Backstroke 2.1.0
User Manual

LLNL-SM-719981

Markus Schordan

March 15, 2017

Contents

1 Backstroke

An up-to-date version of this document can be found at `https://github.com/LLNL/backstroke/docs/user-manual.pdf`

1.1 Introduction

Backstroke is a tool for reversible computation and supports C++11. It is a command line tool for generating reversible C++ source code and is distributed with a runtime library that can maintain all data necessary to support reversibility. The runtime library implements a so called Runtime State Store (RTSS) and is called rtss library. Backstroke imlements a technique for incremental state saving (also called incremental check-pointing).
The command line tool Backstroke takes as input arbitrary C++ source code (C++98 or C++11) and generates as output instrumented reversible C++11 source code. The instrumented code contains function calls to the runtime library rtss. Therefore the runtime library must be linked with the code generated by Backstroke. When the instrumented reversible code is executed, additional program state information is stored in the RTSS which allows to reverse (or "undo") the effects of the forward execution. The rtss library is implemented in C++11.

1.2 Terminology

Backstroke generates reversible *forward-functions* and provides through its rtss library a *reverse-function* and a *commit-function*. The functionality of reversibility is implemented by combining all three kinds of functions. Backstroke provides a

method for *incremental state saving* for which it generates the reversible forward-functions from the original (possibly) irreversible functions. The reversible forward functions are *instrumented* versions of the original functions. The instrumentation implements function calls to the RTSS. The reverse-function and commit-function are implemented in the rtss library and use the additional information stored by the reversible forward-functions. This additional information is used to *reverse* (or *undo*) all effects of memory modifications and memory allocations of the forward-functions. The commit-function performes the *deferred deallocation* of memory.
Backstroke generated code can be run either in *reversible mode* or *non-reversible mode*. The mode can be selected at runtime. In non-reversible mode no information is stored in the RTSS. Since the modes can be switched at runtime, this feature can also be used to manually optimize a program and reduce the amount of information stored at runtime.

2 Installation

Backstroke requires the installation of the ROSE infrastructure which can be downloaded from `http://www.rosecompiler.org`. The compilation of Backstroke requires the ROSE library to be available. Backstroke has no other software dependencies beyond those of ROSE.

2.1 Installation of ROSE

ROSE should be compiled in C++11 mode (-std=c++11) because Backstroke needs to be compiled in C++11 mode. A configuration for ROSE can be:

```
${ROSE_SOURCE}/configure CXXFLAGS=-std=c++11
   --prefix=${ROSE_INSTALL} --with-boost=${BOOST_HOME}
   --enable-languages=c++
```

where `ROSE_SOURCE` is the directory where the source tree of ROSE is located, `ROSE_INSTALL` is the directory where ROSE is to be installed, and `BOOST_HOME` is the path to an installed version of the BOOST library. The option `enable-languages` allows to select a subset of the languages supported by ROSE (this reduces installation time). For Backstroke only C++ is required. Also see the ROSE installation guidelines for more information on installing ROSE. Note that ROSE must be compiled in a directory different to its source directory.

2.2 Installation of Backstroke

After the installation of ROSE, ensure that the ROSE library path (<ROSE_INSTALL>/lib) is added to LD_LIBRARY_PATH and that the path to the installed ROSE tools in <ROSE_INSTALL>/bin is added to the PATH variable on Linux and made available on the command line.
When Backstroke is compiled it uses the ROSE tool rose-config to obtain all configure information from the ROSE installation. Therefore no configuration of Backstroke is necessary. This version of Backstroke requires at least ROSE version 0.9.8.5 (see file ROSE_VERSION in the ROSE distribution) and has been tested with GNU g++ 4.8. When Backstroke is run, it checks at runtime whether the version of the installed ROSE library is sufficient or whether an update of ROSE is necessary.
The procedure for installing Backstroke is therefore straightforward:

1. Install ROSE.
2. In the Backstroke top directory run 'make', 'make install'.
3. Additionally, 'make check' and 'make check-extended' can be run to perform a number of tests.

