Granular Cloning: Intra-Object Parallelism in Ensemble Studies

Philip Pecher[1], John Crittenden[2], Zhongming Lu[3], Richard Fujimoto[1]
[1]School of Computational Science and Engineering
[2]School of Civil and Environmental Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
[3]Beijing Normal University, Beijing Shi, 100875, China
philip161@gmail.com, john.crittenden@ce.gatech.edu, zhongming.lu@bnu.edu.cn, fujimoto@cc.gatech.edu

ABSTRACT

Many runs of a computer simulation are needed to model uncertainty and evaluate alternate design choices. Such an ensemble of runs often contains many commonalities among the different individual runs. Simulation cloning is a technique that capitalizes on this fact to reduce the amount of computation required by the ensemble. Granular cloning is proposed that allows the sharing of state and computations at the scale of simulation objects as small as individual variables, offering savings in computation and memory, increased parallelism and improved tractability of sample path patterns across multiple runs. The ensemble produces results that are identical to separately executed runs. Whenever simulation objects interact, granular cloning will resolve their association to subsets of runs though binary operations on tags. Algorithms and computational techniques required to efficiently implement granular cloning are presented. Results from an experimental study using a cellular automata-based transportation simulation model and a coupled transportation and land use model are presented providing evidence the approach can yield significant speed ups relative to brute force replicated runs.

KEYWORDS

Cloning; multiprocessors; parallel algorithms; parallel simulation; algorithms; performance; scale; speedup; acceleration; experimentation; scenario; design; shared computation; incremental simulation

ACM Reference Format:

Philip Pecher, John Crittenden, Zhongming Lu, and Richard Fujimoto. 2018. Granular Cloning: Intra-Object Parallelism in Ensemble Studies. In Proceedings of SIGSIM Principles of Advanced Discrete Simulation (SIGSIMPADS' 18). ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3200921.3200927

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy
 ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200927

1. INTRODUCTION

Computer simulation shows its strength by its flexibility to model a vast set of domains. However, its power is limited by the accuracy of the underlying model and the computational performance of the simulations. This paper focuses on alleviating the latter constraint. State changes are either driven by event-scheduling or by time-stepping though simulated time. The method described in this paper, termed granular cloning, can be utilized in both time-traversal modes, in both serial and parallel computer architectures, and – for parallel systems – in both optimistic and conservative synchronization protocols. Granular cloning exploits redundant computations across multiple runs of a model at the scale of individual simulation objects. Since it only affects common computations the results are equivalent to explicitly enumerating these runs. The runs are differentiated by random inputs and/or variations in the model scenarios.

In stochastic simulation, randomness is injected in input variables to model uncertainty of the system under investigation (SUI). The performance of granular cloning is improved with the use of common random numbers because the pseudorandom number steams can be shared by all the logical runs. If one wishes to introduce variability, granular cloning can be further improved by altering individual elements of the stream or augmenting the stream with more elements.

In many simulation studies, the objective is to find a feasible policy that leads to desirable output statistics. The number of feasible configurations often grows exponentially with the number of controllable discrete factors one is considering. Consider a network with ten nodes where one wishes to place one of three different router models at each node. The number of possible configurations is 3^{10}. While there are methods for efficiently searching the often nonconvex configuration space, many runs will still be required. For each scenario, it is also desirable to complete many runs to obtain a point estimate of the output statistic with low bias and spread.

When variation is introduced into a set of runs of a certain model, the collective is referred to as an ensemble. An ensemble study may have a set of objectives, including characterizing (e.g., for validation) or optimizing certain model parameters [15]. Whatever the way variation is introduced, random input or scenario variation, granular cloning superimposes all runs of an ensemble into one run and only explicitly stores the actual and most granular state differences across these runs, rather than larger supersets of state. Ensembles may be batched together

based on a partitioning that is favorable for granular cloning. These conditions are discussed later. For example, suppose we model a particle p in a neighborhood of other particles and some gaps of "empty" space. We wish to investigate the evolution of the neighborhood for two different polarity values of p (two different *versions*). If most of the simulation state remains the same for both versions, it would be inefficient to explicitly store two sets of the entire state. We only allocate memory for neighboring particles if they are affected differently for a version compared to the other version. The number of objects with state differences for a given version in this example tends to grow monotonically with simulated time. Explicit enumeration of all runs would be wasteful because of the common states and behaviors across the runs. The technique is generally applicable for Monte-Carlo simulation, although one could construct adversarial models where the runs are completely different at model initialization, in which case the approach collapses into explicit enumeration of runs. At the same time, the technique incurs a small overhead whenever different simulation objects interact with each other. However, there are heuristics that can be used to reduce the overhead cost, which depend on the object interactions of the underlying application.

The next section describes related work. In Section 3, we define the notation and terminology from the simulation cloning literature [1] along with cloning concepts. Section 4 describes how cloning can be extended to state objects within LP's. Section 5 brings attention to the quantitative conditions under which computational performance can be harvested without loss in output accuracy; this is expressed in a simple mathematical inequality that is intuitive from a geometric view. Experimental results of use cases are given in Section 6, varying key parameters that affect the computational performance.

2. RELATED WORK

The concept of cloning memory segments has been used extensively since Von Neumann introduced it for fault tolerance in 1956 [3]. Since then, it has influenced modern relational database systems, distributed memory management, and other areas of computer systems. In addition to describing this historical context in detail, [1] contains a survey of domain-specific simulation models where cloning has been used previously. Common themes in the relevant literature include computation sharing (reuse of common state or events) and incremental simulation.

Reducing the execution time and memory requirements of replicated simulation instances without affecting the accuracy of output statistics has been investigated in both domain specific and general (yet, typically, simulation-engine-specific) settings. Domain-specific use cases generally allow more state to be logically shared by grouping together similar replications at suitable times during the execution, and/or otherwise exploit the underlying application's specific characteristics for space/time efficiency. Relevant examples include the following.

In [4], Pecher, Hunter, and Fujimoto present a lazy evaluation and speculative execution scheme for physical cloning of vehicle objects in microscopic traffic simulation. Vehicle objects that are independent of output statistics propagate tags containing the replication numbers they "touch." Their numbers are compared to vehicle objects that have a dependency on output statistics. If there is a version match between the vehicle types, a physical clone must be launched via a rollback to a relevant saved state for explicit computation. Lentz et al. [5] apply incremental cloning to test different signature paths in a digital logic simulation for potential faults. The "offspring" (clone) of a "parent" is limited to four logical output values. Vakili [6] uses a so-called *standard clock* scheme (an implementation of *single-clock multiple systems*, or *SCMS*) approach to execute – on a SIMD computer system - the same events simultaneously across multiple synchronized replications, which are parametrized differently (e.g., different service disciplines in a queue). The injected positive correlation makes this approach especially useful in simulation-based optimization (ranking & selection), albeit for a restricted class of applications.

