Proceedings of the 2015 Winter Simulation Conference
L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.

TIME WARP STATE RESTORATION VIA DELTA ENCODING

Justin M. LaPre, Elsa J. Gonsiorowski, Christopher D. Carothers

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA

John Jenkins, Philip Carns, and Robert Ross

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA

ABSTRACT

Optimistic simulation yields impressive performance gains for many models. State saving is a quick way to provide the rollback mechanism required for this approach, but it has some drawbacks: it may not handle models with massive states or be able to support memory-constrained systems. This work presents a novel approach to state saving by storing only the relative changes caused by an event. Compressing these deltas allows retaining a greater number of noncommitted events and allows Time Warp to further exploit parallelism in a window less constrained by memory limitations. By compressing the data, we realize greater returns in performance and avoid memory limitations on event / state sizes. Compression ratios over 200 are observed; and, despite chosen pathological conditions, state restoration is fast and efficient. Runtimes are often faster using delta encoding than are their conservative counterparts, without the need for complex reverse code or large memory consumption.

1 INTRODUCTION

Parallel discrete event simulation (PDES) relies on a global synchronization algorithm across the set of nodes executing the simulation. Traditional synchronization algorithms are *conservative* in terms of event processing: no event is processed without a guarantee of local serializability. That is, before processing an event at time t, each node ensures that it has received and processed all events with a timestamp less than t. This is maintained through the use of *lookahead*: a predetermined minimum length of time delay for each event. After a period of global communication all nodes are synchronized to a global virtual time (GVT). Each node can then process any events with a timestamp less than $GVT + lookahead$. For some models, lookahead can be difficult to compute (see Section 5.6 of Fujimoto 1999).

In contrast to conservative simulation, less rigid synchronization algorithms exist. These are referred to as *optimistic* synchronization algorithms. The most well-known is Time Warp (Jefferson 1985). Issues surrounding high-performance Time Warp PDES have been largely solved. Various state-saving approaches have been investigated (Lin et al. 1993; Rönngren et al. 1996) as well as novel methods of state reconstruction (Vulov et al. 2011; LaPre et al. 2014). While much progress has been made, some models still prove difficult to insert into a Time Warp context. For example, erratic and unpredictable state modifications given a large state space remain difficult to model effectively: too many possibilities must be accounted for in order to return the state to its prior values while using reverse computation.

978-1-4673-9743-8/15/$31.00 ©2015 IEEE

While state saving can overcome this particular problem, the memory footprint required for Time Warp to truly shine may be prohibitive. For example, the IBM Blue Gene/Q A2 processor has only 16 gigabytes of memory per node while supporting up to 64 hardware threads. This can constrain certain configurations to 256 megabytes of memory for the statically linked binary, simulator data structures, and optimistic memory (e.g., state copies). A 1-megabyte state could have at most 256 copies given a combined binary and simulator state size of 0 (which is not possible). Clearly, large state sizes can severely limit the discovery of any inherent parallelism in the model.

Tools such as *git* (Chacon 2009) seem to overcome similar issues through the use of technologies related to `diff` (Hunt and McIlroy 1976). Such tools compute *deltas* to store differences from one version to another. Storing these deltas may require similar amounts of space as state saving (the fact that two revisions are identical must be retained after all). Given the *discrete* nature of PDES, we can exploit the likelihood that state differences between events should be small. Small `diff` sizes imply a largely unchanged state (and therefore deltas containing many zeroes), which compresses well.

When selecting a compression algorithm, should precedence be given to size or speed? As already stated, the size of the `diff` should be small. Clearly this answer should imply that a fast algorithm is favorable over one that is space efficient. Given this conclusion, LZ4 was chosen because it is an extremely fast compression algorithm (Collet 2014a). It is used in the ZFS filesystem for block-based, on-the-fly compression (Kiselkov 2013) as well as the squashfs filesystem under Linux (Torvalds 2014).

LZ4 (Collet 2014b) is a fast compression algorithm requiring a small amount of space overhead for compression. While LZ4 compression speeds are typical of LZO (which is faster than gzip), decompression speeds can be faster than LZO (Larabel 2013).

The contribution of this paper is that it provides a stop-gap solution for models that contain events that are not well suited for using reverse computation or consume significant amounts of memory, making copy-state approaches infeasible. For example, some model events may require a complex nesting of while-loops that cannot be easily or efficiently reversed (LaPre, Gonsiorowski, and Carothers 2014). Delta encoding solves this issue by computing state change deltas only after an event has completed execution. Additionally, the delta encoding approach provides the benefits of incremental state saving but without requiring the specific identification of which state elements change. Moreover, because delta encoding is done on a per-event basis, reverse computation and delta encoding can be mixed thereby enabling modelers to take advantage of reverse computation in events for which it is well suited (e.g., constructive assignments and simple loop constructs).

The remainder of this paper is organized as follows. Section 2 describes the ROSS Time Warp engine and LZ4 compression scheme. Section 3 describes our approach to realizing state-delta compression within the ROSS kernel. In Section 4 we evaluate our parallel performance results. Related work is reviewed in Section 5. In Section 6 we present our conclusions and briefly discuss future work.