The files of the Backstroke distribution are installed as

- <ROSE_INSTALL>/bin/backstroke
- <ROSE_INSTALL>/include/backstroke/rtss.h
- <ROSE_INSTALL>/lib/librtss.so

Note that the rtss library is used by the code generated by Backstroke, but not by Backstroke itself – the tool Backstroke requires an installation of ROSE to be run, but not the rtss library. In contrast, an application that has been instrumented by Backstroke, requires the rtss library, but not Backstroke when being executed.

3 Using Backstroke

3.1 What Backstroke Does

Backstroke generates reversible forward-functions from the provided original code. The reversible forward-functions execute the statements in the same or-

der as the original program, but store additional information for all memory modifying operations. Memory modifying operations are all forms of assignment, memory allocation, and memory deallocation. By using the RTSS and based on the information that is stored when the instrumented forward code is executed, all effects of assignments and memory allocation can be reversed. Memory deallocation is deferred until an event is committed. The effects of an event function are reversed or committed by calling the respective function in the RTSS. For example, in the case of optimistic parallel discrete event simulation (PDES), the event (forward), the reverse function, and the commit function, are set up to be called by the simulator.
If the tool Backstroke is applied to all code that is directly or indirectly called from an event function then the user does not need to write any reverse code. The reversible forward code is generated by the tool Backstroke and the reverse and commit operations are provided as part of the rtss library.

3.1.1 Forward-Reverse-Commit Paradigm

The forward-reverse-commit paradigm is applied as a combination of the Backstroke generated reversible forward code and the reverse and commit functions that are provided by the rtss library.
The reverse and commit functions in the rtss library only depend on the state information that is recorded when the reversible (instrumented) forward-functions are executed.

3.1.2 Reversible versus Non-Reversible Runtime Mode

The recording of additional state information is optional and only active when the mode for recording state information is enabled. By default the recording of this state information is disabled. This means that any code that has been run through the tool Backstroke can still be used as before (without recording any additional information) – provided it is linked with the rtss library.
Only when the respective functions of the rtss library for beginning and ending a reversible execution are called, the reversible mode with recording information is enabled. Backstroke generated code behaves exactly like the original code when run in non-reversible mode, except for the mode flag which is checked at the entry of RTSS functions and all delete operations are duplicated in the RTSS when executing in non-reversible mode. In reversible mode all `delete` operations are deferred until the RTSS commit function is called.

3.1.3 Generated File

For a given input file Backstroke generates one output file. The output file contains instrumented forward-functions for all functions that are present in the input file. The input file is pre-processed and therefore the output file contains all included headers (which are transformed as well).

3.2 What Backstroke Does Not Do

Backstroke does not perform the integration of the generated code into the original application. The original application must be modified by the user to take the generated forward-functions into account and link with the rtss library.
Backstroke does not determine whether all functions that are called from a provided translation unit (a C++ implementation file) have also been provided to Backstroke for instrumentation. It is the responsibility of the user to provide all code that is directly or indirectly called within a check-pointed code region for instrumentation to Backstroke. Note that external functions that do not change the program state with respect to the check-pointed code region do not have to be instrumented.
Backstroke does not analyze input/output to streams external to an instrumented program; it does not undo the reading of data from a standard input stream (e.g. std::cin), and it does not defer the writing of data to a standard output stream (e.g. std::cout). Note that Backstroke offers a feature to turn on/off the recording of data and pragma directives to request not to instrument code. The combination of these features allows to control and protect input/output commands to some extent. See Sections 3.5 and 3.7 for more information on these features.

3.3 Backstroke Command Line Options

```
Backstroke [Options] INPUTFILE
```

`--help` : prints list of all command line options.

`--status` : prints status messages during code generation. Without this option Backstroke does not generate any messages.

`--rtss-header` : automatically includes the rtss library header 'rtss.h'. This is important when using the C++Std library in an application.

--trace : prints a trace of all transformations. The format is "line:column: original-expression => transformed-expression".

--no-preprocessor : turns off pre-processing of input files (consequently, included files are not transformed!).

--no-optimization : turns off optimization for the instrumentation of assignments to local variables.

--access-mode NUM : select the RTSS access mode: option 1 for the xpdes interface [default]. Option 2 selects the explicit use of RTSS objects.

