LLNL-JRNL-745586

Generation of Reversible C++ Code for Optimistic Parallel Discrete Event Simulation

M. Schordan, T. Oppelstrup, D. Jefferson, P. Barnes

February 1, 2018

New Generation Computing

Generation of Reversible C++ Code for Optimistic Parallel Discrete Event Simulation

Markus Schordan · Tomas Oppelstrup · David Jefferson · Peter D. Barnes, Jr.

Cite as: Markus Schordan, Tomas Oppelstrup, David R. Jefferson, Peter D. Barnes Jr.: Generation of Reversible C++ Code for Optimistic Parallel Discrete Event Simulation. New Generation Computing 36(3): 257-280 (2018)

Abstract The reversible execution of C/C++ code has been a target of research and engineering for more than a decade as reversible computation has become a central notion in large scale parallel discrete event simulation (PDES). The simulation models that are implemented for PDES are of increasing complexity and size and require various language features to support abstraction, encapsulation, and composition when building a simulation model. In this paper we focus on parallel simulation models that are written with user-defined C++ abstractions and abstractions of the C++ Standard Library. We present an approach based on incremental state saving for establishing reversibility of C++ and an evaluation for a kinetic Monte-Carlo simulation implemented in C++. Although a significant runtime overhead is introduced with our technique, it is an enormous win that it allows using the entire C++ language, and have that code automatically transformed into reversible code to enable parallel execution with the Rensselaer's Optimistic Simulation System (ROSS).

1 Introduction

Reversible computation has become a key concept in optimistic parallel discrete event simulation (PDES) [2,15]. It is essential to achieve high performance for large scale models. In optimistic PDES reversibility is required to perform a distributed rollback in case a conflict is detected due to the optimistic execution.

Jefferson started the subject of rollback-based synchronization in 1984 [9] with presenting the *Time Warp mechanism.* This mechanism performs a distributed rollback by transmitting so called antimessages to "unsend" previ-

Lawrence Livermore National Laboratory

ously sent messages and by restoring a state as a snapshot of an old state. In 1999 Carothers et. al published the first paper to suggest using reverse computation instead of snapshot restoration as the mechanism for rollback [2]. It is written in terms of very simple and conventional programming constructs (C-like rather than C++ -like) and instrumenting the forward code to store near minimal trace information to allow reversing of side effects when needed. That paper considers discrete event simulation as one of several applications of virtual time, but in fact it was then and is now the primary application. Although the term "virtual time" is used, one can safely read it as "simulation time".

Barnes et. al demonstrated in 2013 [1] how important reverse computation can be in the practical application area of optimistic PDES. In [1] a thorough historical investigation was presented of all published results on the PHOLD benchmark from 1995 to 2013 and new results were presented with the highest event rate of 504 billion events per second achieved at Lawrence Livermore National Laboratory on Blue Gene/Q in 2013. The reverse code was handwritten, and methodologically we know that this is unsustainable for large models involving several libraries, highlighting the need for a way of automatically generating reverse code from forward code. This is what we address with the work presented in this paper - to have a tool available, Backstroke, for establishing the reversibility of C++ code.

One of the major challenges in reversible computation is to avoid information loss in the forward computation. In a trace-free reversible language the reversibility of a program (backward determinism) is ensured, among others, by assertions at join points of the control flow [24]. In the absence of such assertions, auxiliary data structures are needed to trace the lost information. Optimized tracing schemes, such as incremental checkpointing, also called incremental state saving, are described by Perumalla [15]. In this paper we present an approach for applying incremental state saving such that it allows to establish reversibility for any C++ code.

We will briefly describe reversible computation and its use for optimistic parallel discrete event simulation, and then describe how we produce reversible code from ordinary, possibly non-reversible, code written in standard C++. We go beyond our previous work in [21] by (i) also using C++ Standard Library abstractions for the implementation of the discrete event simulation model, (ii) start with a model code where all additional code necessary in hand-written reverse code to preserve destroyed information is not available, and (iii) present an evaluation of the simulation model running on a different hardware architecture and for a much bigger problem size as previously. This also demonstrates the advantage of our source-to-source approach in comparison to binary instrumentation [3,5], as we can move to a different hardware architecture without adapting our system to a new hardware architecture. The overhead of instrumentation is addressed in [4] by also allowing a runtime decision with multi-coding to determine if the instrumented code is not suited (because its cost does not pay off), and an alternative method to state reconstruction based on coasting forward should be instead exploited. This decision can be taken at run time depending on the variation of the workload. In this paper we focus on the instrumentation approach, but a combination with those other methods is an interesting option to reduce overheads. Another alternative is to implement the generation of reverse handlers for a lower-level representation in a compiler, such as the LLVM, allowing to leverage all existing backends of a given compiler infrastructure for code generation. This was demonstrated with LORAIN [10] in 2014. In some scenarios, such as the use of shared libraries for which the source code is not available, combining our approach with approaches operating on binaries [3], would allow to combine the advantages of both approaches.

Another approach is the use of a reversible programming language. The reversible programming language Janus [25,23] also allows to generate C code for the forward and reverse code. This allows to combine implementations in C/C++ with reversible functions that are implemented in Janus. For the presented model this is not an option because most of all computations involve floating-point operations and floating-point arithmetic is inherently irreversible due to rounding and truncation at run-time. Reversing floating-point operations without accumulating errors requires some form of tracing. We consider such a combination in future work with a PDES model that operates on scalar types only.

At the source-level the challenge is to address the full range of C++ language constructs, and as we use the C++ Standard Library's classes and functions we also need to ensure that language extensions and built-ins used in these headers still allow to establish reversibility of C++ with our approach. In comparison to reverse code generation the advantage of our approach is that it suffices to only instrument all forms of assignments and memory allocation and memory deallocation operators, whereas all other language constructs can remain as in the original program.

Since the change in hardware architectures, from one generation to the next is a daunting aspect of high-performance computing, this paper also demonstrates a key aspect of our work in combination with previous work [20,21] which we want to emphasize with the presented new results: our approach allows to run models on various hardware architectures. For users it is also relevant that the model itself can be developed on any computer having available a C++ compiler and MPI for message passing, because the compiled simulation code can be run on a laptop as well as on a super computer.

1.1 Discrete Event Simulation

Discrete event simulation (DES) is a simulation paradigm suitable for systems whose states are modeled as changing *discontinuously* and *irregularly* at discrete moments of simulation time. State changes occur at simulation times that are calculated dynamically rather than determined statically as in time-stepped simulations. Most irregular systems whose behavior is not describable by continuous equations and do not happen to be suitable for simple time-stepped models are candidates for DES. Efficient *parallel* discrete event simulation (PDES) is much more complex than the sequential version. There are two broad approaches to resolving the PDES synchronization issue, called *conservative* and *optimistic* [6]. We will discuss optimistic PDES, which requires reversibility, in more detail in the next section. Recently Omelchenko and Karimabadi have developed an asynchronous flux-conserving DES technique for physical simulations [11]. Their preemptive event processing approach to parallel synchronization complements standard optimistic and conservative strategies for PDES.