2 BACKGROUND

In this section we discuss Rensselaer's Optimistic Simulation System (ROSS), briefly discuss the problem of delta encoding, and then discuss data compression techniques.

2.1 ROSS

The simulator chosen for this study is ROSS (Carothers, Bauer, and Pearce 2002). ROSS is capable of both conservative and optimistic execution modes. It is written primarily in the C language and is built on top of an MPI (Gropp, Lusk, and Thakur 1999) layer, allowing it to leverage some built-in MPI primitives such as *all-reduce*. Its focus is on both speed and efficiency: Barnes et al. (2013) demonstrated event rates over half a trillion events per second. While ROSS is capable of (and, in fact, largely derives its speed from) reverse computation, this work is focused on models for which a reverse event handler would be difficult to implement.

Logical processes (LPs) process *events* in time-stamp order. Inter-LP communication is possible by sending a message to another LP. While global (in-order) ordering is maintained during conservative simulation, this may severely limit the amount of parallelism achieved by optimistic simulation. Out-of-order execution may reveal nondependencies (not to be confused with *antidependencies*, Hennessy and Patterson 1996) that the modeler was unaware of or was unable to anticipate.

LPs are only allowed to share state (or otherwise interact with one another) through time-stamped messages. This allows for a well-defined entry-point to alter an LP via state modification. By restricting the access of state variables to messages only, order is enforced within the API and leaves the ROSS runtime free to schedule events effectively.

2.2 Delta Encoding

The problem of finding similarities in data is not new. One well-known approach is the *longest common subsequence* (LCS) problem. Typically used on two strings of data, the problem seeks to find the longest sequence common to both input strings. It can be solved through dynamic programming as longer solutions are built upon earlier and shorter subsolutions.

The `diff` program (Hunt and McIlroy 1976) is a tool based on LCS for discovering changes to data, in this case files. The output of the `diff` program consists of lines that differ between two "versions" of the same file. Building upon `diff` is the Revision Control System (RCS) (Tichy 1982), which specializes in tracking changes to source code files. RCS stores multiple revisions of files efficiently by saving the differences relative to a fixed point in the file's history as opposed to keeping a separate and complete copy for every unique revision.

2.3 Data Compression

To reduce the size of the LP state-deltas, we turn to various data compression techniques. Typically, data exhibits some degree of redundancy. Eliminating these redundancies will necessarily reduce the size of the data. When data redundancy is high (e.g., there exist many zeroes in it), these compression schemes can yield impressive size reductions. Compression schemes fall into two camps: lossy and lossless. Lossy (de)compression may not recreate the data 100% accurately, but the results are suitable for some uses; for example, the MP3 audio coding format discards extreme high and low frequencies, which the human ear is likely unable to perceive. Conversely, lossless (de)compression will not lose any data whatsoever. For obvious reasons, we opt for lossless compression. While many zeroes are expected between inter-event state values, in general little else is known about the composition of the state data or the resulting deltas. This situation is to be expected because ROSS is simply a simulation engine: it has no further information pertaining to the model specifics. Therefore, a universal coding scheme is required.

LZ77 (Ziv and Lempel 1977) is a compression scheme that is both lossless and universal. It performs compression through the use of a variable-sized "sliding window" within which it matches substrings. Matches are encoded by a tuple of values: the *length* of the match, the *offset* indicating its location within the window, and the next symbol following this matching. Given these encodings, it is simple, efficient, and fast to recreate the original data.

While LZ77 provided the conceptual foundation for many data compressors that followed, many tweaks and optimizations were applied; and the LZ77 family of encoders has greatly improved since its inception. LZ4 (Collet 2014b) is one such derivative with an emphasis placed on simplicity and speed. Simplicity aside, its speed may be attributed to several design choices including early exiting upon detection of incompressible data and sacrificing high compression ratios in favor of improved execution time. Emphasis was placed on speed of compression since the state delta will require compression after each event is executed. Furthermore, decompression speed is approximately 3-4$\times$ faster than the corresponding compression time. Although any lossless compression format could be used in delta encoding, we chose to use LZ4 by default because it offers a good tradeoff between speed and compression ratio for general-purpose compression.

3 APPROACH AND IMPLEMENTATION

Our approach is straightforward. We use ROSS in an optimistic (Time Warp, Jefferson 1985) mode. To have minimal changes to the ROSS kernel, we require that delta encoding functions be called explicitly by the model from within forward and reverse event handlers. In the forward event handler, we require a call to `tw_snapshot()` before any state is changed, as well as a call to `tw_snapshot_delta()` before exiting the function. In the reverse event handler, a call to `tw_snapshot_restore()` returns the state to its previous values.

```
int tw_snapshot_delta(tw_lp *lp, size_t state_sz) {
    int i, ret_size = 0;
    char *cur_state = lp->cur_state;
    char *snapshot = lp->pe->delta_buffer[0];

    for (i = 0; i < state_sz; i++)
        snapshot[i] = cur_state[i] - snapshot[i];

    ret_size = LZ4_compress(snapshot, lp->pe->delta_buffer[1], state_sz);
    if (ret_size < 0)
        abort();

    lp->pe->cur_event->delta_buddy = buddy_alloc(ret_size);
    assert(lp->pe->cur_event->delta_buddy);
    lp->pe->cur_event->delta_size = ret_size;
    memcpy(lp->pe->cur_event->delta_buddy, lp->pe->delta_buffer[1], ret_size);

    return ret_size;
}
```

Listing 1: `tw_snapshot_delta()` implementation. This function has been simplified to save space.