--access-operator-arrow : generates operator '`->`' for accessing the RTSS member functions in transformed code (this option is irrelevant if access-mode 1 (xpdes) is used).

--access-operator-dot : generates operator '`.`' for accessing the RTSS member functions in transformed code (this option is irrelevant if access-mode 1 (xpdes) is used).

--stats : prints transformation statistics.

--stats-csv-file FILE writes transformation statistics in CSV format to file FILE.

--version : prints the version number of Backstroke.

For an input file `FILE` Backstroke generates `backstroke_FILE`. With the options `--status` and `--trace` informative output can be generated during the translation. Informative output is generated on the standard output stream.
Example: When Backstroke is invoked with `backstroke example.C` then the generated code is written to `backstroke_example.C`.

3.4 Backstroke Library API

3.4.1 The Backstroke Runtime State Store (RTSS)

The rtss library interface that is accessed in the Backstroke *generated* code consists of the following 5 API functions.

```
T& avpushT(T& lhs)
T* registerAllocationT(T* mem)
void registerDeallocationT(T* mem)
T* allocateArrayT<T>(size_t numberOfElements)
void registerArrayDeallocationT(T* deleteOperand)
```

Backstroke generates calls to the above functions in transformed forward code. If the type `T` is a built-in type then the invocation of `avpush` stores information in the runtime library's instantiated class RunTimeStateStore. This information is used by the reverse and commit functions (see below for more details). If type `T` is not a built-in type, then the invocation of `avpush` has no effect. The function also performs a run-time check whether an object is stored on the applications runtime stack or in the heap. The backstroke library only stores information if data is stored in the heap (it would be wrong to restore stack allocated data after the invocation of an event function has finished). The other 4 functions for memory allocation and deallocation take pointer value arguments only.
Backstroke has two options to instrument different forms of invocation of rtss library API functions. By default Backstroke makes no assumptions about the user code and generates calls to the xpdes interface.

3.5 Integrating Reversible Code

By default Backstroke generates function calls to the functions listed in 3.4.1 within the xpdes namespace. The user does not need to provide any additional code.
The following code shows how an event function that has been processed by Backstroke can be used when using the default mode (access-mode=1).

```
#include <iostream>
#include "backstroke/rtss.h"
State { ... };

state=new State();

try {
  // this function is necessary to allocate and
  // initialize the runtime state store (RTSS)
  // and allocates memory for RTSS
  xpdes::initializeRTSS();
```

```
  for(int i=0;i<10;i++) {
    xpdes::beginForwardEvent();
    event(state); // Backstroke transformed event function
    xpdes::endForwardEvent();
  }
  for(int i=0;i<5;i++) {
    xpdes::reverseEvent();
  }
  for(int i=0;i<5;i++) {
    xpdes::commitEvent();
  }

  // deallocates memory RTSS and clean up
  xpdes::finalizeRTSS();
} catch(Backstroke::Exception& e) {
  std::cout<<"Backstroke exception raised: "
           <<e.what()<<")"<<std::endl;
} catch(std::exception& e) {
  std::cout<<"std exception raised: "<<e.what()<<")"<<std::endl;
}
```

Additionally two functions can be called to enable or disable the recording of data that is required to support incremental checkpointing. By default recording of data is disabled. Therefore you can run any code through Backstroke and use it as before. The functions `xpdes::beginForwardEvent` and `xpdes::endForwardEvent` will internally enable and disable the recording of data mode.
In the above code a class `State` is used to represent some state object. The first loop performs 10 calls of the Backstroke transformed forward event function. Then we reverse 5 event function calls, and commit the remaining 5 in the third loop. Thus, in the result, we see the result of 5 event function calls.
The functions `beginForwardEvent()` and `endForwardEvent()` use internally the following two functions to enable/disable the recording of information:

```
void xpdes::enableDataRecording();
void xpdes::disableDataRecording();
bool xpdes::dataRecording();
```

These functions are public functions and can also be called in user code to reduce the recording of information. However, by default, these functions do not need

to be called by the user and the mode is automatically enabled/disabled by the `beginForwardEvent` and `endForwardEvent` functions.
If a wrong use of the Backstroke API is detected, an exception of type Backstroke::Exception is thrown. The example shows how such an exception can be catched and determined to be a Backstroke::Exception. The function `what` gives a brief explanation of the error. See section 5.1 for more information.

