CHEX: Multiversion Replay with Ordered Checkpoints

Naga Nithin Manne∗
Argonne National Lab.
Lemont, IL, USA
nithinmanne@gmail.com

Shilvi Satpati
DePaul University
Chicago, IL, USA
ssatpati@depaul.edu

Tanu Malik
DePaul University
Chicago, IL, USA
tanu.malik@depaul.edu

Amitabha Bagchi
IIT, Delhi
Delhi, India
bagchi@cse.iitd.ac.in

Ashish Gehani
SRI
Menlo Park, CA, USA
ashish.gehani@sri.com

Amitabh Chaudhary
The University of Chicago
Chicago, IL, USA
amitabh@uchicago.edu

ABSTRACT
In scientific computing and data science disciplines, it is often necessary to share application workflows and repeat results. Current tools containerize application workflows, and share the resulting container for repeating results. These tools, due to containerization, do improve sharing of results. However, they do not improve the efficiency of replay. In this paper, we present the multiversion replay problem, which arises when multiple versions of an application are containerized, and each version must be replayed to repeat results. To avoid executing each version separately, we develop CHEX, which checkpoints program state and determines when it is permissible to reuse program state across versions. It does so using system call-based execution lineage. Our capability to identify common computations across versions enables us to consider optimizing replay using an in-memory cache, based on a checkpoint-restore-switch system. We show the multiversion replay problem is NP-hard, and propose efficient heuristics for it. CHEX reduces overall replay time by sharing common computations but avoids storing a large number of checkpoints. We demonstrate that CHEX maintains lightweight package sharing, and improves the total time of multiversion replay by 50% on average.

PVLDB Reference Format:
doi:10.14778/3514061.3514075

PVLDB Artifact Availability:
The source code, data, and/or other artifacts are available at https://github.com/depaul-dice/CHEX.

1 INTRODUCTION
Suppose that Alice is researching different image classification pipelines. She has a large labeled set of images and progressively tries different combinations of preprocessing steps and neural network architectures. For example, she may replace an entire step with one that is more sophisticated but slower, or vice-versa. As she makes changes, she keeps a copy of the previous versions in separate Jupyter notebooks. We call these her different program versions: they are similar to different experiments in scientific computing. Once done, Alice would like to share her different program versions with Bob so that he can independently repeat and regenerate the results and verify Alice’s work. We say Bob faces the multiversion replay problem: executing all the versions given to him by Alice as efficiently as possible where (i) many versions repeat some of the same preprocessing steps, and (ii) without reusing any of Alice’s own computation. In this paper, we address this problem.

Collaborative scenarios such as the one above arise routinely in scientific computing and data science, where sharing, repeating, and verifying results is common. Several tools have been recently proposed for sharing and reproducing such scenarios [1, 8, 22, 41–44, 57]. These tools audit the execution of a program, and create a container-like package consisting of all files referenced by the program during its execution. This package can then be used to repeat results in different environments. These tools have much to offer; they do not, however, exploit the efficiency possible by solving the multiversion replay problem. As the reproduction of results becomes increasingly time consuming [53], addressing such problems is critical.

One of the above tools is the Sciunit system [1, 55], developed by some of the co-authors. It allows multiple versions of a program to be included in the same package and shared for repetition. We noted that having two or more versions in the same package sets up a natural opportunity for reusing computations that are often common across versions—i.e., a number of versions may perform the same computations for quite some time before they branch out as the researcher tries out different options. But, in order to accurately identify computations that can be reused across versions, we need to be able to determine the point to which the execution of two versions can be treated as equivalent and from which point the execution branches. We develop a methodology to identify common computations in program code fragments or cells of the versions. This methodology depends on lineage audited during program execution [38, 42, 43]. For repetition, at Bob’s end, we share computational state across versions in the form of checkpoints. Let us now see with an example how sharing computational state across versions via checkpoints creates an opportunity to optimize computational time when repeating multiple versions.

Suppose that Alice has shared with Bob a package with three versions. Assume Alice has developed her code using a notebook,
and her first version is divided into two cells that take time 1 minute and 10 minutes, respectively (Figure 1 left). Her second version has the same first two cells but she adds a third cell. Her last version, which processes the dataset, has the same first cell, but diverges after that, and cells 2 and 3 now takes 11 and 2 minutes, respectively. Now, during repetition, if Bob checkpoints cell b while computing v1 and then restores that checkpoint for v2, he can complete v2 in just 1 unit of time (saving 11 units in v2). This checkpoint is, however, not useful for v3 since cell b has been changed to d in this version. Observe, that although a is common across all three versions, checkpointing a, instead of b, is not optimal. In the example on the right, on the other hand, checkpointing a is the better option since the bulk of the computation takes place in a. The example above leads to the question: Why doesn’t Bob just checkpoint both a and b? Storage is cheap after all! Indiscriminate caching, however, is not a practical solution: In machine learning examples, e.g., checkpointing all cells across versions (as our experiments indicate) can lead to a memory requirement in the range of 50-550GB, for even moderately-sized multiversion programs. Thus, we consider a limited in-memory space as this avoids additional I/O costs from checkpointing. Limited space leads to an optimization problem: As we will show, given limited space and multiple versions with different cost and size estimates of cells, deciding checkpoint locations for efficient replay of all versions is an NP-hard problem. We present efficient heuristics to solve this problem.

The CHEX system. We present CHEX, a system for efficient multiversion replay that uses recorded lineage shared in container-like auditing systems to (i) determine when the program state is identical across versions, and (ii) decides which common computations to save in an in-memory, limited-size cache, and where to continue recomputing. Effectively, CHEX computes an efficient plan for Bob to use his cache to repeat Alice’s multiversion program with minimum computation cost. Subsequently CHEX repeats the computation according to this plan.

To execute the multiversion program, we first need a plan for sharing and reusing computational state across versions. A possible approach would be to reuse program elements such as the output of functions, expressions, or jobs. Such reuse approaches were examined in [16–18]. However, these methods make assumptions about the programs—they are limited to programs with no side-effects and apply to specific (functional or interpreted) programming languages. Such assumptions are too restrictive in a sharing scenario.

Our approach is program-agnostic and we, instead, use checkpoints. A checkpoint saves the computational state at a specific program location so that the same program can be restored from the location at a later time. To share computations, we extend checkpoint-restore to checkpoint-restore-switch, in which a system checkpoints a common computational state and and restores it later to resume a different version of the program.