In order to realize these minimal changes to the ROSS API, the ROSS core required the addition of certain flags to indicate whether delta encoding was to be used for the current execution. Additionally, the ROSS random number generation (RNG) code (based on the CLCG4 implementation, L'Ecuyer and Andres 1997) required adding a counter to the generator itself; this enables the "undoing" of RNG calls to ensure determinism. Because of the (potentially) varying number of unique RNG streams for a given ROSS model, however, automatically reversing all RNG operations is not currently possible and requires manual intervention.

In addition to these changes, the ROSS runtime requires a memory allocation in which to store the compressed delta. Unfortunately, the ROSS API does not support runtime memory allocation. To overcome this problem, we implemented a *buddy system allocator* (Knowlton 1965).

A classical buddy system allocation scheme starts with one large block of memory (which is allocated during system initialization). When a memory request of size m is made, the size of the request is rounded up to m', the nearest power of 2 such that $m' \geq m$. (The remaining memory, `sizeof(`m'`)-sizeof(`m`)`, is referred to as *internal fragmentation* and is one of the main drawbacks to using a buddy system allocator.) That request is then passed on to the buddy system, which will repeatedly halve the memory blocks until the size of the block and the size of the request are equal. The memory block is then returned to fill the original request. Metadata regarding individual blocks (e.g., block size, current usage) are stored within the blocks themselves. The size of the initial buddy block can be set at runtime.

We also had to decide where to store our allocations at a logical location within the simulation. We chose to store the pointer to the allocated memory within each event, although no allocation takes place until `tw_snapshot_delta()` is called from within the forward event handler. By delaying allocation until this point, we allow for varying-sized allocations as well as eliminating the need for sending deltas

with their events. While events may travel to different processors, memory allocations should be considered local and stationary. In the event that ROSS supports LP migration, this decision may need to be revisited.

```
void rc(phold_state * s, tw_bf * bf, phold_message * m, tw_lp * lp) {
    long count = m->rng_count;
    while (count--) {
        tw_rand_reverse_unif(lp->rng);
    }
    tw_snapshot_restore(lp, lp->type.state_sz, lp->pe->cur_event->delta_buddy,
                        lp->pe->cur_event->delta_size);
}
```

Listing 2: Reverse event handler used for this study.

Given these additions to ROSS, we are now able to describe the behavior and functionality of the delta encoding API. ROSS will proceed to execute events on all LPs in timestamp order. At the beginning of the (forward) event handler, `tw_snapshot()` should be called. This will effectively `memcpy()` the LP state data into a preallocated snapshot buffer. Immediately before exiting this event handler, `tw_snapshot_delta()` must be called. Within `tw_snapshot_delta()`, we derive the delta by subtracting the snapshot buffer from the current state. See Listing 1 for the implementation of our `tw_snapshot_delta()` function. Following that, we compress our snapshot with `LZ4_compress()` which returns the resulting compressed data size. With that piece of data in hand, we can request a suitably-sized block of memory from the buddy allocator and perform a `memcpy()` to copy the data into the memory block and store it in the current event.

Should an event be committed, we simply reclaim the delta buffer. If an event gets rolled back, we must undo our RNG calls and restore our state to its previous values. The RNG calls can be reversed with a simple `while` loop by taking advantage of new ROSS RNG counter functionality requested by the ROSS community. Our state can be restored by calling `tw_snapshot_restore()`, which behaves as follows. The compressed delta information is uncompressed, and the resulting delta is reverse-applied to the current event state. The memory block containing the delta information is deallocated to the buddy allocator, which may or may not be able to coalesce adjoining free blocks into a larger block. See Listing 2 for a condensed version of the actual code used with comments removed for brevity.

We note that the code in Listing 2 will be largely the same *regardless of the specific model*. The delta encoding API is data-agnostic and will restore the state to their previous values regardless of the type or size of the data. In fact, the only modifications to Listing 2 would be using multiple RNG streams, since each would need to be fully reversed in order to maintain determinism.

One notable exception, however, is pointers to data. To see why, one must observe that a pointer can remain the same while changing the data at the pointed-to location. Pointers to data *within* the LP state will of course be handled properly, although C has no mechanism to enforce such invariants. LP state with pointers is not currently supported for the following reasons: state data should almost always be *local* as sharing data will typically introduce other issues, and one often can retain the same functionality with equivalent code that does not use pointers.

4 PERFORMANCE STUDY

The target for this performance study is a modified version of the synthetic PHOLD (Fujimoto 1990) benchmark herein referred to as *BPHOLD*. The BPHOLD model sends messages to other LPs *remote* percent of the time, where *remote* is a variable specified at runtime. Additionally, we added an array of 4,090 `long int` data (amounting to 32,720 bytes) to the LP state. During each event, we uniformly choose a random number between 1 and 1,022 (one-quarter of the state size) and change that many values in the state to randomly generated `long int` values. To avoid bias, we randomly shuffle the positions

Table 1: Event rates (events / second) for the various configurations.

