can be overwritten as needed. In theory, more synchronization protocols (*e.g.*, other algorithms for conservative synchronization) can be added without altering our design.

Conservative

For conservative synchronization, each node determines the nodes it is influenced by. Each model provides a lookahead function, which determines the lookahead depending on the current simulation state. Within this time interval, it is an error if a model raises an event, thus violating its previous promise. The node uses this information to compute its earliest output time (EOT), and writes out the value in shared memory through the use of C++11 synchronization primitives.

Optimistic

For optimistic synchronization, each node needs to keep track of all intermediate simulation states. This needs to be done carefully, in order to avoid unnecessary copies, and minimize the overhead induced for each transition function. We use Mattern's GVT algorithm [12] to determine the Global Virtual Time (GVT) using at most $2n$ synchronization messages. This process runs asynchronously from the simulation itself. Once the GVT is found, all nodes are informed of the new value, after which fossil collection is performed.

3.2 Transparancy

A user must only provide one model, implemented in C++11, which can be simulated by each synchronization kernel. The exception is conservative synchronization: a lookahead function is required, whereas this is not possible in other synchronization kernels. Two options are possible: either a lookahead function is always provided, even when it is not required and possibly not used, or using a default lookahead function if none is defined.

Always defining a lookahead function might seem redundant, especially if users will never use conservative synchronization. The more attractive option is for the simulation tool to provide a default lookahead function, defined by the minimum detected time advance. This lookahead value is most likely too small, but will prevent causality errors at the cost of performance. Depending on the model, simulation performance might still be faster than sequential simulation.

Defining a lookahead function is therefore recommended in combination with conservative synchronization, but is not a necessity.

4. PERFORMANCE

In this section, we evaluate the performance of our simulation tool. We show that the inclusion of multiple synchronization protocols does not decrease our performance to the point where it is unusable. To this end, we compare to adevs, one of the most efficient simulation kernels at this time [6]. A comparison is made for both the CPU and memory usage of both sequential simulation and parallel simulation.

We start of with a comparison of sequential simulation, to show how adevs and dxex relate in this simple case. We not only show that our approach does not influence performance negatively, but also that our main simulation algorithm, similar to the one of PythonPDEVS, is significantly faster than the one found in adevs. Similar differences, compared to adevs, can also be seen in the parallel simulation benchmarks. In the parallel simulation benchmarks, the benefit of our different synchronization protocols is also indicated.

For all benchmarks, results were all well within 1% deviation of the average, such that only the average is used in the remainder of this section. The same compilation flags were used for both adevs and dxex benchmarks ("`-O3`"). To guarantee comparable results, no IO was performed during the benchmarking phase. Simulation traces were used to verify that both adevs and dxex models have exactly the same behaviour. All benchmarks were performed using Linux, but our simulation tool works equally well on Windows and Mac.

4.1 Benchmarks

We use a selection of benchmarks, based on those found in the literature. Three different types of benchmark are defined, each for a different purpose:

1. *Queue* model, based on the HI model of DEVStone [10], creates a chain of atomic models, which are hierarchically nested in each other. A single generator will push events into the queue, which get processed by the processors after a fixed or random delay. It takes two parameters: width

Figure 1. Queue model for depth 3 and width 2.

Figure 2. HighInterconnect model for four models.

Figure 3. PHOLD model for four models, split over two nodes. In parallel simulation, each coupled model is simulated at a different node.

and depth, which determine the widht and depth of the hierarchy. This benchmark shows how the complexity of the simulation kernel behaves with an increasing amount of atomic models, and an increasingly deep hierarchy. If the processing delay is fixed for all processors, further insight is provided in the collision handling performance of the simulation kernel. An example for width 2 and depth 3 is shown in Figure 1.

2. *PHOLD* model, presented by [7]. It creates a set of n atomic models, and each model has exactly $n-1$ output ports: one for every other atomic model. Couplings are made such that each atomic model is directly connected to each other atomic model, such that every atomic model can directly send an event to every other atomic model. After a random delay, atomic models will send out an event to a randomly selected output port. Output port selection happens in two phases: first it is decided whether the event should be sent to an atomic model inside or outside of this coupled model. Afterwards, a uniform selection is made between the possible ports. It takes one parameter: the percentage of remote events, which influences the fraction of messages routed to other coupled models. This benchmark shows how the simulation kernel behaves in the presence of many local or remote events. An example for four models, split over two nodes, is shown in Figure 3.