The challenge of checkpoint-restore-switch, however, is determining locations at which to checkpoint, since ideally programs may be checkpointed after each instruction. Even if we decide at a fixed number of program locations, before reusing a checkpoint we must verify that two versions share the same computational state at a given program location. In this paper we solve this dual challenge by showing that when a program is divided into cells, computational state can be shared across versions by using fine-grained execution lineage. Dividing a program into cells is used in read-evaluate-print (REPL) programming environments, which are increasingly popular [26]. However, CHEX does not necessitate that a user employ REPL style; it transparently divides a program into REPL-style cells. This paper contributes the following:

- Maintains lightweight package sharing. CHEX does not require users like Alice to share checkpoints as part of the shared package. Instead, CHEX audits the execution of each version to record execution details. We note that in reproducibility settings, it is not desirable to allow Alice to share her checkpoints since that defeats the purpose of reproducibility.
- Merging versions based on lineage. CHEX compares fine-grained lineage to check if the program state is common across cell versions. It combines versions into an execution tree.
- Deciding checkpoint location. Given that the multiversion replay problem under space constraints is computationally intractable, we rely on depth-first-search (DFS) traversals of the execution tree to help us identify a subset of possible checkpointing decisions for the execution units of the program. We call the members of this subset DFS-based replay sequences. We propose two heuristic algorithms for deciding which cell state to checkpoint such that the multiple versions can be replayed in a minimum amount of time.

Experiments on real and synthetic datasets. We experimented with real machine learning and scientific computing notebooks as well as synthetic datasets, showing that CHEX improves the total time of multiversion replay by 50%, or correspondingly replays twice the number of versions in a given amount of time. We show that the overheads of creating execution trees is significantly lower than the gain from replay efficiency.

Working prototype system: We have developed a prototype CHEX system, which given an execution tree performs multiversion replay. CHEX currently uses standard auditing methods, developed by us [1, 55], to build execution trees and determine cell reuse. For multiversion replay, we extended these methods to work with interactive Jupyter notebooks, as well as, transform regular programs to REPL-style computation via code-style paragraphs.

2 BACKGROUND

We briefly describe the REPL environment under which CHEX operates. CHEX is not limited to the REPL environment but this is easy to illustrate visually so we adopt this for ease of exposition. We discuss generalization to other environments in Section 9.
We denote a REPL program by an ordered list of cells, variables and objects used by the program at that point — intuitively:

- lineage
- code hash
- computation time
- gram state, $p_s$

Editors executions and provides the following details about each program just before the first cell is executed, includes the value of the program state just before cell $i$ is executed. The state of the program at the end of a cell’s execution depends on its (i) initial environment, (ii) code that is run, and (iii) external input data. The environment is determined by the execution state at the start of the cell. Thus, (i) and (ii) are captured via $g_{i-1}$ and $h_i$. Further, every external input data file $f$ is accessed via a system call event. For each such event, we record a hash of its contents of $f$ in $E_i$.

Figure 3 shows the audited information for the two programs, $L_1$ and $L_2$. The ordered set of system events for the third cells of the two programs are shown in the shaded box below.

![Figure 2: An illustration of REPL programs. The left program $L_1$ trains a machine learning model (resnet18) on a training dataset and evaluates its accuracy on a test dataset. The right program $L_2$ is the same, except that it adds a preprocessing step to the training dataset. $L_1$ has 5 cells, $L_2$ has 6 cells.](image)

REPL or Read-Evaluate-Print-Loop is a programming environment. A popular example is the Jupyter notebook. As shown in Figure 2, it contains code partitioned into cells. Developers typically use a separate cell for each "step" of the program: preprocessing the dataset, training the model, etc. This allows them to interactively test each step before writing the next. One restriction is that control flow constructs, such as if-blocks, loops, cannot be split across cells. We denote a REPL program by an ordered list of, e.g., the left program in Figure 2 is denoted $L_1 = \{x_1, x_2, \ldots, x_5\}$, and the right program as $L_2 = \{y_1, y_2, \ldots, y_6\}$.

In a typical REPL execution, cells are executed in sequence from the first to last. While the Jupyter notebook allows out-of-order cell execution, we do not consider such execution. (We elaborate on this constraint further in Section 9.) The state of the program at the end or beginning of each cell is termed the program state. The program state at any point of execution consists of the values of all variables and objects used by the program at that point — intuitively, it is all the contents of the memory associated with the program.

To see why $g_i$ is defined so, we note that the execution of the program code in cell $i$ (and the code in previous cells) resulted in $p_{si}$. Therefore, $p_{si}$ at the end of a cell’s execution depends on its (i) initial environment, (ii) code that is run, and (iii) external input data. Every such event, we record a hash of its contents of $f$ in $E_i$.

Figure 3 shows the audited information for the two programs, $L_1$ and $L_2$. The ordered set of system events for the third cells of the two programs are shown in the shaded box below.

![Figure 3: Auditing of programs $L_1$ and $L_2$ in terms of $\delta, sz, h, g$. Events in $E_i$ show a forked process, open of an external file along, and read of data content, denoted by its hash value. We determine how to check if $E_i$ across versions is equal in Section 6.](image)

3 CHEX OVERVIEW

As we see in Figure 2, the two programs behave the same till the end of the third cell ($x_3$ in $L_1$, $y_3$ in $L_2$) and then diverge. If the audited lineage, as shown in Figure 3, is established to be the same, then the program state at the end of $x_3$ can be used before $y_3$, i.e., we can skip executing cells $y_1$ to $y_3$. CHEX uses recorded lineage to determine when the program state is identical across versions, and decides which common computations to save. We now present a high-level block diagram of CHEX in Figure 4.

CHEX has two modes: audit and replay. It is used in audit mode to audit details of executions on Alice’s side. Details of multiple executions, i.e., the $\delta, sz, h$ and $g$ of each cell across versions are represented in the form of a data structure called the Execution Tree. We discuss the execution tree and how it is created in detail in Section 6. CHEX creates a package of all Alice’s versions and their data, binary, and code dependencies, along with the execution tree. This package can now be shared with Bob.
We now describe the multiversion replay problem. Figure 6 summar- 
ed via system calls in detail in Section 6. Intuitively, establishing 
cell equality using execution lineages. We will discuss how this is 
of cells that are equal across versions. The tree branches at a cell 
mon nodes till a branch in the tree correspond to the subsequence 
and a branch at $L_1$, resulting in $L_2$ and a branch at $a$, the last common node across $L_1$ and $L_2$. Similarly, in $L_3$ there is a dataset change to $L_2$ at cell $e$, resulting in $L_3$ and a branch at $c$, the last common node across $L_2$ and $L_3$. The com- mon nodes till a branch in the tree correspond to the subsequence 
of cells that are equal across versions. The tree branches at a cell 
ode, subsequent to which cells are not reusable. CHEX computes 
cell equality using execution lineages. We will discuss how this is 
done via system calls in detail in Section 6. Intuitively, establishing 
cell equality makes program states reusable across versions.