Unlike these prior efforts, the granular cloning approach described here does not rely on domain-specific properties. Although domain-specific methods can exploit knowledge of the given application, general techniques are by definition more flexible. As discussed next, more general techniques have been developed. It is possible that combinations of these techniques complement each other if they exploit different aspects of the underlying application.

An example of a more general technique is the parallel cloning scheme introduced by Hybinette and Fujimoto in [1] that applies to the distributed LP event-scheduling paradigm. It is summarized in Section 3 because our granular cloning scheme will be described with similar terminology and notation. Just as with parallel cloning, granular cloning also makes heavy use of set-manipulation algorithms. Furthermore, the underlying mechanism for both algorithms consists of sharing state logically, while incrementally allocating physical memory. However, granular cloning offers a much finer grained mechanism that focuses on cloning individual objects rather than entire logical processes, and is agnostic to the underlying simulation executive paradigm. Chen, Turner, Cai et al. have extended the distributed cloning algorithms for the High Level Architecture (HLA) in [16] along with HLA's data distribution management (DDM) in [17].

Ferenci et al. [7] presents an incremental scheme termed *updatable simulation*, where one first logs the events and sample path of a baseline run and hopes to reuse the state modifications from it for subsequent runs. Subsequent runs then determine what horizon of logged events can be reused at a given timestep where an event is to be processed. In the ideal case, a sequence of $r >> 1$ events can be composed together, rather than separately, to act on the current state. If the runs are sufficiently different, the event horizon r will be zero at each event-processing timestep and nothing can be reused. The *staged simulation* technique discussed in Walsh and Sirer [8] caches previous event invocations, decomposes them, reuses previously computed and/or similar results, and reorders restructured events for their efficient scheduling. Stoffers et al. [12] extends automatic memoization to impure functions where side effects are permitted (subject to a few constraints). Granular cloning differs from these approaches in that it does not rely on reusing previously computed results.

Granular cloning is perhaps most closely related to recent work by Li, Cai, and Turner [18] which focuses on tree-based cloning algorithms for agent-based models. A cloning tree contains nodes that map to simulation instances (versions) that have a

different parameter than their parents. Each such instance is associated with two data structures: *AgentPool* and *Context*. The former contains agents whose characteristics are unique to that instance, while the latter contains references to shared agents in ancestor nodes. During the execution of an instance, the relevant agents from a child's ancestors are copied. A child instance performs clone condition checking as follows. If a *Context* agent (shared with an ancestor node) senses a parameter variation or an agent in the respective *AgentPool*, a clone is generated and moved to the *AgentPool*. Execution of simulation instances within the clone tree occur level-by-level in a breadth-first manner. In contrast to this simulation instance-hierarchical scheme, granular cloning does not use a tree hierarchy; rather, it associates tags with object realizations and clone condition checking occurs by comparing the tags of interacting objects. Granular cloning accesses relevant objects in arrays rather than by tree traversal. A bit masking scheme is used to efficiently implement version management. Superficially, granular cloning applies to granular subsets of state, while [18] applies to agent objects, though potentially one could adapt the algorithm from [18] to encompass the same scale. Furthermore, at any given timestep, the parallelism in [18] is limited by the available nodes at any level of the clone tree (batch-by-batch); in granular cloning, all objects may be executed concurrently at any timestep.

Granular cloning utilizes a kind of data dependency detection for interacting objects (or state variables thereof). Granular cloning compares properties of versions associated with these objects. Detecting various properties of data dependencies with the end goal of accelerating simulation execution has been studied in other works as well. Quaglia and Baldoni [19] investigate data dependencies associated with a simulation object *over several events*. The scheme can be used to accelerate certain optimistic simulations by increasing event-level parallelism. The definition of "intra-object parallelism" in [19] differs from the usage in this paper and in [10]. In [19], the parallelism refers to the state transitions (i.e., events) that an object is exposed to, while here (and in [10]) it refers to the variations in the object's state. Marziale et al. [20] measure the *amount of* data dependencies across processes in order to create suitable groups of these processes ("granular LPs") dynamically in a Time Warp-based environment. The grouping that is derived from dependencies in [20] applies to subsets of state (logical processes), while they apply to subsets of runs in this paper. Furthermore, the grouping sequence for granular cloning is unique.

3. CLONING PARALLEL SIMULATIONS

Hybinette and Fujimoto [1] introduce the concept of simulation cloning in the context of distributed and interactive simulations where the analyst can intervene and dynamically evaluate alternative futures (*clones*, identified by a *version* number that is incremented chronologically: i = 1, 2, 3, …) at *decision points* and find a policy that offers advantages over others. A decision point must specify how a given clone differs in its state from a given reference version and at which timestep it occurs. However, different courses of action may also be defined a priori to evaluate different scenarios and inject uncertainty into certain input variables; they can also be triggered by certain conditions at runtime. Clones are generated by specifying them at decision points or through certain interactions between logical processes (LPs). The underlying system architecture used in [1] is based on distributed LPs and the executive operates with the event-scheduling paradigm. The physical system is mapped into a model in the computer and the state of this model is partitioned into LPs (identified with j = A, B, C, …). During runtime, LPs exchange time-stamped messages to change their state and schedule new events; an LP only changes its state after processing an event (scheduled by itself or with a message received from another LP). Consider a simulation that evaluates two scenarios where, at some timestep $t_b > 0$, some partial component of the state differs among them.

Simulation cloning makes use of *computation sharing* because:

- the sample path up to the decision point is only computed once. The set of *virtual logical processes* (*VLP*) being mapped to by each clone are shared by a shared memory region in the computer system, namely the corresponding *physical logical processes (PLP)*, and
- only the VLP's that differ in their state after the decision point are explicitly duplicated (as a *PLP*) in memory and processed separately and in parallel; the VLP's that are identical continue to be shared within a corresponding PLP. As the given clone starts affecting state in other VLP's, further PLP's will be incrementally spawned. An important point with respect to one key contribution of this paper is that the algorithm in [1] clones an entire VLP, even if there is just a small difference in a state object within the VLP.

VLP's exchange virtual messages among each other, but each virtual message maps to a single physical message and each VLP maps to a single PLP. Whenever several virtual instances are mapped to a particular physical instance (that is, if they have the same state), computations are essentially shared. A given VLP is identified by (i) the *LP identifier* and (ii) the *version* of the clone it refers to. A given version is associated with a set of VLP's for every LP in the model.

It is useful to define some notation prior to delving into the actual cloning algorithm. As mentioned before, the entire state of a model is partitioned into LP's (identified by *j= A, B, C,…*) and each version (identified by *i = 1, 2, 3, …*) maps to a set of VLP's. Subsequently, we will refer to a particular VLP as *V(i,j)*. If, for fixed *j*, several VLP's map to the same PLP (only if they share the same state), the corresponding PLP is denoted *P(i,j)* and *i* is the minimum version number of the VLP's pointing to *P(i, j)*.