1.2 Optimistic Parallel Discrete Event Simulation

The general approach to PDES is to divide the simulation and its state into semi-independent parallel units called LPs (logical processes) that generally execute concurrently and communicate asynchronously. Each simulated event (state change) is executed within one LP and affects only that LP's state. Any event may schedule other events for future simulation times. Events scheduled for other LPs must be transmitted to them as event messages with a timestamp indicating the simulation time when the event should be executed. Arriving event messages get enqueued in the event queues of the receiving LPs in increasing time stamp order.

What makes PDES so complex is the synchronization problem. Every LP must execute all of its events in strictly non-decreasing timestamp order despite the fact that it does not know in what order events may arrive or what timestamps they may carry. At any hypothetical global snapshot of the parallel simulation taken at a single instant of wall clock time some LPs will be ahead in simulation time and some will be behind, and which LPs are ahead or behind may change during execution.

Conservative synchronization uses conventional process blocking primitives along with extra knowledge about the simulation model (called *lookahead* information) to prevent the execution from ever getting into a situation in which an event message arrives at an LP with a timestamp in its past. Conservative synchronization is limited to models with static communication graphs.

Optimistic synchronization, by contrast, allows dynamic communication graphs. As a result, there is a danger of a *causality violation* when an LP that is behind in simulation time, e.g. at t_1, sends an event message with a (future) timestamp $t_2 > t_1$ that arrives at a receiver that has already simulated to time $t_3 > t_2$. In that case the receiver has already simulated past the simulation time when it *should* have executed the event at t_2, but it would be incorrect to execute events out of order. Whenever that occurs, the simulator rolls back the LP from t_3 to the state it was in at time t_2, cancels all event messages the LP sent after t_2, executes the arriving event, and then re-executes forward from time t_2 to t_3 and beyond. All event executions are therefore *speculative* or *provisional*, and are subject to rollback if the simulation gets into local causality trouble. Most of the time that does not happen and the simulation proceeds forward in parallel.

Each LP computes its local virtual time (LVT) based on the time stamps of event messages it receives. Because of rollbacks the LVT can also be reset to an earlier point in time. The global virtual time (GVT) is defined to be the minimum of all of the LVTs. Several algorithms exist to compute an estimate of the GVT during the simulation. Any events with time stamps older than GVT can be *committed* because it is guaranteed that they never need to be reversed. For more detail see [15,9]. That events are committed once they are older than GVT, allows to delete all information that may have been stored to enable reversibility.

1.3 Motivation for Generation of Reversible Code

In this paper we assume the parallel simulation model is written in C++. Each event is the execution of some event method $E()$ that makes changes to the state variables of the simulation. If that event has to be rolled back to deal with a causality violation, then the simulator needs a way to exactly reverse all of the side effects of $E()$ to return the simulation to the exact state it was in before $E()$ was executed.

However, a C++ method $E()$ will generally destroy information during its forward execution. It will usually overwrite or update some state variables, destroy control information (e.g. by $forgetting$ which branch it took at a conditional), and may also delete data structures on the heap. It is not possible in general to write a reverse function $E'()$ that can restore information that was actually destroyed by $E()$. But we can frame the problem differently and achieve our purpose.

Instead, for an event method $E()$ written in C++ (with return type void), we generate a derived method, $E^+()$ that is identical to $E()$ except that it is instrumented to save in an auxiliary data structure a trace of all of the information that $E()$ would destroy. The simulator uses that saved trace information to undo the side effects of $E^+()$, and it also destroys the auxiliary data structure that $E^+()$ created.

If $E()$ does any memory deallocation, we do not actually do the deallocation in $E^+()$ since it cannot be reversed. Instead we defer the actual deallocation of an object (but not the call to its destructor) to be done at *commit time* when we can be sure that $E^+()$ will never need to be reversed. Once the entire simulation has progressed (on all nodes) beyond a certain simulation time t and there is no message in the network with a receive time $< t$, it is guaranteed that no event at a simulation time $< t$ will ever need to be reversed. At this point the commit function is called for all events at simulation times earlier than t. I/O and certain other issues are also handled at commit time, but they are beyond the scope of this paper.

Our approach is also described in general by K. S. Perumalla in [15] (p.132) as incremental check pointing “Among the checkpointing schemes, incremental checkpointing is in general the most efficient scheme, but is also one of the more challenging ones to implement." - also pointing out that incremental checkpointing can severely interfere with cache behavior and introduce a significant performance penalty in particular in cases when entire data structures are modified. In combination with the above mentioned commit time, it is an instance of the forward-reverse-commit paradigm as described in [15] (Chapter 7.3). An event can be reversed if necessary, or it is committed once it is certain that it never needs to be reversed.

In [18] a method is presented to implement incremental state saving in simulation kernels based on C++. In contrast to our approach, this method is not completely transparent. In particular, the user has to explicitly declare state variables as `State<>` or `RefState<>` objects and might have to explicitly make some casts which were not previously needed.

Our work significantly goes beyond the scope of our work in [21] which excluded the use of C++ Standard Library abstractions. The complexity that comes with using the C++ Standard Library implementation in the model's implementation is that we need to transform all templates and types provided in C++11 Standard header files and ensure that these reversible types can also be used by code that is not supposed be reversed (e.g. Backstroke's C++ runtime library implementation itself). We demonstrate our complete approach to C++ by applying it to a model in a scalable kinetic Monte-Carlo C++ application. The results presented in this paper are based on version 2.1.2 of our tool Backstroke that can generate reversible forward code for the given model code without any intervention or requirements on the user. To the best of our knowledge, this is the very first paper that presents the transformation of a full C++ simulation model using also C++ Standard Library abstractions and also transforms all used system headers. In [21] we transformed C++98 code and excluded the use of the C++ Standard Library.

In this paper we apply our approach also to C++11 Standard Library classes and functions in the C++11 header files and investigate the use of Backstroke in a more general setting: combining reversible (transformed) code with non-reversible code and also running reversible code in reversible and non-reversible *mode*. It is the combination of these features that makes our approach applicable to a wide range of use-cases. We discuss these features in detail when describing the use of the C++ Standard Library, calling ROSS simulator functions from our model code for sending MPI messages (i.e. communication between nodes of BlueGene/Q), and also the shared use of the C++ Standard Library abstractions in our model and in code that is not supposed to be reversed. For example, the data structures that are used to record all modifications of memory by Backstroke generated code at runtime, are implemented using C++ Standard Library abstractions but are not supposed to be reversed, instead they are only used to achieve reversibility.