3. *HighInterconnect* model, a merge of the HI model of DEVStone [10] and PHOLD [7]. It creates a structure similar to the one from PHOLD, but instead of $n-1$ output ports, every atomic model has only a single output port. All models are still connected to each other, but through the use of broadcasting: every model will receive a generated event. It takes one parameter: the number of models. This benchmark investigates the complexity of the routing algorithm. An example for four models is shown in Figure 2.

We opted to deviate from the DEVStone benchmark, as DEVStone tends towards unrealistic models since all internal and external transitions occur simultaneously. In our benchmark models, there is always the option for simultaneous transition functions (*fixed time advance*), or scattered transition functions (*random time advance*). Furthermore, they defined the use of an artificial load function, which easily skews the result, making the actual simulation algorithm barely comparable.

4.2 Sequential Simulation Execution Time

Despite our core contribution being mainly in the parallel simulation, we still value a comprehensive comparison of sequential simulation results. First, and foremost, as parallel simulation results are tightly linked to the sequential simulation results: parallel simulation merely adds a synchronization layer over different, essentially sequential, simulation kernels. Second, since parallel simulation results are validated through the use of adevs. To provide a more comprehensive comparison to adevs in the parallel simulation benchmarks, sequential simulation results need to be compared. Only the Queue and HighInterconnect models are relevant for sequential simulation.

Queue

In the Queue model, we increase both the width and depth simultaneously, causing a quadratic growth in the number of atomic models. As can be seen in Figure 4, dxex considerably outperforms adevs. Through careful analysis of profiling results, we determined that adevs spends much time in handling simulation messages, whereas this is mostly avoided due to the differently designed simulation algorithm of dxex. Both simulation tools have quadratically increasing execution times, though dxex is much faster thanks to its more efficient simulation control algorithms.

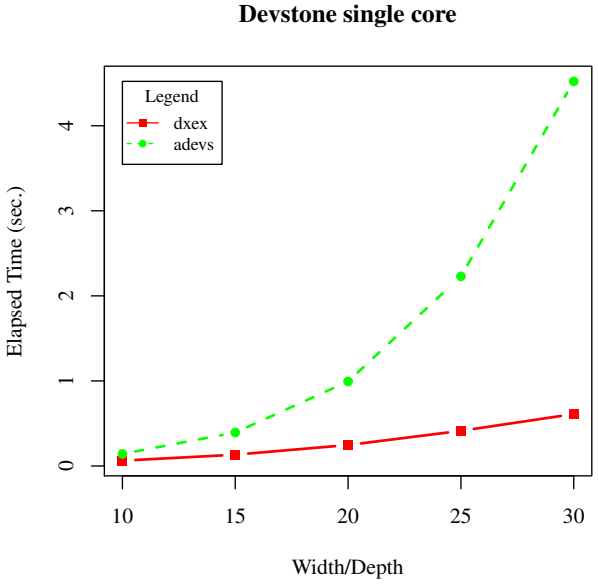

Figure 4. Queue benchmark results

HighInterconnect

In the HighInterconnect model, we increase the number of atomic models, thus quadratically increasing the number of couplings. As can be seen in Figure **??**, adevs now outperforms dxex by a fair margin. Analysis showed that this slowdown is caused by the high amount of exchanged events. Event creation is found to be much slower in dxex than it is in adevs, even despite the use of memory pools in dxex.

4.3 Parallel Simulation Execution Time

By default, the benchmarks use 4 kernels for parallel simulation.

Devstone

The flattened models are allocated to kernels by giving each kernel a distinct section of the chain, resulting in a low ratio of inter-kernel to intra-kernel messages. For the optimistic case,