Our assumptions. Our basic assumption is that Bob wishes to 
independently verify the results from Alice’s versions but is time 
constrained to repeat all her versions. We do not make any assump-
tions on the types of edits that differentiates one version from the 
next. Thus, Alice can change values of parameters, specifications of 
datasets, models, or learning algorithms. She can also add or delete 
entire cells. We illustrate possible changes via red boxes (Figure 2) 
across program versions. We only assume that edits result in valid 
executions, which do not terminate in an error, and, each version 
is executed in the natural order, top to bottom.

4 THE MULTIVERSION REPLAY PROBLEM

We now describe the multiversion replay problem. Figure 6 sum-
maries the symbols used in Section 2. In the replay mode, CHEX 
inputs an execution tree, $T$, and a fixed cache size, $B$, to solve the 
multiversion replay problem. We define the execution tree as:

**Definition 1. (Execution Tree)** An execution tree $T = (V, E)$ is 
a tree in which each program state is mapped to a node and equal 
program states across the different versions are mapped to the same 
node. Each root to leaf path in $T$ corresponds to a distinct version $L_i$.

**Example.** Figure 5 shows the execution tree created from five 
versions. In this tree, each root to leaf path corresponds to version $L_i$. 
In $L_1$ there is an edit to settings of the program at cell $b$, resulting in 
$L_2$ and a branch at $a$, the last common node across $L_1$ and $L_2$. Similarly, in $L_3$ there is a dataset change to $L_2$ at cell $e$, resulting in $L_3$ and a branch at $c$, the last common node across $L_2$ and $L_3$. The com-
mon nodes till a branch in the tree correspond to the subsequence 
of cells that are equal across versions. The tree branches at a cell 
ode, subsequent to which cells are not reusable. CHEX computes 
cell equality using execution lineages. We will discuss how this is 
done via system calls in detail in Section 6. Intuitively, establishing 
cell equality makes program states reusable across versions.

The multiversion replay problem is an optimization problem that 
arises when multiple versions of a program, each previously exe-
cuted, are replayed as a collection. Once the multiversion program 
is represented as an execution tree it is clear that there is some 
advantage in not replaying the common prefixes of this tree.

**Example.** If we replay the five versions of Figure 5 sequentially we 
incurs total cost of 129. On the other hand, assuming a cache size 
of 25, if we store the checkpoint at common prefixes, restore-switch 
the checkpoint later to avoid computing the common prefix for the 
next version, and evict the previous checkpoint to store a new 
one, the replay cost is reduced to 114 as shown in the first replay 
sequence of Figure 7. In the second figure we see that a different 
set of checkpointing decisions can improve the cost even when the 
cache size remains the same. Finally we see that increasing the 
space to 50 further improves replay costs to 95.

Under these operations, and given an execution tree and a fixed 
amount of space for storing checkpoints, the multiversion replay 
problem aims to determine a replay sequence that has the minimum 
replay costs. We define a general replay sequence as follows:

**Definition 2 (Replay sequence).** Given execution tree $T = 
(V, E)$ and a cache of size $B$, a replay sequence $R$ consists of $m$ steps 
such that step $t$ specifies the operation $O_t$ performed and the resulting 
state of the checkpoint cache $S_t$, i.e.,

$$R = \{(O_t, S_t) : 0 \leq t \leq m\}$$

We will use the term replay order interchangeably with the term 
replay sequence.

At the initial step $S_0$ is empty and the root of the tree is computed. 
At any given step $t$, $O_t$ is one of the following four types. Here, $u_j$ and $u_k$ are nodes in $V$, the vertices of $T$.

- Compute $CT(u_j)$: computes $u_j$;
- Checkpoint $CP(u_j)$: checkpoints $u_j$ into the cache;
- Restore $RS(u_j, u_k)$: restores a previous checkpoint $u_j$ in 
cache and switches to $u_k$ where $u_j = parent(u_k)$; and
- Evict $EV(u_j)$: evicts a previous checkpoint $u_j$ from cache;

...
\[ \delta(R) = \sum_{j \in [0, m)} \delta_{O_j}, \]

in which \( \delta_{O_j} = \delta_j \), when \( O_j = CT(u_j) \), and \( \delta_{O_j} = 0 \) otherwise.

In MVR-P, we assume the cost of checkpoint, restore-switch, and evict operations to be negligible. Thus, the only cost considered is the cost of computing the cells. Determining the minimum cost replay order leads to a natural trade-off between computational cost of cells and fixed-size cache storage occupied by the checkpointed state of the cells. Thus, to optimally utilize a given amount of storage we must determine for each cell whether its next cell be recomputed, or some other cell be recomputed by checkpointing the state of the current cell. We state that determining the replay order is computationally hard.

**Theorem 1.** **MVR-P is NP-hard.**

We show that the decision version of MVR-P is NP-hard. Given an execution tree \( T \), a cache size parameter \( B > 0 \) and a total cost parameter \( \Delta > 0 \), define \( RP(T, B, \Delta) \) to be the decision problem with answer YES if there is a replay sequence of \( T \) with cost at most \( \Delta \) and size of cache at most \( B \), and with answer NO otherwise.

The proof is by reduction from the decision version of bin packing. In outline, the proof works by constructing an execution tree whose depth nodes have checkpoint sizes corresponding to the size of the items to be packed into bins in the bin packing problem. The \( B \) of \( RP(T, B, \Delta) \) is set to the size of the bins. In order to force caching, we keep \( \Delta \) small and add nodes below the depth 1 nodes so that each of the level one nodes has to be cached when first computed. We are able to show that by carefully adding subtrees below the depth 1 nodes we are able to prove a tight relationship between the two problems, i.e., the bin packing decision problem gives a YES answer iff \( RP(T, B, \Delta) \) gives a yes answer.

We omit the proof due to space restrictions, referring the reader to the full version of this paper available at [34].