Cores	512	2,048	8,192
Conservative	8,700.3	28,262.4	92,574.1
Delta-Encoding Optimistic	241,197.0	947,506.9	3,637,916.3

of each value we replace. The goal of these modifications is to mimic a model having a large amount of state and that would be difficult to reverse by using reverse computation.

We also evaluate a model implementation of the Optimized Link State Routing (OLSR) protocol as described in LaPre et al. (2012). Link state routing protocols such as OLSR contain a (often sizeable) global "map" of the network in each node (when referring to OLSR objects the term "node" is often used; in our model, an OLSR node is represented by 1 LP). Changes in the map are typically localized and small due to events such as node discovery. Nodes exchange periodic HELLO messages to discover each other and construct a 1-hop and 2-hop neighbor set. Multipoint relay (MPR) nodes are chosen to retransmit any messages overheard from any node within its 1-hop neighbor set. Coupled with topology control (TC) messages which disseminate connectivity information, a network topology can be established and routes can be calculated. We simulate 262,144 OLSR LPs each of which consists of 22,672 bytes of various repositories for neighbor, MPR, and topology tuples to name a few; inter-node messages have the capacity to change one or more of these. For a more detailed discussion of OLSR or its simulation model, interested readers are referred to RFC 3626 (Clausen and Jacquet 2003) or LaPre et al. (2012).

4.1 Experimental Setup

Our target platform for this work was the IBM Blue Gene/Q at the Rensselaer Polytechnic Institute Center for Computational Innovations. Each node of the Blue Gene/Q contains 18 cores: 16 for task-level computation, 1 for OS functionality, and 1 fail-over core in the event that another core stops functioning properly. Each core has 4 hardware threads capable of running their own MPI context. Thus, each node can run $16 \times 4 = 64$ independent MPI ranks. Because of memory constraints of the model under consideration, however, we run only 16 ranks per node. The cores operate at 1.6 GHz, and each node has 16 gigabytes of memory and 32 megabytes of L2 cache. These nodes are connected by a 5-D torus network capable of 2 GB/s (Chen et al. 2011).

4.2 Performance Results

4.2.1 BPHOLD

These results were collected with the following flags: `--start-events=16 --extramem=2048 --report-interval=0.10 --clock-rate=1600000000.0 --lookahead=0.0001 --gvt-interval=512 --batch=8 --nlp=16 --remote=1.0 --end=128`. In this study, we always use 100% remote events. We use conservative mode compared with delta encoding in optimistic mode. For this study, we added the `--buddy_size` option and set it to 26. This allocates $2^{26} = 64$ megabytes per processor for our buddy allocator.

Figure 1 shows the effective compression ratio compared with the percentage of state data that was changed. We say "effective" here because we do not actually compress the state but rather only the resulting delta relative to a given event. Figure 1 also shows that as the amount of state changed increases, so too does the resulting size, necessarily driving the compression ratio down.

When the percentage of the state changed is close to zero, LZ4 compresses the deltas at compression ratios over 200. As more state data is changed, however, the compression ratio decays quickly. Changing 3% of the state yields a compression ratio of 20, and changing 6.5% yields a compression ratio of 10. This trend continues to lessen and at 25% of state change, we arrive at a compression ratio of 3.

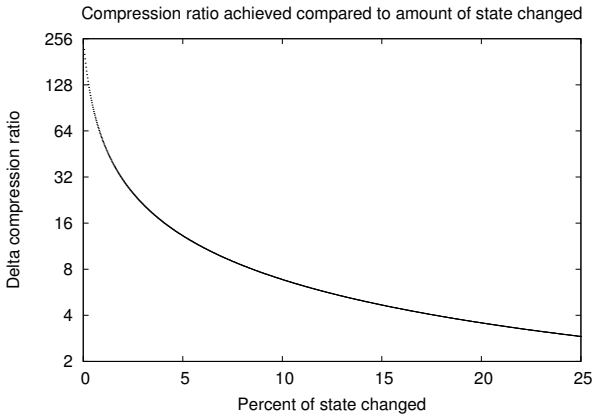

Figure 1: Compression ratios for an increasing amount of state changed. For a small amount of state change, we see compression ratios of over 200. As we approach 25% change the compression ratio approaches 3. This plot contains 1,022 points.

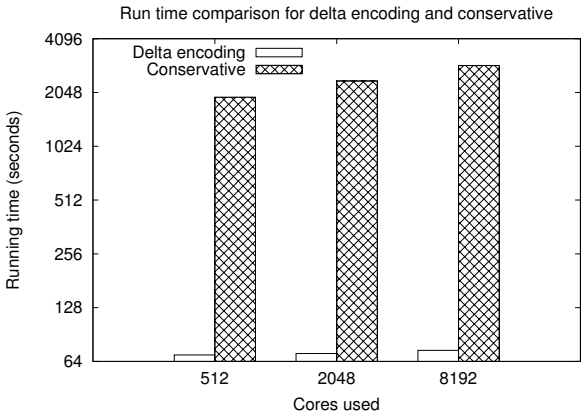

Figure 2: Running times for delta-encoding optimistic mode and conservative modes vs. the number of cores used. All models were run with 100% remote messages. Delta-encoded optimistic runs were on average $\approx 35\times$ faster than conservative runs.

