LORAIN: A Step Closer to the PDES “Holy Grail”

Justin M. LaPre, Elsa J. Gonsiorowski, and Christopher D. Carothers
Department of Computer Science, Rensselaer Polytechnic Institute
Troy, NY U.S.A.
laprej@cs.rpi.edu, gonsie@cs.rpi.edu, chrisc@cs.rpi.edu

ABSTRACT

Automatic parallelization of models has been the “Holy Grail” of the PDES community for the last 20 years. In this paper we present *LORAIN – Low Overhead Runtime Assisted Instruction Negation* – a tool capable of automatic emission of a reverse event handler by the compiler. Upon detection of certain instructions, LORAIN is able to account for, and in many cases reverse, the computation without resorting to state-saving techniques. For our PDES framework, we use Rensselaer’s Optimistic Simulation System (ROSS) coupled with the LLVM [18] compiler to generate the reverse event handler.

One of the primary contributions of this work is that LORAIN operates on the LLVM-generated Intermediate Representation (IR) as opposed to the model, high-level source code. Through information gleaned from the IR, LORAIN is able to analyze, instrument, and invert various operations and emit efficient reverse event handlers at the binary code level.

This preliminary work demonstrates the potential of this tool. We are able to reverse both the PHOLD model (a synthetic benchmark) as well as Fujimoto’s airport model. Our results demonstrate that LORAIN-generated models are able to execute at a rate that is over 97% of hand-written, parallel model code performance.

Keywords

discrete event simulation; compiler analysis; reverse computation; slicing

1. INTRODUCTION

Since the early 1990’s, the “Holy Grail” for parallel discrete-event simulation has been the automatic parallelization of event- and process-driven models which yield performance improvements on par with hand-written models. The belief in this goal is rooted in the Time Warp protocol’s ability to be “blind” with respect to a model’s event scheduling behavior both in terms of space (e.g., which logical processes (LPs) communicate) and time (i.e., model lookahead) [12, 17]. The promise of Time Warp has been its ability to automatically uncover parallelism within a discrete-event model. On the surface this appears to be a very easy problem to solve compared with the more general problem of automatically parallelization of a serial piece of software. That has clearly turned out to not be the case.

At the 1994 PADS conference, Fujimoto correctly asserted that the PDES “Holy Grail” would not be reached in that century [22]. This view was driven from experience in the automatic parallelization of the Simscript II.5 models which were built for and executed by a Time Warp kernel [29]. The principal challenge was LP state-saving, more specifically the issue of sharing state among LPs. This was addressed via *Space-Time Memory*, however, its functionality came with potentially large overheads that diminished overall parallel performance. There were also related issues in that Simscript allowed an event to examine the event list.

To address part of the Time Warp state-saving challenge, Carothers, Perumalla, and Fujimoto [9] devised a new approach called *reverse computation*. As the name implies, only the control state (e.g., which `if`-block was taken) is stored during the forward processing of any event, as opposed to incrementally or fully saving an LP’s state changes. To support the rollback operation, a reverse event handler is created that uses the control state information to select which control path to take during event rollback processing. It then reverses or undoes each state change. Due to the fact that many operations performed on state are *constructive*, such as an increment or decrement on integer statistics, the model’s execution time is dramatically reduced by using this approach. Performance increases with reverse computation have been attributed to significant improvements in cache and TLB hit rates within the overall memory hierarchy [9].

Reverse computation has been applied to a number simulation areas including Hodgkin-Huxley neuron models [20], Internet models [32], HPC network models [19, 21], particle simulations [26], and gate-level circuit models [15]. Most recently, Barnes et al. [5] executed the PHOLD benchmark with reverse computation on nearly two million cores and was able to achieve an event-rate exceeding half a trillion events per second.

Besides improving overall Time Warp simulator performance, reverse computation also opened the door to revisit the question of automatic model parallelization. In 1999, Perumalla created the first reverse C compiler called *RCC* [25]. This was a source-to-source compiler that took a model built to a specific PDES engine (in this case Georgia Tech Time

SIGSIM-PADS’14, May 18–21, 2014, Denver, CO, USA.
Copyright 2014 ACM 978-1-4503-2794-7/14/05 ...$15.00.
http://dx.doi.org/10.1145/2601381.2601397.

Warp) and produced both the instrumented forward event handlers as well as the reverse event handler. RCC was based on a direct statement by statement reversal of the forward event handler's C code statements. The key to this approach is that the model developer had to insert pragma statements to provide hints to RCC in order to produce correct forward and reverse event handlers. This work demonstrated for the first time that an automatic parallelization tool could produce event processing code that was as good as the hand written code (i.e., having negligible performance loss).

The most recent effort to automatically parallelize event-driven models with reverse computation has been the *Backstroke* framework [16, 30]. Here, the ROSE source-to-source compiler framework is used to develop a set of algorithms that operate on an Abstract Syntax Tree (AST) to produce both a modified forward event processing code as well as the reverse event handler code. What is unique about Backstroke is that it takes on the challenge of working with models developed in C++ at the source-level.

In this paper, we introduce a new automatic parallelization approach called *LORAIN – Low Overhead Runtime Assisted Instruction Negation*. LORAIN is a collection of compiler passes that leverage the LLVM [18] compiler framework to create both a modified forward event handler, which records the control state, as well as the corresponding reverse event handler for a specific Time Warp simulator's API. For this research, ROSS (Rensselaer's Optimistic Simulation System) [5] is used.

The key contribution of this work is that it operates not on the source code of the model but on LLVM's Intermediate Representation (IR). This can be viewed as a very high level abstract machine instruction set. LLVM IR instructions use Static Single Assignment (SSA) [10] which aids in the analysis of the model code and forward/reverse code generation process. LORAIN's overall approach is twofold: first, it analyzes the forward handler per the specifics of the ROSS API; second, it synthesizes the proper state-restoring reverse handler. The analysis stage includes evaluating the IR to find locations at which *destructive* assignments take place. Transformations are also made that make certain code patterns more amenable to reversal. In the synthesis stage, LORAIN passes the instrumented forward IR into secondary transformation pass which traverses the Control Flow Graph (CFG) and finds and reverses "store" instructions. Once the model's IR is complete, LLVM can directly produce an executable, as opposed to modified source code. Thus, the developer need not be concerned with additional compilation steps of model source code nor do they need to insert pragmas.

The principal advantage of operating at the LLVM IR level is LORAIN gains independence from the syntax complexities that occur at the source code level, such as multi-expression if-statements that contain potentially destructive assignment statements. The LLVM compiler design-view endorses this separation between language syntax and representation by enabling nearly all compiler passes to operate exclusively at the IR level and not on the front-end's AST (which is contained in a separate tool called *clang*).

The remainder of this paper is organized as follows: Section 2 describes the ROSS Time Warp engine and LLVM compiler. Section 3 describes our LORAIN framework. The specific details on how LORAIN performs the IR modification and generation is presented in Section 4 for two models along with parallel performance results. Finally, additional related work and conclusions are presented in Sections 5 and 6 respectively.