## 5. HEURISTIC SOLUTIONS

From Theorem 1 we know that it is unlikely we will find polynomial time solution to MVR-P. Accordingly, we present two efficient heuristics for this problem. Both heuristics restrict our exploration of the search space to solutions in which the execution order of the nodes of the execution tree corresponds to a DFS traversal of the tree—a natural, simple order in which to approach the replay of the tree. In order to formalize this notion we present some definitions. In the following, for sake of brevity, we specify only the compute \( CT(u_j) \) type operations in replay sequences. The other operations (checkpoint, restore, evict) are separately specified. In this briefer format, each step of a replay sequence is of the form

\[ RS(u_j):CT(u_j): u_{CP}(u_j): u \]

**Definition 3 (Ex-Ancestor replay sequence).** Suppose \( T = (V, E) \) is an execution tree. Given any replay sequence \( R = \{(u_i, S_t) : 1 \leq t \leq T\} \) we define its first appearance order to be \( i_1 < i_2 < \cdots < i_{|V|} \) such that \( u_{i_j} \) is the first appearance of a node \( u_j \) of \( T \) in \( R \). We call the indices \( i_1, \ldots, i_{|V|} \) as first appearances and all other indices as
repeat appearances. For a replay sequence \( R \), for each \( j \in \{2, \ldots, |V|\} \) the sequence of cells \( u_j, u_{j+1} \ldots u_{j-1} \) is called the helper sequence for \( v_j \). If the helper sequence for \( v_j \) forms a path from an ancestor of \( v_j \) to \( v_j \) for each \( j \) then \( R \) is called an ex-ancestor replay sequence.

We observe that in an ex-ancestor replay sequence if the helper sequence of \( v_j \) is non-empty then it either begins with the root of \( T \) or with a node whose parent is in \( S_j \). If \( j = 1 \), the helper sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance.

We illustrate this definition with an example. Consider the tree in Figure 5. Assume for now that cache size \( B = 0 \) and consider the following replay sequence:

\[
a, b, d, g, k, o, a, c, e, h, l, a, c, f, i, m, a, c, f, i, n, p, a, c, f, j
\]

where bold font indicates repeat appearance nodes. Here the indices 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 14, 15, 16, 21, 22, and 26 are first appearances and all others are repeat appearances. Let’s take the example of node \( n \). It’s first appearance index is 21. Its helper sequence extends from indices 17 to 20 and contains \( a, c, f, i \). This is a simple path in the tree beginning from the node containing \( a \) which is an ancestor of \( n \). In fact it is easy to verify that this sequence is an ex-ancestor replay sequence.

The question arises: Are there meaningful replay sequences that are not ex-ancestor replay sequences? For example, would it make sense to modify \( n \)’s helper sequence and make it \( a, c, e, f, i \). It appears that the extra computation of \( e \) is superfluous and so a priori it is not obvious that such replay sequences are meaningful from the point of view of efficient replay. Therefore we focus on ex-ancestor replay sequences. We conjecture that an optimal solution to MVR-P will be such a sequence.

Definition 4 (DFS-based replay sequence). Suppose \( T = (V, E) \) is an execution tree for a collection of traces \( C \). We say a complete and minimal replay sequence \( R \) is a DFS-based replay sequence if \( R \) is an ex-ancestor sequence and the first appearance order of \( R \) is a DFS-traversal order of \( T \).

Note that first appearance sequence of the example discussed below Definition 3 gives us a DFS-traversal of \( T \). Hence this is a DFS-based replay sequence for the tree of Figure 5.

Now assume a cache size \( B = 25 \) and the caching decisions made according to the first replay sequence in Figure 7.1 The corresponding replay sequence is similarly:

\[
\begin{align*}
a, b, d, g, k, o, a, c, e, h, l, a, c, f, i, m, a, c, f, i, n, p, a, c, f, j
\end{align*}
\]

Since \( a, c, f, i \) and \( f \) are cached at appropriate junctures, the only node with a non-empty helper sequence is \( n \) and the length of this sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance creation.

We are able to explicitly bound the number of DFS-based replay sequences.

Proposition 1. Suppose \( T = (V, E) \) is an execution tree for a collection of traces \( C \) such that \( |V| = n \) and the height of \( T \) is \( h \). Let \( b_u \) be the number of children of \( u \in V \) and let

\[
\bar{b} := \frac{1}{n} \sum_{u \in V} b_u \log b_u.
\]

Then, the number of DFS-based replay sequences of \( T \) is \( O(2^n \bar{b} \log h \bar{b} + h \bar{b}) \).

Proof. Let us fix a DFS traversal order. The helper sequence preceding (and including) each node can be at most \( h \) in length and hence the length of a replay sequence can be no longer than \( h \bar{b} \). Since each helper sequence is an ex-ancestor path to a node of the tree, we can have at most \( h \) choices of a helper sequence at each node. Therefore there are at most \( 2^{h\bar{b}} \) different sequences that can qualify to be DFS-based replay sequences. Note that at each point of one of these sequences of cells we can decide to either cache the cell that we have just computed (this may require the eviction of something previously cached) or not to cache it. Hence there are at most \( 2^{nh} \) replay sequences associated with each of the \( h^n \) different sequences that we got for a single DFS traversal order. This gives us an upper bound.

To compute the number of DFS traversal we simply permute the children visited by DFS at each step to get \( \prod_{u \in V} b_u \) which can be rewritten as \( 2^{nh} \) by using Stirling’s approximation. Multiplying we get the result.

Since \( h \) is \( \Omega(\log n) \) from Proposition 1 it appears that the space of possible solution is superexponential. In order to control the complexity of our solutions we restrict the solution space in two different ways and define two heuristics.

5.1 Persistent Root Policy Greedy Algorithm

Within the space of DFS-based replay sequences, for our first heuristic, we propose the following caching policy: A cell can be cached only when it is first computed. Once cached the cell remains in the cache till every leaf of the subtree rooted at the node containing cell is computed. We call this the DFS Persistent Root policy.

Given a DFS traversal order this policy reduces the size of the solution space to \( O(2^n) \) which is still exponential in the size of the tree. We present a greedy algorithm called Persistent Root Policy Greedy (PRP) that helps find a good solution in polynomial time. We present the listing of PRP as Alg. 1. The algorithm begins with the baseline cost (stored in min) of a DFS-based replay sequence in which no node is cached and seeks out the node of the tree whose addition to the list \( S \) achieves the maximum improvement over the baseline. This process continues incrementally while it is possible to include another node in the list. The process will stop when the subroutine \( DFS\text{Cost} \) tells us that there is no node remaining in \( V \setminus S \) that can be included in \( S \). Typically this will happen because for every node \( u \) remaining in \( V \setminus S \), the cache will be full when it is encountered in the DFS order. In such a situation \( DFS\text{Cost} \) will return \( \infty \). This algorithm takes \( O(n^2) \) time to find each candidate to include in the list of nodes to be cached, and there are potentially \( O(n) \) such nodes. Therefore the time complexity of this algorithm is \( O(n^3) \). However there is no guarantee of optimality.