Running times for 512, 2,048, and 8,192 cores are shown in Figure 2 for both conservative and optimistic mode using delta encoding. Table 1 contains the event rates for the corresponding configurations. Optimistic mode using delta encoding for all three core counts is on average $35\times$ faster than the corresponding conservative runs (at this particular value of lookahead).

These increased runtimes can be attributed to the time spent in computing GVT (Jefferson 1985). GVT represents the smallest unprocessed event timestamp in the system. In ROSS, calculating GVT requires periodically calling `MPI_Allreduce()`, a blocking function. As Figure 3 shows, the time spent computing GVT for conservative runs dwarf their optimistic counterparts. As Fujimoto stated, "...conservative algorithms rely on lookahead to achieve good performance." (Fujimoto 1999) The comparatively small lookahead values contribute to an overall lower-performing simulation.

While no reverse event handler exists for the complex BPHOLD model, a comparison of delta encoding to reverse computation is still warranted. See Figure 4 for the results from an *unmodified PHOLD* model. Carothers and Perumalla (2010) demonstrate that small lookahead values can cripple simulator performance but recover quickly with modest increases; Figure 4 exhibits the same trend. The optimistic approach with reverse computation always outperforms the other two approaches for the given lookahead values. Delta encoding maintains runtimes within a constant factor while conservative synchronization runtimes are inversely proportional to the lookahead.

Another important result supported by Figure 4 is that conservative synchronization continues to enjoy improved performance as lookahead increases. For lookahead value of 0.01, conservative is approaching delta encoding performance and at 0.1 the conservative approach overtakes the delta-encoding runtime. These observations clearly show that delta encoding is not universally appropriate. For models with relatively large lookahead values, conservative performance is near optimal, and an optimistic approach may not be fruitful.

4.2.2 OLSR

These results were collected with the following flags: `--report-interval=0.10 --clock-rate=1600000000.0 --lookahead=0.0001 --gvt-interval=16 --batch=16 --buddy_size=26 --end=128`. Delta encoding is compared against both conservative and optimistic mode using copy state saving (CSS). In this particular experiment, the size of the model consumed too

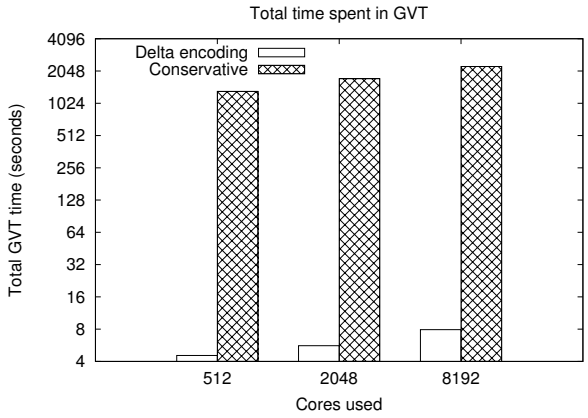

Figure 3: Total time spent calculating GVT vs. the number of cores used. GVT calculations negatively impact conservative simulations with low lookahead values; increasing the core count only exacerbates this trend.

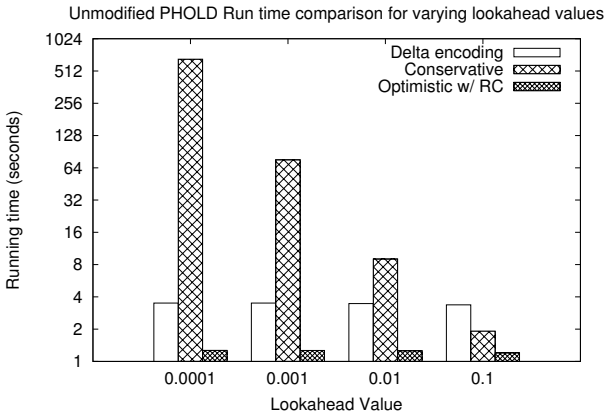

Figure 4: Running time for all configurations vs. lookahead. The lookahead value can drastically affect the runtimes of the BPHOLD runs. This data was collected by using 512 cores and options identical to the previous results (aside from changing the lookahead).

much memory when running in optimistic mode and the core usage had to be reduced to one quarter of the available cores per node, or 4 out of 16. It is worth noting that the delta encoding implementation was able to utilize all 16 cores and likely would have continued to scale further to take advantage of either 2 or 4 hardware threads per core.

Whereas the BPHOLD results demonstrated the weak scaling properties of delta encoding, the OLSR results show its utility within the domain of strong scaling. 262,144 OLSR LPs were distributed over 128, 512, and 2,048 cores. The resulting performance is shown in Figure 5. At 128 cores, conservative runtimes outperform delta encoding: $\approx$ 430 seconds were spent on LZ4 operations (ROSS reports the maximum across all cores). However, at 512 and 2,048 cores the runtime penalty for LZ4 compression is disbursed across a greater number of cores resulting in 110 and 28 seconds, respectively. These reduced penalties allow delta encoding to outperform conservative runtimes within an optimistic framework.