In Section 2 we present our approach to generating reversible C++11 code. In Section 3 we describe the kinetic Monte-Carlo simulation model and in Section 4 an evaluation of the optimistic parallel execution on our IBM BlueGene/Q supercomputer with the ROSS simulator. All reversible code used to implement the reverse function as required with the ROSS simulator, is generated automatically and no user intervention is necessary. In Section 5 we present the related work. In Section 6 we conclude on the observed performance and what future potential we see for further performance improvements.

2 Generation of Reversible Code

Our approach is a variant of incremental state saving (also called incremental check pointing) and the forward-reverse-commit paradigm as described in [15] (Chapter 7.3). This paradigm allows us to address situations in which a program fragment can be executed optimistically “ahead of time”, but is found to be incorrect (“too optimistic”), and requires re-execution from a previous state of execution.

This situation occurs in parallel discrete event simulation with optimistic synchronization. We need to reverse events if they turn out to be not on the correct execution path (e.g. because an event that was transmitted on the network arrives with a timestamp that is older than the one that has already been simulated). After the event has been reversed we can then re-execute the event, but also taking into account the events that had occurred with an older time stamp.

Our approach requires that we transform only the forward event function. The forward code is transformed such that it records additional information in a data structure that is used by the reverse and commit methods. No code is generated for the reverse and commit method. They share the same implementation for all variants of transformed forward event codes. We have implemented our approach in a tool called *Backstroke* as source-to-source transformation based on the compiler infrastructure ROSE [17].

In the following sections we describe the code transformation operations, recorded data at runtime, and how the recorded data is used by the reverse and commit methods in the following sections.

2.1 Code Transformations for Intercepting Memory Modifying Operations

For our approach it is sufficient to intercept all memory modifying operations. Measured in bits, we may store more information than is actually destroyed, but we do not need to store control flow information, because we only restore the sequence of memory locations that have been modified through assignments in one execution of the event method. When the execution of the event method is reversed, we restore the memory locations in reverse order of their modification. For this purpose of restoration, it is irrelevant which execution path was executed by the event method. One can also consider this to be an execution trace of all addresses and their old values before they are overwritten by an assignment. By restoring all those memory locations to their previous value, we can restore the program state before execution of the event function. In addition, we also undo all memory allocation when the effects of an execution are reversed.

This approach allows us to address the effects of memory modifications for all of C++. With C++ come a number of language constructs in addition to the language constructs in C that make the generation of reverse code that can be *executed* in reverse very difficult. This applies in particular to C++ exceptions which are used in many modern C++ codes.

We consider three kinds of memory modifying operations: assignment operators, memory allocation, and memory deallocation. C++ offers 15 different assignment operators, which can modify the memory for all built-in types, two kinds of operators for memory allocation (single object and arrays) and two kinds of operators for memory deallocation (single object and arrays). The number of variants of the **`new`** and **`delete`** operator are increasing with each new standard of C++. In this paper we focus on C++11 and address the C++11 variants of these operators. The only C/C++ language construct not addressed yet are bitfields, since they require a different transformation as the address operator cannot be applied to bitfields in C/C++.

In the following section we define the transformations for all the C++ operators that must be transformed in the forward method.

2.2 Forward Code Generation for Assignment Operators

C++ offers 15 memory modifying operators, eleven variants of assignment and four variants of increment/decrement operators:

1. Assignment: E_1 **`=`** E_2
2. Assignment with additional operation: E_1 *op* E_2
   where *op* ∈ {**`+=`**, **`-=`**, **`*=`**, **`/=`**, **`%=`**, **`&=`**, **`|=`**, **`^=`**, **`<<=`**, **`>>=`**}
3. pre/post increment/decrement operators: *op* E, E *op* where *op* ∈ {**`++`**,**`--`**}

For each of the 15 operators we apply a transformation that enables us to record the old value stored at the address that is modified by an assignment and the address itself. The code transformation of assignments is only applied to operations on built-in types. For our approach it is important to be aware of the fact that memory can only be modified through built-in types. In C++ the assignment of user-defined types is defined by a default assignment operator (see Section 2.6). If the default assignment operator is user-defined we transform this implementation in the same way as all other forward code. If the default assignment operator is not user-defined, we generate a reversible default assignment operator. This way, the assignment of user-defined types is addressed as well. In C++11 two kinds of assignment operators exist. If a default C++11 copy assignment operator is not provided by the user, we generate a reversible one. If a user-defined C++11 move assignment operator exists, it is transformed; analogous for the copy assignment operator. If a C++11 move assignment operator is not defined, it defaults to using a default

C++11 copy assignment operator, for which we generate a reversible variant. The C++11 move semantics might require further refinements, but we are not aware of any codes where reversibility is not established with above rules.

For assignment operators we define the transformation α which intercepts all 15 forms of C++ assignments. The code that is introduced by this transformation is applied to all 15 kinds of assignment operators as a unified operation (including pre- and post-increment/decrement operators). The transformation α is introduced as follows for the different kinds of assignment:

1. E_1 `=` $E_2 \Longrightarrow \alpha(E_1)$ `=` E_2
2. $E_1\ op\ E_2 \Longrightarrow \alpha(E_1)\ op\ E_2$
3. $op\ E \Longrightarrow op\ \alpha(E)$, $E\ op \Longrightarrow \alpha(E)\ op$

For example, let `p` be a pointer to an object and `x` be a data member of this object. Then an assignment of `y` to this member variable can be written as `p->x=y`. The transformation α instruments the left-hand-side of the assignment. The right-hand-side of the assignment remains unmodified as we only require the address of the left hand side of an assignment to access the old value (i.e. the value before assignment). Hence, we perform the transformation α(`p->x`)=`y`. The transformation α introduces a call to a function in the Backstroke library that takes as argument a reference parameter and returns the very same reference. This allows us to keep this transformation local to expressions and we never need to transform any control flow or normalize code. An example of a transformed code involving assignments and memory allocation is described in Section 2.5.

2.3 Addressing Dynamic Memory Allocation

To address memory allocation we introduce transformation β, and for deallocation we introduce transformation γ. This concept is essentially the same as described in [15] (Chapter 13), but extended for C++ constructors and destructors, and in particular for C++ array allocation and deallocation. The correctness proof provided in [15] also applies for our approach. The only difference is that we split the execution of the destructor from the actual deallocation. In contrast, constructor calls are not separated from the actual allocation. We only need to store the address of the allocated object.