2. BACKGROUND

2.1 ROSS

In this study, we will be using ROSS [8], an open-source, discrete event simulation engine that supports both conservative and optimistic (Time Warp) simulation (`http://ross.cs.rpi.edu`). ROSS is written in C and assembly language atop MPI, with targeted support for supercomputing platforms such as the IBM Blue Gene/Q. ROSS takes a different approach from other Time Warp simulation systems which use bulk state-saving. Instead, ROSS utilizes reverse computation to undo state changes programmatically. To help achieve this programmatic reversal, ROSS employs a reversible random number generator. Through reverse computation, ROSS enables quick state restoration, often many times faster than classical state-saving.

Within ROSS, MPI Tasks are represented as *Processing Elements* (PEs). PEs contain multiple *Logical Processes* (LPs) capable of independently executing *events* in time-stamp order. Events are created by sending messages between LPs and inter-PE messages are permitted. During optimistic simulation, out of order execution is permitted until a temporal anomaly is detected. For example, an LP receives an event in its past. At this point ROSS uses reverse computation to incrementally undo all operations that potentially modified the LP's state. Through multiple anti-messages, the LP's state is reversed until the system can process the erroneous event in proper time-stamp order. At this point, normal forward execution of events will resume.

A central assumption of all ROSS models is that LPs only share state by exchanging time-stamped event messages. This is a typical restriction of many PDES systems and allows ROSS to scale to hundreds of thousands of processors or more. Currently, Space-Time Memory (STM) [29] is not supported in ROSS but could be in the future. Design and implementation of STM on a million core supercomputer is still an open PDES problem. Additionally, static or global variables are not supported unless they are read-only once the model initialization step is complete. This assumption greatly simplifies LORAIN's analysis and instrumentation steps.

The functions within the ROSS API that need to be reversed are relatively straightforward. All of the RNG functions in ROSS rely on the `tw_rand_unif()` macro, which itself calls the `rng_gen_val()` function. Other functions exist, such as `tw_rand_integer()` and `tw_rand_exponential()`, which must be caught and handled appropriately. All other function calls are either re-emitted or ignored as they are immaterial to the various LORAIN passes.

Finally, the function prototypes for the forward and reverse event handlers must be identical and are critically important for this work; all parameters play a role: `void name(state *lp_state, bitfield *bf, message *msg, LP *lp);`, where `lp_state` is the pointer to the LP state, `bf` is a pointer to the control state bit field, `msg` is the pointer to the current event's message data and `lp` is the pointer to the current LP. LORAIN's use of the ROSS API will be discussed further in Section 3.

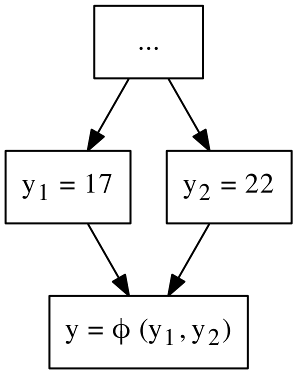

Figure 1: SSA ϕ construction. y is assigned the appropriate value based on the path taken through the control flow graph.

```
%1 = load i32* @test_add_x , align 4
%2 = add nsw i32 %1, 1
store i32 %2, i32* @test_add_x , align 4
```

Figure 2: A use-def chain example. The `store` instruction uses %2, which in turn uses %1. Using this information, we can generate a topological ordering of an instruction's dependencies.

2.2 LLVM

LLVM is a compiler framework developed at University of Illinois at Urbana-Champaign by Lattner and Adve [18]. The virtual instruction set is very similar to RISC, where most instructions are represented as three-address code. Instructions for accessing memory are an exception. Memory is only accessed via `load` and `store` instructions. LLVM utilizes SSA [10] assigning a value to each identifier exactly once. Strict typing is used to ensure consistency within virtual instructions. LLVM has a modern C++ code base and enjoys heavy interest from academia and industry alike.

The LLVM virtual instructions are collated into *basic blocks*. Basic blocks are sequences of instructions that *will* execute once they have been entered (i.e. there are no exits other than completing the block). Each basic block concludes with exactly one *terminator* instruction such as a jump or `return` instruction.

A key feature of LLVM is its application of SSA virtual registers. Here, new SSA virtual registers are assigned the result of any operation that generates a value e.g., `%22 = add %x, %y` adds `x` and `y` and assigns that value to the temporary `%22`. `%22` may be used in future computations requiring that value, for example common sub-expression elimination. Having the same value in multiple paths through the CFG can be problematic. SSA solves this at merge points with ϕ functions. See figure 1. For example, `y` is assigned the output of a ϕ function on y_1 and y_2. The ϕ function "knows" which path was taken to get here and is therefore able to choose the correct value.

Finally, LLVM is object oriented and many classes derive from the *Value* class such as *Argument*, *BasicBlock*, and *Instruction* to name a few. To be precise, the *Instruction* class is a subclass of the *User* class, which itself is a subclass of the *Value* class. The *User* class keeps track of *Value*s it uses and is helpful in the construction of *use-def* chains [2]. LORAIN uses use-def chains to ensure that all instruction dependencies are satisfied (see figure 2).

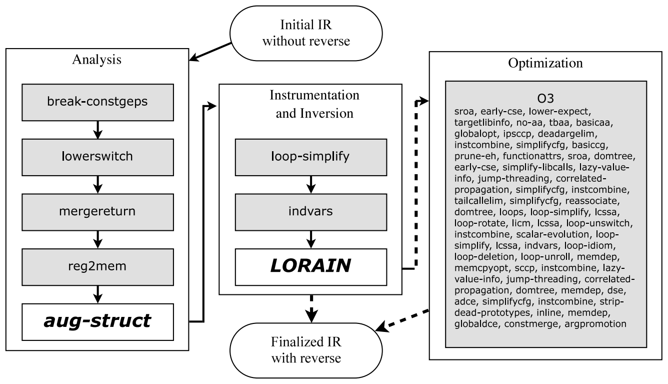
Initial IR without reverse
Analysis
break-constgeps
lowerswitch
mergereturn
reg2mem
aug-struct
Instrumentation and Inversion
loop-simplify
indvars
LORAIN
Finalized IR with reverse
Optimization
O3
sroa, early-cse, lower-expect, targetlibinfo, no-aa, tbaa, basicaa, globalopt, ipsccp, deadargelim, instcombine, simplifycfg, basiccg, prune-eh, functionattrs, sroa, domtree, early-cse, simplify-libcalls, lazy-value-info, jump-threading, correlated-propagation, simplifycfg, instcombine, tailcallelim, simplifycfg, reassociate, domtree, loops, loop-simplify, lcssa, loop-rotate, licm, lcssa, loop-unswitch, instcombine, scalar-evolution, loop-simplify, lcssa, indvars, loop-idiom, loop-deletion, loop-unroll, memdep, memcpyopt, sccp, instcombine, lazy-value-info, jump-threading, correlated-propagation, domtree, memdep, dse, adce, simplifycfg, instcombine, strip-dead-prototypes, inline, memdep, globaldce, constmerge, argpromotion

Figure 3: The various LLVM passes that are run by LORAIN. LLVM-authored passes are shown in gray, our passes are shown in white.