To implement simulation cloning, one must alter the simulation executive to incorporate (i) *message cloning* and (ii) *process cloning*. Message cloning simply refers to a simple mechanism whereby a message sent from a PLP (sharing several versions in a set called *VSendSet*) and received by a PLP (sharing several versions in a set called *VRcvSet*) must be forwarded to other PLP's mapped to from versions contained in *VSendSet* that are not in *VRcvSet*. Process cloning occurs when a PLP is mapped to by versions not contained in a receiving message. The respective message may cause a modification in the state of all versions that the PLP represents, which would clearly be invalid for the states that are not in the message's *VSendSet*. The full process cloning algorithm from [1] is (S^C denotes the complement of set S with respect to the universe of versions):

(1) Get VSendSet from message
(2) Get VRcvSet from local state
(3) VPsUnaffected = $VSendSet^C \cap VRcvSet$
(4) VPsMoveToClone = VPsUnaffected
(5) VPsRequested = VSendSet ∩ VRcvSet
(6) If (!(PhysicalReceiver ∩ VPsRequested)) then
VPsMoveToClone = VPsRequested
(7) If (VPsMoveToClone ≠ {}) then
Cloned PLP = min(VPsMoveToClone)

4. GRANULAR CLONING

4.1 Intuition

A straightforward extension to simulation cloning approach from [1] is to replace the process cloning step with one that only clones state differences. An associative map is maintained for each PLP that maps a given version *i* to a record that contains information regarding *where* and *how* a PLP differs from shared state. In Figure 1, an LP modeling an airport initially shares the state for versions 1 and 2 and receives an airliner arrival event that only occurs in version 1. To maintain correctness for version 2, simulation cloning now copies the entire LP state prior to handling of the arrival. Granular cloning instead only duplicates the state that the arrival event handler would modify. These duplicated state variables are now valid for version 2 (airliner absent), while the variables they were copied from may now be safely overridden for the version 1 arrival event.

Relative to the parallel cloning scheme from Section 3 granular cloning should perform well if a given LP owns

1. a sufficiently large partition of the entire state space,
2. event handlers where state changes tend to be fragmented over the state variables, rather than over its entire state,
3. event handlers where unmodified state variables are insensitive to the changes to the state variables that are modified.

One could, of course, simply take the parallel cloning scheme and redefine each LP to encompass only a single state object. In doing so, however, one potentially increases the communication between the LP's which can significantly degrade performance.

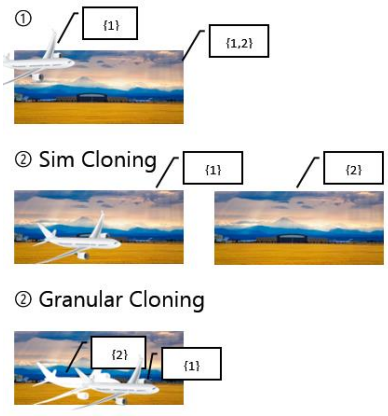


Figure 1: Rather than duplicate the entire LP, granular cloning only duplicates the state that changes. ©Free-Photos, Creative Commons 0

If - instead of the event-scheduling based approach - the simulation executive uses a timestepped agent-based model (ABM), which may be part of a federation, one can associate each agent, and any state object in general, with a set of versions (a *version set*). This not only comes with all the benefits discussed before (i.e., savings in space and time because of shared computations), but also through increasing the parallelism in a shared memory machine (under certain circumstances), especially if the number of replications is a soft constraint. For a given version *v* (out of *V* total runs), an ABM typically executes, over several iterations (depending on the input model), for each timestep < *T*, the behavior of *e* < *E* agents. Within a fixed version, timestep, and entity, a kernel (or, transition function) is typically executed that reads from various state objects of the previous timestep, and writes to the given agent's state objects for the current timestep.

Figure 2 shows a so-called *time-space diagram* (see [11]) for a concrete example.

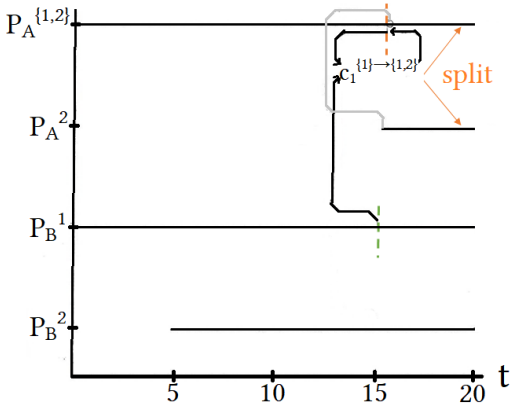


Figure 2: Time-space diagram for granular cloning split.

The horizontal axis measures simulated time and the vertical axis contains discrete state space objects (in memory). The lines within the quadrants represent sample paths, here denoting simply the presence of a given physical state object. The state objects use a similar notation as that used for PLP's in Section 3. Here, the superscript refers to the versions mapping to the state object and the subscript denotes the identifier for the state object. At *t*=15, the *kernel* of *A* for its sole version set {1, 2} is executed, depicted as c_1. A kernel is a sequence of instructions computed for a version set of a single agent (called the *kernel object*) and is not permitted to write to any other object. If any write operation in the kernel depends on a read operation of other objects, the intersection of the read sources' version sets is compared to the version set of the kernel object. If the kernel object contains members not present in the intersection of the read sources, the kernel object splits. If one were to not split the kernel object, the state of the kernel object would be overwritten for the versions that are contained in it, but not in the read sources, potentially leading to errors.

4.2 Formulation

As summarized before, a timestepped ABM - from a high-level view - typically has the following operational scheme:

```
for s in scenarios
   initialize_model(s)
   for t in timesteps
      for a in agents
         a.kernel /*also called transition
                  or step */
```

Algorithm 1: Agent-based model - operational outline (traditional)

When granular cloning is applied to these kind of models, it obviates the need to explicitly enumerate the scenarios (since all of them are superimposed). However, at runtime, granular cloning needs to perform some bookkeeping, in order to trace dependencies between objects that are now associated with versions (*versioned objects*). As the simulation progresses, versioned objects that originally shared state across all versions start to diverge from the shared state to a version-specific state. From a high-level view, the operational scheme for granular cloning applied to a timestepped ABM looks like:

```
initialize_model(scenarios)
for t in timesteps
   for a in agents
      for vo in a.versionedobjects
         a.GC_interaction /*kernel with
                     granular cloning */
```

Algorithm 2: Agent-based model - operational outline (granular cloning)

In some sense, the outermost loop in Algorithm 1 has been logically replaced with the innermost loop in Algorithm 2. However, we hope to iterate over fewer than scenarios.number (the total number or runs, or versions) in the innermost loop of Algorithm 2 by exploiting common state across the versionedobjects. The degree of reduction (or state-sharing) needs to be sufficiently high to offset the additional overhead that occurs in a.GC_interaction (in Algorithm 2) relative to a.kernel (in Algorithm 1). Most of what follows in this subsection is concerned with the details of the last line (a.GC_interaction) in Algorithm 2. In fact, a.GC_interaction is presented as Algorithm 3 and represents the high-level granular cloning kernel management by ensuring that versioned objects that diverge from any shared state have the kernel applied to them as well. In order to (i) report potential diverging splits in versioned objects and to (ii) dispatch the actual kernel function on versioned objects, Algorithm 3 calls Algorithm 4 (GC_transition()). (i) is computed in Algorithm 5 (handle_conflicts()). The diagram in Figure 3 summarizes this granular cloning mechanism.

