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

1.1 DEVS

The family of DEVS [28] formalisms serve as a common basis for most other discrete event formalisms. Of interest in this paper are the 2 key formalisms: Dynamic Structured [1] and Parallel [5] and their implementation. This project uses the Direct Connection [4] algorithm, so from a simulation kernel's perspective only Atomic Models exist linked to each other by ports.

1.2 Parallel computing

Parallel execution of a PDEVS simulation can lower overall runtime and increase the bound on the state space, thereby enabling simulation of more complex systems in the same time-frame. While the shared memory parallelism offered by most modern hardware does not raise the state space bounds, it can reduce the runtime and offers more direct communication and control between entities involved in synchronization compared to distributed simulation.

1.3 Motivation

Adevs [21] offers a very fast conservative synchronized shared memory PDEVS simulator, but no optimistic synchronized variant. The latter can be significantly faster, especially in simulations where the runtime behaviour of the simulation is hard to predict.

The matured parallelism features of C++11 were used in this project with the dual aim of writing standard-compliant (and thus portable) code and without losing access to powerful low-level threading primitives.

1.4 Solution

The usage of the Direct connection algorithm makes reusing adevs kernel hard. Devs-ex-machina (dxex) is based on PythonPDEVS [26](PyPDEVS) but differs in design and implementation. The core aim of the project is to offer a deterministic simulation kernel where the simulation modeller is shielded as much as possible from the kernel implementation, without sacrificing performance. As in PyPDEVS, a model need be written only once for use in the different simulation kernels (with the exception of a non-trivial lookahead). The tracing framework from PyPDEVS was ported to allow optional verification of simulations.

1.5 Time

The DEVS formalisms have $\mathbb{R}$ as time base, but any implementation has to decide on an enumerable representation of time. In principle any type with well defined operators can be used as template parameter, but from a performance point of view a type fitting in a machine word offers obvious advantages. In practice, dxex defaults to word sized unsigned integer. Although this significantly reduces the possible range of virtual time, it avoids approximation errors. Furthermore, the notion of ϵ as a minimum distance between two distinct time points need not to be established. This task is non-trivial for floating point approximations.

Our time representation is extended with a causality field with a range equal or greater than the maximum nr of models in a single kernels (2^{48} by default). If A and B are imminent at time t then $t[1]_a < t[1]_b \oplus t[1]_b < t[1]_a$, while $t[0]_a == t[0]_b == t[0]$. This allows a deterministic ordering of concurrent events.

2. BACKGROUND

In this section, we provide a brief introduction to two different synchronization protocols for parallel simulation, and the features offered by C++11 that aid in our implementation.

2.1 Conservative Synchronization

Conservative synchronization is defined by the invariant that no model will advance in time before it has received all input from any influencing model.

This requires the concepts of earliest output time (eot) and earliest input time (eit), which define the timespan within which a model can safely advance.

The eit of any model is the minimum of all eot values of (directly) influencing models. A model can simulate up to (but not including) eit, then waits until that value is increased. An important disadvantage here is that the influenced-by relation is always defined at model(link) creation, not at runtime. A model that can influence another, but never does, can severely slow down the protocol. Deadlock between models that influence each other and end up waiting on each other can be broken/avoided by a variety of means. In this simulator, the CMB [3] null-message protocol is used.

In our implementation null-time is the timestamp a model is guaranteed to have passed in simulation. More precisely, a null message of time t is a guarantee that any output with timestamp t-ϵ is already sent.

Conservative synchronization relies explicitly on information provided by the model creator in the form of a lookahead, a relative timespan during which the model is insensitive to outside events. This lookahead can be non-trivial to calculate, a simulation writer will in general not be able to determine the exact lookahead of models involved in an experiment without having run the experiment.

Conservative kernels can operate if there exists a cyclic dependency between them, but at a quite severe performance penalty, as seen in section 4.