3. LORAIN

LORAIN makes the key assumption that only instructions that affect memory have the capacity to alter state. Therefore, restoring the contents of memory to its previous values is sufficient to effectively negate an erroneous event. By inverting the CFG and reversing the stores along the basic block path taken through the function, the contents of memory following a reverse function call should be restored to its original values (assuming there is no bit loss in the instruction reversal step). Currently, LORAIN only reverses the LLVM store instruction, although other instructions do exist which may alter memory. In practice, these are rarely seen.

Most operations are capable of being programmatically inverted (i.e., they are constructive). However, for those that are not we must resort to state-saving. Values which are destructively assigned must store their data elsewhere to ensure proper reversal. This is explained further in Section 3.1

LORAIN's reversal process begins with a standard C language file which adheres to the ROSS API and includes the forward event handler as well as a declaration for its reverse event handler. ROSS builds a table of function pointers which is used by the simulation runtime to determine which function to call. Therefore, ROSS requires all model-specific function handlers to be declared ahead of time thereby necessitating the declaration of the reverse event handler. Note that at this stage the function body of of the reverse event handler has yet to be defined. This file is then compiled down to its corresponding IR bitcode. The resulting bitcode file can then be passed into the LLVM `opt` tool. `opt` is used to run code transformation and analysis passes. The following list of compiler passes are utilized by LORAIN's overall work-flow. Figure 3 shows how these pass are organized and ordered. The passes developed in this paper are highlighted in bold italics.

- break-constgeps: de-optimizes the instruction sequence for analysis purposes (e.g., struct member references).

This pass is from the latest version of the open-source SAFECode compiler [1, 11].

- lowerswitch: convert `switch` instructions into a sequence of `if` conditions. This allows the developer to *not* have to implement `switch` instructions.
- mergereturn: create a single `return` point and have all other earlier termination instructions jump to it.
- reg2mem: de-optimize the SSA code into non-SSA form. This allows for easier modification of the IR.
- ***aug-struct***: append members to the `message` struct which enables state-saving for destructively-assigned LP state members (see Section 3.1).
- loop-simplify: apply transformations to natural loops to more easily allow future analysis and transformations.[1]
- indvars: simplify loop induction variables.[1]
- ***LORAIN***: This step is implemented as several passes for automatic reversal of parallel discrete-event simulation models. These passes are described in the Sections 3.2 and 3.3.

3.1 Analysis & Message Augmentation

A first pass is executed by LORAIN to find all state values which are destructively overwritten. Upon detection of such a value, the value's type must be retained as well as other identifying aspects of that value. The pass iterates over all `store` instructions, marks them with metadata, and saves them in a set. At this point, the `message` struct provided by the ROSS model is augmented with types matching the overwritten values. This provides a location to store values that may be overwritten by the current event. This approach has been used extensively by hand written models, such as the Yaun et al. TCP model [32].

Given a forward event handler, we need to evaluate all store instructions to the LP state. First, we find all "destructive" stores. Destructive stores are stores in which the original value cannot be recovered given the final value and sequence of steps taken. For example, `i = i * 3;` is reversible: simply divide the new value of `i` by 3. `i = 27;` is non-reversible: given 27, there is no way of knowing the previous value for `i`. If any destructive stores exist, we must make space to save these values prior to overwriting them. We do this by augmenting the `message` struct. This requires modifying the forward and reverse event handler function prototypes. These destructive store instructions are marked with LLVM metadata indicating they have already been taken care of and will not be reversed.

3.2 Instrumentation

Instrumentation is a pass done primarily to facilitate proper reversal. For readability and debugging purposes, all basic blocks are renamed. Basic blocks with two or more predecessors are found and the edge from the predecessor to the current basic block is *split* and an empty basic block is inserted (see figure 4). An instruction to turn on a particular bit in the bit field is inserted in these blocks. The purpose of this construction is to ensure bits are toggled so it is possible to retrace the path through the function's CFG. This leverages the existing bit field `bf` of the ROSS model event handler that developers use in their hand-written models.

[1]This pass is not exemplified by the examples in this paper, though is typically required.

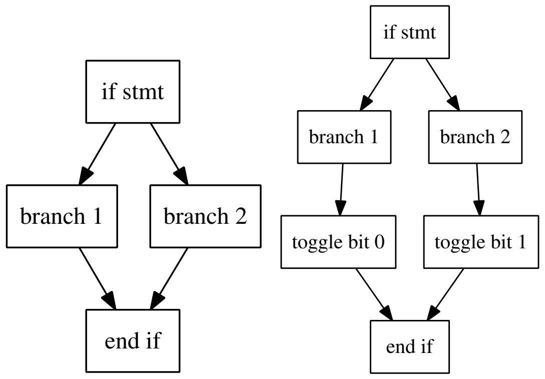

Figure 4: The CFG of the original function *F* (left subfigure) and the modified CFG (right subfigure). The branch 1 and branch 2 blocks are split and instructions to toggle the appropriate bit fields are inserted into the newly created basic blocks.

The next part of the instrumentation pass deals with the `store` instructions. For each `store` instruction LORAIN analyzes it uses and determine if it affects the memory of the LP state. If the `store` instruction has no bearing on the LP state, it is marked with metadata to be ignored in future LORAIN passes. If the `store` instruction affects the LP state but is destructive, we save the value for future rollback. The value is saved in the augmented `message` struct which is created during the `aug_struct` pass as previously noted in Section 3.1. These instructions are also marked with metadata since the reversal is handled in the opposite order with a restorative assignment operation from the `message` struct back to the LP state.

3.3 Inversion

We begin our reverse event handler function with its prototype. Since the forward and reverse event handlers have identical types, we simply request a function type matching the forward event handler. We must do this for ROSS compliance.

For each basic block in our forward event handler we create an empty analog block in our reverse event handler. The entry block always has certain special properties, namely stack allocations for local variables and stack-allocated temporary variables for arguments. A bidirectional, one-to-one mapping is established between the individual forward and reverse basic blocks.

Next, LORAIN analyzes each forward basic block separately. Instructions which are important enough to be reversed are placed on a stack. LORAIN then goes through the stack to build the corresponding reverse basic block. For each forward instruction, LORAIN takes advantage of the LLVM's InstVisitor class which provides a Visitor pattern [14] implementation for all available instruction types. We have the potential to visit any of the following instructions:

- `visitSExtInst:` sign extend the given value. LLVM requires all operands in a comparison are the same length.
- `visitBitCast:` cast between compatible types.
- `visitGetElementPtrInst:` all array and struct accesses must use this instruction to perform proper pointer arithmetic.
- `visitTerminatorInst:` all basic blocks finish with a terminator instruction (e.g., a `return` instruction).
- `visitAllocaInst:` a stack space allocation instruction.
- `visitStoreInst:` a store instruction (can access memory).
- `visitLoadInst:` a load instruction (can access memory).
- `visitCallInst:` for the sake of this work, this should be viewed as a standard function call.
- `visitBinaryOperator:` standard three-address code binary operator. For example, arithmetic operations, shifting operations, etc.