Let T be a built-in type or a user defined type (class, struct, or union) and E be an expression. Then for every occurrence of the operators **`new`** and **`delete`** in a program, we introduce the following transformations

1. **`new`** T**`()`** $\Longrightarrow \beta_1$(`T`).
2. **`new`** T**`()[`**E**`]`** $\Longrightarrow \beta_2(T,E)$.
3. **`delete`** $E \Longrightarrow \gamma_1(E)$.
4. **`delete[]`** $E \Longrightarrow \gamma_2(E)$.

For array allocation the expression E specifies the size of the array. For the C++ **`delete`** operator (e.g. **`delete`** x, where x denotes the address of an object) we generate code that invokes the destructor of the respective type, store the address of the object to be deallocated, and defer the actual memory deallocation to commit time (i.e. the memory of the object is deallocated when the commit function in the Backstroke library is invoked by the simulator).

2.4 Semantics of the Transformation Operators for Memory Allocation and Deallocation

Array deallocation has the additional complication that we also need to know (i) the size of the array (which is not explicitly provided in the source code), (ii) apply the destructor for each array element in reverse of the order constructors were called in the forward code, and finally (iii) to deallocate the memory allocated for the array and the memory location storing the size.

We completely replace the array allocation and deallocation by our own allocation scheme. Our allocation scheme adds a word for storing the size of the array. The pointer returned by our allocation function refers to the actual memory behind the added size field. This mimics the same behavior as the code that is usually generated by C++ compilers, where the size of an array is stored in addition to the array elements. However, it is not standardized where exactly this size information is stored by a compiler generated code, therefore we need to maintain this separately. This enables us to ensure that the call to the destructor and the actual memory deallocation can be separated and it is also compiler independent.

2.5 Example: Original and Transformed Model Code Fragment

In Figure 1 we show a Backstroke transformed code. It is a code fragment from the std::vector type used by our model code and the allocator and deallocator functions used in STL new_allocator class. In our model we use various C++Std data types (e.g. vector, deque, pair and iterators). We use our own user-defined type for the map implementation. The reason is that std::map is only partially implemented the C++Std library headers, but also has parts of its implementation in the linked C++Std library. This would require to also transform the entire Standard Library implementation and link with this instrumented Standard Library instead of the installed Standard Library. Since this would impact the performance of all C++ functions, including I/O functions, we use our own map implementation. In future work we will also consider to transform the entire C++ Standard Library code that is provided in the shared library. In this paper we describe our approach for transforming all code that comes with the C++ Standard Library headers.

The example fragment contains assignments, memory allocation, and memory deallocation. In general, assignments to local variables are not instrumented because they are auto-allocated in C++ and destroyed after execution of a function (usually allocated on the stack). All instrumentations are implemented as functions in the **`xpdes`** namespace and therefore easily visible in the

Listing 1: Reversible Forward Code

```
   template<typename _Tp>
     class new_allocator
     {
     public:
     ...
     pointer
       allocate(size_type __n, const void* = 0)
       {
 if (__n > this->max_size())
   std::__throw_bad_alloc();
 return static_cast<_Tp*>((xpdes::registerOperatorNewT(::
       operator new(__n * sizeof(_Tp)))));
       }
       void
       deallocate(pointer __p, size_type)
       { xpdes::operatorDeleteT(__p); }
     }
...
template<typename _Alloc>
  class vector<bool, _Alloc> : protected _Bvector_base<_Alloc
       >
   {
     ...
     public:
     ...
       push_back(bool __x)
     {
       if (this->_M_impl._M_finish._M_p != this->_M_impl.
            _M_end_of_storage)
         (xpdes::avpushT(*(xpdes::avpushT(this->_M_impl.
              _M_finish))++)) = __x;
       else
         _M_insert_aux(end(), __x);
     }
}
```

Fig. 1: Example code fragment from the Backstroke instrumented code of the C++Std Library, showing the allocate and deallocate functions of the class new_allocator and the vector class implementation of the function push in the vector header file.

generated code. The α transformation introduces the `xpdes::avpushT(used)` function call. For the **`new`** operator the transformation β is applied. It introduces the function call `xpdes::registerOperatorNewT` for the operator **`new`** in the global namespace (allocating memory of the provided size) and `xpdes::registerAllocationT` for the operator `new` that can be invoked for a specific type. The transformation γ introduces similar functions for deallocation. This Backstroke runtime library function introduced by γ records the pointer of the object memory to be deallocated and defers the deallocation until commit time. It also invokes the destructor of the respective class explicitly without deallocating memory.

2.6 Default Assignment Operators of User-Defined Types

C++ allows to implement a user-defined assignment operator for any user-defined type. A user-defined assignment operator function is invoked whenever an assignment operator is used for this user-defined type. This way alternative semantics for assignment can be implemented. We utilize this C++ feature by generating reversible default assignment operators, if the operator is not provided by the user. In this case the compiler generates a default assignment operator, but since we require a reversible assignment operator we generate an alternative implementation. A provided user-defined assignment operator replaces the default implementation. If the user provided an assignment operator we transform the existing one (like any other function).

2.7 Code Generation

The generated code contains calls to Backstroke library functions that have been introduced. All transformations are local expressions. Backstroke's Runtime Library is linked against the transformed forward code. The execution of the forward code stores all data necessary to restore any previous state in the computation of the forward function. All data is maintained in data structures of Backstroke's runtime library.

2.8 RTSS Operations

Backstroke's runtime library uses a Run Time State Storage (RTSS) to manage the data necessary to restore states to support reversibility. Since our approach follows the Forward-Reverse-Commit paradigm, the essential data structure used internally in the RTSS is a double ended queue. The forward code pushes data on one end of the queue, and the reverse code pops data from the very same end of the queue. The commit function, in contrast, pops data from the other end. Note that we maintain a data queue for each event invocation, and thus, after restoring an event its data queue is guaranteed to be empty after reversal or commit. Note that maintaining this order is important in cases where data for the same memory location is restored multiple times (because multiple writes to the same memory location were recorded).

In the reverse method we deallocate memory that has been allocated, but may also restore data in this very same memory. Hence, the order of data restoration and undoing of memory allocation (i.e. deallocation) must be maintained in the reverse function. The commit function only disposes data stored by the forward function and performs the deferred memory deallocation (as a result of the `delete` operator). Since deallocated memory blocks cannot overlap, and no data is restored in the commit function, the order is irrelevant - however, it is conceptually clean to perform it in the same order as the forward function, as memory deallocation is a deferred operation. We do not consider input/output operations in this paper, but the order is also significant when completing deferred output in the commit function. Therefore, we consider a double ended queue as the appropriate abstraction for maintaining data for the forward-reverse-commit approach. Handling of input and output (currently only performed in the initialization of model data and after the end of the simulation) is the subject of future work.