Granular cloning for agent-based models is differentiated from conventional execution in that it needs to preserve the integrity of the superimposed sample path. Before executing any transition function, granular cloning first ensures that no version-specific state of the kernel object is being wrongly overwritten for a dependency that does not match that version (this check is done in a function called handle_conflicts()). If there is a version conflict, the current kernel object needs to be split. The split object is handled after the current kernel object.

At a high level, the kernel management algorithm for a set of kernel objects – here, versionedobjects, is as follows:

```
VersionObj[] split_versionedobjects
for VersionObj versioned_obj in versionedobjects
   GC_transition(versioned_obj,
     split_versionedobjects)
while split_versionedobjects.has_elements
   GC_transition(split_versionedobjects.pop,
     split_versionedobjects)
```

Algorithm 3: High-level granular cloning kernel management

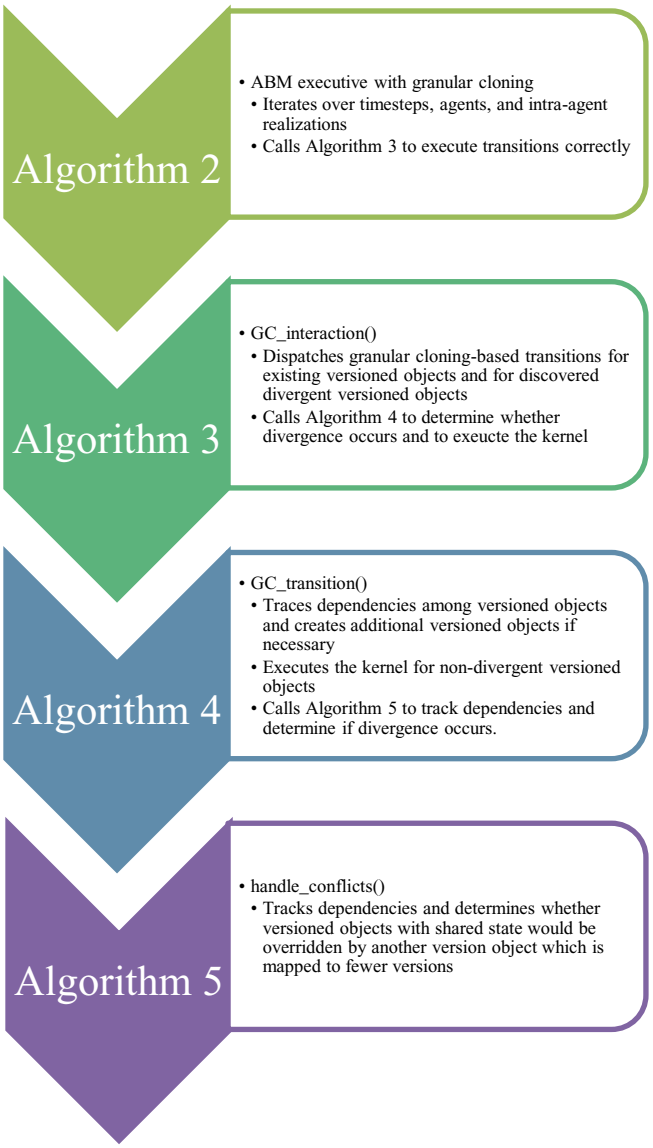


Figure 3: The granular cloning mechanism

split_versionedobjects maintains potential split kernel objects that fragment from the currently considered kernel object and still need to be handled by the kernel. For example, suppose a given kernel object, versioned_obj, is associated with versions 1 and 3, and the granular cloning transition manager GC_transition() determines that versioned_obj needs to be split from the version set {1,3} into {1} and {3}. In this case, GC_transition() will only override the kernel object corresponding to the intersection of the versions sets of all read sources. Suppose, in our example, the intersection is {1}. Now, the kernel object for {3} is still unhandled and still needs to be processed. In the while-block of the granular cloning kernel manager, we simply execute the kernel for all these split objects until they are all handled.

A given object's version set is encoded in unsigned integers where bits are set for corresponding version indices; granular cloning makes heavy use of bit manipulation. For example, the following binary value encodes a mapping to versions 1 and 3 (1-based indexing) among 32 total versions:

0000 0000 0000 0000 0000 0000 0000 0101

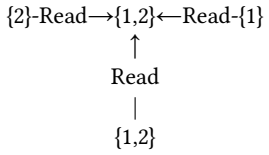
The granular cloning-wrapped kernel function GC_transition() performs the following steps:

```
GC_transition()(VersionObj vo, VersionObj[]
 splits)
   VersionObj[] potential_splits =
    vo.handle_conflicts()
   Versions unaffected_versions
   if potential_splits.has_splits
      splits.emplace_back(potential_splits)
      unaffected_versions =
       potential_splits.get_versions()
   vo.skernel(unaffected_versions)
```

Algorithm 4: Low-level granular cloning kernel wrapping

As previously mentioned, splits is an output argument that will store unhandled versioned objects in case we need to split the current kernel object vo. Whether splitting will occur depends on the comparison of the version set of vo and the versions of all objects we read in the kernel (i.e., all dependencies of the kernel object vo). The handle_conflicts() method determines if the version set intersection of all objects that are being read in the kernel (all dependencies) are a strict subset of the kernel object vo's version set. If they represent a strict subset, the unhandled split version objects need to be stored in splits. The version set of the unhandled split version objects are maintained in the unaffected_versions variable. Finally, we execute the superimposed kernel, skernel(), for the kernel object vo, but only for the versions that are not included in the unhandled splits – that is, the logical complement of unaffected_versions. For the complement of unaffected_versions we can be sure that we can read the corresponding version sets from all the read sources because handle_conflicts() already filtered out the maximal version set we can manage (by the definition of set intersection).

In handle_conflicts(), a tie breaker rule will determine what version number serves as the foundation of integrity checking. An arbitrary selector, like the filtering for the minimum version, will denoted with SEL. Once that version is determined, one must filter neighboring objects for corresponding sets. Consider the following example:



In this case, the min-selector uses version 1. The intersection of {1,2} and {1} is {1} and so we split {1,2} into {1} and {2}. We update {1} with the kernel and leave the state mapped to version set {2} unaffected; {2} will be handled in the while-loop of Algorithm 3 later. The state of the kernel object (after the split, only associated with {1}) does not only depend on the state of the versioned object to the right, but also on one on the bottom. The while-loop of Algorithm 3 ensures that any fragment of the current kernel object will be handled by the read sources that are omitted (those fragments will potentially be split further).