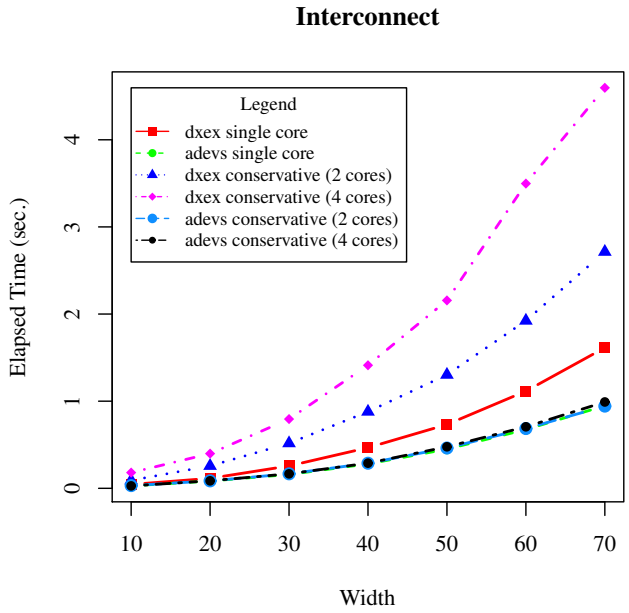

Figure 5. Interconnect

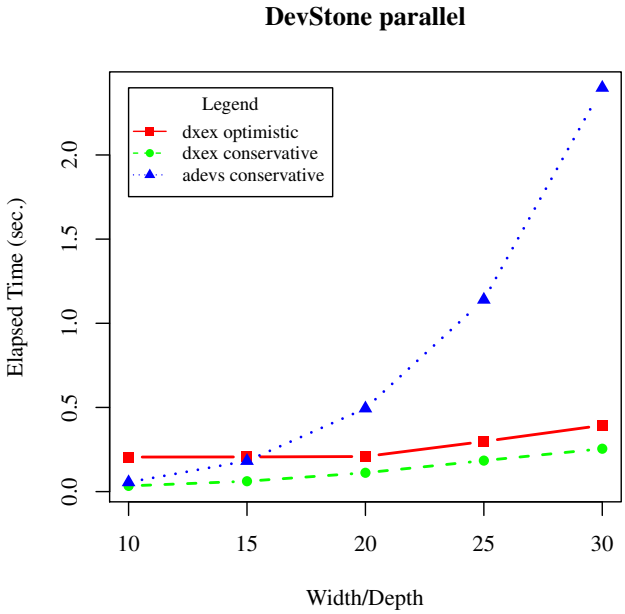

Figure 6. DevStone parallel

this can cause more reverts since the kernels will start to drift faster as the model-count increases. Furthermore, optimistic synchronization is quite sensitive to an increase in kernels, since the delay before a revert propagates, increases. This benchmark requires a specific warm-up time; for $n = d \times w$ models, it takes timeadvance$() * (n-1)$ transitions to activate the last model in the chain. When using parallel processing, this can be reduced to timeadvance$() * (n\frac{kernels-1}{kernels} - 1)$ before the last kernel becomes active.
Figure shows that dxex outperformance adevs (in both optimistic and conservative case).

PHold

In Phold, the allocation is specified in the benchmark itself. Each kernel manages a single node with a constant set of subnodes. The parameter R (which is here set to 10) determines the percentage of remote destination models.
The dynamic dependency graph is a very sparse version of the static dependency graph, penalizing the conservative case. The lookahead is ϵ, so the conservative case spends most of its time crawling in steps of ϵ. Since the dependency graph between kernels is a complete graph, this is not a simulation that scales in our implementation. For N kernels, each kernel has to query the null-time of $N-1$ kernels, resulting in O(N^2) polling behaviour. In a non-cyclic simulation with a non-trivial lookahead (like in, e.g., Devstone), that choice does pay off (see Figure). Adevs's lesser performance is due in part to their lookahead management, which after profiling shows to spend a non-trivial amount of time in exception handling code.
The optimistic case suffers little from the above problems; due to the high interconnectivity, however, a cascading revert is still possible. With the percentage of remotes equal to 100, Phold reflects interconnect in behaviour, which is why the R value is not dimensioned here. A revert is very expensive in PHold due to our usage of C++11's random nr generators. The cost of a revert is dominated by the recalculation of destination models, not in allocating/deallocating states/messages. Once a revert happens the drift between kernels increases fast, increasing the likelihood of more reverts. Despite all this, optimistic can for low R values quickly exploit the uncertainty that slows down conservative in this benchmark.

Interconnect

In Interconnect, the set of atomic models form a complete graph (w.r.t. connections); each model broadcasts messages to the entire set.
Allocation cannot avoid cycles and the resulting dependency graph between kernels remains a complete graph. The runtime dependency graph is almost immediately identical to the static graph.
The conservative case still shows the same issues as in PHold, with the key difference , for a fixed time advance, the lookahead is equal to the timespan between transitions. The scaling issue is identical as with PHold.
In this benchmark, the optimistic case runs very quickly out of memory. With c kernels and N atomic models, a single revert undoing k transitions will lead to $(N-1) \times k$ messages that need to be recreated, plus $(N-1)\frac{c-1}{c} \times k$ anti-messages that need to be sent.