Figure 6 shows the memory consumption of a single core in the simulation for various componenets of the simulator while running under delta encoding or optimistic mode with CSS. Note that identical amounts of memory were consumed for LP state in both cases. Additionally, no buddy system memory exists in the CSS case. However, as the CSS LP copy is stored within the ROSS events themselves, each event will grow substantially (in this case, from 2,248 to 24,920 bytes).

5 RELATED WORK

Jefferson's Time Warp (Jefferson 1985) is a complex system requiring careful tuning of several parameters to achieve speedup. Early Time Warp implementations required checkpointing (state saving), which, when performed blindly, is a potentially expensive operation. Lin et al. (1993) proposed an approach for determining how frequently checkpointing is required. Bauer and Sporrer (1993) chose a contrasting approach, namely minimizing the memory required for checkpointing data by storing overwritten values in a predetermined location. Steinman (1993) uses a similar approach and additionally uses C++ operator overloading for simple values and a "rollback queue" for operations that are not easily reversed. The implementations of the latter two papers are typically referred to as *incremental state saving*. Rönngren et al. (1996) developed a largely transparent approach requiring modifying the state declarations requiring restoration functionality. West and Panesar (1996) automatically modified the binary itself to instrument

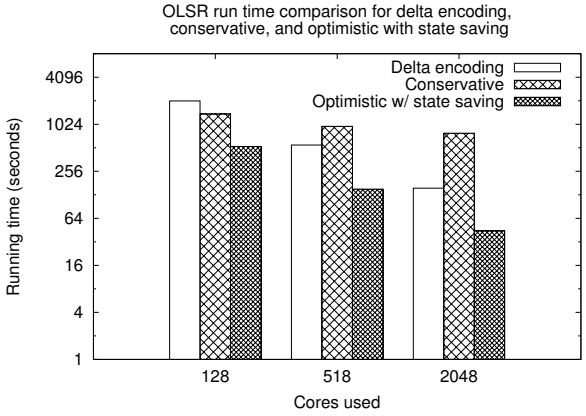

Figure 5: Strong scaling running times for delta encoding, conservative, and optimistic (with CSS) modes vs. the number of cores used.

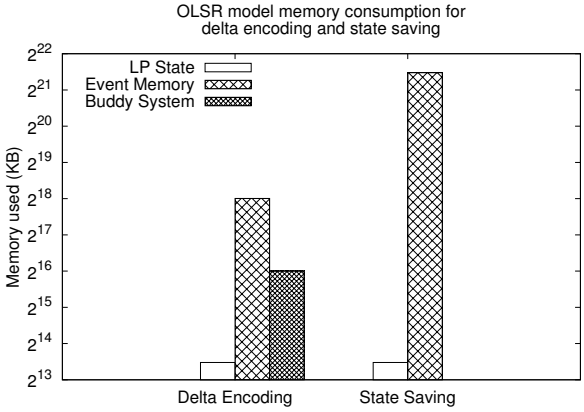

Figure 6: OLSR memory consumption of a single core for delta encoding vs. optimistic with CSS plotted on a log scale. The sum of memory consumed for delta encoding is $\approx \frac{1}{8}$ the memory consumed for CSS.

and state save values before they are overwritten. Pellegrini and Quaglia (2014) developed an OS-level memory tracking scheme that allows the developer to write code in a sequential manner regardless of the PDES environment.

Carothers, Perumalla, and Fujimoto (1999) and Perumalla (1999) proposed reverse computation as well as automatic reversal of event handlers. Building off those earlier results, Vulov et al. (2011) developed Backstroke, a framework capable of parsing C++ and emitting reverse event handlers. LaPre, Gonsiorowski, and Carothers (2014) developed LORAIN, a tool capable of automatically generating reverse event handlers in a language-agnostic fashion by targeting the LLVM (Lattner and Adve 2004) compiler framework.

Hunt and McIlroy (1976) describe the `diff` program that computes the difference between two files using an algorithm to solve the longest common subsequence. Tichy (1982) builds on `diff` to construct the Revision Control System (RCS), which is useful for tracking changes in text files. Descendants of RCS such as the Concurrent Versions System (CVS) (Grune 1986) and Subversion (Collins-Sussman, Fitzpatrick, and Pilato 2004) continue to be used today. Hunt, Vo, and Tichy (1998) compared the various algorithms for delta encoding and benchmarked them in various ways.

Huffman et al. (1952) developed Huffman coding, a method of finding shortest prefix codes based on character frequencies. Ziv and Lempel (1977) created LZ77, a lossless compression scheme that examining smaller fixed-size "windows" of data and exploiting similarity between data. Both LZO and LZ4 are closely related to LZ77, although subtle implementation differences have allowed them to surpass LZ77 in various ways. LZO (Oberhumer 2015) placed an emphasis on speed, while LZ4 (Collet 2011, Collet 2014b) aimed for simplicity in its format and code base. A much lower emphasis was placed on compression ratio, allowing for quick yet sub-optimal compression. DEFLATE (Deutsch 1996) combines LZ77 followed by a Huffman coding pass, ultimately yielding a smaller size than either approach alone but at greater computational cost.

6 CONCLUSIONS AND FUTURE WORK

We have presented an initial framework for state saving using delta encoding and compression. It performs well in situations dealing with massive states where reverse computation may not be practical. Furthermore, this approach is both transparent and universal. Model developers are freed from the worry of having to

change their forward event handlers in any way; and, for many models, the template reverse event handler provided should suffice.