Without an automated runtime that can inspect all object identifiers of a kernel invocation, it is necessary for the user to specify what objects the kernel accesses. In many ABM's the read sources consist of some neighborhood – in a given coordinate system - around the kernel object. For example, in Conway's Game of Life [14], a given kernel object (a cell that can assume a state of 1 or 0), which is located in a 2D grid, reads from its eight neighbors. In the general handle_conflicts() method (listed in Algorithm 5), it is assumed that one can access the entire state of the previous timestep (OLD) as well as the read sources relative to the kernel object (NBR) by some defined addressing scheme (i.e., coordinate system for most ABM's). There are more general variants for handle_conflicts()and some of them may be more efficient for a particular application. One can also engineer optimized versions for specific application where certain characteristics are exploited. For example, one fairly common characteristic among cellular automata is that all the versions are present (with mutual exclusivity) for each element of NBR.

```
VersionObj handle_conflicts()
  Version ref_version = SEL(this.versions)
  VersionObj[] read_sources =
   get_sources(this, OLD, NBR)
  VersionObj[] ref_vo = filter(read_sources,
   ref_version)
  Version working =
   intrsct(getversions(ref_vo),this.versions)
  For Version v in
   working.get_onehots_except(ref_version)
    working = intrsct(working,
     getversions(filter(read_sources, v)))
  if working != this.versions
    VersionObj res(this.state, working)
    res.set_splits(true)
    return res
  else
    VersionObj res(WILDCARD_S, WILDCARD_V)
    res.set_splits(false)
    return res
```

Algorithm 5: One possible general variant of handle_conflicts()

Algorithm 5 first arbitrarily picks a reference version from the kernel object's version set with the selector function SEL. It then proceeds to collect all the kernel dependencies and filters those objects with the reference version that was just selected. A working version set is then used to determine whether there is version-consistency between the dependencies' (ref_vo) and the kernel object's respective version sets. The intersection ensures that we enter skernel() with the maximal version set that is still common among all dependencies (and the kernel object itself) and preservers correctness. The for-loop ensures that transitive dependencies of non-reference versions are captured. To crystalize this in a small example:

$$\{2,4\}\text{-Read}\rightarrow\{1,2,3\}\leftarrow\text{Read-}\{1,2\}$$

Using a min-selector, we would have a reference version of 1. working would capture $\{1,2\}\cap\{1,2,3\}=\{1,2\}$ before entering the for-loop. If we did not have the for-loop intersections, the skernel() would later only read the dependency on the right (associated with version set {1,2}). However, the kernel object is also associated with version 2. Version 2 depends on the left-versioned object that is associated with {2,4}. Therefore, to preserve correctness, one must include transitively linked versions as well. The for-loop grabs all non-reference versions from working and then tries to find these transitive dependencies. The conditional branch at the end simply reports back to the caller whether a split needs to occur. If there are elements in the kernel object that are not present in the relevant read sources, a split occurs. Otherwise, the version sets are synchronized which is indicated by a flag in the returned object. The skernel() method behaves exactly like the traditional kernel with the exception that we have to filter for the version set of the kernel object among the read sources, namely for vo.get_versions() \ unaffected_versions (where \ is the set minus operator) and separate out the split version set from the kernel object. Read sources that are associated with version sets that represent a strict superset of the kernel object version set are treated exactly like those that are associated with vo.get_versions() \ unaffected_versions. The additional versions in the superset simply refer to state being shared with other versions. Since we do not write to the read objects, the presence of the additional versions is irrelevant.

For event-scheduled simulation executives, granular cloning is compatible with both optimistic and conservative synchronization protocols. For the latter, each LP will proceed to execute local events up to the synchronization point without violating local causality. The only difference compared to the traditional approach is that messages and objects are associated with versions and, whenever events are processed, the comparison of "version set"-tags occurs (via the granular cloning algorithms described). For situations in which the lookahead value is state-dependent, local causality is guaranteed if the lookahead values are derived *across all versions of a given LP* (there may exist further optimizations whereby correctness can be ensured if this is relaxed). For optimistic synchronization protocols, the same granular cloning algorithms are used during normal execution. However, during rollbacks one needs to ensure that anti-messages annihilate messages with the same version set and that state saving captures the version sets associated with simulation objects (message acknowledgements are also version set specific). A straggler message may trigger a "global" rollback (i.e., for all versions) even if it only affects a few versions. Although forward progress is guaranteed with a global rollback (as in traditional Time Warp), it may cause rollback thrashing. To counteract this, it may be possible to modify the granular cloning rollback mechanism to preemptively query a rollback's version dependencies across LPs and only affect an isolated subset of the versions, while the execution of unaffected versions can proceed in parallel (since the rollback does not apply to them). These extensions are beyond the scope of this paper.

4.3 Optimizations and Heuristics

As previously mentioned, it is possible to optimize the general approach for handle_conflicts() for a specific kernel. For example, sometimes a particular state value of the kernel object determines that few other objects need to be read than if the state of the current kernel object is assumed to be unknown. However, it should be emphasized that this is not necessary for granular cloning - unlike some other methods for computation sharing. Algorithm 6 shows a simplified version of an optimized handle_conflicts(). It exploits the application-specific fact that the kernel reads from at most one object and that we can "look ahead" which object will be read based on the current state of the kernel object.

```
VersionedObj handle_conflicts()
  if this.state == STATE0
    VersionedCell[] neighbor =
     OLD[get_index()+NBR[0]]
    for VersionedObj lvn in neighbor
      /*assume neighbor is sorted by min
      Version*/
      if versions.has_a_match(
       lvn.get_versions()) and
       !versions.is_contained_in(
       lvn.versions)
        return VersionedCell(
          this.state,
          this.versions.diff(
           this.versions.intersection(
            lvn.versions
            )
          )
        )
  else if this.state == STATE1 /* remaining cases */
  return VersionedCell() //no conflict
```

Algorithm 6: An application-specific variant of the handle_conflicts() method (applied on the kernel object) that exploits the property that there is at most one object dependency by peeking at the kernel object's state and then filtering the read sources.

Algorithm 6 first decides where in the old state of the model environment, OLD, it needs to look for the relevant object. As mentioned, the correct location depends on the state of the kernel object. In order to avoid displaying redundant code, only one of the conditional branches is shown completely (the one for STATE0). The algorithm then iterates over the versioned instances of the object and tries to discern whether the relevant instance is in (version-)conflict with the kernel object's version set. If so, it returns the split object. If not, the end of the function will return an empty object. The caller will then be able to discern whether the returned object is split or not.

This is just a small sample of possible optimizations. There are many properties that can be exploited, ranging from (i) dense and globally-sensitive version sets to (ii) sparse, localized, and clustered version sets. In applications that fall under the former category (i), it may be the case that explicitly enumerating sample paths corresponding to single versions after the branching point outperforms the general handle_conflicts() scheme; in this case, the computations are only shared up to the branching point. For applications in the latter category (ii), where one wishes to simulate many versions, it may prove fruitful to abandon the unsigned integer representation and instead adopt a compressed representation of version sets. Indices could be used to map to particular version sets in aggregate data structures.