PRP is a greedy algorithm that seeks, at each iteration, to pick for caching the vertex of the execution tree that minimizes the cost. However, it can be easily modified to choose a vertex that minimizes the cost incurred per unit of cache memory consumed. Normalizing by size is a common measure for object caches [7, 32], We experimentally study both these variants in Section 7. We will refer to the cost-minimizing version as PRP-v1 and the ratio minimizing version as PRP-v2.
5.2 Parent Choice Algorithm

We now present a second heuristic that, while still not being optimal, searches a superset of the portion of the solution space searched by PRP. For each $u \in V$ it seeks to partition the children of $u$ into two sets: $P_u$ of nodes for which it is better to cache $u$ for the computation of the corresponding child subtrees, and $\overline{P}_u$ for which it is not. As in Persistent Greedy, caching choices once made persist here as well.

The listing of the essential recursive Parent Choice is presented as Alg. 2. When called with $(u, S)$ we explore the situation in which we are given the set $S$ of ancestors of $u$ that will be in cache while the subtree rooted at $u$ is computed. In case $u$ happens to be a leaf, no further decisions are needed, and we simply return the cost of computing $u$ given cache $S$ (Lines 2-4). Else, we need to determine what is best for each child $u_i$ of $u$: Should the subtree rooted at $u_i$ be computed with $S$ as is, or is it better to augment the cache with $u$ (denoted $S_{au}$)? In the former the subtree may be forced to recomputed $u$ multiple times, in the latter cache space which may be more useful down the subtree is used up. The two costs are computed recursively (Lines 14-15), and the child is assigned to the set $P_u$ or $\overline{P}_u$ corresponding to the lower cost (Lines 16-19). Note that when adding $u$ to the cache is infeasible, i.e. $|S_{au}| > B$, we make the first choice for each node, i.e. assign them all to $P_u$ (Lines 7-10). Finally, we return the cost value up the recursion stack (Line 20).

The essential recursive algorithm PC needs to be implemented using standard dynamic programming memoization and backpointers (see, e.g. [11]). Once a call with input $(u, S)$ is complete, the corresponding return cost value and the two sets $P_u$ and $\overline{P}_u$ are recorded. The initial call is with $(\text{root}(T), \emptyset)$. This returns the cost of the optimal replay sequence for the entire $T$. To construct the replay sequence itself, “follow the backpointers”: Start with $u = \text{root}(T)$ and $S = \emptyset$. If for the corresponding call, $P_u$ is not empty, compute $u$ (possibly by restoring the closest ancestor in the current cache) and checkpoint it. Update $S$ to include $u$. Then recursively compute the subtrees rooted at the nodes in $P_u$. Next update $S$ to remove $u$. Following this, recursively compute the subtrees rooted at the nodes in $\overline{P}_u$, if any.

The above implementation takes time and space proportional to the total number of child nodes encountered over all recursive calls. For each $u \in V$, at most one recursive call is made for each possible set of ancestors in the cache. The number of different ancestor sets is at most $2^B$. Thus the total time taken is $O(2^B \sum_{u \in V} n_u)$.

---

We do not explicitly show the cost of computing $u$ in order to cache it for $P_u$. This cost is offset by the same cost incurred by the first child subtree in $\overline{P}_u$, as shown, but not actually paid since $u$ is already in cache when it is executed. The case of $P_u = \emptyset$ has a further optimization that is possible; see the full version of this paper [34].
6 EXECUTION TREE

We now discuss how CHEX constructs the execution tree at Alice’s end. As per Definition 1, an execution tree merges equal program states of different versions into a single node in the tree. Given the per cell values of state computation time $\delta_i$ and size $sz_i$, state lineage $g_i$, and state code hash $h_i$, we use the following conditions to identify equal program states:

**Definition 5 (State equality).** Given two program versions $L_1$ and $L_2$, state $ps_i$ in $L_1$ is equal to state $ps_j$ in $L_2$, denoted $ps_i = ps_j$, if and only if (i) $h_i = h_j$, (ii) $g_i = g_j$, and (iii) $\delta_i$ and $sz_i$ costs are similar.

In other words we say that two states are equal if they are reusable *i.e.*, they are (i) equal at code syntactic level, (ii) after cell execution, result in the same state lineage (note state lineage of $i^{th}$ cell depends on state lineage of previous cell), and (iii) have roughly similar execution costs. Program state does not remain equal when cell code is edited, which changes the hash value of that cell and any subsequent cell. Similar states across versions also do not remain equal if costs change drastically, *i.e.*, computed on different hardwares (*viz.* GPU vs CPU). Equating state lineage depends on the granularity at which the system events are audited. Since in CHEX, lineage is audited at the level of system calls, there are some pre-processing steps that are necessary to establish equality, such as accounting for partial orders, abstracting real process identifiers, and accounting for hardware interrupts. We describe these issues below.

Lineage equality implies that at end of cell $i$ of version $L_1$, $g_i$ is the same as that at end of cell $i$ of version $L_2$. This is true if and only if the sequence of system call events (and their parameters)—till $i$ in $L_1$ and $i$ in $L_2$—exactly match. But if a cell, *e.g.*, forks a child process, which itself issues system calls, then each version’s sequence will contain the parent calls and the child process calls interleaved in possibly different orders.

In Figure 3 the parent process forks a child and then issues a ‘mem’ memory call, and the child process itself issues ‘exec’, ‘open’, and ‘read’ calls. As the figure shows, it is possible that in the sequence for version $L_1$ the ‘mem’ access is before the ‘read’, while for $L_2$ it is after. If we want to correctly determine that the state in $L_1$ is identical to that in $L_2$ at this point, we need to recognize that the sequence of system calls is an arbitrary total order imposed on an underlying partial order. The partial order for $L_1$ and $L_2$ is identical, while the total order can differ.