Delta encoding refines incremental state saving by evaluating changes to the LP state as a whole and saving only the changed information. A change in any individual element is treated as a change in the whole of the state, negating the need to detect fine-grained changes. While hinted at earlier in the paper, this work provides not only a simple path toward optimistic simulation but also a reasonable fix for systems with limited memory.

We have demonstrated that delta encoding is a viable substitute for both state saving and reverse computation. However, a hybrid solution may yield even better results. Reverse computation coupled with delta encoding for complex and unpredictable state modifications is a topic worthy of investigation. Pairing delta encoding with an automated solution, such as Backstroke (Vulov et al. 2011) or LORAIN (LaPre, Gonsiorowski, and Carothers 2014) removes the need for overly complicated model development.

ACKNOWLEDGMENTS

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

REFERENCES

Barnes, Jr., P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. "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, 327–336. New York, NY, USA: ACM.

Bauer, H., and C. Sporrer. 1993, Mar. "Reducing Rollback Overhead In Time-warp Based Distributed Simulation With Optimized Incremental State Saving". In *Simulation Symposium, 1993. Proceedings., 26th Annual*, 12–20.

Carothers, C. D., D. Bauer, and S. Pearce. 2002. "ROSS: A High-Performance, Low-Memory, Modular Time Warp System". *Journal of Parallel and Distributed Computing* 62 (11): 1648 – 1669.

Carothers, C. D., and K. S. Perumalla. 2010. "On Deciding Between Conservative And Optimistic Approaches On Massively Parallel Platforms". In *Winter Simulation Conference'10*, 678–687.

Carothers, C. D., K. S. Perumalla, and R. M. Fujimoto. 1999, July. "Efficient Optimistic Parallel Simulations Using Reverse Computation". *ACM Trans. Model. Comput. Simul.* 9 (3): 224–253.

Chacon, S. 2009. *Pro Git*. 1st ed. Berkely, CA, USA: Apress.

Chen, D., N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. 2011. "The IBM Blue Gene/Q Interconnection Network And Message Unit". In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 26:1–26:10. New York, NY, USA: ACM.

Clausen, T. H., and P. Jacquet. 2003. "Optimized Link State Routing Protocol (OLSR)". Technical report.

Collet, Yann 2011. "LZ4 Explained". http://fastcompression.blogspot.in/2011/05/lz4-explained.html. Accessed: 2015-03-03.

Collet, Yann 2014a. "LZ4 - Extremely Fast Compression Algorithm". https://github.com/Cyan4973/lz4. Accessed: 2015-03-03.

Collet, Yann 2014b. "LZ4 (latest version)". http://fastcompression.blogspot.com/p/lz4.html. Accessed: 2015-03-03.

Collins-Sussman, B., B. Fitzpatrick, and M. Pilato. 2004. *Version Control With Subversion*. O'Reilly Media, Inc.

Deutsch, P. 1996, May. "DEFLATE Compressed Data Format Specification Version 1.3". RFC 1951 (Informational).

Fujimoto, R. M. 1990, October. "Parallel Discrete Event Simulation". *Commun. ACM* 33 (10): 30–53.

Fujimoto, R. M. 1999. *Parallel And Distributed Simulation Systems*. 1st ed. New York, NY, USA: John Wiley & Sons, Inc.

Gropp, W., E. Lusk, and R. Thakur. 1999. *Using MPI-2: Advanced Features Of The Message-passing Interface*. Cambridge, MA, USA: MIT Press.

Grune, D. 1986. "Concurrent Versions System, A Method For Independent Cooperation". Technical report, IR 113, Vrije Universiteit.

Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture (2nd Ed.): A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Huffman, D. A. et al. 1952. "A Method For The Construction Of Minimum Redundancy Codes". *proc. IRE* 40 (9): 1098–1101.

Hunt, J. J., K.-P. Vo, and W. F. Tichy. 1998, April. "Delta Algorithms: An Empirical Analysis". *ACM Trans. Softw. Eng. Methodol.* 7 (2): 192–214.

Hunt, J. W., and M. D. McIlroy. 1976. "An Algorithm For Differential File Comparison". Technical report, Bell Laboratories.

Jefferson, D. R. 1985. "Virtual Time". *ACM Trans. Program. Lang. Syst.* 7 (3): 404–425.

Kiselkov, Sašo 2013. "LZ4 Compression". http://wiki.illumos.org/display/illumos/LZ4+Compression. Accessed: 2015-03-03.

Knowlton, K. C. 1965, October. "A Fast Storage Allocator". *Commun. ACM* 8 (10): 623–624.

LaPre, J. M., C. D. Carothers, K. D. Renard, and D. R. Shires. 2012, October. "Ultra Large-Scale Wireless Network Models Using Massively Parallel Discrete-Event Simulation". *Autumn Simulation Multi-Conference (AutumnSim'12)*:Digital distribution only (no page numbers).

LaPre, J. M., E. J. Gonsiorowski, and C. D. Carothers. 2014. "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, 3–14. New York, NY, USA: ACM.