Priority network model

The priority benchmark is composed of a single server generating a stream of $0 \leq m \leq n$ messages at fixed time intervals, interleaved with a probability p for a priority message to n receivers (see Figure 7).

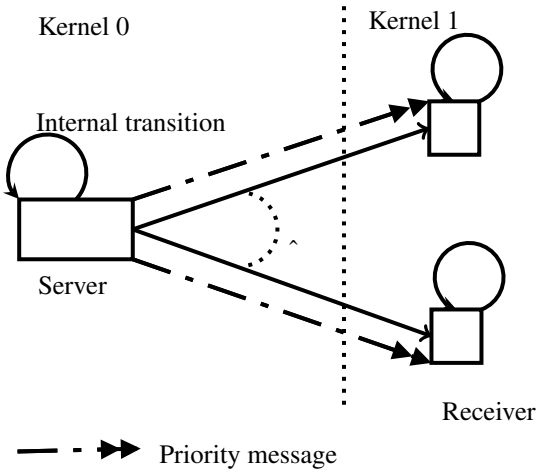

Figure 7. The priority network model

This defaults the lookahead for the receivers to ϵ, but this time there is no scaling effect, nor are there cycles in the dependency graph. This model therefore highlights the basic strengths/weaknesses of both synchronization protocols. Receiving models are allocated on another kernel than the server and have an internal transition so that they will not wait for the incoming messages.

A key difference here with the other benchmarks is that a state (in the Receiver instances) is very cheap to copy/create. The kernel holding the server will never revert since it is a source in the dependency graph. The optimistic case will therefore not suffer the same performance hit in recreating the states as it does in PHold. The overhead in the optimistic case is entirely due to the factor m, which will quickly dominate in increasing buffers of received messages. Interestingly, the parameter p does not clearly favour either synchronization protocol (see Figure). While this removes any possibility for a lookahead, the conservative case can quickly bridge the timespan between fixed messages since there is no cyclic dependency. Lightweight states prevent performance loss due to reverts in the optimistic case; only the overhead in event handling in the optimistic case eventually becomes the deciding factor.

4.4 Memory Usage

Platform and tools

Both dxex and adevs use tcmalloc as memory allocator. Additionally, dxex uses memory pools to further reduce the frequency of expensive system calls (malloc/free/sbrk/mmap/...). Tcmalloc will only gradually release memory back to the OS, whereas our pools will not do so at all. If memory has been allocated once, it is from a performance point of view better to keep that memory in the pool. This is one reason why memory utilization is best measured by peak allocation. Profiling is done using Valgrind's massif tool [13]. The platform used for memory profiling has an i5-3317U Intel CPU and 8GiB RAM with a page size of 4,096KiB, running Fedora 22 (kernel 4.2.6).

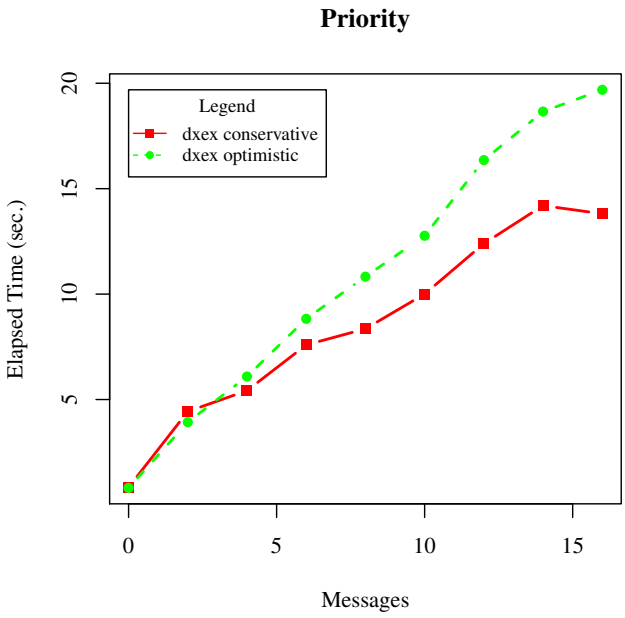

Figure 8. Priority

Measure

Adevs passes messages by value, dxex passes a pointer. The runtime effects of this choice have already been demonstrated in the Interconnect benchmark, so, in this section, we measure memory usage in number of allocated pages combining text, stack and heap memory for the program profiled. For the OS and/or user, this is the actual memory footprint of the application. It is important to note that, especially in the optimistic case, not all this memory is always in use by the kernels. During simulation, the pools will generally not return deallocated memory to the OS, but keep it for later reuse.