The Terminator instruction requires special consideration. In LLVM, all basic blocks end with a single terminator instruction with no exceptions. Typical terminators are branches to other basic blocks, function calls, or return instructions. A critical observation is that, based on the terminator instructions alone, we are able to reconstruct the control flow graph or in this case, its inverse.

For example, if a given basic block B from the forward event handler F has no predecessors, we can conclude that this block is in fact the entry block. In the reverse event handler F', we must make its counterpart, B', the exit block. Similarly, if B has one predecessor P, then clearly there is an unconditional jump from P to B. This requires the reverse edge from B' to P'. Two or more predecessors means there are multiple paths through the CFG to B. Reversing this will require an instruction that supports two or more possible exits, namely a `switch` statement terminating block B'. Determining which route to take involves evaluating the bit field corresponding to this particular branch point, see figure 5.

Another key instruction is `store`. If it is unmarked with metadata it is reversed. Upon detecting a store instruction in B, we first check to see if it exists in our forward-to-reverse value mapping, i.e. if we have previously seen it from another path. If so, this store has already been successfully reversed and inserted into F'. This could have happen during dependency lookups. If we have not encountered this value before, all of the instruction's dependencies must be found and also reversed. All reversals are again carried out by the InstVisitor class. Here, for each *use* of the value being stored, we must recursively visit each member of the instruction's use-def chain. When all dependencies are satisfied, we can construct the final store and insert it into F'. Any store instruction we have visited is saved in the forward-to-reverse value mapping.

Some function call instructions must also be reversed. For example, RNG calls must be "uncalled" to preserve deterministic behavior. In particular, LORAIN supports the

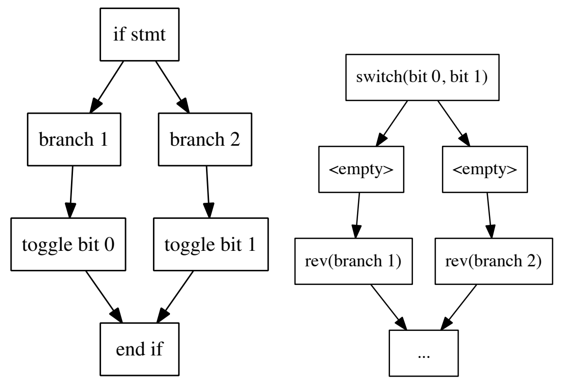

Figure 5: The forward event handler (left subfigure) and reverse event handler (right subfigure), which are inverses of one another. Based on which bit is set, we can find our way back through the CFG.

RNG calls that are provided with the ROSS framework. Some ROSS macros may also ultimately call RNG functions. This is not a problem as the macros have already been expanded by the time we examine the IR. LORAIN's passes catch all RNG calls and emit the corresponding reverse RNG calls. These are supplied by the ROSS framework. All other unrecognized function calls, e.g. `printf()`, are not reversed.

Finally, all function exits must be coalesced; this can be accomplished by using LLVM's `mergereturn` pass. This is required due to the C language invariant that any function must have exactly one entry point. If F had multiple exits, F' would have multiple entry points: a nonsensical proposition for the C language. Having a single exit blocks also gives us an ideal location to restore the values that were previously state-saved in the augmented message struct. This corresponds to destructive store instructions which were marked with metadata.

4. MODELS & PERFORMANCE STUDY

To investigate LORAIN's performance, two complete ROSS models were selected. First, we present the PHOLD model [12] and the entirety of its reversal. This demonstrates a full LORAIN reversal and consists of several complex calls to the RNG. Second, we present Fujimoto's airport model [13]. This is a larger model which demonstrates additional features of LORAIN such as recovering from both constructive and destructive state changes.

4.1 PHOLD

PHOLD is simple benchmark for discrete-event simulations [12]. Upon startup, the system is primed with n starting events. These events are then processed in time-stamp order on their respective starting LP. Only during event processing can another event be dispatched.

The PHOLD pseudo-code is shown in figure 6. Observe that there are potentially two or three RNG calls depending on whether or not the `if` block is entered. Figure 7 is the corresponding basic block representation of figure 6. Figures 8 through 15 are the actual IR from the basic blocks contained within this function. As a whole, we denote the IR of forward event handler with the name F.

```
initialization;
if tw_rand_unif() <= percent_remote then
    dest = tw_rand_integer(0, ttl_LPs - 1);
else
    dest = local_PE;
end
if dest < 0 || dest >= (g_tw_nlp * tw_nnodes()) then
    tw_error(TW_LOC, ”bad dest”);
    return;
end
tw_event *e = tw_event(dest,
tw_rand_exponential(mean));
tw_event_send(e);
```

Figure 6: PHOLD forward event handler.

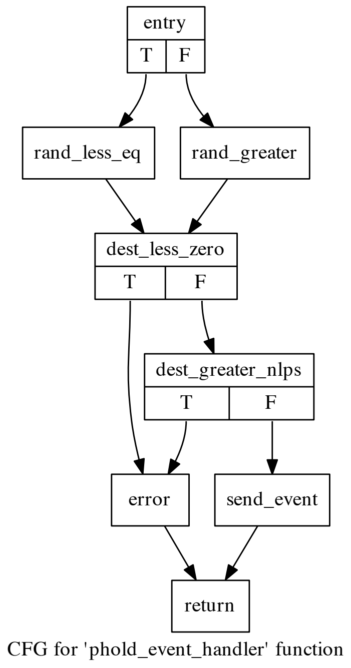

Figure 7: The CFG of the PHOLD forward event handler. The basic block titles have been renamed for clarity.

The `entry` (figure 8) block begins by creating four stack allocations to hold the program parameters and one additional allocation for the `dest` variable. All four parameters are stored into their allocations. Next, a zero value is written over the bit field parameter. The `tw_rng_stream` is extracted from the LP parameter and `rng_gen_val()` is called on it to determine whether PHOLD is generating a remote event or not. If it does, then PHOLD jumps to `rand_less_eq`, otherwise it jumps to `rand_greater`. Note, that in LLVM IR, `getelementptr` is used for calculating memory addresses when dealing with `struct` or array indexing, `bitcast` is used to transform the integer bit field to the appropriate type, and `fcmp` is a floating-point comparison instruction.

