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:
Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. CHEX: Multiversion Replay with Ordered Checkpoints. PVLDB, 15(6): XXX-XXX, 2022.
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, but (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,

*Work done as part of a summer internship at DePaul.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514075

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

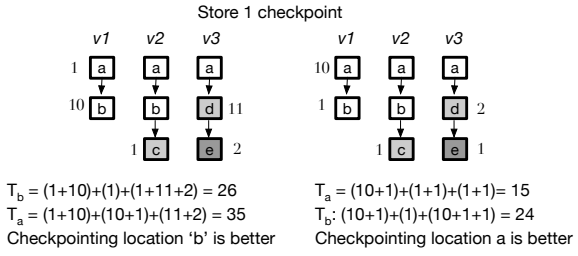


Figure 1: Deciding checkpoint location depends on cost and size estimates of the cells.

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.

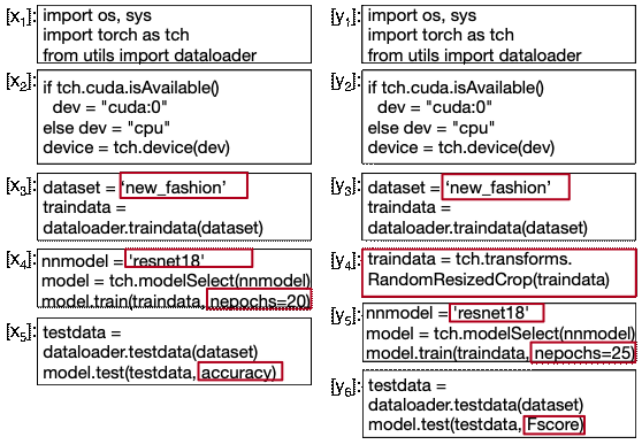


Figure 2: An illustration of REPL programs. The left program L_1 trains a machine learning model (resnet18) on a training dataset and evaluates its accuracy on a test dataset. The right program L_2 is the same, except that it adds a preprocessing step to the training dataset. L_1 has 5 cells, L_2 has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

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 cells, *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. So, e.g., for the program $L = [x_1, x_2, \ldots, x_5]$, the corresponding program states are $[ps_0, ps_1, \ldots, ps_5]$, in which ps_{i-1} denotes the program state just before cell x_i is executed. The state ps_0, which is just before the first cell is executed, includes the value of the environment and any initial input.

CHEX works in combination with an *auditing system* which monitors executions and provides the following details about each program state, ps_i:

- **computation time**, δ_i, the time to reach the program state ps_i from its predecessor ps_{i-1},
- **size**, sz_i, size of the program state ps_i,
- **code hash**, h_i, computed by hashing code in cell i, and
- **lineage**, g_i, which is determined by combining the predecessor cell's lineage with the sequence of system events that are triggered by program instructions in the cell i and the hashes of the associated external data dependencies. Thus, $g_i = (g_{i-1}, h_i, E_i)$, where E_i is the ordered set of system events in cell along with the hash of the content accessed by the event. Initially, $g_0 = \{\}$.

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 ps_i. Therefore, ps_i 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.

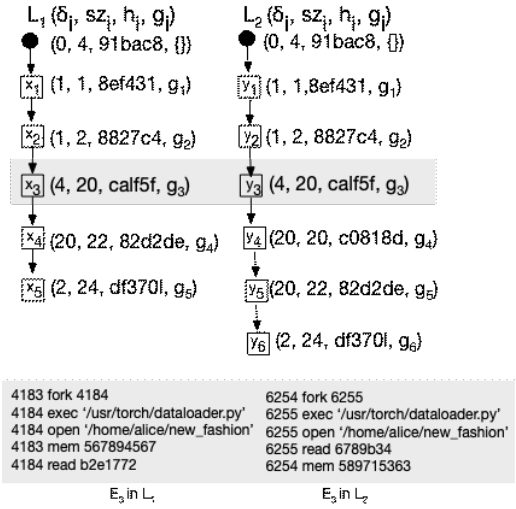


Figure 3: Auditing of programs L_1 and L_2 in terms of δ, sz, h, g. Events in E_3 show a forked process, open of an external file along, and read of data content, denoted by its hash value. We determine how to check if E_3 across versions is equal in Section 6.