2.2 Optimistic Synchronization

Optimistic synchronization allows causality errors to occur but recovers using a roll-back mechanism, the most common of which is Timewarp [15]. Whenever a kernel receives an event with a timestamp in the kernel's past, the state of the kernel (and all models)reverts to that time. Because an optimistic kernel does not wait for other kernels, it can achieve a lower runtime than conservative. The cost of this approach is higher complexity of the code and larger memory usage for state/message saving. In contrast to conservative, optimistic does not rely on any domain specific information, it is only sensitive to runtime use of connections, not the probability that they might occur.

If the (runtime) dependency graph contains a cycle, optimistic can suffer a series of cascading reverts. The cost of reverts can be reduced by lazy cancellation and/or lazy re-evaluation [8].

2.3 Global Virtual Time

To avoid exhausting memory in state/event saving, optimistic synchronization relies on the concept of global virtual time[15]. In optimistic simulations, GVT is defined as the lowest timestamp of any unprocessed event.

Intuitively this is the simulation timepoint that is certain to

be preserved, corresponding exactly with a sequential simulation. In conservative, the minimum simulation time of all kernels is the LBTS, or in terms of null messages: the least timestamp of any null message in transit. The GVT calculation is vital to safely commit unrecoverable transactions such as IO (e.g. tracing), releasing memory, ...

2.4 C++11 Parallelism Features

C++11 offers a wide range of portable synchronization primitives in the Standard Library, whereas in earlier versions one had to resort to non-portable (C) implementations. More importantly, C++11 is the first version of the standard that actually defines a multi-threaded abstract machine memory model in the language. Our kernels use a wide range of threading primitives and atomic operations. As an example, eot/eit/nulltime are exchanged not as messages, but as reads/writes to atomic fields shared by all kernels. This avoids the otherwise unavoidable latency penalty by mixing simulation messages with synchronization messages. For an in-depth study, see [6]. Most modern compilers support the full standard, allowing the kernels to be portable by default on any standard compliant platform.

3. FEATURES

3.1 Based on PythonPDEVS

The simulator is based on PythonPDEVS, and provides the following features:

1. Direct Connection

2. Dynamic Structured DEVS

3. Termination function. If specified, a termination function is applied every simulation round to each model to test whether the simulation can terminate. Only available in single-threaded simulation.

4. State/Message can have any payload type. Different message types can be used together within the same simulation.

5. Tracing An asynchronous, thread safe and versatile tracing mechanism allows exact verification of the simulation.

6. Optimistic and Conservative synchronization of PDEVS.

The implementation tries to adhere to the C++ principle that you don't pay performance-wise for what you don't use. For this reason, the support for a termination function for the multi-threaded kernel was abandoned, as it is non-trivial to implement and had a non-negligible impact on the runtime, even when not in use. Another example is the state saving mechanism, which is only used for optimistic parallel simulation and has no performance impact in conservative parallel simulation.

The tracing is not comparable to adevs's listener interface. To be usable in optimistic simulation, the tracing of the simulation has to be reversible and only be committed at GVT points. Furthermore, the framework itself has to be thread safe and deterministic so that a simulation will always produce the exact same output. The following features from PyPDEVS are not present

1. Activity tracking and relocation

2. Serialization

3. Interactive control.

4. Distributed simulation

Serialization in this context is the ability to save/load a complete simulation to disk, not the state saving mechanisms required for TimeWarp.

Model allocation is done by a derivable allocator object which the user can implement to arrange a more ideal (domain-specific) allocation. If this is omitted, a default (non-activity-aware) allocation stripes the models over the simulation kernels.

Debugging tools such as a logger and a graph visualizer are included which can track activity with respect to allocation for later study, but not online/dynamic as is possible in PyPDEVS.

3.2 Different Synchronization protocols

Synchronization is required for PDEVS to prevent or recover from causality violations if the simulation is executed in parallel. Preventing causality violations (conservative) typically requires domain-specific information to compensate for performance loss, while recovering from causality errors (optimistic) requires a state to recover to.