The `rand_less_eq` block (figure 9) again extracts `tw_rng_stream` from the LP and calls `rng_gen_val()`. `ttl_lps` is

```
%1 = alloca %struct.PHOLD_state*, align 8
%2 = alloca %struct.tw_bf*, align 8
%3 = alloca %struct.PHOLD_message*, align 8
%4 = alloca %struct.tw_lp*, align 8
%dest = alloca i64, align 8
store %struct.PHOLD_state* %s, %struct.
    PHOLD_state** %1, align 8
store %struct.tw_bf* %bf, %struct.tw_bf** %2,
    align 8
store %struct.PHOLD_message* %m, %struct.
    PHOLD_message** %3, align 8
store %struct.tw_lp* %lp, %struct.tw_lp** %4,
    align 8
%5 = load %struct.tw_bf** %2, align 8
%6 = bitcast %struct.tw_bf* %5 to i32*
store i32 0, i32* %6, align 4
%7 = load %struct.tw_lp** %4, align 8
%8 = getelementptr inbounds %struct.tw_lp* %7,
    i32 0, i32 7
%9 = load %struct.tw_rng_stream** %8, align 8
%10 = call double @rng_gen_val(%struct.
    tw_rng_stream* %9)
%11 = load double* @percent_remote, align 8
%12 = fcmp ole double %10, %11
br i1 %12, label %13, label %21
```

Figure 8: PHOLD `entry` block.

```
; <label>:13                    ; preds = %0
%14 = load %struct.tw_lp** %4, align 8
%15 = getelementptr inbounds %struct.tw_lp* %14,
    i32 0, i32 7
%16 = load %struct.tw_rng_stream** %15, align 8
%17 = load i32* @ttl_lps, align 4
%18 = sub i32 %17, 1
%19 = zext i32 %18 to i64
%20 = call i64 @tw_rand_integer(%struct.
    tw_rng_stream* %16, i64 0, i64 %19)
store i64 %20, i64* %dest, align 8
br label %25
```

Figure 9: PHOLD `rand_less_eq` block.

loaded and 1 is subtracted from it and is then passed into `tw_rand_integer()`. The returned value is assigned to the `dest` variable.

```
; <label>:21                    ; preds = %0
%22 = load %struct.tw_lp** %4, align 8
%23 = getelementptr inbounds %struct.tw_lp* %22,
    i32 0, i32 1
%24 = load i64* %23, align 8
store i64 %24, i64* %dest, align 8
br label %25
```

Figure 10: PHOLD `rand_greater` block.

```
; <label>:25                    ; preds = %21, %13
%26 = load i64* %dest, align 8
%27 = icmp ult i64 %26, 0
br i1 %27, label %35, label %28
```

Figure 11: PHOLD is `dest_less_zero` block.

```
; <label>:28                  ; preds = %25
%29 = load i64* %dest, align 8
%30 = load i64* @g_tw_nlp, align 8
%31 = call i32 @tw_nnodes()
%32 = zext i32 %31 to i64
%33 = mul i64 %30, %32
%34 = icmp uge i64 %29, %33
br i1 %34, label %35, label %36
```

Figure 12: PHOLD `dest_greater_nlps` block.

```
; <label>:35                  ; preds = %28, %25
call void (i8*, i32, i8*, ...)* @tw_error(i8*
    getelementptr inbounds ([58 x i8]* @.str1,
    i32 0, i32 0), i32 90, i8* getelementptr
    inbounds ([9 x i8]* @.str2, i32 0, i32 0))
br label %47
```

Figure 13: PHOLD `error` block.

```
; <label>:36                  ; preds = %28
%37 = load i64* %dest, align 8
%38 = load %struct.tw_lp** %4, align 8
%39 = getelementptr inbounds %struct.tw_lp* %38,
    i32 0, i32 7
%40 = load %struct.tw_rng_stream** %39, align 8
%41 = load double* @mean, align 8
%42 = call double @tw_rand_exponential(%struct.
    tw_rng_stream* %40, double %41)
%43 = load double* @lookahead, align 8
%44 = fadd double %42, %43
%45 = load %struct.tw_lp** %4, align 8
%46 = call %struct.tw_event* @tw_event_new(i64
    %37, double %44, %struct.tw_lp* %45)
call void @tw_event_send(%struct.tw_event* %46)
br label %47
```

Figure 14: PHOLD `send_event` block.

```
; <label>:47                  ; preds = %36, %35
ret void
```

Figure 15: PHOLD `return` block.

The `rand_greater` block, as shown in figure 10, loads the LP identifier into `dest`. The `dest_less_zero` block shown in figure 11 compares `dest` with zero. If it's less, PHOLD jumps to the error block. Otherwise, it jumps to the `dest_greater_nlps` block. The `dest_greater_nlps` block (figure 12) calls `tw_nnodes()` and multiply the result with `g_tw_nlp`. If `dest` is greater than the product, jump to the `error` block. Otherwise jump to the `send_event` block. The `error` block (figure 13) is the error state. Despite the appearance, this state will terminate the program. The `send_event` block (figure 14) creates a random number using `tw_rand_exponential()` to which we add `lookahead`. PHOLD then calls `tw_event_new()` with `dest` as the destination LP, a new future time-stamp, and the current LP as the third argument. Finally, PHOLD calls `tw_event_send()` and jumps to the `return` block. Last, PHOLD's `return` block, shown in figure 15) exits from the event handler function.

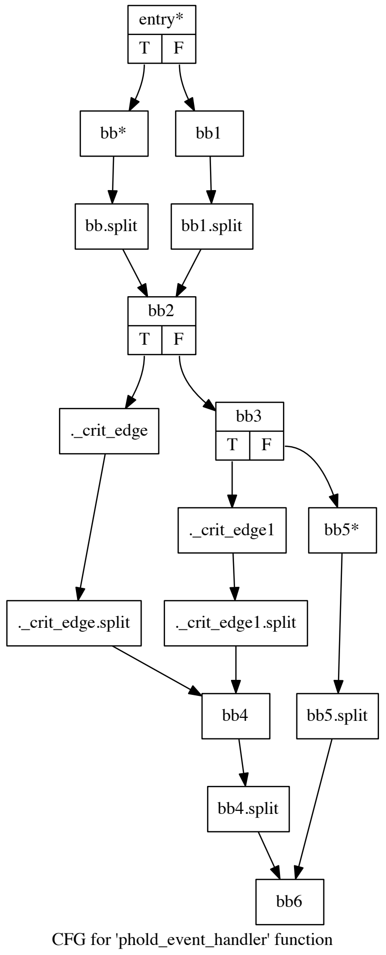

Figure 16: The CFG of the instrumented PHOLD forward event handler function. RNGs calls are marked with *.

With the forward event handler IR of F in hand, LORAIN must both instrument it (figure 16) and then invert the resulting function in order to generate F'. Please note that there are no stores to the simulation (LP) state in the PHOLD model. There are therefore no stores that need to be reversed to generate F'. There are, however, varying numbers of RNG calls that must be un-called. By retracing our steps in reverse through the CFG of F', all RNGs will be successfully rolled back as shown in figure 17.

4.2 Airport

While PHOLD is fairly simple and requires no state changes to be reversed, Fujimoto's airport model [13] does require handling of state modifications while simultaneously maintaining a degree of simplicity. As ROSS has an implementation of this, the airport model was deemed an ideal target for demonstrating further reversing capabilities. Additionally, this model contains examples of all three operations which must be reversed: constructive operations, destructive operations, and random number generation.