In our implementation, we essentially reconstruct the underlying partial order when we detect asynchronous computation, and match it to identify equality of program states in different versions. This is achieved by separating the events into PID-specific sequences and then comparing corresponding sequences. The above comparison can only be established when process identifiers are abstracted to their logical values. Memory accesses cannot be abstracted and we just count the number of accesses in a cell. Comparison must also account for external inputs in addition to system events. As Figure 3 shows the hash of external dataset file ‘new fashion’ is changed from ‘b2e1772’ to ‘6789b34. Thus, the two cells cannot be equated even though the order of system call sequence in $E$ is the same.

A related nuance is due to hardware interrupts. If $P_1$ experiences a hardware interrupt and $P_2$ does not, we make the safe choice: assume the program states are not equal. (It is easy to make the opposite choice, by simply ignoring hardware interrupts, if so desired.)

7 EXPERIMENTAL EVALUATION

We now describe CHEX’s implementation and present an extensive evaluation of CHEX for multiversion replay.

**Implementation.** CHEX is implemented in C and Python. CHEX relies on Sciunit [1] for monitoring the application on Alice’s side and it relies on Checkpoint/Restore in Userspace (CRIU) [10] to checkpoint/restore program states. CHEX maintains a ramfs cache to maintain checkpoints. These checkpoints are of the process corresponding to the REPL program and not of the container that Sciunit creates.

We use CRIU as a checkpointing mechanism. This is precisely to enable checkpoint of a process independent of its programming language. CRIU does not freeze the state of the container but just the application process. Currently, CHEX is integrated with the IPython kernel. In future, we plan to integrate CHEX with Xeus [9], which will help us extend CHEX to C programs as well. For the purposes of reproducibility we have made available the code for the audit and replay mode of CHEX at [40].

We used a combination of real-world applications and synthetic datasets for evaluation. We ran all our experiments on a 2.2GHz Intel Xeon CPU E5-2430 server with 64GB of memory, running 64-bit Red Hat Enterprise Linux 6.5. The heuristics were developed in Python 3.4.

**Real-world Applications.** We searched GitHub and identified compute- and data-intensive notebooks, *i.e.*, the programmer had already divided the code into cells. Most of these notebooks were published as artifacts in specific domain conferences (pre-established to be reproducible), and they were described as compute- and data-intensive.

We used four neural network machine learning applications (ML) and two scientific computing (SC) applications. Table 1 describes the characteristics of these notebooks. For the majority of the applications, the number of versions was determined in consultation with the notebook authors, by identifying meaningful changes to parameter values. Other notebooks were changed similarly. Total replay cost is the time to run all the versions with no cache. Total checkpoint size is the space required if each cell of the corresponding execution tree is checkpointed. Cell compute range and Cell checkpoint size represents the range of cell compute time and checkpoint size ranges, respectively. The changed parameter row mentions application parameters that were changed to create versions. The only way we created versions was by changing parameters. We did not modify any part of the programs in any other way.

The case of the parameter epochs in ML notebooks is special. In our case, the ML notebooks embed deep neural networks, in which typically the compute-intensive part is the back propagation during the training phase. Back propagation is usually implemented as an iterative for-loop, whose upper bound is defined by the epochs parameter. Changing epochs will change the training length and the

---

1Native serializations, *viz.* Pickle, provide only a slight performance benefit (1-2%).
Table 1: Six Real-world Applications

<table>
<thead>
<tr>
<th>Dataset</th>
<th>ML1</th>
<th>ML2</th>
<th>ML3</th>
<th>ML4</th>
<th>SC1</th>
<th>SC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Network</td>
<td>Prediction</td>
<td>Image</td>
<td>Time Series</td>
<td>Analysis</td>
<td>Image</td>
</tr>
<tr>
<td>Changed parameter</td>
<td>version (40)</td>
<td>year (20)</td>
<td>input size (30)</td>
<td>quality (15)</td>
<td>[2]</td>
<td>[44]</td>
</tr>
<tr>
<td>Number of versions</td>
<td>20</td>
<td>24</td>
<td>32</td>
<td>36</td>
<td>12</td>
<td>23</td>
</tr>
<tr>
<td>Total (no-cache) replay cost (s)</td>
<td>0.5</td>
<td>0.7</td>
<td>1.0</td>
<td>1.2</td>
<td>1.5</td>
<td>1.8</td>
</tr>
<tr>
<td>Total checkpoint size (GB)</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
</tr>
<tr>
<td>Cell compute range (s)</td>
<td>0.0002</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0006</td>
<td>0.0007</td>
</tr>
<tr>
<td>Cell checkpoint size (GB)</td>
<td>0.0001</td>
<td>0.0002</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0006</td>
</tr>
<tr>
<td>Max. Branch-out Factor</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>0.5</td>
<td>0.6</td>
</tr>
<tr>
<td>Max. Version Length</td>
<td>1000</td>
<td>2000</td>
<td>3000</td>
<td>4000</td>
<td>5000</td>
<td>6000</td>
</tr>
</tbody>
</table>

Figure 8: Performance of DFS algorithms on 6 real-world applications. X denotes the size of the largest checkpoint cell as specified in the last row of Table 1. The y-axis is truncated to show finer performance variations between algorithms.

Table 2: Three Synthetic Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CI</th>
<th>DI</th>
<th>AN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Compute-intensive</td>
<td>Data-Intensive</td>
<td>Analytical</td>
</tr>
<tr>
<td>Max. Branch-out Factor</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>Max. Version Length</td>
<td>2000</td>
<td>4000</td>
<td>6000</td>
</tr>
<tr>
<td>Number of versions</td>
<td>20</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Total (no-cache) replay cost (s)</td>
<td>10000</td>
<td>20000</td>
<td>30000</td>
</tr>
<tr>
<td>Cell compute range (s)</td>
<td>100 - 1000</td>
<td>100 - 1000</td>
<td>100 - 1000</td>
</tr>
<tr>
<td>Total storage size (GB)</td>
<td>20</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Cell checkpoint size (GB)</td>
<td>0.5</td>
<td>0.6</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Baseline. IncPy [17, 18] avoids recomputing a function with the same inputs when it is called repeatedly or across program versions. Despite our best attempts we could not get IncPy to run with our real datasets. IncPy is not longer actively maintained and is Python 2.7 based which creates conflicts with more recent notebooks. We simulated the Vizier system, by taking one notebook version at a time [6], and using the simple caching policy that is used for Vizier: Least Frequently used (LFU), which is a standard caching algorithm. We adapt LFU to our case by checkpointing every cell of the first version of a notebook till the cache space fills up. As subsequent versions arrive, the cache eviction policy is decided by the measure frequency × # of nodes in subtree/cell size, i.e., retaining cells which are used frequently and are responsible for larger subtree, normalized by their size. Least recently used, another standard caching algorithm, is not relevant in our case due to the depth-first replay order.
7.1 Experiments
We first evaluate the benefit different algorithms provide in terms of reduction in replay time. We then evaluate the overhead of operating CHEX.