Conservative

A conservative kernel will determine which kernels it is influenced by. This information is constructed from the incoming connections on all hosted models. The process is only 1 link deep, since an influenced kernel will in turn be blocked by others deeper in the graph.

A model should provide a lookahead function which returns, relative from the current time, the timespan during which the model cannot change state due to an external event. This information is collected for all models hosted on the kernel, and the minimum is set as the lookahead of that kernel.

The kernel will calculate its earliest output time and write this value in shared memory. The eit of the kernel is then set as the minimal eot of all influencing kernels.

For garbage collection (of sent messages) the LBTS/GVT is calculated as $\min_{\forall i \in \text{influencors}}(\text{nulltime}[i]) - \epsilon$.

Optimistic

The optimistic kernel requires from the hosted model only that copying the state is well-defined, which is provided in the base State class for the user. The kernels use Mattern's [17] GVT algorithm with a maximum of 2 rounds per iteration to determine a GVT. This process runs asynchronously from the simulation itself. Once found, the controlling thread informs all kernels of the new value, which they can use to execute garbage collection of old states and (anti)messages.

The user need only provide one implementation of a model for use with both synchronization protocols. A lookahead function is desired to accelerate conservative, but is not required. In the absence of a user supplied lookahead, the kernel assumes it cannot predict beyond its current time $+ \epsilon$, creating a lockstep simulation.

The implementation details such as defining the copy semantics of a State are provided (but can be overridden).

From the user's perspective, the multi-threaded aspect of the kernel is not exposed.

3.3 Performance Improvements

Continuous profiling of the kernels in several benchmarks highlighted the following key bottlenecks:

Heap

Events are passed by pointer, avoiding a possibly expensive copy of the payload at the cost of allocation overhead.

Allocation becomes prohibitively expensive in highly connected models, so to reduce that overhead we use thread_local memory pools for states and events, and optionally replace the system malloc with calls to tcmalloc[10]. In this way allocating threads do not block each other. If desired, arena-pools are available for single-threaded simulation. A disadvantage is increased complexity in ownership semantics. The creating kernel is responsible for destruction, but this can only be guaranteed up to GVT. Experiments with synchronized pools proved slower than malloc/free.

The kernels initially used strings as identifiers, as is done in PyPDEVS. Profiling quickly indicated this to be a performance bottleneck. C++ strings are heap allocated variable sized objects with an atomic reference count, not immutable objects as in Python. Access of that reference count across threads is expensive, as are the calls to malloc/free the string implementation makes to create/destroy new objects, or copy existing.

Strings are more intuitive to work with from a user's standpoint, so as a compromise the user can reference models/ports by string name. Once simulation starts all objects use integral identifiers for performance. This also increased usage of the constexpr feature of C++11 in, amongst others, timestamps and message headers.

Raw pointers

While an important C++11 feature in general, our initial usage of smart pointers for some types of objects was misplaced. Used across threads, the reference counting becomes very expensive, and the (de)allocating of memory caused significant contention between threads. Models are still held by a smart pointer, as is a kernel, but a message is a raw pointer to compacted memory.

Locking

Locking between kernels uses mostly atomic operations, where we can occasionally leverage memory orderings to only pay for synchronization when we need it. Messages are exchanged via a shared set of queues each with a dedicated lock.

On a higher level, we avoid the sending of synchronization messages entirely by writing the timestamp directly into shared memory.

Sending of antimessages is fairly cheap in our implementation, since only the modified pointer to the original message is sent to the receiving kernel.

Schedulers

PyPDEVS has a wide range of schedulers for the user to choose from, with performance of each depending on the simulation type. Profiling showed in our case that, for a c++

implementation, the heap implementation used in adevs was faster than any of the schedulers we had tested before. Unlike most node based heaps, this scheduler uses a fixed size array where a heap is rebuilt or modified in place depending on the amount of items to update. Items are only updated, never removed.