3.5.1 Example

Let us assume the event function was implemented in a file called `event.C`. When Backstroke is invoked with `'backstroke event.C'` it generates the file `backstroke_event.C`. This file, `backstroke_event.C`, contains the Backstroke transformed event function that is invoked in the above example code. Additionally, the compiled rtss library must be linked with the example code.

3.6 Explicit Use of Runtime State Store Objects

If the xpdes interface functions are used the explicit use of runtime state store objects is not necessary. However, in case one wants to maintain multiple different RTSSs, this is the way to go.
With command line option `--access-mode=2` Backstroke assumes a variable `rts` of type RunTimeStateStore to be available and used in the generated code. For example, a generated access is `rts.avpushT(x)`. Alternatively, an arrow operator can be generated (instead of '.') by providing the option `--access-operator-arrow`. For example, this will generate `rts->avpushT(x)`.
Similar to the functions in the xpdes namespace, the user code must invoke 3 functions to (i) initialize the runtime state store (once), (ii) mark the beginning of an event which creates internal data structures for an event, and (iii) mark the end of an event which enqueues the created data structures for a reverse or commit function. The following minimal example shows the concrete function calls that are required in the user code.

```
#include "backstroke/rtss.h"
State { ... };
RunTimeStateStore rtss;
rtss.reset();
rtss.init_stack_info();
```

```
state=new State();
for(int i=0;i<10;i++) {
  rtss.beginForwardEvent();
  event(state); // calls the Backstroke transformed event function
  rtss.endForwardEvent();
}
for(int i=0;i<5;i++) {
  rtss.reverseEvent();
}
for(int i=0;i<5;i++) {
  rtss.commitEvent();
}
```

In the above code a class `State` is used to represent some state object. The variable `rts` of type RunTimeStateStore is declared. It holds all information necessary to reverse and commit executed forward functions. The first loop performs 10 calls of the Backstroke transformed forward event function. Then we reverse 5 event function calls, and commit the remaining 5 in the third loop. Thus, in the result, we see the result of 5 event function calls.

3.7 Backstroke Pragma Directives

Backstroke supports pragma directives to allow the user to direct what code of the original program is transformed and how.

3.7.1 The Map Directive

```
#pragma reversible map forward = original
```

Above directive can be applied to any statement in the original forward code. Backstroke does not apply any code transformation to this code and generates the original code verbatim in the generated forward-code.
The directive can be applied to any statement, block, or function definition. For example:

```
#pragma reversible map forward = original
state->x++;
#pragma reversible map forward = original
{
  state->x++;
  state->x++;
}
state->y++;
```

Above directive is applied to the statement `state->x++` and to the block containing two updates of `state->x`. If the directive is applied to a block, the entire block is excluded from being reversed. Any other code, not marked with the exclude directive, is transformed (e.g. `state->y++` in the above example). For example, this directive can be used for code that computes statistics in the original code that should *not* be reversed. For functions, the directive must be added to the definition of the function (the implementation). It is not sufficient to add it to the function declaration only. If it is added to the function declaration it has no effect. See Section 5.3 for a more elaborate example.
The Backstroke pragma directives can be used in combination with any other pragma directives. Backstroke only considers pragmas with the prefix "reversible", all other pragmas are ignored. The generated transformed forward code contains all pragmas as they were present in the original code.

3.8 Using Backstroke with ROSS

If Backstroke is used in combination with ROSS, the RTSS runtime library must be compiled with `-DRTSS_LIBRARY_WITH_ROSS`. This allows to include the `ross` header file and provide some additional functionality for handling ROSS logical processes (LPs). With ROSS the `RunTimeLpStateStore` must be used. The benchmarks in [1, 2] were run with this mode with ROSS on LLNL's Vulcan BG/Q machine.

3.8.1 Integration of Backstroke Generated Code with ROSS

To simplify the use of Backstroke, it is best to have the event function and all functions that are called by the event function in files separate from the simulator code. The Backstroke generated code, which implements the transformed forward-function, can then be used to replace the original functions.