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_4. 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 δ, 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.

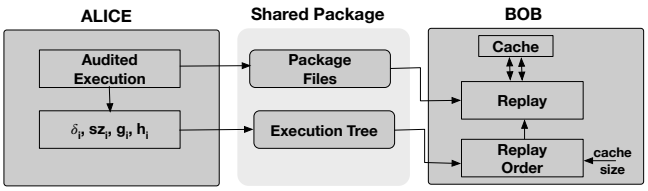


Figure 4: CHEX Overview.

CHEX is used in replay mode on Bob's side. It first determines an efficient *replay sequence* or replay order, i.e., a plan for execution that includes checkpoint caching decisions. To do so **CHEX** inputs a cache size bound, B, and then executes a heuristic algorithm on the execution tree received from Alice to determine the most cost efficient replay sequence for that cache size. Computing a cost-optimal replay sequence for multiple versions of a program with a cache bound is an NP-hard problem as we show in Section 4 and so we describe some efficient heuristics for this purpose (Section 5). Finally, once the replay sequence is computed, **CHEX** uses this replay sequence to *compute, checkpoint, restore-switch* REPL program cells or *evict* stored checkpoints from cache.

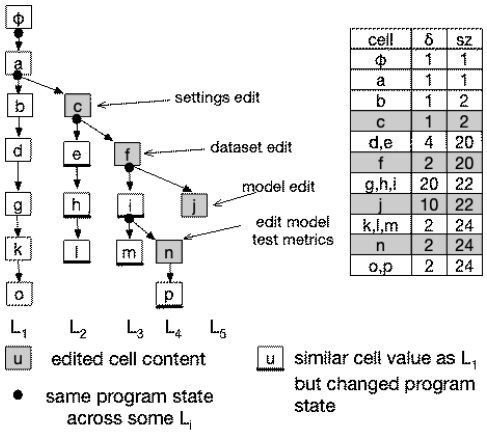
Our assumptions. Our basic assumption is that Bob wishes to independently verify the results from Alice's versions but is time constrained to repeat all her versions. We do not make any assumptions on the types of edits that differentiates one version from the next. Thus, Alice can change values of parameters, 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 summarizes 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 common nodes till a branch in the tree correspond to the subsequence of cells that are equal across versions. The tree branches at a cell node, 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.



cell	δ	sz
Φ	1	1
a	1	1
b	1	2
c	1	2
d,e	4	20
f	2	20
g,h,i	20	22
j	10	22
k,l,m	2	24
n	2	24
o,p	2	24

Figure 5: The enhanced specifications of versions (without lineages for simplicity) represented as a tree. The cell i appears similar to h but has a changed program state due to edited f. Both m and n proceed from i's state.

The multiversion replay problem is an optimization problem that arises when multiple versions of a program, each previously executed, 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 incur 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 of 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 checkpointed 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;

L_j	REPL program version
x_i	Cell in L_j
h_i	Hash of source code in x_i
g_i	Cumulative hash of source code and ext. dependencies till x_i
ps_i	Program state at end of x_i
sz_i	Size of ps_i
δ_i	Computation time to reach ps_i from ps_{i-1}
T	Execution tree combining overlapping L_j's
B	Fixed cache size
R	Replay sequence for T
O_t	Operation at step t in R
S_t	Set of program states in cache after step t in R
$\delta(R)$	Computation time for R

Figure 6: Notation used in Section 2 and Section 4

The cache size can never exceed B, i.e. $|S_t| \leq B$ for $0 \leq t \leq m$. Further, an operation O_t at step t can only be performed on u_j, u_k under the following constraints:

- *Checkpoint from working memory:* A node in the execution tree is checkpointed only if it was computed in some previous step, after which there are only some evictions (to make space), if at all, *i.e.,* if $O_t = CP(u_j) \implies S_t = S_{t-1} \cup \{u_j\}$ and $O_{t-i} = CT(u_j)$, for some $1 \leq i \leq t$, and $O_{t'} = EV(u_{t'})$ for $t - i < t' < t$.
- *Restore from cache and switch to child:* A node is restored only if it was in cache in a previous step, and without altering cache state, switches to one of its children in the execution tree, which is computed next *i.e.,* if $O_t = RS(u_j, u_k) \implies u_j \in S_{t-1}, S_t = S_{t-1}, O_{t+1} = CT(u_k)$.
- *Evict from cache:* A node is evicted from cache and alters its state, *i.e.,* if $O_t = EV(u_j) \implies u_j \in S_{t-1}, S_t = S_{t-1} - \{u_j\}$.
- *Continue computation:* Continue computing a node if its parent was being computed or if its parent was restored, *i.e.,* if $O_t = CT(u_j) \implies O_{t-1} = CT(parent(u_j))$ or $O_{t-1} = RS(parent(u_j), u_j)$ and $S_t = S_{t-1}$ or t= 1 and u_j=root of the tree T.

We assume the operations generate *complete* and *minimal* sequences. A replay sequence is complete if all leaf nodes of the tree T appear in R, and is minimal if no u_j that is in cache is recomputed.

Problem 1 (The Multiversion Replay Problem (MVR-P)). *Given tree $T(V, E)$, the multiversion replay problem is to find a complete replay sequence R that minimizes*

$$\delta(R) = \sum_{i=0}^{|R|=m} \delta_{O_t},$$

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

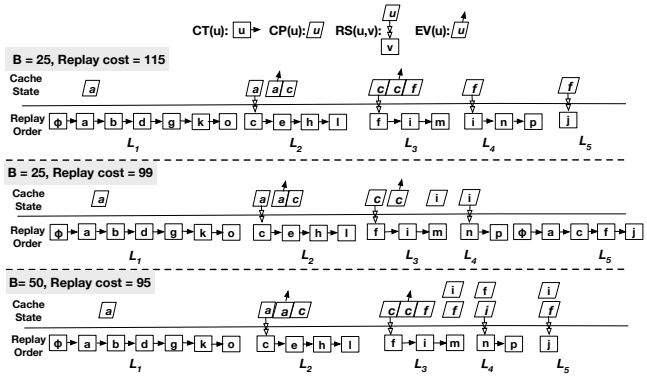


Figure 7: Replay sequences for the execution tree in Figure 5 showing use of operations and the state of the cache.

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

We show that the decision version of MVR-Pis 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 Δ 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 1 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 Δ 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 (u_t, S_t) specifying that at step t, u_t is computed, and the resulting cache is S_t.

Definition 3 (Ex-Ancestor replay sequence). *Suppose $T = (V, E)$ is an execution tree. Given any replay sequence $R = \{(u_t, S_t) : 1 \leq t \leq T\}$ 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 v_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* $u_{i_{j-1}+1} \ldots u_{i_j-1}$ *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 $S_{i_{j-1}}$.

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:

$$a, b, d, g, k, o, \mathbf{a}, c, e, h, l, \mathbf{a}, \mathbf{c}, f, i, m, \mathbf{a}, \mathbf{c}, \mathbf{f}, \mathbf{i}, n, p, \mathbf{a}, \mathbf{c}, \mathbf{f}, j$$

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: $a, b, d, g, k, o, c, e, h, l, f, i, m, \mathbf{i}, n, p, j$. Since a, c, 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 and third sequence in Figure 7 the number of recomputations are similarly three (**a**, **c**, **f**) and zero respectively.

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 node* $u \in V$ *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* $O(2^{n(h+\log h+\bar{b})})$.

[1] All three replay sequences in Figure 7 are DFS-based.