## 4. PERFORMANCE

4.1 Sequential Simulation
Preamble

The benchmark models are identical in functionality (between adevs/dxex versions). Dxex uses integer timestamps, adevs uses floating point. Tests show there is little difference performance wise (both types fit in a machine word)

Benchmarks ran on a i7-2670QM, 16GiB Ram, Arch Linux (4.2.5 kernel), G++ 5.2. Interpretation of the benchmark results is done using call-graphs generated by perf, a selection of those call-graphs can be found in the code repository. For the runtime plots perf is used in combination with R.

Note that a fixed timeadvance (e.g. devstone) is set at 100.

Devstone

The Devstone [11] benchmark is highly hierarchical, using Direct connection the dxex kernels can exploit this whereas adevs needs to walk the coupled model's structure to pass events.
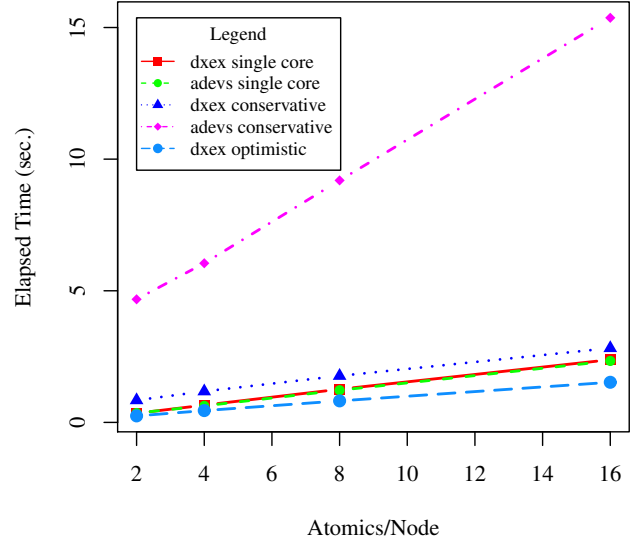
Devstone single core
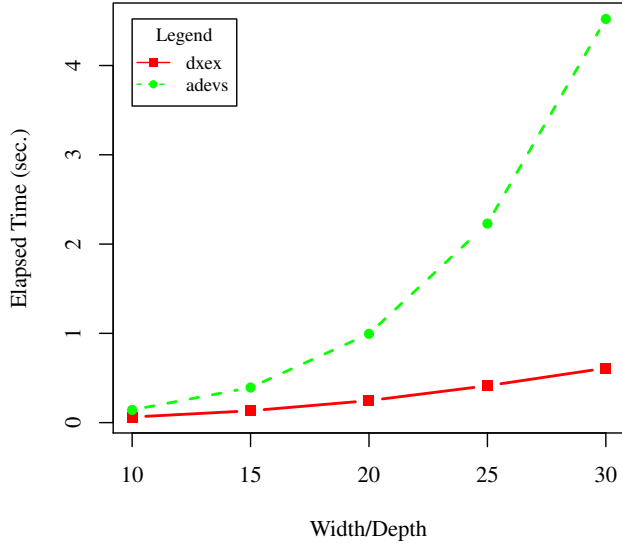
PHold

PHold [7] is a parallel oriented benchmark, sequential runtime is measured only as a baseline. A model selects the destination of a message at runtime using a state-saved RNG, which takes up a non-trivial amount of runtime.

Phold

Interconnect

Interconnect [24] is a benchmark where all models broadcast, creating a complete graph in dependencies between models. As the model count increases, we see the expected quadratic increase in runtime for both adevs and dxex, but an increasing penalty for dxex. Profiling shows this is entirely due to the heap allocation of messages, which even though minimized by using memory pools remains significant.