4 Transformed C++11 Language Constructs

Backstroke transforms the following language constructs:

- The left-hand-side of assignment operators (including increment and decrement operators).

- The expression `new`. It correctly detects and does not transform placement `new` operators.
- The expression `delete`.
- Calls to overloaded global C++ new and delete operators.
- Detects implicit (non-implemented) default assignment operators and generates reversible default copy assignment operators.
- Generates a friend declaration in classes where the destructor is protected or private. This is necessary to allow functions inside the rtss library to call the destructor in the commit function. Note that this does not change the interface of the original class.

Note that Backstroke does not generate reversible move assignment operators, but only reversible copy assignment operators. In those very restricted cases were an implicit default move assignment operator would exist (without an existing copy assignment operator), it defaults back to the either existing or generated reversible default copy assignment operator (according to C++11 semantics). Since Backstroke generates a reversible copy assignment operator for all cases of an implicit copy assignment operator, all such classes with implicit move assignment operators default now back to this generated reversible copy assignment operator. In this context note that unless otherwise specified, all standard library objects that have been moved from are placed in a valid but unspecified state. That is, only the functions without preconditions, such as the assignment operator, can be safely used on the object after it was moved from. In case of a default fall back on the copy assignment operator the original (valid) state remains [1]
Backstroke instrumented code defers memory deallocation (triggered by variants of `delete`) to the commitEvent function. Memory is only deallocated if the `xpdes::commitEvent` function is called.

4.1 Supported Built-In Types

Backstroke only stores data of built-in types. It does not need to store any other data. The supported built-in types are:

[1] If a case is found in future where this reasoning is not sufficient for not generating move assignment operators, the generation of reversible move assignment operators will be added to Backstroke.

- bool
- signed char
- signed short int
- signed long int
- signed long long int
- unsigned char
- unsigned short int
- unsigned long int
- unsigned long long int
- float
- double
- long double
- pointer (to any type)

Note that reversible copy assignment operators are generated for all user-defined types. All templates are transformed as well. The implementation in the RTSS uses a fall-through technique for user-defined types and resolves the type and whether data should be stored or not at compile time using C++11 compile time predicates. Hence, the type is resolved at compile time of the instrumented application code and therefore there is no runtime overhead for determining the type of an instrumented expression inside a template declaration although the type is not known when transformed by Backstroke (which happens before the application is compiled).

5 Examples

5.1 Detected Errors and Exceptions of the RTSS Library

In case of an error the rtss library throws an exception of type Backstroke::Exception. This exception class inherits from std::exception and can be

caught as std::exception or explicitly as Backstroke::Exception. A handler for calls to the rtss library can be written as:

```
try {
  // backstroke library functions calls
} catch(Backstroke::Exception& e) {
  std::cout<<"Backstroke exception raised:"<<e.what()<<std::endl;
} catch(std::exception& e) {
  std::cout<<"std exception raised:"<<e.what()<<std::endl;
}
```

The rtss library can throw an exception with the following error messages (produced by calling the member function `what()`) of the class Backstroke::Exception.

5.1.1 Wrong Use of the rtss Library

The rtss library detects some wrong usage of its API. In this case an exception is thrown containing the following error text(s) (which can be shown with the member function what() of the Backstroke::Exception object. See the example in Section 5.1).

"reverseEvent: no existing event" : the reverseEvent function was called but no data for an event is available.

"commitEvent: no existing event" : the commitEvent function was called but no data for an event is available.

5.1.2 Internal Errors

The rtss library also checks its supported types and size requirements of maintained application data. The following internal errors can be reported as a thrown exception of type Backstroke::Exception:

"unknown event record allocation mode" : If an allocation mode was chosen that does not exist. RTSS supports two allocation modes.

"restoring unknown built-in type" : If a built-in type is detected that is not supported. Data is restored when the function reverseEvent is called.

"disposing unknown built-in type" : If a built-in type is detected that is not supported. Data is disposed when the function commitEvent is called.

"size of unsupported built-in type" : If a built-in type is detected with a size that is not supported.