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 hn. 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 h^n 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 to not 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 $2^{n\bar{b}}$ 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 *DFSCost* 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 *DFSCost* will return ∞. This algorithm takes $\theta(n^2)$ 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**.

Algorithm 1 A greedy algorithm that takes as input execution tree T, and a cache size parameter B. It outputs list S of nodes to be cached under the DFS Persistent Root policy.

```
1:  function PRP(T, B)
2:      S ← ∅
3:      f ← True ▷ f is True while greedy is able to extend its solution
4:      r ← root(T)
5:      Set min ← DFSCost(r, S, B, 0) ▷ The function gives us the cost of a DFS-
        based replay sequence for T given a list of nodes S that must be cached when first computed.
6:      while f is True and S ≠ V do
7:          f ← False
8:          for each u ∈ V \ S do
9:              if DFSCost(r, S ∪ {u}, B, 0) < min then
10:                 f ← True ▷ We can extend the solution
11:                 u* ← u ▷ u* is the current best candidate
12:         if f is True then
13:             S ← S ∪ {u*}
14:             min ← DFSCost(r, S, B, 0)
15:     return S
1:  function DFSCost(u, S, B, b) ▷ u is a node of T; b is the cache budget used by
    the path from the root of T to u. Called with u = root(T) and b = 0 this returns the cost of
    computing the entire tree.
2:      if u ∈ S and b + sz_u > B then
3:          return ∞ ▷ Cache size infeasibility detected
4:      c_u ← cost of computing u from nearest ancestor in S
5:      if u has no children then
6:          return c_u
7:      sum ← 0
8:      for each v that is a child of u do
9:          if u ∈ S then
10:             sum ← DFSCost(v, S, B, b + sz_u)
11:         else
12:             sum ← DFSCost(v, S, B, b) + c_u ▷ u is not cached so must be
        recomputed for each child
13:     if u ∈ S then
14:         sum ← sum + c_u ▷ u must be computed once
15:     return sum
```

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_{+u}). In the former the subtree may be forced to recompute 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. $|S_{+u}| > B$, 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).[2]

Algorithm 2 A recursive algorithm the computes for a tree rooted at u the lowest DFS-based replay cost for a given cache S. The child subtrees of u are allowed to choose between executing with S or in addition caching u.

```
1:  function ParentChoice(u, S)
2:      if u is a leaf then
3:          a ← nearest ancestor of u in S
4:          return cost of computing u from a
5:          ▷ If a doesn't exist, return cost of computing u from scratch.
6:      S_{+u} ← S ∪ {u} ▷ S_{+u} is cache that also includes u.
7:      if size of cache S_{+u} > B then
8:          ▷ Caching u is not a option; process its children with S.
9:          P_u ← ∅; P̄_u ← Children(u).
10:         return Σ_{u_i ∈ P̄_u} ParentChoice(u_i, S)
11:     P_u ← ∅, P̄_u ← ∅
12:     ▷ P_u will collect the nodes for which caching parent u is cheaper.
13:     for each u_i ∈ Children(u) do
14:         cost(u_i, S_{+u}) ← ParentChoice(u_i, S_{+u})
15:         cost(u_i, S) ← ParentChoice(u_i, S)
16:         if cost(u_i, S_{+u}) ≤ cost(u_i, S) then
17:             P_u ← P_u ∪ {u_i}
18:         else
19:             P̄_u ← P̄_u ∪ {u_i}
20:     return Σ_{u_i ∈ P_u} cost(u_i, S_{+u}) + Σ_{u_i ∈ P̄_u} cost(u_i, S)
```

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^h. Thus the total time taken is $O(2^h \sum_{u \in V} b_u)$.

[2] 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 $\overline{P}_u = \emptyset$ 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 δ_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) δ and sz 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[3]. 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

[3]Native serializations, *viz.* Pickle, provide only a slight performance benefit (1-2%).