7.1.1 Comparing decrease in replay cost via different algorithms. Persistent Root Policy (PRP) and Parent Choice (PC) make different choices with respect to cells that must be retained in cache for recomputation. In this experiment, we evaluate how those decisions compare with the (LFU) baseline. Recall that PRP has two versions: PRP-v1, in which we cache checkpoints greedily based on contribution to reduction in cost, and PRP-v2, in which we normalize the cost reduction by the checkpoint size.

To compare algorithmic performance, we choose a cache size that is equal to the largest checkpoint size in a notebook and compute total replay time. The y-axis is initialized with a non-zero value to show finer comparisons between algorithms. For both PC and PRP algorithms on real-world applications, as is expected, Figures 8(a)-(f), show decreasing compute times (y-axis) as the cache size is increased (x-axis). We also see PRP and PC always perform substantially better than LFU, and PC reduces total compute cost more than either of the PRP versions.

Both these result trends are not exhibited in Figure 8(e) (SC1) and, to an extent, in Figure 8(d) ML4. In (e), as we observe, none of the algorithms, including the baseline LFU, show any benefit of caching. This is because in this notebook only the last cell of each version is compute-intensive, and none of the intermediate cells are cache-worthy. In (d), similarly, most computation is towards the later cells; PRP and PC still find some ways to optimize which LFU cannot find. The effect of reuse of intermediate results is well-demonstrated when comparing ML4 and SC2 which exhibit similar total replay costs. However, there is a much greater reduction in total replay cost in SC2 (from 100% to 18%) as there are several compute-intensive pre-processing steps in the earlier cells of the notebook, where as in ML4 most computation occurs towards the later cells.

Analyzing deeper we also observe these trends: (i) Sometimes, initially, PRP performs better, and this happens due to small cache size effect, since PC becomes a clear win with some additional cache space; (ii) PRP-v1 performs better than PRP-v2 indicating that eviction on a cost/size measure leads to more greedy eviction policy where checkpoints are evicted which need to be recomputed later; and finally (iii) ML1, ML3 and SC2 are compute-intensive notebooks. Using the PC algorithm, these notebooks show a reduction of 60-65% in their compute time at a size of the cache which is at most double the size of the largest checkpoint cell in the notebook. This indicates that smart algorithms can provide significant benefits even with small cache sizes. We obtain similar results for synthetic datasets, and, for lack of space, only include the figures for synthetic results in the extended version of the paper [34].

7.1.2 Determining number of versions replayed with fixed cache size. We also examine the direct benefit of a system like CHEX for users. For most users CHEX will be configured with a given amount of cache space. Users, however, have time constraints. Thus we determine, for given cache sizes, number of versions that can be replayed with CHEX in a given amount of time, on the AN dataset.

Figure 9 presents the result (number of versions (y-axis)) for the amount of time it takes to replay them (x-axis) for a given cache size, the value being either: no cache, 0.25GB, 0.5GB, and 1GB. The Figure shows that a user can run 50% more number of versions by doubling the space for the same fixed amount of time. To be able to run larger number of runs for the same amount of time has implications for scaleable collaborative sharing and artifact evaluation use cases.

7.1.3 Time and space required to run CHEX. We first determine the cost of auditing an application in CHEX.

Cost of Auditing. CHEX performs auditing of state for each version of an application in terms of computation time $\delta$, state size $sz$, state code hash $h$, and state lineage $g$. We report both normal execution and audited execution as a percentage of the total time of using CHEX on a real application.

Amongst these audited quantities, the primary overhead is the additional time required to audit the application for state lineage, i.e., $g$. We further divide time to audit for $g$ into time required to (i) monitor and log system events in the application, and (ii) the time required to compute the hash of any external content that is referenced. As Figure 10(a) shows, a 15-25% of total auditing overhead is added across all applications. We are reporting 5 out of six applications as ML2 has relatively insignificant running time to begin with.

Also the time to perform cell equality and construct the execution tree is negligible. Alice shares a package with the execution tree, the size of which is less than 1KB.
Cost of computing cache eviction decisions. We have shown the multiversion replay problem to be NP-hard; PRP and PC are heuristic algorithms, and thus have some time and space cost of making the cache eviction decisions. In this experiment, we measure the cost of using PRP and PC algorithms in comparison to LFU in terms of running time, space used, and number of times checkpoint/restore call was made. We experimented with the AN synthetic dataset.

The variability in running time of the algorithms with cache size is negligible (~[0.05%]). Therefore, we fix the cache size to 1GB and show the variation with respect to two other parameters, the number of nodes in the tree, i.e., tree size and the different algorithms. Figure 11 (a) shows that PC is better than PRP in terms on run-time overhead, but in terms of space, it has a cost as shown next.

PRP has negligible state maintenance as it uses the execution tree to determine the order. However, PC takes storage, because it has to store all possible combinations of execution orders for different cache eviction sizes to get the most optimal one. Figure 11 (b) shows the increase in storage for different tree sizes as cache size is increased. Despite these differences, we highlight that the runtime and memory overheads of both algorithms is much lower (0.5-2%) than the overall compute time and storage of multiversion execution of any given real dataset.

The above experiment measures decision-making time and space. In practice to implement the decisions we must account for in-memory checkpoints and restore (C/R) time. In general, time to C/R are proportional to the size of the checkpointed state and are negligible. So we measured the number of times C/R were performed to check if small C/R costs adds to the overall latency of multiversion execution (Figure 11 (c)). As we see C/R costs are negligible, and algorithm decision making accounts for the primary cost of CHEX.

Apart from the experiments reported above, we attempted a comparison between PC and an optimal algorithm, using the AN dataset for comparison. For optimal, we wrote our problem, the MVR-P, as an Integer Linear Program (ILP) and attempted to solve it with the Couenne optimizer [30]. The Couenne timeout was set as ten minutes. For tree size of 2-6 nodes, Couenne finished finding a solution in less than 10 seconds, but after that the time starts increasing exponentially. At 12 versions and an execution tree of 20 nodes, the optimal solution could not be found within the set time out. On increasing the number of versions, it took more time to find the optimal solution than naive replay (without cache). On the other hand, as we show in Figure 11(a) we took milliseconds to find a solution for more than a tree-size of 30.