Each airport in the airport model maintains three variables: `In_The_Air`, `On_The_Ground`, and `Runway_Free`. The first two are integers: the number of planes in the airport's

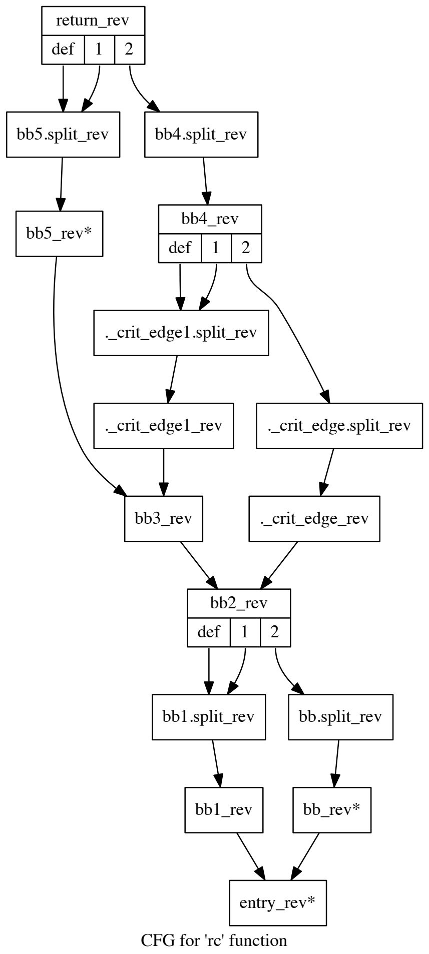

Figure 17: The CFG of the reverse PHOLD event handler. Reverse RNG calls are marked with *.

air-space and the number of planes on the ground at this particular airport. `Runway_Free` is a boolean indicating whether or not the runway is available. In this model, each LP represents an airport and an event represents the action of an airplane (arrive, land, depart). As different events are scheduled, the airplane "moves" between airports.

As aircraft flow from airport to airport, these variables are adjusted appropriately. When a plane x moves from airport A to airport B, the `In_The_Air` variable at A is decremented by one while B's is incremented by one. If x is destined for airport B, `Runway_Free` must be checked to determine whether or not the runway is available for a landing. If so, a landing event must be scheduled. `In_The_Air` will again be decremented and `On_The_Ground` will be incremented. After some amount of time, this aircraft may again be scheduled to depart this airport.

Figure 18 demonstrates a basic block from the forward event handler while figure 19 demonstrates its corresponding basic block from the reverse event handler. These basic blocks represent the forward and reverse cases of the departure event. The blue region in the forward decrements (in the reverse it increments) the state variable `On_The_Ground`. The green and red regions handle random number generation in both forward and reverse directions.

```
bb4:                  ; preds = %LeafBlock1
%91 = load %struct.airport_state** %0, align 8
%92 = getelementptr inbounds %struct.
    airport_state* %91, i32 0, i32 2
%93 = load i32* %92, align 4
%94 = add nsw i32 %93, -1
store i32 %94, i32* %92, align 4
%95 = load %struct.tw_lp** %3, align 8
%96 = getelementptr inbounds %struct.tw_lp* %95,
    i32 0, i32 7
%97 = load %struct.tw_rng_stream** %96, align 8
%98 = load double* @mean_flight_time, align 8
%99 = call double @tw_rand_exponential(%struct.
    tw_rng_stream* %97, double %98)
store double %99, double* %ts, align 8, !jml !10
%100 = load %struct.tw_lp** %3, align 8
%101 = getelementptr inbounds %struct.tw_lp*
    %100, i32 0, i32 7
%102 = load %struct.tw_rng_stream** %101, align 8
%103 = call i64 @tw_rand_integer(%struct.
    tw_rng_stream* %102, i64 0, i64 3)
%104 = trunc i64 %103 to i32
store i32 %104, i32* %rand_result, align 4, !jml
    !10
store i64 0, i64* %dst_lp, align 8, !jml !10
%105 = load i32* %rand_result, align 4
store i32 %105, i32* %.reg2mem, !jml !10
br label %NodeBlock20
```

Figure 18: Demonstration of constructive operations. For example, the blue region consists of subtracting 1 from the On_The_Ground state variable. The green and red regions perform RNG calls.

```
bb4_rev:                  ; preds = %NodeBlock20_rev
%31 = load %struct.tw_lp** %0
%32 = getelementptr %struct.tw_lp* %31, i32 0,
    i32 7
%33 = load %struct.tw_rng_stream** %32
%34 = call double @rng_gen_reverse_val(%struct.
    tw_rng_stream* %33)
%35 = load double* @mean_flight_time
%36 = load %struct.tw_lp** %0
%37 = getelementptr %struct.tw_lp* %36, i32 0,
    i32 7
%38 = load %struct.tw_rng_stream** %37
%39 = call double @rng_gen_reverse_val(%struct.
    tw_rng_stream* %38)
%40 = load %struct.airport_state** %3
%41 = getelementptr %struct.airport_state* %40,
    i32 0, i32 2
%42 = load i32* %41
%43 = sub i32 %42, -1
store i32 %43, i32* %41
br label %LeafBlock1_rev
```

Figure 19: Demonstration of "reverse" constructive operations. The blue region consists of adding 1 to the On_The_Ground state variable. The green and red regions are reverse RNG calls.

Likewise, as destructive operations are performed on the various values in the forward event handler, they must be collated and their values saved for possible future restoration. These values are determined at an earlier stage by an analysis pass and passed to the current stage via metadata. The forward event handler is configured to save these values at the beginning of the function (before their values are overwritten). The emitted reverse event handler will restore the values in its (single) exit block.

The time deltas for each event will be generated by ROSS' reversible RNG. For example, when a landing event is scheduled, it must be at some (bounded) random time in the future from the current event time. RNG calls must be un-called so as to not disturb the random sequence. In other words, if RNG R produces r_1 and is un-called, the very next call on R must produce an identical value to r_1.

4.3 Experimental Setup

For our experimental study, we consider the following three code versions for both PHOLD and airport models:

- *O0: unoptimized.* Both models were passed through LORAIN and the modified forward and synthesized reverse event handlers were compiled without applying any further optimization (`-O0`)
- *O3: optimized.* Both models were passed through LORAIN but the modified and forward and synthesized reverse event handlers were further optimized (`-O3`)
- *HW: hand-written.* Both models had modified forward and reverse event handlers written by hand and used optimization level `-O3`. We emphasize here that the LORAIN passes are not used.

The ROSS framework itself is always compiled with `-O3` turned on regardless of which approach we were using. The results were gathered using a 2.1 GHz 64 core AMD machine with 512 GB of RAM running GNU/Linux and clang 3.2. Only 32 cores were ever used in experimentation.

The PHOLD model is configured with 16 LPs and 1 event per LP when run on 2 cores (small workload case) and 1,048,576 LPs with 16 events per LP when run on 32 cores (large workload case). The airport model is configured 1024 airport LPs with 1 plane event per airport LP using 2 cores (small workload case) and 1,048,576 airport LPs with 16 plane events per airport LP using 32 cores (large workload case). All box-plot graphs were generated by running each simulation 100 times.

A first pass ROSS model verification relies on a simulation's net event count. This count indicates the number of committed events that occurred for a specific experiment. It is necessary (but not sufficient) that the net event count is identical for all parallel (any number of processors) and serial runs of the same model configuration. This approach was used as a sanity check when model development was done by hand.

Table 1: Median performance of PHOLD and airport models measured in net events per second.