Table 1: Six Real-world Applications

Dataset:	ML1	ML2	ML3	ML4	SC1	SC2
Description	Neural Networks [49, 50]	Stock Prediction [27, 28]	Image Classification [33]	Time-Series Forecast [13]	Gas Market Analysis [2]	Spatial Analysis [45, 46]
Changed parameter	models, hyperparameters, test metrics, datasets, epochs				datasets and input parameters	
Number of versions	25	24	32	36	12	23
Version Length	9 - 13	9	7 - 8	17	18	33
Total (no-cache) replay cost (s)	33390	298	2127	10696	7126	10826
Cell compute range (s)	0.0005 - 1073	0.0003 - 8.5	0.008 - 50	0.01 - 240	0.0003 - 926	0.0002 - 224
Total checkpoint size (GB)	57	37	106	566	13	14
Cell checkpoint size (GB)	0.2 - 1.8	0.2 - 0.38	0.4 - 2	1.3 - 11	0.077 - 0.100	0.040 - 0.050

Table 2: Three Synthetic Datasets

Dataset:	CI	DI	AN
Description:	Compute-intensive	Data-Intensive	Analytical
Max. Branch-out Factor	4	4	4
Max. Version Length	6	6	6
Number of versions	20	20	20
Total (no-cache) replay cost (s)	~20000	~5000	~20000
Cell compute range (s)	100 - 600	100	100 - 600
Total storage size (GB)	~22	~18	~18
Cell checkpoint size (GB)	0.5	0.1 - 0.6	0.1 - 0.6

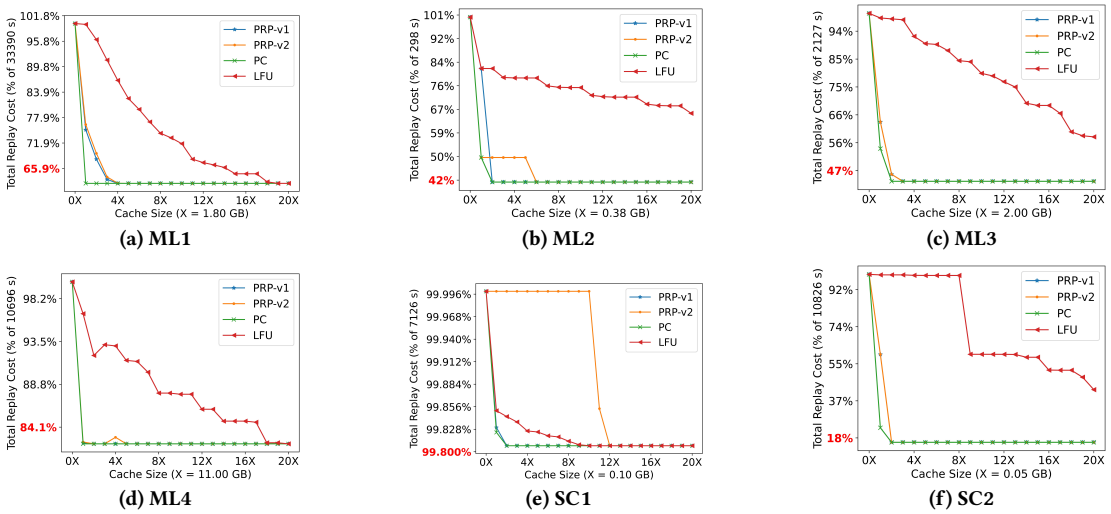


Figure 8: Performance of DFS algorithms on 6 real-world applications. X denotes the size of the largest checkpoint cell as specified in the last row of Table 1. The y-axis is truncated to show finer performance variations between algorithms.

number of iterations in the for-loop. Such a change to create a new version, however, will also re-run the entire training phase again, which will include the training iterations performed in the previous version. Therefore to create a version when the change is to *epochs*, we do not modify the value in-place. Instead we add a new cell. This cell consists of the author-provided training loop but with a incremental range of epochs starting from the last epoch value of the previous cell. This way of modifying the epoch parameter introduces no change to the code and corresponds to incremental training, which is often used in ML to take advantage of previous computations.