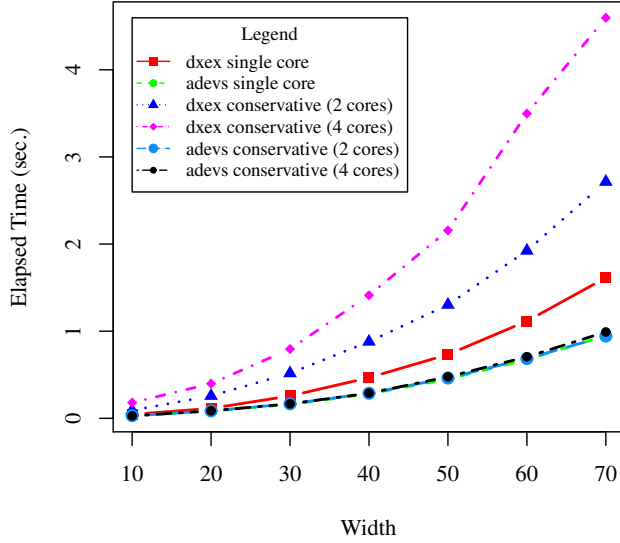### 4.2 Parallel Simulation
Devstone

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 optimistic, this can cause more reverts since the kernels will start to drift faster as the model-count increases. Furthermore, optimistic is quite sensitive to an increase in kernels, since the delay before a revert propagates increases. Of note as well is the warm-up time this benchmark requires, for $n = d \times w$ models, it takes timeadvance$() * (n-1)$ transitions to activate the last model in the chain. For parallel this can be reduced to timeadvance$() * \left(n \frac{kernels-1}{kernels} - 1\right)$ before the last kernel becomes active.

PHold

In Phold allocation is specified in the benchmark itself. Each kernel manages a single node with a constant set of sub-nodes. The parameter R (here 10) determines the percentage of remote destination models.

The dynamic dependency graph is a very sparse version of the static dependency graph, penalizing conservative. Lookahead is ϵ, so conservative spends most of its time crawling in steps

Interconnect

DevStone parallel
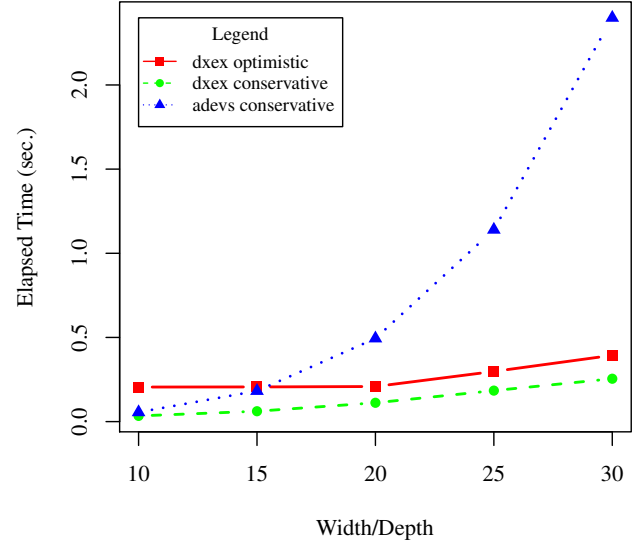
of ϵ. Since the dependency graph between kernels is a complete graph, this is not a simulation that scales in our implementation. For N kernels, each kernel has to query the null-time of N-1 kernels, resulting in $O(N^2)$ polling behaviour. In a non-cyclic simulation with a non-trivial lookahead (e.g. devstone), that choice however does pay off. 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. Optimistic suffers little from the above problems, however due to the high interconnectivity a cascading revert is still possible. With the percentage of remotes = 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. Again, this could be significantly reduced using lazy-cancellation. 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 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, the resulting dependency graph between kernels remains a complete graph. The run-time dependency graph is almost immediately equal to the static graph. Conservative still faces the same issues as in PHold, with the key difference that for a fixed time advance lookahead is equal to the timespan between transitions. The scaling issue is identical as with PHold.

In this benchmark optimistic 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 send.

Priority network model

The priority benchmark is composed of a single server generating a stream of $0 <= m <= n$ messages at fixed time intervals, interleaved with a probability p for a priority message, to n receivers.