Experiment	O0	O3	HW
2 core PHOLD	946,164.80	971,834.45	974,658.30
32 core PHOLD	6,070,813.30	6,155,326.35	6,178,446.40
2 core Airport	1,921,355.20	2,039,775.40	2,062,018.85
32 core Airport	6,713,835.70	6,854,997.75	7,044,712.05

4.4 Performance Results

The performance graphs for the PHOLD and airport models configured with 2 processors and small workload are

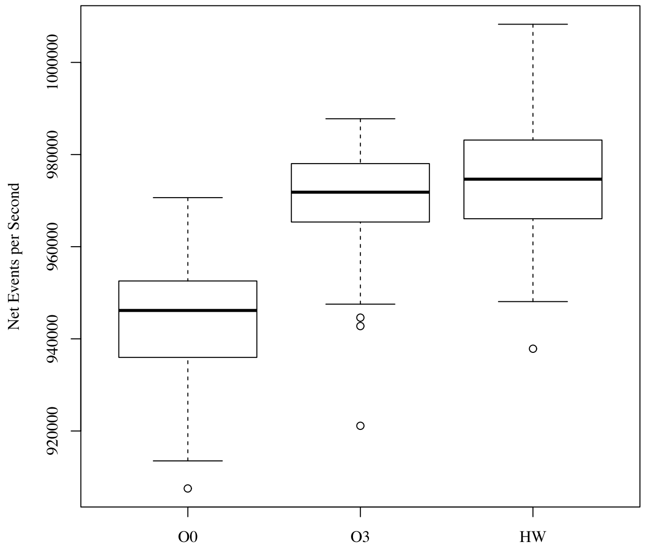

Figure 20: The performance of the PHOLD model using 2 cores. The three plots are O0, the unoptimized generated IR, O3, the optimized generated IR, and HW, the model including a hand-written reverse event handler.

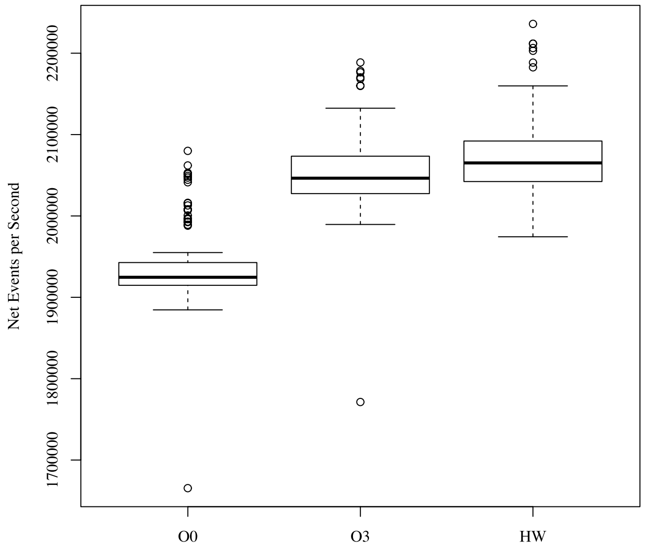

Figure 21: The performance of the airport model using 2 cores. The three plots are O0, the unoptimized generated IR, O3, the optimized generated IR, and HW, the model including a hand-written reverse event handler.

shown in figures 20 and 21. The 32 core, large workload performance graphs are shown in figure 22 for PHOLD and figure 23 for the airport model. For both models, the unoptimized version generated by LORAIN is always worse than the other two executions. However, the gap narrows between optimized LORAIN model code and the hand-written approach. Using the median from the hand-written approach as the baseline, the LORAIN-generated airport model was only 1% less efficient while running with 2 cores and 2.7% less efficient when running with 32 cores. The PHOLD

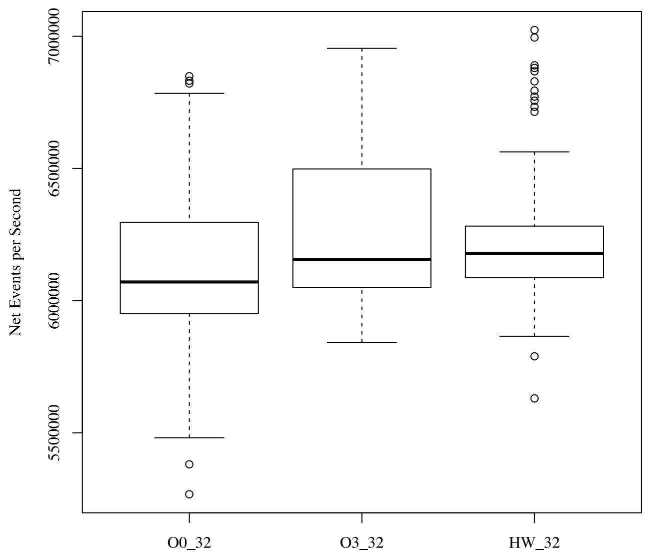

Figure 22: The performance of the PHOLD model using 32 cores. The three plots are O0, the unoptimized generated IR, O3, the optimized generated IR, and HW, the model including a hand-written reverse event handler.

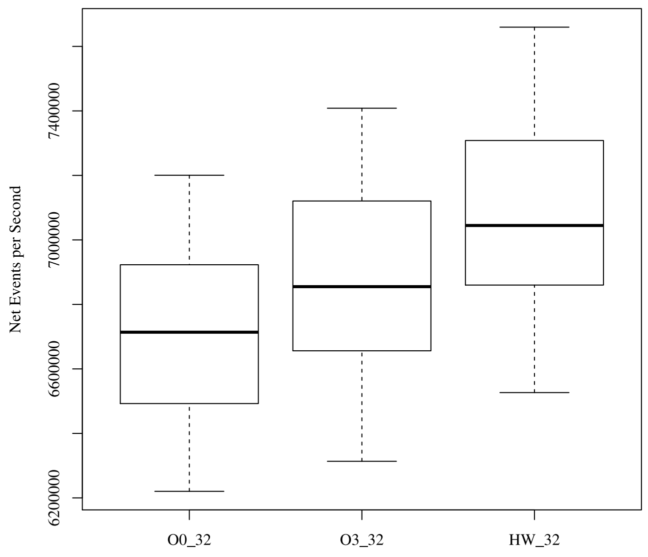

Figure 23: The performance of the airport model using 32 cores. The three plots are O0, the unoptimized generated IR, O3, the optimized generated IR, and HW, the model including a hand-written reverse event handler.

model was 0.3% less efficient than the corresponding hand-written model when run with 2 cores and only 0.4% less efficient when run on 32 cores.

As expected, hand-written instrumentation and reverse event handlers always outperform their synthesized counterparts. Despite the hand-written approach always having the best performance, the optimized LORAIN approach is able to come very close to the hand-written performance in all cases. Although the hand-written model appears to be more efficient in figure 23, table 1 indicates a median performance difference of 189,714.30 events-per-second (for models that are executing at a rate of 6 to 7 million events-per-second) between hand-written and optimized LORAIN code.

Notably, the range of the performance results was larger than one might expect. This is attributed to OS jitter [6]. The Linux machine on which we performed these measurements had quite a few extraneous processes, network traffic, and other external factors that would contribute to a variance in the runtimes. Additionally, no attempt at locking the MPI processes to specific PEs, which may have tightened the results slightly, was made.