Synthetic. To test the sensitivity of our heuristics we randomly generated synthetic execution trees with different costs and sizes. We controlled the tree structure using the following parameters:

max. branch out factor: The maximum number of branches possible at a node. Each branch is constructed with a 50% probability. This leads to trees in which many nodes have a single child. This is what we have observed in real notebooks.

max. version length: The number of cells in each version. In general, the length for each version is different because of the randomization described above.

max. number of versions: The number of leaves in the execution tree generated by using *max. branch out factor* and *max. version length*. Using the above parameters, we generate three synthetic datasets:

- Compute intensive (**CI**): In the **CI** tree, the compute cost (δ) of cells is high and the checkpoint cost (sz) is modest.
- Data intensive (**DI**): In the **DI** tree, the checkpoint cost (sz) of cells is high and compute cost (δ) is modest.
- Analytic (**AN**): In the **AN** tree, compute and checkpoint costs *i.e.,* δ and sz increase with *version length.*

Table 2 presents the total compute time and total storage size as well as the compute and storage ranges per cell.

Baselines. 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 $frequency \times \#\ of\ nodes\ in\ subtree/cell\ size$, 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 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.

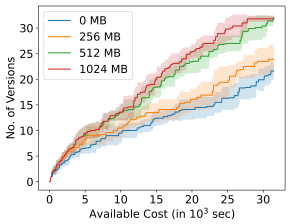


Figure 9: For given cache sizes, number of versions that can be replayed with CHEX in a given amount of time, on the AN dataset.

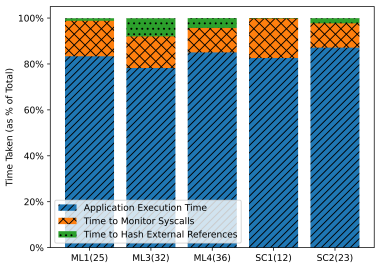


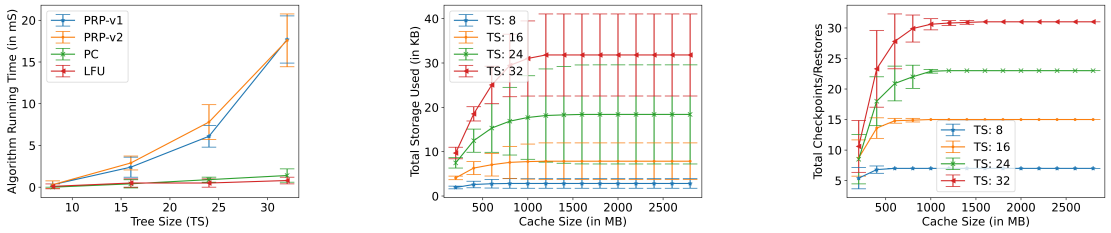
Figure 10: The overhead of auditing δ, sz, g and h in real world applications with > 5 minutes of replay cost.

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 δ, 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.



(a) (b) (c)

Figure 11: Algorithm complexity of AN workloads: (a) running time, (b) storage size for PC, and (c) number of checkpoints/restore-switch for PC

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

common subexpressions may blow up the computation time because a large join may be required. Our scenario appears similar but 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 state can be stored 'as-is'. In SciInc [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.

9 DISCUSSION

We now discuss any assumptions that **CHEX** makes and our results. We assumed that **CHEX** works with REPL cells, but, in general, we do not constrain users like Alice to program with REPL interfaces. If the code is not developed via a REPL interface, **CHEX** preprocesses it into cells, akin to a program developed via a REPL interface, before monitoring. This preprocessing takes care to not split functions or control flows into separate cells. Thus every input program is automatically transformed into an equivalent REPL program and then entered into the **CHEX**.

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.

REFERENCES

[1] 2017. Sciunit. https://sciunit.run/. [Online; accessed 10-Sep-2021].