In summary, the essential operations in the RTSS for supporting Backstroke generated reversible forward functions in combination with the reverse and commit functions which are provided by the RTSS, are shown in Table 1.

Original	Forward	Reverse	Commit
new	register obj allocation	deallocate obj	dispose obj allocation
delete	register obj deallocation	dispose obj deallocation	deallocate obj
l=r	store (addr(l),val(l))	restore (addr(l),val(r))	dispose (addr(l),val(r))

Table 1: Summary of operations performed by the Run Time State Storage (RTSS) in the Backstroke Library.

In addition, the RTSS offers API functions to turn the recording of data on or off. The Backstroke Library uses this feature also itself by turning off recording of data when storing data in C++ Standard Library data structures (e.g std::deque), because its own data structures are not supposed to be reversed. We also use this feature in an interface to the ROSS simulator. This feature also allows to manually optimize code by reducing the overhead for pure functions which do not change the state of the simulation. We have applied this for one function (nextflip) in the KMC model - although the impact is minimal, it may have a good potential for users to optimize their code manually, but without writing reverse code. However, it does require to understand reversibility.

2.9 Transformation Statistics for the KMC Model

In Table 2 we show the transformation statistics for the KMC simulation model that we evaluate in this paper. We show results for the optimized version where we detect local variables, which are guaranteed not to be state variables. Local variables are straight forward to detect in C++ based on their declaration. Note that this does not apply to data structures *referred* to by local variables which may have been assigned an alias to shared data from an input parameter. For those, precise static analysis can help to further reduce instrumentation overhead. In C++ const declarations can be further leveraged as well, but an analysis also needs to take const casts (used for eliminating declared constness) into account. We consider these methods for future work, and present results with optimized instrumentation overhead for local variables. However, note that instrumentation only causes data accesses being recorded if the corresponding data is stored in the heap (see section 2.11 for more details).

For local variables we do not need to record any information, nor do we need to check whether it is allocated on the stack or heap, since all local variables are stored on the runtime stack. We can therefore eliminate the instrumentation for such assignment. The columns T1-T6 show the number of instrumentations by language construct as follows

T1 : Assignment operator expression statements (e.g. `x=y;`)
T2 : Object memory allocation with operator **`new`**
T3 : Array memory allocation with operator **`new[]`**

T1	T2	T3	T4	T5	T6	Total
656	4	5	4	3	158	830

Table 2: Overview of the number of applied transformations on the KMC Model code including the used C++Std library headers for the six different transformed language constructs.

T4 : Object memory deallocation with operator **`delete`**
T5 : Array memory deallocation with operator **`delete[]`**
T6 : Default assignment operators that are not implemented by the user and for which a reversible version is generated.

In the last column we show the total number of transformations. For the KMC model and the included C++11 Standard header files `std::vector` and `std::deque` (and all headers included by those) the pre-processed original file has 27154 lines of code, the Backstroke generated code has 28236. Thus, Backstroke generates 1082 additional lines of code for 158 default assignment operators, that's about 7 lines per default assignment operator function. For data members of type array loops are generated to copy all array elements. The transformation of the entire code takes 22.5 seconds with Backstroke 2.1.2.

2.10 C++ Templates

Any modern non-trivial C++ code involves templates. Since our transformation is performed source-to-source we need to take templates into account. For each type used in a template the compiler generates a separate instance of the template with the parameter replaced by the actual type. We transform the original template definition without considering instantiations. We discuss this in detail for the assignment operator and why this is guaranteed to be correct in combination with the specific semantics implemented in the Backstroke library functions.

An assignment that occurs in the template function can be either instantiated for (a) a built-in type, or (b) for a user-defined type. For a built-in type (e.g. `int`) the left hand side of the assignment must be passed to a Backstroke library function to record its address-value pair to support reversibility. For a user-defined type we do not need to record the address-value pair for the left hand side variable, because any actual memory-modifying assignments will occur in the underlying reversible assignment operator for that type.

Since we transform the original template (and not the instantiated code) the code transformation is the same for both cases. Here C++11 predicates, which determine at compile time whether a type is a built-in type or a user-defined type, come to rescue. The implementation in the Backstroke library uses these predicates such that the C++11 compiler can determine at compile-time whether the provided type to the library function `avpushT` is a built-in type or a user-defined type. For the user-defined type the function performs a no-op (i.e. fall-through branch). For built-in types code is compiled such that the address-value pair is stored. Note that transformed assignment operators contain an `avpushT` for each data member of the user-defined type. The use of these conditional C++11 compile time predicates in the runtime library enables us to keep the required source-to-source transformations simple. Note that operators for built-in types cannot be overloaded in C++. That is why we need to perform a source-to-source transformation (otherwise it would suffice to implement reversible assignment operators for built-in types).

Memory allocation and deallocation operators are independent of the assignment operator semantics. They are only involved when the result of a **`new`** operation (i.e. a pointer to allocated object memory) is assigned to a variable. Assignment of values and memory allocation and deallocation is treated orthogonally by the separate transformations α, β, and γ.

2.11 Stack vs Heap - The Necessary Check

The Backstroke Runtime Library must ensure that data stored on the runtime stack of the event is not restored in the reverse function because once the event function has been executed, all elements on the event function's runtime stack are popped from the stack by the C++ runtime system. Therefore a compiler/system dependent test is performed for each address-value pair that are about to be stored in the Run Time State Storage (RTSS). If the pointer refers to a stack address, then the pointer is not stored in the RTSS. This is necessary, because otherwise we would restore memory (in the reverse function) that is no longer valid because the stack frame associated with a previously-executed function is no longer valid (or possibly overwritten). On the other hand, if it is a heap pointer, then the pointer is stored in the RTSS and the reverse (or commit) functions use this information to perform the proper operations to restore (or remove) the memory state.

Hence, no information about stack allocated memory is ever stored in the RTSS, only *modified* heap allocated memory is duplicated in the RTSS.

The Backstroke Runtime Library is initialized at the beginning of the simulation. For our model this also involves initializing the simulation with data read from configuration files. Hence, the code performing Input/Output (similar for generating the results of the simulation) is not part of the event function and therefore not reversed.

The Backstroke Library stores the current start address and end address of the thread's stack (currently we use POSIX pthread library function calls to determine the stack's start address and the length of the stack).

3 C++ KMC Model

In order to verify robustness and correctness of Backstroke, we have chosen to apply it to a grain evolution simulation using a real world parallel kinetic Monte-Carlo code written for crystal kinetics. The code is named SPOCK

(Scalable Parallel Optimistic Crystal Kinetics) and is built on the ROSS discrete event simulator by C. Carothers et al., which uses the Time Warp algorithm to achieve a scalable and robust parallelization. The following sections introduce the ROSS simulator, the kinetic Monte-Carlo method, and the grain evolution application.