5. RELATED WORK

Beyond Parallel Simscript II.5 [29], RCC [25] and Backstroke [30], there are number of previous related works within the PDES community that have a similar goal. For example, Nicol and Heidelberger [22] create the Utilitarian Parallel Simulator (UPS) which creates a set of modeling language extensions that enable model components developed for serial execution to execute in parallel using a conservative event scheduling approach. A similar idea is that of the High-Level Architecture (HLA) [13] except that the end goal is one of simulator interoperability and not improved parallel performance. The idea here is that a heterogeneous, federation of serial simulators is combined and made to interoperate with much less effort than a fully, integrated rewrite of all models executing within the federation. In the context of network simulation, Riley et al. [27] create a specialized approach called *Backplane* that enables the federations of different network simulators. They demonstrate the capability of *Backplane* using a parallel version of *ns* (pdns), GTNetS and GloMoSim.

Outside of the PDES community, there has been growing interest automatic reverse execution of programs, especially in the context of program debugging. Akgul et al. [3] provide an assembly language level reverse execution approach for the purposes of application debugging. Biswas and Mall [7] provide an reverse execution approach for purposes of debugging. The earliest work (e.g., 1988) in reverse execution for debugging appears to have been done by Pan and Linton [23]. This early work is especially interesting because there target was not serial programs but parallel programs that are much harder to debug. Like other approaches, it involved some varying degree of program state-saving to support the reverse program execution.

LORAIN's overall approach is similar to Weiser's slicing methodology [31]. Slicing is a method of reducing the program down to specific *slicing criterion*, or points of interest (in this case, LLVM Values), as well as the remainder of the program that may possibly affect it. This subset of the program is referred to as a *slice*. Slicing is useful in many areas [28]; applications include debugging purposes, parallelization, and more directly related to this work, reverse engineering.

Bahi and Eisenbeis [4] present a lower bound on the spatial complexity of a DAG with reversible operations and an approach for finding the minimum number of registers required for a forward and backward execution of a DAG. They also define an energy-based garbage collection metric as the additional number of registers needed for the reversible execution of the original computation. Experimental results report that garbage size is never more than 50% of the DAG size.

Finally, Perumalla [24] provides an excellent survey of the field of reverse/reversible computing.

6. CONCLUSIONS & FUTURE WORK

In this work we have presented LORAIN, a tool capable of automatic analysis and instrumentation of a specific forward event handler and the emission of a reverse event handler capable of inverting all model state changes. Early results show the automatically generated code is on par with handwritten reverse event handlers. LORAIN currently only supports models adhering to the ROSS simulator API.

LORAIN can deliver over 99% the performance of handwritten models and our results have shown no less than 97% of the performance on a more complex model. While at an early stage, having such a tool will undoubtedly ease the burden on model developers and allow them to more quickly develop optimistic models that better exploit current and future HPC systems.

Currently, LORAIN does not support inter-procedural analysis or transformations, though such support may be added in the future. Preliminary loop support exists although loop handling improvements are certainly a high-priority addition. All primary simulation languages such as C, C++, and Fortran are supported in LLVM, paving the way for future support in LORAIN.

Additionally, we are interested in leveraging other LLVM advances that may help to improve the overall quality of LORAIN's modified forward and synthesized reverse event handler functions. For example, Zhao et al. [33] develop a proof technique for proving SSA-based program invariants and compiler optimizations. Here, they apply this technique and formally verify a variant of LLVM's `mem2reg` transformation and show the verified code generation performs on par with LLVM's unverified implementation.

The overall issue of verification is very much an open question in the context of reverse computation especially at the IR level. It is clear from the *Backstroke* compiler that source-to-source tools can maintain program semantics that may enable improved reverse code transformations. However, it is unclear if verification of the generated code at the LLVM IR level is equally good as source-to-source reverse code.

7. ACKNOWLEDGMENTS

This research was partially supported by the Office of Science of the U.S. Department of Energy under Contracts DE-SC0004875 and DE-AC02-06CH11357.

8. REFERENCES

[1] SAFECode (latest version). `http://sva.cs.illinois.edu/index.html`, 2014.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Trans. Softw. Eng. Methodol.*, 13(2):149–198, Apr. 2004.

[4] M. Bahi and C. Eisenbeis. Spatial complexity of reversibly computable dag. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 47–56, New York, NY, USA, 2009. ACM.

[5] 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.

[6] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 11(1):3–16, 2008.

[7] B. Biswas and R. Mall. Reverse execution of programs. *SIGPLAN Not.*, 34(4):61–69, 1999.

[8] C. Carothers, D. B. Jr, and S. Pearce. ROSS: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648 – 1669, 2002.

[9] 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.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[11] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157, New York, NY, USA, 2006. ACM.

[12] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, Oct. 1990.

[13] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[15] E. Gonsiorowski, C. Carothers, and C. Tropper. Modeling large scale circuits using massively parallel discrete-event simulation. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 127–133, Aug 2012.

[16] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. In *Compiler Construction*, pages 81–100. Springer, 2012.

[17] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[19] N. Liu and C. D. Carothers. Modeling billion-node torus networks using massively parallel discrete-event simulation. In *Proceedings of the 2011 IEEE*

Workshop on Principles of Advanced and Distributed Simulation, PADS '11, pages 1–8, Washington, DC, USA, 2011. IEEE Computer Society.

[20] C. J. Lobb, Z. Chao, R. M. Fujimoto, and S. M. Potter. Parallel event-driven neural network simulations using the hodgkin-huxley neuron model. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, pages 16–25, Washington, DC, USA, 2005. IEEE Computer Society.

[21] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete event simulation. In *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS12) held as part of SC12*, 2012.

[22] D. Nicol and P. Heidelberger. Parallel execution for serial simulators. *ACM Trans. Model. Comput. Simul.*, 6(3):210–242, July 1996.

[23] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD '88, pages 124–129, New York, NY, USA, 1988. ACM.

[24] K. Perumalla. *Introduction to Reversible Computing.* CRC Press, 2013.

[25] K. S. Perumalla and R. M. Fujimoto. Source-code transformations for efficient reversibility. 1999.

[26] K. S. Perumalla and V. A. Protopopescu. Reversible simulations of elastic collisions. *ACM Trans. Model. Comput. Simul.*, 23(2):12:1–12:25, May 2013.

[27] G. F. Riley, M. H. Ammar, R. M. Fujimoto, A. Park, K. Perumalla, and D. Xu. A federated approach to distributed network simulation. *ACM Trans. Model. Comput. Simul.*, 14(2):116–148, Apr. 2004.

[28] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[29] J.-J. Tsai and R. M. Fujimoto. Automatic parallelization of discrete event simulation programs. In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, pages 697–705, New York, NY, USA, 1993. ACM.

[30] 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.

[31] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[32] G. Yaun, C. D. Carothers, and S. Kalyanaraman. Large-scale TCP models using optimistic parallel simulation. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, PADS '03, pages 153–, Washington, DC, USA, 2003. IEEE Computer Society.

[33] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. *SIGPLAN Not.*, 48(6):175–186, June 2013.