Since we only found optimal solution for small trees, in terms of the quality of the solution, we found the replay cost of PC similar to optimal. For larger tree sizes, we anticipate it to give better cost estimates, but given the large running time of optimal for larger and complex instances, we assess, it is not worth it. Finally, the overhead of implementing the decisions in CHEX is too small to be measured and often smaller than the variance between multiple runs.

8 RELATED WORK

Tools and hubs for sharing and reuse. Sharing and replaying is essential for verifying, and reproducing complex applications. Several user-space virtualization based tools have recently been proposed to enable sharing and repeating computations [8, 19, 22, 41, 43, 55]. These tools do not address multiversion replay. In a virtualization package, code and data remain separate as files or databases [43]. Computational notebooks, which combine code and data, have received wide attention recently for sharing and use [26]. Notebook sharing, like package sharing, is easy but (re-)execution across versions remains sequential. Nodebooks [58] and Vizier [6] are specialized notebook clients that support and store notebook versions at a cell level. Neither, however, compute deltas between versions or trade computation for storage. Our work complements specialized notebook systems used for interactive development [24], and given lineage from these systems [31], replay can be enabled.

Execution lineage. There are several provenance models for capturing execution lineage [52]. In this paper, we adopt the system-event trace analysis process that is also used in other whole system provenance tracking methods [3, 15, 51].

Data caching. Data management systems have a rich history of employing object caches that tradeoff space for time to improve performance of applications. Semantic caching allows caching of query results [12, 48], web-object caching allows caching of web objects [7, 23], and query-based object caching allows database object caching based on queries [32]. In all of these works, the workload sequence is not known. In the multi-query scenarios [48] the workload is presented as set of queries and hence there is the possibility of caching the results of common sub-expressions and reusing them across queries. However, efficient reuse in the multi-query setting primarily involves searching through the space of query answering plans to identify plans that could potentially lead to optimal reuse. In certain cases not finding the optimal plan and blindly reusing

Figure 11: Algorithm complexity of AN workloads: (a) running time, (b) storage size for PC, and (c) number of checkpoints/restore-switch for PC.
common subexpressions may blow up the computation time because we do not have the wiggle room provided by the semantics of a query, nor the potential pitfalls associated with blind reuse.  

**State management for recomputation.** [54] provides an excellent survey of state management for computation. State can be recomputed from lineage or can be stored ‘as-is’. In Scilnc [56] state is recomputed from lineage that is versioned. Versioned lineage or causality-based versioning [36, 56] leads to correct computation of state for incremental replay. In this work, on the contrary, we are concerned with state that is stored ‘as-is’. Several works store ‘as-is’ state—this state is state of a variable, query, program, or configuration [54]. Similar to [20, 21, 25, 39], in this work, our operator is program state. However, in these works the purpose is fault-tolerance, and so the system periodically checkpoints but does not consider space limitations. We determine a limited number of checkpoints of program state to save in-memory space, and using lineage, choose to simply recompute when efficient. To reduce space an alternative would be to incrementally checkpoint as explored in differential flows [35, 37] and query re-optimization [29]. These approaches are not extendable to checkpoints of program state, which is an in-memory map. Very recently checkpointing was used to improve efficiency, but the checkpoint frequency is periodic [14].  

**Checkpoint location.** Deciding when to checkpoint has received attention in HPC scheduling [5, 47]. A primary objective is to minimize the amount of computation that needs to be redone in case the system fails. In HPC workflows, the checkpoint also has an overhead. We consider machine learning and scientific computing programs in which the checkpoint overhead is nearly zero.

Closer in spirit to our work is the DataHubs [4] system that seeks to maintain multiple versions of large data sets without fully replicating them. In this system some versions are stored fully materialized and others are stored only as deltas linked to other versions. The problem is to trade off total storage required versus time taken to recreate a version. At a glance, it is possible to think that the program states of the cells of our multiversion program can be aligned with the data sets considered in DataHubs. However, the fundamental difference is that DataHubs assumes each version of a data set has already been created the first time. Thus, they assume that at least one version of the data set is stored in its entirety. In CHEX, the equivalent thing would be for Alice to share some of the program states generated in her execution with Bob. This defeats the entire purpose of independent repetition by Bob.

We have assumed multiple versions for a given program. We make no assumptions on the types of edits that constitutes a version on Alice’s side. Thus, Alice can change values of parameters, specifications of datasets, models, or learning algorithms. She can also add or delete entire cells. In practice we have found such versions to not correspond to development versions but as separate branches in version-control repositories. In workflow systems they also correspond to independent, but related, experiments.

We have only demonstrated a scenario in which Alice shares notebooks with Bob for multiversion replay. A more evolved back-and-forth sharing of packages, one that accounts for any previous multiversion replay decisions to be persisted, will require further changes both to the system and the algorithm. In such a scenario, if the caches persist, some intermediate results are available for free and the algorithm needs to accommodate for that accordingly. This scenario is part of our future work.

Finally, our experiments show that CHEX significantly decreases the replay time for notebooks and allows a user to execute a far higher number of versions in a given amount of time. The benefit arises particularly for notebooks where pre-processing or training steps are compute and data-intensive. In particular, if all computation is conducted in the last cell, then opportunities for optimization on intermediate results reduce drastically. In this case, one option is to encourage the developer to further divide the last cell, which creates further opportunities of optimization. If the cell cannot be divided, then one may employ a hybrid approach of using function-based caching within this cell. This may, however, require some analysis of the program in the last cell.

10 CONCLUSION

In this work we have highlighted the need for improving the efficiency of multiversion replay. Our work shows that execution lineage can be used to establish cell equality and reuse shared program state to optimize replaying of multiversions. We show that optimizing is not trivial and, given a fixed cache size, MVR-P is NP-hard and present two efficient heuristics for reducing the total computation time. We develop novel checkpoint-based caching support for replaying versions and show that CHEX is able to reduce the compute time of several machine learning and scientific computing notebooks using a cache size that is smaller than the checkpoint size of a notebook.

In the future, we wish to extend CHEX for queries and the standard database provenance model. This problem seems akin to how we previously extended provenance-based application virtualization [42] to database virtualization [43]. We also wish to explore how CHEX can incorporate program restructuring, which happens during interactive notebook development leveraging recent provenance models developed in this area [6, 24, 31] and developing corresponding online algorithms.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918 and a Department of Energy Fellowship.