Results

Devstone

adevs	adevs con	dxex	dxex con	dxex opt
44	70	42	75	363

Table 1. Devstone 40x40 t5e5, unit MiB, 4 kernels (if parallel)

Since, in the conservative case, messages are passed by pa ointer, a GVT/LBTS implementation is required to organise the garbage collection. This inevitable delay explains the higher memory usage compared to adevs.
Optimistic's TimeWarp requires state/event saving, and its GVT algorithm is more complex (with a resulting higher latency) than the LBTS calculation in the conservative case. Moreover, the differences in LP virtual times are far larger compared to conservative time synchronization. All these factors explain the heavier memory usage. Devstone (flattened) is allocated in a chain. Leafs in the dependency graph will therefore do a lot of unnecessary simulation before having a revert, leading to an increased memory pressure. Unlike conservative and sequential execution, memory usage in the

optimistic case varies greatly depending on scheduling of kernel threads and drifting between kernels.

PHold

adevs	adevs con	dxex	dxex con	dxex opt
40	x	37	61	682

Table 2. Phold n 4 s 16 t1e6 r 10, unit MiB, 4 kernels (if parallel)

With only 10% of all messages being inter-kernel, we expect conservative to have memory consumption near that of the single threaded implementation, since intra-kernel messages are reclaimable after each round. The counterintuitive high memory usage can be explained by conservative's stalled round behaviour which occurs whenever a kernel cannot advance (eit == time). In such a round messages are sent out but the kernel does not execute any transitions until it has received all input from all influencing kernels.
With lookahead ϵ this then leads to a high frequency of polling on shared null-times, which are used to determine lbts and thus garbage collection. The lbts calculation will not wait until a new value is found, since this can create unwanted contention with the simulation. The resulting longer time intervals within which no new lbts is found delay memory deallocation. Adevs' conservative fails to complete the benchmark under valgrind even with a significantly reduced load. Optimistic exhibits the expected high memory usage.

Interconnect

In section 4.2 the parallel performance of this benchmark is further explained. Interconnect highlights dxex's lower performance due to message allocation overhead.

Priority Network

The priority network model is detailed in section 4.1.
The high memory usage of the conservative case is largely due to the garbage collection implementation. Kernel 0 is the initiator of the LBTS calculation but since it holds only models independent of any other (server), it will finish simulation very fast leaving the other kernel bereft of an updated LBTS. The kernel holding the server will after simulation wait on the receiving kernel until it can prove all sent messages can be deallocated.

5. RELATED WORK

5.1 PythonPDEVS

Dxex is closely related to PyPDEVS in design and philosophy. PyPDEVS allows anyone who grasps the PDEVS formalisms to immediately simulate his/her model without having to consider the kernel implementation. A Python based implementation offers the advantage of very fast prototype/run/evaluate cycles, this can't be matched by a C++ simulator. It should be noted that, once the kernels have been compiled (in shared libraries), the actual compilation time of the model is small enough to make prototyping possible. Advanced features such as activity based relocation and the performance gains this results in, are still unique to PyPDEVS.

adevs	adevs con	dxex	dxex con	dxex opt
39	39	35	43	259

Table 3. Interconnect w 20 t5e5, unit MiB, 2 kernels (if parallel)

dxex	dxex con	dxex opt
35	450	651

Table 4. Priority model n 128, m 16, p 10, t2e8, unit MiB, 2 kernels

5.2 Adevs

Adevs is still under active development, allowing for an exact comparison in performance and features. It remains a very fast simulation engine for the PDEVS formalism, but it lacks an optimistic synchronization implementation. Due to the fact that adev does not flatten the Coupled Models, it's performance degrades significantly if the hierarchy becomes deeper. Dxex, on the other hand, manages lookahead with more overhead than adevs, leading to a performance difference as the model count increases. Finally, adevs employs the full dependency graph this in contrast with dxex kernels, which only observe 1-edge removed nodes.