5. EXPECTED PERFORMANCE FOR AGENT-BASED GRANULAR CLONING

Intuitively, one expects better performance (more shared computation) if the state variation is introduced late and if there is a slow spread of version-differentiated sample path relative to the entire state space. Better performance is also expected if the overhead from granular cloning is much smaller than the transition function.

Consider an agent based simulation (*ABS*) that simulates E agents over T *(>0)* timesteps (t is a particular timestep) over R runs (r is a particular run). At a given run, timestep, and for a given agent, a kernel (or, transition function) with work W_{kernel} is executed. Although the W_{kernel} for granular cloning is slightly more complex than the one for traditional execution, the same factor will be used for both: The only additional granular cloning overhead is a potential split of the kernel object (which is a constant-time operation). Both the traditional and the granular cloning kernels access interacting objects in constant time. The key for the former is the object id, while for the latter it consists of both the object id and relevant version set. After p % of these timesteps, decision points are injected whereby the state $S(t)$ differs from the baseline sample path (which is identified with version id, $r = 1$). At each timestep after $\lceil p*T \rceil$, each agent reads the state of E_{read} $(\leq E)$ agents. $1 \leq \varrho(t) \leq R$ is a function that – for granular cloning - returns the average number of version sets at timestep t across all agents E. $\kappa_{conflicts}$ (> 1) scales W_{kernel} by the additional overhead from handle_conflicts().

The speedup of granular cloning vs. traditional replicated execution is approximated by:

$$\frac{E \times W_{kernel} \times T \times R}{E \times \kappa_{conflicts} \times W_{kernel} \times \sum_{t=0}^{T} \varrho(t)} = \frac{T \times R}{\kappa_{conflicts} \times \sum_{t=0}^{T} \varrho(t)}$$

Without a mechanism for re-merging split objects, $\varrho(t)$ is monotonically increasing. Furthermore, $\varrho(t)$ cannot exceed R for any t because a version number is the most atomic element of an object and, by assumption, no more than R runs are simulated. Therefore, $T \times R \geq \sum_{t=0}^{T} \varrho(t)$. Figure 4 shows three example growth rates of $\varrho(t)$ with overlaid trendlines. The fastest-rising curve $\varrho'(t)$ would not be expected to yield a significant speedup and, depending upon the implementation of handle_conflicts(), the scaling with $\kappa_{conflicts}$ may even dominate the payoff expression above the curve $T \times R - \sum_{t=0}^{T} \varrho(t)$. The growth rates primarily depend on the percentage of the total state space that is being read in an object's kernel at a point in time. A constant number (without any noticeable bias for detecting certain types of state variables) would likely yield the linear $\varrho''(t)$ divergence pattern in Figure 4.

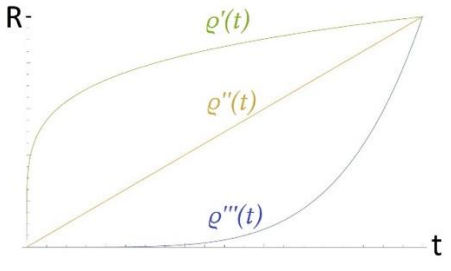


Figure 4: Lower growth rates yield better performance for granular cloning.

6. EMPIRICAL EVALUATION

The performance of granular cloning is evaluated with two models, both of which are described in more detail later. In brief, they consist of:

(i) a cellular automaton- (CA-) based traffic simulation on a 1D grid, which is a boundary case of the Nagel-Schreckenberg traffic model [13].

(ii) a distributed setting that simulates a basic integrated transportation and land use model (ITLUM), where one process executes (i) and another process executes a land use model from Lu et al. [2]. The execution of both processes occurs in a synchronous fashion per timestep.

The ITLUM benchmark is a useful test case because in many usage scenarios replicated runs will often only modify one model, e.g., the land use or the transportation model, but not both. As such, this is an interesting test case to explore the benefits of cloning in general, and granular cloning in particular. Further, ITLUM represents a class of simulations of practical interest to communities concerned with urban planning and the sustainable growth of cities.

For both models, the number of runs is limited to 32; the version tags assigned to objects are 32-bit unsigned integers. In settings with more runs, one could assign wider tags for even greater speedups. The machine that generated these results uses an Intel Core i7 6700 ™ (4 physical cores, 8 logical) and 32GB DDR4 memory. All programs were compiled from C++ and the distributed implementation uses MSMPI. Unit tests were successfully completed to ensure that states of the granular cloning match the explicit execution of the same runs.

The next three subsections discuss the conceptual models, and the two subsections thereafter report the performance results. Opportunities for attractive applications as well as limitations in adversarial problems due to tradeoffs are highlighted.

6.1 Conceptual Model: CA Traffic Simulation

A 1D CA models one road in a neighborhood. Each cell of the array is either set or cleared, the former representing a cell by a vehicle and the latter the absence of a vehicle. At discrete timesteps every vehicle will move forward (towards the right) by one cell if the cell ahead is clear. Vehicles "wrap around" when they reach the rightmost cell. The initial array state is populated randomly. This ruleset is referred to as Wolfram Rule 184 which is also used to model several other systems. However, there is a slight modification relative to this behavior that is described by Wolfram Rule 184. A vehicle will sample random cells in front of it and not move forward if the perceived traffic intensity is deemed too high, namely with probability $p = 1-[0.9+0.1*ESTIMATED\_TRAFFIC\_INTENSITY]$, where *ESTIMATED_TRAFFIC_INTENSITY* is determined by the ratio *[number of sampled cells that are occupied]/[number of sampled cells]*. In the experimental design, the number of sampled cells is varied with powers of two to illuminate its impact on the speedup. The length of the 1D array and the number of timesteps are also varied in the performance section.

6.2 Conceptual Model: Land Use Model

The land use model used is a translation from the Netlogo model of Lu et al. [2] into C++. A detailed description of the model is beyond the scope of this paper, but can be found in the reference guide of [2]. At a high level, the ABM maintains a 24x24 grid that represents 9 square miles of greenfield, which is developed with single-family dwellings over a 30-year period. It also contains on the order of 100 input, output and state variables. The agents in the model include homebuyers and developers, which interact with the local government (taxation and infrastructure improvements) and the grid. In each of the 30 time steps:

1. 1000 homebuyers bid on up to 10 properties,
2. property transactions are settled,
3. new properties are constructed, and
4. local taxes are collected to improve the infrastructure.

The model was developed to compare two stormwater development policies against each other to find out how many apartment homebuyers they would incentivize.