3.1 ROSS Simulator

ROSS is a general purpose discrete event simulator developed at RPI by C. Carothers et. al. [7]. Within ROSS, a simulation consists of a set of logical processes (LP's) that communicate with each other through time-stamped event messages. A discrete event process formulated in this way is called a ROSS model. To implement a new model in ROSS one needs to write an initialization function that sets off the initial state of each LP, and an event function which is responsible for processing a received event message for a given LP. The event function also has the opportunity to send further event messages.

After initialization the simulation logically progresses by processing any event messages in simulation time order using the provided event functions for the LP's.

ROSS has been developed over more than 10 years, and is mature software. It has the capability of running simulations in parallel using either conservative or optimistic synchronization. The Time Warp mechanism is the versatile and scalable option in ROSS for running simulations in parallel. Time Warp is an optimistic approach, where each processor employs speculative execution to process any event messages it is aware of. Causality conflicts, such as when a previously unknown message which should already have been processed is received, are handled through local roll back. During roll back the effects of messages that were processed in error are undone.

In order to use Time Warp in a ROSS model, a reverse event function must be provided. The reverse function is responsible for undoing the state changes that the forward event function incurred for the same message.

The power of Backstroke in this context is that it can automatically generate the appropriate reverse event function for a given forward event function. For complicated discrete event models, this greatly increases productivity since less code needs to be written by hand. In addition, it is much less error prone and reduces the code maintenance burden, compared to hand written reverse code. It can not be stressed enough that even a very minute bug in the reverse code, so that it only almost reverses the effect of the forward event code, is disastrous for Time Warp simulations. Small state differences can make the code address out of bounds, cause exceptions, and result in infinite loops.

3.2 Model: A Parallel Kinetic Monte-Carlo Application

We have applied Backstroke to the event processing code of SPOCK (Scalable Parallel Optimistic Crystal Kinetics), a parallel kinetic Monte-Carlo application for simulation of growth and morphology evolution of crystals.

We will here give an overview of the kinetic Monte-Carlo method, and of crystal evolution simulations. After that we will describe the application of Backstroke to SPOCK and give some performance metrics.

In the kinetic Monte-Carlo method, a physical system is modeled as a set of interacting objects, and their evolution is described by a sequence of discrete events that affect the set of objects.

In our crystal grain simulation, we model a piece of solid as a grid of unit elements. Each unit element represents a microscopic piece of material, big enough to be able to exhibit a well defined crystal orientation, but much smaller than typical grain sizes. These unit elements are commonly called spins, since the nature of grain evolution resembles evolution of magnetic domains, commonly studied by the Ising model in which unit element has a binary state: spin up or down. A spin that has at least one neighbor of a different orientation than itself is defined a boundary spin, and the set of these spins comprise the grain boundaries. It is the evolution and morphology of these boundaries that is of interest in grain evolution simulations.

An event in the grain evolution model is a change in crystal orientation of a spin. The time of the change, and the next orientation of a given spin are given by stochastic variables whose distributions depend on the orientation of the neighboring spins. Typically, it is favorable for a spin to choose a new orientation that maximizes the number of neighbors with that same orientation.

A high level description of a grain evolution algorithm is as follows:

1. Initialize the orientations of all spins.
2. For each spin, sample an event from the corresponding time and orientation distributions. This event is inserted in the event queue, and serves as a putative event for this spin.
3. Execute the earliest event in the event queue, and advance the simulation time to that of this event.
4. For all spins whose time and orientation distributions changed due to the executed event, retract their current putative events, and sample new ones. Sample a new event also for the spin whose event was just executed. Insert these events in the event queue.
5. If the earliest event in the event queue is past the end time of the simulation, stop. Otherwise, go to 3.

Some notes about this algorithm: For short ranged interactions, in 2D each spin typically has 4 or 8 neighbors, while in 3D this number is often 6 or 26. For longer range interactions the number of neighbors can be much greater. Typically, at each executed event, as many events as the number of neighbors are retracted, and new events are sampled. This means that the vast majority of scheduled events are retracted. This is in contrast to most event driven simulations where retractions are rare. The algorithm description above allows plenty of room for optimization in these grain simulations: often, new orientations are restricted to those present in the neighbors. Therefore, spins in the bulk of a grain, i.e. one which as the same spin as all its neighbors, can not change, and thus no events need to be generated or kept for these spins, which can be the vast majority of spins in the simulating. This saves both memory and computational resources.

4 Evaluation

As a case for our evaluation of Backstroke, we have chosen to use the 2D Potts grain growth model, where the spins are vertices in regular Cartesian grid on a 2D torus, and each spin interacts with its 8 nearest neighbors. The possible orientations, or spin values, are represented by integers, and a spin can only change orientation to that of one of its neighbors.

The event code was not written with Backstroke in mind, and is a relatively complex C++ code making full use of e.g. templates and overloaded operators. The forward event code size is around 1800 lines including the implementation of all used data types.

To run our grain evolution tests, we choose two system sizes: A smaller system with 192×192 spins divided into a grid of 24×24 LP's, which we run for 10 simulation time units, or a total of 1053890 events, and a bigger system with 1536×1536 spins in 256×256 LP's. We run the big system for 2 time units, or a total of 47633718 events.

The tests were run on an IBM BlueGene/Q supercomputer with 16 cores per node, using up to 8192 cores. We evaluated the correctness of our simulations by recording for each simulation the final number of committed events as well as intermediate information about the number of orientation changes at well defined simulation times. There is complete agreement of these numbers between all simulations, parallel and sequential, and hand written and automatically generated reverse code. The code used in our runs was compiled with GNU g++ 4.7.2 with the optimization flags:

```
    -O3 --finline-limit=1000000
```

It is crucial to allow a high amount of inlining to get good performance, as it enables high-level and low-level optimizations on the code instrumentations and the inlined Backstroke library functions.

We include performance data both for simulations using hand written forward and reverse code, and using reversible code generated by Backstroke from the hand written forward code.