This defaults 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 a internal transition so that they will not wait for the incoming messages.
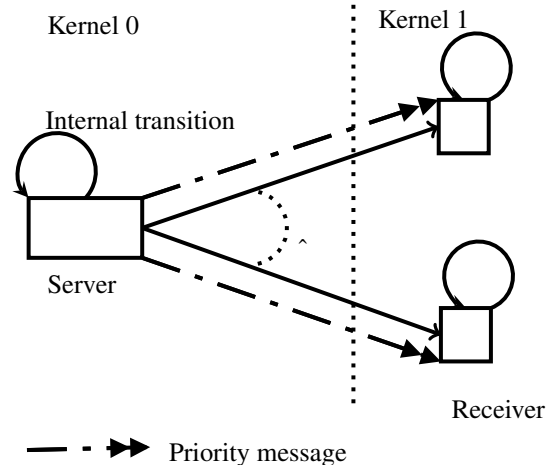
A key difference here with the other benchmarks is that a state (in the Receiver instances) is very cheap to copy/create. The kernel holding the server will never revert since it is a source in the dependency graph. Optimistic will therefore not suffer the same performance hit in recreating states as it does in PHold. The overhead in optimistic is entirely due to the factor m, which will quickly dominate in increasing buffers of received messages. Interestingly, the parameter p does not

Priority
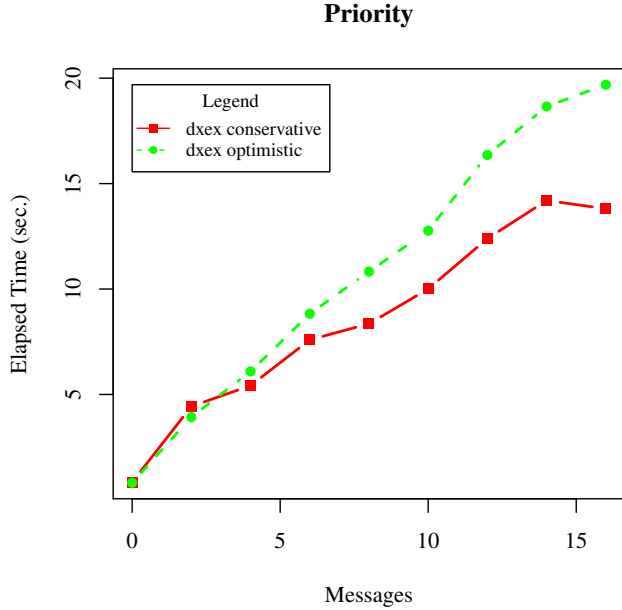
clearly benefit either synchronization protocol. While this removes any possibility for a lookahead, conservative can quickly bridge the timespan between fixed messages since there is no cyclic dependency. Lightweight states prevent performance loss due to reverts in optimistic, only the overhead in event handling in optimistic eventually becomes the deciding factor.

4.3 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. [20] 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).

Measure

Adevs passes messages by value (in container by reference), we pass by pointer. The runtime effects of this choice are already 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 case of optimistic, 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

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

Adevs	AdevsCon	DX	DXCon	DXOpt
44	70	42	75	363

Since conservative passes messages by pointer, it needs a GVT/LBTS implementation to organise 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 conservative. Additionally, 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 severe memory pressure. Unlike conservative and sequential execution, memory usage in optimistic varies greatly depending on scheduling of kernel threads and drifting between kernels.

PHold

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

Adevs	AdevsCon	DX	DXCon	DXOpt
40	x	37	61	682

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.

Table 3. Interconnect w 10 t5e5, unit MiB, 2 kernels (if parallel)

Adevs	AdevsCon	DX	DXCon	DXOpt
39	39	35	43	164

Priority Network

The priority network model is detailed in section 4.1. Conservative's high memory usage 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.

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

DX	DXCon	DXOpt
35	450	651

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 but it should be noted that once the kernels are compiled (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.