[2] Bahuisman. 2018. Natural-Gas-Model. https://github.com/bahuisman/NatGasModel. [Online; accessed 10-Dec-2021].

[3] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2013. OPUS: A Lightweight System for Observational Provenance in User Space. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*. 1–4.

[4] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1346–1357.

[5] Mohamed-Slim Bouguerra, Denis Trystram, and Frédéric Wagner. 2012. Complexity analysis of checkpoint scheduling with variable costs. *IEEE Trans. Comput.* 62, 6 (2012), 1269–1275.

[6] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Friere. 2020. Your notebook is not crumby enough, REPLace it. In *Conference on Innovative Data Systems Research (CIDR)*.

[7] Pei Cao and Sandy Irani. 1997. Cost-aware www proxy caching algorithms.. In *USENIX Symposium on Internet Technologies and Systems*. 193–206.

[8] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *SIGMOD'16*. 2085–2088.

[9] Jupyter Community. 2016. C++ implementation of the Jupyter Kernel protocol. https://github.com/jupyter-xeus/xeus.

[10] The CRIU Community. 2019. Checkpoint/Restore In Userspace. https://criu.org/. [Online; accessed 8-Jan-2019].

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Chapter 15.

[12] Shaul Dar, Michael J Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*. 330–341.

[13] Joseph Eddy. 2019. Time-Series Forecasting. https://github.com/JEddy92/TimeSeries_Seq2Seq. [Online; accessed 10-Dec-2021].

[14] Rolando Garcia, Eric Liu, Vikram Sreekanti, Bobby Yan, Anusha Dandamudi, Joseph E Gonzalez, Joseph M Hellerstein, and Koushik Sen. 2020. Hindsight logging for model training. *Proceedings of the VLDB Endowment* 14, 4 (2020), 682–693.

[15] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*. 101–120.

[16] Pradeep Kumar Gunda, Lenin Ravindranath, Chandu Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*. 75–88.

[17] Philip Guo and Dawson Engler. 2011. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 287–297.

[18] Philip J. Guo and Dawson Engler. 2010. Towards Practical Incremental Recomputation for Scientists: An Implementation for the Python Language. In *Proceedings of the 2nd Workshop on the Theory and Practice of Provenance (TAPP'10)*. 6–6.

[19] Philip J. Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. 21–21.

[20] Doug Hakkarinen and Zizhong Chen. 2012. Multilevel diskless checkpointing. *IEEE Trans. Comput.* 62, 4 (2012), 772–783.

[21] Jeong-Hyon Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. 2007. A cooperative, self-configuring high-availability solution for stream processing. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 176–185.

[22] Yves Janin, Cédric Vincent, and Rémi Duraffort. 2014. CARE, the Comprehensive Archiver for Reproducible Execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering (TRUST '14)*. 1–7.

[23] Shudong Jin and Azer Bestavros. 2000. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE, 254–261.

[24] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*. 17–17.

[25] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. 2008. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment* 1, 1 (2008), 574–585.

[26] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–11.

[27] Xinyi Li. 2020. Stock Prediction Using Financial News. https://github.com/AI4Finance-LLC/Financial-News-for-Stock-Prediction-using-DP-LSTM-NIPS-2019. [Online; accessed 5-Dec-2021].

[28] Xinyi Li, Yinchuan Li, Hongyang Yang, Liuqing Yang, and Xiao-Yang Liu. 2019. DP-LSTM: Differential privacy-inspired LSTM for stock prediction using financial news. *33rd Conference on Neural Information Processing Systems (NeurIPS 2019) Workshop on Robust AI in Financial Services: Data, Fairness, Explainability, Trustworthiness, and Privacy* (2019).

[29] Mengmeng Liu, Zachary G Ives, and Boon Thau Loo. 2016. Enabling incremental query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1705–1720.

[30] Robin Lougee-Heimer. 2003. Convex Over and Under ENvelopes for Nonlinear Estimation. https://www.coin-or.org/Couenne/. [Online; accessed 21-July-2021].

[31] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.