In summary we find in our measurements that the simulation with Backstroke generated reversible code is 2.8 to 3.6 times slower than with the hand written forward and reverse code. The performance details can be found in the left panel of Fig. 2 where the y-axis units are committed events per second. In Fig. 3 we compare the speed (in units of committed events per second) of the

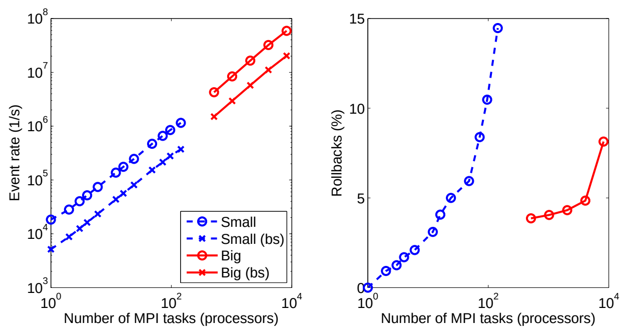

Fig. 2: The performance of Potts model of different sizes using Backstroke instrumented code, as well as hand written reverse code. The big system (in red) consists of 1536 × 1536 spins, divided into a grid of 256 × 256 = 65536 LP's, and was run for 2 time units. The small system (in blue) is 192 × 192 spins, divided into a grid of 24 × 24 = 576 LP's, and was run for 10 time units. *(left)* Aggregate event rate as a function of number of MPI tasks (processors), with ×'s labeling original hand written code, and ∘'s Backstroke generated code. *(right)* Rollback fractions as a function of number of MPI tasks. The data for original and backstroke generated codes are nearly indistinguishable, and only shown for the original code

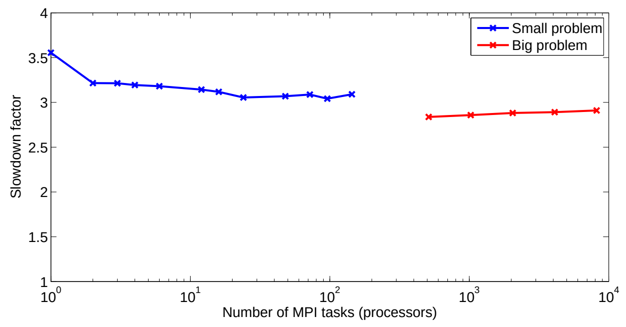

Fig. 3: The slowdown factor of using the Backstroke instrumented code compared to hand written reverse code.

hand written code to the Backstroke generated code by showing the slowdown factor. The slowdown factor is defined as the speed with the hand written code divided by the speed with the Backstroke generated code.

The highest penalty of Backstroke is seen in sequential execution. In this case the simulator uses a simplified event scheduler with significantly reduced book-keeping overhead. For all parallel cases, the overhead of Backstroke instrumented code is in the range of a factor 2.8-3.2. This can be compared with our previous results which show a factor of 3.4 in the best case.

In the right panel of Fig. 2 we show the effect on how many events are rolled back as a function of the number of processors. Interestingly, the numbers for the original and Backstroke instrumented codes are nearly indistinguishable and we show the results only for the original code.

If the fraction of rollbacks is high, it means that the model is running out of inherent parallelism, and the simulation becomes inefficient. As can be expected, the number of rollbacks increase with the number of processors, as more and more parallelism is exploited. In our simulations we see up to about 15% of the total number of executed events rolled back, which although impacting the run time by $\sim 20\% \left(\frac{1}{1-0.15} + \text{rollback cost}\right)$ can be considered relatively efficient.

Simultaneously with increasing number of rollbacks we see in Fig. 3 the relative performance of the original and Backstroke instrumented codes remaining relatively constant ($< 6\%$ deviation between the parallel executions of the small simulation, and $< 3\%$ for the big simulation), and can infer that the data restoration incurred by the Backstroke rollback function does not add significant overhead compared to the hand written reverse code.

5 Related Work

Perumalla and Alfred Park discuss the use of Reverse Computation for scalable fault tolerant computations [16]. The paper is limited in a number of ways, but they make a fundamental point, which is that Reverse Computation can be used to recover from faults by mechanisms that are much faster than check pointing mechanisms. The approach presented in this paper uses incremental checkpointing, which reduces the amount of memory that needs to be stored for establishing a checkpoint by only storing the changes to a state, instead of storing the values of all variables (and memory) defining the state of a program.

In [10] Justin LaPre et. al discuss an approach called Low Overhead Runtime Assisted Instruction Negation, LORAIN. LORAIN is able to account for, and in many cases reverse, the computation without resorting to state-saving techniques. Similar to our presented work, it also uses Rensselaer's Optimistic Simulation System (ROSS), but couples it with the LLVM compiler to generate the reverse code. The reverse code generation is limited as it cannot handle more sophisticated C++ language features such as virtual functions and exceptions, but since it operates on the LLVM IR, it is independent of the source level language, but is limited to LLVM supported backends. Loops also require user-provided information, which in contrast, our approach does not. An alternative fine-grain time-sharing Time Warp system is presented in [13]. It can be run on multi-core Linux machines and makes systematic use of event preemption in order to dynamically reassign the CPU to higher priority events/tasks.

The approach in [22,8] is similar to [10] as it takes control flow into account and generates code for computing additional information required to reconstruct the execution path that had been take in the forward code, but operates at the source code level. It has similar limitations w.r.t. to C++ language constructs with complicated control flow, in particular virtual method calls and exceptions, which are not addressed.

Our approach presented in this paper is different to [2,22,10] as we do not need to take any control flow information into account. For example, in [22] only scalar data types and programs without function calls are considered. Our work presented in this paper allows to address all of C++. However, the drawback of our approach is that it is likely to generate a high overhead in the forward code. Therefore, our approach should only be considered as a base level for establishing reversibility of arbitrary irreversible (i.e. information destroying) C++ code that can benefit from the above mentioned approaches for reverse code generation. Approaches to reverse code generation that allow reverse execution usually induce smaller overhead in the forward code as our approach if information is destroyed, or no overhead at all for reversible programs (which do not destroy information).

In [14] an autonomic system is presented that can utilize both an incremental and a full checkpointing mode. At run time both code variants are available and the system switches between the two variants, trying to select the more efficient checkpointing version. With our approach we optimize the incremental checkpointing by reducing the number of instrumentations at compile time by determining with static analysis which memory locations are part of the event object's state and only need to be instrumented. In [3] an instrumentation technique is applied to relocatable object files. Specifically, it operates on the Executable and Linkable Format (ELF). It uses the tool Hijacker [12] to instrument the binary code to generate a cache of disassembly information. This allows to avoid disassembly of instructions at run time. At run time the reverse instructions are built on-the-fly also using pre-compiled tables of instructions. This approach has been further refined to also operate on shared libraries [5]. This approach is similar to our approach as it also pays an instrumentation overhead. The information that it extracts from instructions, the target address and the size of a memory write, is similar to our address-value pairs. With our source-level approach the available type information is sufficient to determine the value size. Since our transformation operates at the source level we can target different architectures without further work, which is important in the field of high-performance computing. Recently progress has been made also in utilizing hardware transactional memory for further optimizing single node performance [19].