5.2 Simple State Variable Update

5.2.1 Original Code

```
struct State {
  int x;
};

void event(State* state) {
  if(state->x > 20) {
    state->x = 0;
  } else {
    state->x++;
  }
}
```

5.2.2 Transformed Forward Code

```
struct State {
  public: State& operator=(const State& other) {
    xpdes::avpushT(this->x)=other.x;
  return *this;
  }
  int x;
};

void event(State* state) {
  if(state->x > 20) {
    (xpdes::avpushT(state->x)) = 0;
  } else {
    (xpdes::avpushT(state->x))++;
  }
}
```

5.3 Example: pragma reversible map forward = original

This directive allows to specify that the associated block or statement of the forward code is mapped to the original code, effectively not performing a transformation on this code. Hence, the original code remains as-is in the generated code and no reverse code is generated for this portion of the code.

5.3.1 Original Code

```
class State {
public:
  State():count(0){}
  int count;
  void update1();
  void update2();
};

#pragma reversible map forward = original
void State::update1() {
  count++;
}

void State::update2() {
  count++;
}

#pragma reversible map forward = original
void f1(State* s) {
  s->count++;
}

void f2(State* s) {
  s->count++;
}

void event(State* s) {
  #pragma reversible map forward = original
  s->count++;
  #pragma reversible map forward = original
  {
    s->count++;
    s->count++;
  }
  s->count++;
  #pragma reversible map forward = original
  {
  }
  s->count--;
}
```

5.3.2 Transformed Forward Code

```
class State {
public:
  public: State& operator=(const State& other) {
    xpdes::avpushT(this->count)=other.count;
    return *this;
  }
State():count(0){}
  int count;
  void update1();
  void update2();
};

#pragma reversible map forward = original
void State::update1() {
  count++;
}

void State::update2() {
  (xpdes::avpushT(count))++;
}

#pragma reversible map forward = original
void f1(State* s) {
  s->count++;
}

void f2(State* s) {
  (xpdes::avpushT(s->count))++;
}

void event(State* s) {
  #pragma reversible map forward = original
  s->count++;
  #pragma reversible map forward = original
  {
    s->count++;
    s->count++;
  }
  (xpdes::avpushT(s->count))++;
  #pragma reversible map forward = original
  {
  }
  (xpdes::avpushT(s->count))--;
}
```

Future versions will allow to also establish mappings to existing reverse code (hand-written or generated by other tools). For example, this will allow to also integrate Janus[2] generated C code, or use existing reverse number implementations in ROSS models.

6 C++ Standard Library and STL Support

Backstroke instruments source code. If code is linked that has not been processed by Backstroke and it modifies state (i.e. is not read-only) then the generated code is not reversible.
The general solution for addressing all of the C++11 library is to process the entire C++11 standard library with Backstroke and link with this reversible C++ standard library instead of the original standard library.

7 Performance

Increasing the amount of in-lining that a compiler's metric decides to perform by default, can significantly improve performance. Inspecting the assembly code also shows that the compiler will also optimize the register allocation, from which the Backstroke generated code can benefit greatly.
Optimization level 3 can reduce the *overhead* of the Backstroke generated instrumentations significantly. For example, to get good performance with GNU g++ 4.8 or later, use the options `-O3 --finline-limit=1000000`.

8 Limitations

- Bitfields are not supported yet.

9 Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, via LDRD project 14-ERD-062. IM release number LLNL-SM-719981.

[2] See http://topps.diku.dk/pirc/?id=janus for more details.

References

[1] Markus Schordan, David Jefferson, Peter Barnes, Tomas Oppelstrup, and Daniel Quinlan. Reverse code generation for parallel discrete event simulation. In Jean Krivine and Jean-Bernard Stefani, editors, *Reversible Computation*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer International Publishing, 2015.

[2] Markus Schordan, Tomas Oppelstrup, David Jefferson, Peter Barnes, and Daniel Quinlan. Automatic Generation of Reversible C++ Code and Its Performance in a Scalable Kinetic Monte-Carlo Application. In proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (PADS), pages 111-122, ACM New York, ISBN 978-1-4503-3742-7, 2016.