6.3 Conceptual Model: ITLUM

The two models from the previous subsections were coupled into an ITLUM. During the 4th phase of the land use model of [2], namely the tax collection and infrastructure improvement phase, the transportation cost savings of a neighborhood are derived from the degradation of the road infrastructure and depreciation of public transit. However, these variables were fixed in the original model. Using an actual transportation simulation to determine these variables dynamically may improve the credibility and validity of the results. The traffic intensity of the neighborhood roads is understood to be proportional to the population density of the neighborhood. These communication ideas inspire the federation sequencing pipeline for the ITLUM shown in Figure 5.

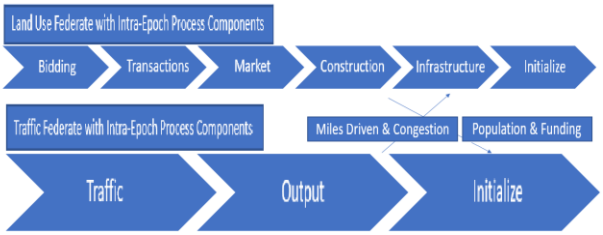


Figure 5: ITLUM Federation Sequencing Pipeline

As the ratio between (i) the execution time of one epoch of the land use node and (ii) the execution time of one epoch of the transportation node approaches 0, the speedup of the ABS dominates the overall speedup.

6.4 Evaluation: CA Traffic Simulation on a Single Process

Figure 6 shows the speedup of the granular cloning approach executing 32 versions in one single replication, compared to the traditional approach of executing a single version for each of 32 replications. Each of the 32 replications alters the state at a uniformly drawn timestep at a uniformly drawn cell. The speedups were determined using measured CPU times for a single process executing the transport model only.

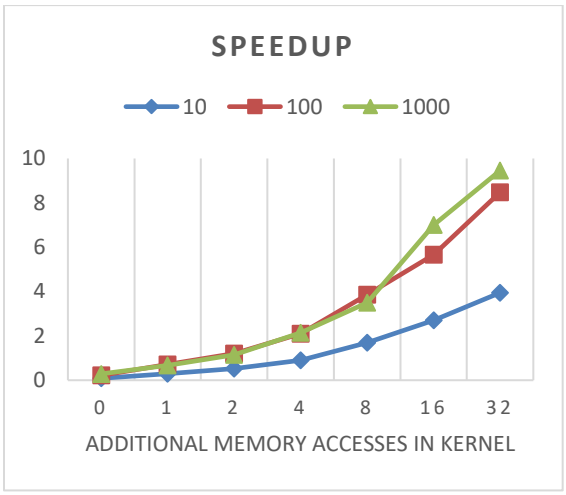


Figure 6: Speedup of the transport model of three different cell-to-timestep ratios

The three lines refer to three different scenarios differentiated by the space-to-time ratio (the cell-count divided by the time-step count). The speedup is less impressive if there are only a few timesteps relative to the numbers of cells because a given version has less time available to propagate its specific behavior into the global state. The speedup increases with the computational load in the transition function as the relative overhead of the granular cloning algorithm diminishes. The computational load is injected by enhancing the traffic intensity sampling resolution of the given driver (see the description of the transportation conceptual model above). It should be noted that merging versions that share the same state together at certain intervals in the granular cloning mode did not improve the performance significantly.

Figure 7 shows, for the 10 space-to-time ratio, the speedup if one realizes the differences among the replications late. As mentioned above, the differences among the 32 replications are uniformly drawn in the 1D grid space and across all timesteps. However, if one restricts the realized timesteps to only be drawn in the last 20% or 50% of timesteps, the version-specific local state perturbations have fewer opportunities to propagate through the global state space. This directly translates into computational savings in memory and execution time. As can be seen in the figure, the jump from the 50% scenario to the 20% scenario is less dramatic. Early branches pose a significant computational burden on granular cloning. In an adversarial scenario, the speedup is less than 1 because the physical sample path is equivalent to that in the traditional approach and one cannot reap the benefits of shared computations up to the decision point. The shared computations after the decision point are out-shadowed by the granular cloning overhead. With regard to the performance model from Section 5, $\varrho(t)$ for this particular model would exhibit a linear growth trend based on the kernel which attempts to detect collisions.

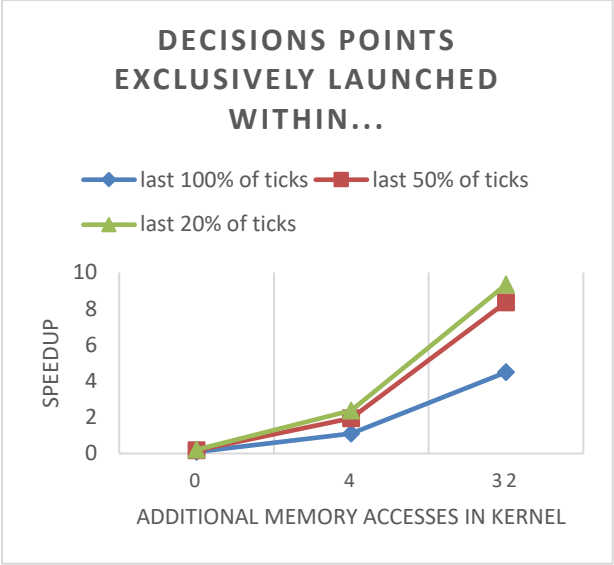


Figure 7: Speedup of the transport model if uniform scenario alterations are drawn in restricted time intervals

6.5 Evaluation: ITLUM on Two Processes

The following tables show the speedup results of the combined ITLUM model described above for 30 simulated years each. The speedup quantities are not measured relative to sequential execution of the two processes separately; rather, they are relative to the parallel execution of the two processes where only the transport model uses the traditional approach of explicitly enumerating all the replications: $S = T_{parallel,traditional} / T_{parallel,granular}$. A column headed by the expression $S_{TI=IC}$ denotes speedup figures that were obtained from models where IC instructions in the transition function for each object's update in the next timestep were computed. The data exchanged between the two simulators in a synchronized/blocking fashion. This is also why the speedup quantities in the following tables are derived from wall-clock intervals. If they were based on CPU time, they could be misleading as a result of the platform context-switching from a blocking process. However, the drawback is that platform-specific factors may pollute the times. Nevertheless, a second set of replications confirmed that the numbers were close to those in the first set.

Table 1: Speedup for cellcount/timesteps = 100 and start of decision points commences at timestep=0

Cell count	$S_{TI=4}$	$S_{TI=32}$
1,000	0.93	1.03
10,000	1.04	3.39

Table 2: Speedup for cellcount/timesteps = 100 and start of decision points commences halfway through

Cell count	$S_{TI=4}$	$S_{TI=32}$
1,000	1.01	1.07
10,000	1.13	3.29

Table 3: Speedup for cellcount/timesteps = 10 and start of decision points commences at timestep=0

Cell count	$S_{TI=4}$	$S_{TI=32}$
1,000	0.99	1.35
10,000	0.92	3.07

Table 4: Speedup for cellcount/timesteps = 10 and start of decision points commences halfway through

Cell count	$S_{TI=4}$	$S_{TI=32}$
1,000	1.00	1.66
10,000	1.11	4.11*

In the entry marked with * was also evaluated with a transition function count of 64 and 128, which resulted in speedups of 4.89 and 5.96, respectively. Since the only speedups that can be reaped in take place in the transport simulator, it determines the overall speedup provided it takes longer per epoch than the land use simulator. Because of reasons discussed before, the instruction count of the driver's transition function has a significant impact on the overall speedup. In general, especially high performance is expected when there is a large number of runs, the variation is introduced late, the behavior (transition function or event handler) is complex in relation to the overhead, and/or when individual objects do not have the opportunity to quickly affect the entire state space in a unique fashion.