6 Conclusion

We have demonstrated an approach to reversible computation that can be applied to C++ and demonstrated it for a model that makes use of templates, operator overloading, memory allocation, the placement **`new`** operator, user-defined complex data structures, and several data types from the C++11 Standard library. We transform irreversible C++ programs that destroy information into instrumented reversible C++ programs. The instrumentation ensures that all potentially destroyed information is preserved by the transformed forward code such that the reverse function provided in the Backstroke Runtime Library can restore a previous program state and the commit function can perform the deferred memory deallocation.

In contrast to other approaches for generating reverse code, we do not need to explicitly take any control flow information into account, but instead store address-value pairs of modified memory locations and record information about all dynamic memory allocation and deallocation. In addition, we generate reversible assignment operators. The reversible assignment operators are crucial for supporting user-defined types that are composed of other user-defined types and built-in types. When the forward function is executed it is guaranteed that only heap allocated memory is stored in the Run Time State Storage (RTSS). With a run time check we ensure that stack allocated memory is never stored in the RTSS as the event function's stack frame is only valid while the event function is executing.

To the best of our knowledge, this is the very first publication where a full scale C++ model using also C++ Standard Library abstractions is automatically transformed such that it can be executed in parallel optimistically on a supercomputer. Our scaling study still shows a significant performance penalty compared to the manually written reverse code. For the big model we observe a performance penalty between 2.8 and 3.6 for the automatically optimized version. Further, the optimization of the RTSS and improved static analysis to reduce the number of program instrumentations have a great potential to reduce this performance penalty. In this paper we focus on correctness and will address improvements in performance in future work.

If a user starts with a new model and uses Backstroke, it is sufficient to write uninstrumented irreversible C++ code (i.e. conservative code). There are no limitations in what features of C++ can be used. Backstroke can be used to automatically transform the model's code and run the code on a parallel computer using the ROSS simulator. This offers a significantly lower entry level for model writers to use parallel discrete event simulation. It is worth noting that even at a performance penalty of a factor of three, parallel execution enabled by Backstroke at almost no extra work can yield many fold speedup on a single multicore workstation and give the opportunity to get 1000-fold speedup on a cluster, compared to original sequential code.

Some of the performance penalty can be addressed at the language level by implementing functions as pure functions (in C++ as const functions that do not modify state) as Backstroke does not need to instrument such functions.

In future work we also plan to incorporate approaches that address reversible languages into Backstroke. One example is the Janus language which has been extended to provide a C backend. Such Janus generated reversible C code is suitable to be combined with Backstroke generated code and offers to avoid the Backstroke induced overhead for reversible code fragments. How to efficiently combine both approaches is subject to future research. We also plan to further investigate the manual optimization of models by using

Backstroke's feature to turn on or off the recording of data as this will show how far our approach can be pushed in terms of performance. Furthermore, alternative implementations of the Backstroke Runtime Library could also be used to monitor memory accesses and memory allocations for other purposes.

7 Acknowledgments

We wish to thank the reviewers for their detailed comments and suggestions. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. IM release number LLNL-JRNL-745586.

References

1. P. D. Barnes, Jr., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. Warp speed: Executing time warp on 1,966,080 cores. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 327–336, New York, NY, USA, 2013. ACM.
2. C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999.
3. D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '15, pages 211–222, New York, NY, USA, 2015. ACM.
4. D. Cingolani, A. Pellegrini, and F. Quaglia. Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. *ACM Trans. Model. Comput. Simul.*, 27(2):11:1–11:26, May 2017.
5. D. Cingolani, A. Pellegrini, M. Schordan, F. Quaglia, and D. R. Jefferson. Dealing with reversibility of shared libraries in PDES. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '17, pages 41–52, New York, NY, USA, 2017. ACM.
6. R. M. Fujimoto. *Parallel and Distribution Simulation Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.
7. A. O. Holder and C. D. Carothers. Analysis of time warp on a 32,768 processor IBM Blue Gene/L supercomputer. In *Proceedings of the European Modeling and Simulation Symposium (EMSS)*, 2008.
8. C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. In M. O'Boyle, editor, *Compiler Construction*, pages 81–100, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
9. D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, July 1985.
10. J. M. LaPre, E. J. Gonsiorowski, and C. D. Carothers. LORAIN: A step closer to the PDES 'holy grail'. In *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, pages 3–14, New York, NY, USA, 2014. ACM.
11. Y. Omelchenko and H. Karimabadi. Hypers: A unidimensional asynchronous framework for multiscale hybrid simulations. *J. Comp. Phys.*, 231(4), Feb. 2012.
12. A. Pellegrini. Hijacker: Efficient static software instrumentation with applications in high performance computing: Poster paper. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 650–655, July 2013.
13. A. Pellegrini and F. Quaglia. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.*, 27(2):10:1–10:25, May 2017.

14. A. Pellegrini, R. Vitali, and F. Quaglia. Autonomic state management for optimistic simulation platforms. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1560–1569, June 2015.
15. K. S. Perumalla. *Introduction to Reversible Computing.* CRC Press Book, 2013.
16. K. S. Perumalla and A. J. Park. Reverse computation for rollback-based fault tolerance in large parallel systems. *Cluster Computing*, 17(2):303–313, June 2014.
17. D. Quinlan, C. Liao, R. Matzke, M. Schordan, T. Panas, R. Vuduc, and Q. Yi. ROSE Web Page. `http://www.rosecompiler.org`, 2014.
18. R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent incremental state saving in time warp parallel discrete event simulation. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, PADS '96, pages 70–77, Washington, DC, USA, 1996. IEEE Computer Society.
19. E. Santini, M. Ianni, A. Pellegrini, and F. Quaglia. Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 145–154, Dec 2015.
20. M. Schordan, D. Jefferson, P. Barnes, T. Oppelstrup, and D. Quinlan. Reverse code generation for parallel discrete event simulation. In J. Krivine and J.-B. Stefani, editors, *Reversible Computation*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer International Publishing, 2015.
21. M. Schordan, T. Oppelstrup, D. Jefferson, P. D. Barnes, Jr., and D. Quinlan. Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '16, pages 111–122, New York, NY, USA, 2016. ACM.
22. G. Vulov, C. Hou, R. Vuduc, R. Fujimoto, D. Quinlan, and D. Jefferson. The Backstroke framework for source level reverse computation applied to parallel discrete event simulation. In *Proceedings of the Winter Simulation Conference*, WSC '11, pages 2965–2979. Winter Simulation Conference, 2011.
23. T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 43–54, New York, NY, USA, 2008. ACM.
24. T. Yokoyama, H. B. Axelsen, and R. Glück. Reversible flowchart languages and the structured reversible program theorem. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming*, pages 258–270, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
25. T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 144–153, New York, NY, USA, 2007. ACM.