[32] Tanu Malik, Randal Burns, and Amitabh Chaudhary. 2005. Bypass caching: Making scientific databases good network citizens. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 94–105.

[33] Nithin Manne. 2020. Image Classification. https://www.kaggle.com/nithinmanne/fashionmnist. [Online; accessed 10-Dec-2021].

[34] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: Multiversion Replay with Ordered Checkpoints. arXiv:2202.08429 [cs.DB]

[35] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[36] Kiran-Kumar Muniswamy-Reddy and David A. Holland. 2009. Causality-based Versioning. *Transactions of Storage* 5, 4 (Dec. 2009), 1–28.

[37] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[38] Yuta Nakamura, Tanu Malik, and Ashish Gehani. 2020. Efficient Provenance Alignment in Reproduced Executions. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*. 6–12.

[39] Bogdan Nicolae and Franck Cappello. 2013. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 155–166.

[40] Tanu Malik Nithin Naga Manne. 2021. The CHEX System. https://bitbucket.org/depauldbgroup/storagevscompute/src/optimal/.

[41] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference*. 377–389.

[42] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using Provenance for Repeatability. In *USENIX Theory and Practice of Provenance (TaPP'13)*. Article 2, 2:1–2:4 pages.

[43] Quan Pham, Tanu Malik, Boris Glavic, and Ian Foster. 2015. LDV: Light-weight database virtualization. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1179–1190.

[44] Quan Pham, Severin Thaler, Tanu Malik, Ian Foster, and Boris Glavic. 2015. Sharing and Reproducing Database Applications. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1988–1991. https://doi.org/10.14778/2824032.2824118

[45] Michael Rilee. 2020. STARE Cookbooks: STARE+Dask-Demo. https://bit.ly/37dlK4B. [Online; accessed 10-Dec-2021].

[46] Michael Rilee, Niklas Griessbaum, Kwo-Sen Kuo, James Frew Frew, and Robert Wolfe. 2020. STARE-based Integrative Analysis of Diverse Data Using Dask Parallel Programming. *Proceedings of ACM SIGSPATIAL conference (SIGSPATIAL'20)* (2020), 417–420.

[47] Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2012. On the complexity of scheduling checkpoints for computational workflows. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 1–6.

[48] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[49] Hojjat Salehinejad. 2020. EPruning (EDropout). https://github.com/sparsifai/epruning. [Online; accessed 10-Dec-2021].

[50] Hojjat Salehinejad and Shahrokh Valaee. 2020. EDropout: Energy-Based Dropout and Pruning of Deep Neural Networks. *arXiv preprint arXiv:2006.04270* (2020), arXiv–2006.

[51] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *International Provenance and Annotation Workshop*. Springer, 155–167.

[52] Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, Remco Vermeulen, Ashish Gehani, Herbert Bos, and Paul Groth. 2016. Trade-Offs in Automatic Provenance Capture *(IPAW 2016)*. Springer-Verlag, 29–41.

[53] Victoria Stodden, Matthew S Krafczyk, and Adhithya Bhaskar. 2018. Enabling the Verification of Computational Results: An Empirical Evaluation of Computational Reproducibility. In *Proceedings of the First International Workshop on Practical*

Reproducible Evaluation of Computer Systems. ACM, 3.

[54] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (2018), 847–872.

[55] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable Research Objects. In *IEEE eScience*. 374–383.

[56] Andrew Youngdahl, Dai Hai Ton That, and Tanu Malik. 2019. SciInc: A Container Runtime for Incremental Recomputation. In *IEEE eScience*. IEEE, 291–300.

[57] Zhihao Yuan, Dai Hai Ton That, Siddhant Kothari, Gabriel Fils, and Tanu Malik. 2018. Utilizing Provenance in Reusable Research Objects. *Informatics* 5, 1 (2018), 14. https://doi.org/10.3390/informatics5010014

[58] K Zielnicki. 2017. Nodebook. https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/ [Online; accessed 10-July-2021].