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. By virtue of not flattening Coupled Models, adevs's performance degrades in increasingly hierarchical models. Dxex manages lookahead with more overhead than adevs, leading to a performance difference as the model count increases. Finally, adevs employs the full dependency graph, contrasted to 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 our single program, with a non-fragmented code-base, 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), dxex is designed specifically for a shared memory architecture and as such any middleware, while feature-rich, would have lead to unacceptable overhead.

6. CONCLUSIONS

Sequentially dxex outperforms adevs in hierarchical simulations while yielding to adevs in broadcast-like simulations. Parallel the results are more nuanced. The sensitivity to broadcast simulations is carried over in the parallel implementation, resulting in performance loss for both synchronization protocols. Optimistic can still achieve good performance if the volume of inter-kernel messages is low. Both are fast in non-cyclic simulations, with conservative better suited whenever state-saving becomes expensive and optimistic when 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.

6.1 Future work

Memory

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

Activity

As shown in [25] 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 optimistic one can use reallocation 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. There is a limited framework implemented to track model activity for debugging purposes, this could be extended to enable the above strategies.

Hybrid

The optimistic implementation could use (null/eot/eit) from conservative to detect and/or reduce the cost of reverts without completely stalling on influencing kernels. Conservative kernels could be extended with runtime information about influencing models, for example if one can guarantee a static dependency is not used for a fixed time-span, this can be removed for that period of (virtual) time. Ultimately the simulation could switch at runtime between protocols based on the information provided by activity tracking. This requires the above mentioned framework, and could only be done at timepoints where a GVT/LBTS time is agreed upon between the kernels. From the model point of view no changes are needed, although a non-trivial lookahead is obviously desired.

ACKNOWLEDGMENTS

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. Chandy, K. M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 4 (Apr. 1981), 198–206.

4. Chen, B., and Vangheluwe, H. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference* (2010), 209–218.

5. 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.

6. 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.

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. Ghemawat, S., and Menage, P. TCMalloc : Thread-Caching Malloc. **http://goog-perftools.sourceforge.net/doc/tcmalloc.html**, Nov. 2005.

11. 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.

12. Glinsky, E., and Wainer, G. New parallel simulation techniques of DEVS and Cell-DEVS in CD++. In *Proceedings of the 39th annual Symposium on Simulation* (2006), 244–251.

13. Himmelspach, J., and Uhrmacher, A. M. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium* (2006), 239–244.

14. Jafer, S., and Wainer, G. Conservative vs. optimistic parallel simulation of devs and cell-devs: A comparative study. In *Proceedings of the 2010 Summer Computer Simulation Conference*, SCSC '10 (2010), 342–349.

15. Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.

16. Kim, K. H., Seong, Y. R., Kim, T. G., and Park, K. H. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS 13*, 3 (1996), 135–154.

17. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (1993), 423–434.

18. Muzy, A., and Nutaro, J. J. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling and Simulation (OICMS)* (2005), 273–279.

19. Muzy, A., Varenne, F., Zeigler, B. P., Caux, J., Coquillard, P., Touraille, L., Prunetti, D., Caillou, P., Michel, O., and Hill, D. R. C. Refounding of the activity concept? towards a federative paradigm for modeling and simulation. *Simulation 89*, 2 (2013), 156–177.

20. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (jun 2007), 89–100.

21. Nutaro, J. J. ADEVS. **http://www.ornl.gov/~1qn/adevs/**, 2015.

22. OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

23. Troccoli, A., and Wainer, G. Implementing Parallel Cell-DEVS. In *Annual Simulation Symposium* (2003), 273–280.

24. Van Tendeloo, Y. Research internship i: Efficient devs simulation.

25. Van Tendeloo, Y., and Vangheluwe, H. Activity in pythonpdevs. In *Activity-Based Modeling and Simulation* (2014).

26. 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.

27. Van Tendeloo, Y., and Vangheluwe, H. PythonPDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Spring Simulation Multiconference*, SpringSim '15, Society for Computer Simulation International (2015), 844–851.

28. 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.

29. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.