Larabel, Michael 2013. "Support For Compressing The Linux Kernel With LZ4". http://www.phoronix.com/scan.php?page=news_item&px=MTI4NjM.

Lattner, C., and V. Adve. 2004. "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, 75–86. Washington, DC, USA: IEEE Computer Society.

L'Ecuyer, P., and T. H. Andres. 1997. "A Random Number Generator Based On The Combination Of Four LCGs". *Mathematics and Computers in Simulation* 44 (1): 99 – 107.

Lin, Y.-B., B. R. Preiss, W. M. Loucks, and E. D. Lazowska. 1993. "Selecting The Checkpoint Interval In Time Warp Simulation". In *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation*, PADS '93, 3–10. New York, NY, USA: ACM.

Oberhumer, MFXJ 2015. "oberhumer.com: LZO Real-Time Data Compression Library". http://www.oberhumer.com/opensource/lzo/. Accessed: 2015-03-03.

Pellegrini, A., and F. Quaglia. 2014. "Transparent Multi-Core Speculative Parallelization Of DES Models With Event And Cross-state Dependencies". In *Proceedings of the 2Nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, 105–116. New York, NY, USA: ACM.

Perumalla, K. S. 1999. *Techniques For Efficient Parallel Simulation And Their Application To Large-Scale Telecommunication Network Models*. Ph. D. thesis, Georgia Institute of Technology.

Rönngren, R., M. Liljenstam, R. Ayani, and J. Montagnat. 1996. "Transparent Incremental State Saving In Time Warp Parallel Discrete Event Simulation". In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, PADS '96, 70–77. Washington, DC, USA: IEEE Computer Society.

Steinman, J. S. 1993. "Incremental State Saving In SPEEDES Using C++". In *Proceedings of the 25th Conference on Winter Simulation*, WSC '93, 687–696. New York, NY, USA: ACM.

Tichy, W. F. 1982. "Design, Implementation, And Evaluation Of A Revision Control System". In *Proceedings of the 6th International Conference on Software Engineering*, ICSE '82, 58–67. Los Alamitos, CA, USA: IEEE Computer Society Press.

Torvalds, Linus 2014. "torvalds/linux". https://github.com/torvalds/linux/commit/7a02d089695a1217992434f03a78aa32bad85b5c. Accessed: 2015-03-03.

Vulov, G., C. Hou, R. Vuduc, R. Fujimoto, D. Quinlan, and D. Jefferson. 2011. "The Backstroke Framework For Source Level Reverse Computation Applied To Parallel Discrete Event Simulation". In *Proceedings of the Winter Simulation Conference*, WSC '11, 2965–2979: Winter Simulation Conference.

West, D., and K. Panesar. 1996. "Automatic Incremental State Saving". In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, PADS '96, 78–85. Washington, DC, USA: IEEE Computer Society.

Ziv, J., and A. Lempel. 1977. "A Universal Algorithm For Sequential Data Compression". *IEEE TRANSACTIONS ON INFORMATION THEORY* 23 (3): 337–343.

AUTHOR BIOGRAPHIES

JUSTIN M LAPRE is a Ph.D. candidate in computer science at Rensselaer Polytechnic Institute. His research interests include parallel discrete event simulation, compilers, and static analysis. Justin enjoys writing code and reading about better ways to build systems using modern practices in software engineering. His email address is laprej@cs.rpi.edu.

ELSA J GONSIOROWSKI is a computer scientist at Rensselaer Polytechnic Institutes Center for Computational Innovations, as well as a part-time Ph.D. candidate. Her research interests include massively parallel discrete-event simulations of gate-level circuit models. Her email address is gonsie@cs.rpi.edu.

CHRISTOPHER D CAROTHERS is a faculty member in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. Prior to joining RPI, he was a research scientist at the Georgia Institute of Technology. He is an NSF CAREER Award winner as well as Best Paper award winner at the PADS workshop for 1999, 2003, and 2009. His email address is chrisc@cs.rpi.edu.

JOHN JENKINS is a postdoctoral appointee at Argonne National Laboratory. He received his Ph.D. in computer science from North Carolina State University in 2013. His research interests include parallel I/O, parallel/distributed storage and analysis systems, and parallel discrete event simulation. His email address is jenkins@mcs.anl.gov.

PHILIP CARNS is a principal software development specialist in the Mathematics and Computer Science division of Argonne National Laboratory and a fellow of the Northwestern-Argonne Institute of Science and Engineering. He received his Ph.D. from Clemson University in 2005. He has served as the lead developer on notable projects such as PVFS, Darshan, and BMI. His current research interests include development, measurement, and simulation of large-scale storage systems. His email address is carns@mcs.anl.gov.

ROBERT ROSS is a senior computer scientist at Argonne National Laboratory and a senior fellow at the Computation Institute of the University of Chicago and the Northwestern-Argonne Institute for Science and Engineering at Northwestern University. He received his Ph.D. in computer engineering from Clemson University in 2000. He then joined the Mathematics and Computer Science Division at Argonne National Laboratory. He was the lead architect of the PVFS parallel file system. He is a member of the MPICH2 development team awarded the R&D 100 award in 2005, and was a recipient of the 2004 Presidential Early Career Award for Scientists and Engineers. His email address is rross@mcs.anl.gov.