5.3 CD++

Different projects on CD++ offer conservative (CCD++) as well as optimistic (PCD++) parallel simulation. In contrast to dxex, neither projects offer both synchronization protocols. CD++ relies on the WARPED kernel. It is a middleware that provides memory, event, file, time and communication scheduling. We did not use the WARPED kernel (nor the underlying MPI) because dxex is designed specifically for a shared memory architecture and, as such, any middleware, however feature-rich, would have lead to an unacceptable overhead.

6. CONCLUSIONS AND FUTURE WORK

6.1 Memory

Our optimistic implementation can benefit from a faster GVT algorithm such as used in [9] or, more recently in [2]. With access to the memory subsystem already in place (pools), the optimistic case could be constrained using per kernel quota, preventing the simulation from exhausting memory. While less sensitive to memory pressure, the same approach could improve the conservative case.

6.2 Activity

As shown in [15], activity and allocation of models across kernels is a key aspect in achieving high performance in any parallel implementation. Allocating models so that there are no dependency cycles between their containing kernels is a first step, but not always possible. For the optimistic case, one can use re-allocation to break (runtime dependency) cycles and/or perform load-balancing. If kernels are unevenly balanced, they will begin to drift fast, causing increasingly more reverts. In dxex, we implement a limited framework to track model activity for debugging purposes; this could be extended to enable the above strategies.

6.3 Hybrid

The optimistic implementation could use (null/eot/eit) from the conservative case to detect and/or reduce the cost of reverts without completely stalling on influencing kernels.
The eot calculation in the conservative case can be enhanced by using the entire dependency graph between kernels (not models). The implementation could detect dependency cycles between kernels and, instead of waiting on dependent kernels, advance as a synced group with better scaling. LBTS calculation could leverage this as well to reduce memory usage, though this is not that stringent in the conservative case. Ultimately, the simulation could switch at runtime between protocols based on the information provided by activity tracking. This requires the above mentioned framework, and can only be done at timepoints where a GVT/LBTS time is agreed upon between the kernels. From the model's perspective, no changes are needed, although a non-trivial lookahead is obviously desired.

In this paper, we presented dxex, a new C++-based PDEVS implementation supporting both sequential and parallel processing models.
We showed that, in the sequential case, dxex outperforms adevs in hierarchical simulations and that adevs produces better results in broadcast-like simulations.
In the parallel case, the results are more subtle. The sensitivity to broadcast simulations is carried over in the parallel implementation, resulting in performance loss for both synchronization protocols. The optimistic case can still achieve good performance if the volume of inter-kernel messages is low. Both synchronization protocols are fast in non-cyclic simulations, whereas the conservative case is better suited whenever state-saving becomes expensive; the optimistic case, on the other hand, is better suited when a lookahead is not available. Finally, the current optimistic implementation is ill-suited for memory constrained systems, especially in simulations where the frequency of reverts is high.

ACKNOWLEDGMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO).

REFERENCES

1. Barros, F. J. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation 7* (1997), 501–515.
2. Bauer, D., Yaun, G., Carothers, C. D., Yuksel, M., and Kalyanaraman, S. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS '05, IEEE Computer Society (Washington, DC, USA, 2005), 39–48.
3. Chen, B., and Vangheluwe, H. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference* (2010), 209–218.
4. Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference*, SCS (1994), 716–722.
5. De Munck, S., Vanmechelen, K., and Broeckhove, J. Revisiting conservative time synchronization protocols in parallel and distributed simulation. *Concurrency and Computation: Practice and Experience 26*, 2 (2014), 468–490.
6. Franceschini, R., Bisgambiglia, P.-A., Touraille, L., Bisgambiglia, P., and Hill, D. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *2014 Imperial College Computing Student Workshop*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014), 40–49.
7. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).
8. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.
9. Fujimoto, R. M., and Hybinette, M. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul. 7*, 4 (Oct. 1997), 425–446.
10. Glinsky, E., and Wainer, G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 2005 9th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications* (2005), 265–272.
11. Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.
12. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (1993), 423–434.
13. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (jun 2007), 89–100.
14. Van Tendeloo, Y. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014.
15. Van Tendeloo, Y., and Vangheluwe, H. Activity in pythonpdevs. In *Activity-Based Modeling and Simulation* (2014).
16. Van Tendeloo, Y., and Vangheluwe, H. The Modular Architecture of the Python(P)DEVS Simulation Kernel. In *Spring Simulation Multi-Conference*, SCS (2014), 387 – 392.
17. Vangheluwe, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design* (2000), 129–134.
18. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.