6.6 Evaluation: 2x4 XOR-HOLD (Event-Scheduling on Eight Processes)

In addition to these agent-based and timestepped models, granular cloning was also compared against traditional simulation cloning in a distributed event-scheduling setting (with 8 processes). The benchmark used here is *2x4 XOR-HOLD* (2 LPs, 4 versions each) using a conservative barrier synchronization based approach. In addition to simply processing and scheduling events (*HOLD*), LPs in XOR-HOLD also own state variables (represented as an integer array) and modify them in their (sole) event handler. More specifically, they access their array at an index randomly sampled by the sender and apply logical exclusive OR of the sender's corresponding value to their own value at that index. Versions are realized as random variations in individual state variables (integers) at the start of the simulation. When the array sizes are set to 100,000 and the number of total events processed per LP set to 300,000, the obtained average speedup is 1.68 (standard error = 0.08). The speedup was again based on wallclock time until the simulation terminates using each method.

7. CONCLUSION AND FUTURE WORK

The granular cloning framework offers performance improvements without loss in accuracy for a broad range of simulation applications where one wishes to execute several runs that share similar sample paths. Instead of being restricted to the scale of extended partitions of the state space, the explicit state representation occurs at the exact minimal scale where state differs. Our experimental results show order-of-magnitude speedup in some cases and illustrate that performance is increased when:

- the number of replications is large,
- the transition functions (in timestepped models) or the event handlers (in event-scheduling models) are relatively complex, and
- the state among runs is similar, especially at the beginning of the run, so that the framework may reuse more shared computations, rather than duplicate redundant work. In many applications, the replication-specific variation tends to explode throughout the state space as simulated time increases.

There are interesting further research questions related to granular cloning. For example, how should one batch together runs of various scenarios to apply simulation-based optimization methods (scenarios vs. random input)? Another research topic lies in approximate computing: one could reduce the overhead of granular cloning interactions selectively, which relaxes correctness; in this fashion, bias is traded off for variance. It is also an open question of whether there are more efficient algorithms to keep track of the state-version associations, especially for common applications. Another open research area is how different simulation acceleration techniques contribute to speedup when they are used in groups across a broad range of benchmarks.

We plan to release a general-purpose library that allows users to encapsulate their simulation objects into granular cloning objects as well as the interactions among these objects into granular cloning interactions. As in the simulation cloning library of [1], the granular cloning runtime will manage the bookkeeping of the varying state of a given object to the replication number (version) internally.

ACKNOWLEDGMENTS

Support for this research was provided by NSF Grants 3676778 and 1745580.

REFERENCES

[1] Maria Hybinette and Richard Fujimoto. 2001. Cloning parallel simulations. Commun. ACM Transactions on Modeling and Computer Simulation (TOMACS), 11(4), 378-407.

[2] Z Lu, D. Noonan, J. Crittenden, H. Jeong, and D. Wang. 2013. Use of impact fees to incentivize low-impact development and promote compact growth. Environmental science & technology, 47(19), 10744-10752.

[3] John von Neumann. 1956. Probabilistic logics and the synthesis of reliable organism from unreliable components. Princeton University Press.

[4] Philip Pecher, Michael Hunter, and Richard Fujimoto. 2015. Efficient Execution of Replicated Transportation Simulations with Uncertain Vehicle Trajectories. Procedia Computer Science 51 (2015): 2638-2647.

[5] Karen Panetta Lentz, Elias S. Manolakos, Edward Czeck, and Jamie Heller. 1997. Multiple experiment environments for testing. Journal of Electronic Testing 11, no. 3 (1997): 247-262.

[6] Pirooz Vakili. 1992. Massively parallel and distributed simulation of a class of discrete event systems: A different perspective. ACM Transactions on Modeling and Computer Simulation (TOMACS), 2(3), 214-238.

[7] Steve Ferenci, Richard Fujimoto, Mostafa Ammar, Kalyan Perumalla, and George Riley. Updateable simulation of communication networks. 2002. Proceedings of the sixteenth workshop on Parallel and distributed simulation pp. 107-114. IEEE Computer Society.

[8] Kevin Walsh and Emin Gün Sirer. Simulation of large scale networks I: staged simulation for improving scale and performance of wireless network simulations. 2003. Proceedings of the 35th conference on Winter simulation: driving innovation. pp. 667-675.

[9] SLX FAQs. http://www.wolverinesoftware.com/SLXFAQs.htm. Accessed: 2018-01-04

[10] James Henriksen. An introduction to SLX [simulation software]. 1995. Simulation Conference Proceedings, 1995. Winter. IEEE.

[11] Richard M. Fujimoto. 1989. The virtual time machine. Proceedings of the first annual ACM symposium on Parallel algorithms and architectures. ACM.

[12] Mirko Stoffers, Daniel Schemmel, Oscar Soria Dustmann, and Klaus Wehrle. 2016. Automated Memoization for Parameter Studies Implemented in Impure Languages. Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation. pp. 221-232. ACM.

[13] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. 1992. Journal de physique I 2, no. 12 2221-2229.

[14] John Conway. The game of life. 1970. Scientific American 223, no. 4

[15] Richard Fujimoto, Conrad Bock, Wei Chen, Ernest Page, and J. Panchal. Research Challenges in Modeling and Simulation for Engineering Complex Systems. 2016. NSF Report

[16] Dan Chen, Stephen J. Turner, Wentong Cai, Boon Ping Gan, and Malcolm Yoke Hean Low. Algorithms for HLA-based distributed simulation cloning. 2005. ACM Transactions on Modeling and Computer Simulation (TOMACS). 15, no. 4. 316-345.

[17] Dan Chen, Stephen J. Turner, Wentong Cai, Georgios K. Theodoropoulos, Muzhou Xiong, and Michael Lees. Synchronization in federation community networks. 2010. Journal of parallel and distributed Computing 70, no. 2. 144-159.

[18] Xiaosong Li, Wentong Cai, and Stephen J. Turner. Cloning Agent-Based Simulation. 2017. ACM Transactions on Modeling and Computer Simulation (TOMACS). 27, no. 2. 15.

[19] Francesco Quaglia and Roberto Baldoni. Exploiting intra-object dependencies in parallel simulation. 1999. Information Processing Letters 70 (3), 119-125.

[20] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. Granular time warp objects. 2016. Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced and Discrete Simulation