SciInc: A Container Runtime for Incremental Recomputation

Andrew Youngdahl, Dai Hai Ton That, Tanu Malik
School of Computing
DePaul University, Chicago, IL, USA
Email: {*ayoungdahl, dtonthat, tanu*}*@depaul.edu*

***Abstract*—The conduct of reproducible science improves when computations are portable and verifiable. A container runtime provides an isolated environment for running computations and thus is useful for porting applications on new machines. Current container engines, such as LXC and Docker, however, do not track provenance, which is essential for verifying computations. In this paper, we present SciInc, a container runtime that tracks the provenance of computations during container creation. We show how container engines can use audited provenance data for efficient container replay. SciInc observes inputs to computations, and, if they change, propagates the changes, re-using partially memoized computations and data that are identical across replay and original run. We chose light-weight data structures for storing the provenance trace to maintain the invariant of shareable and portable container runtime. To determine the effectiveness of change propagation and memoization, we compared popular container technology and incremental recomputation methods using published data analysis experiments.**

I. Introduction

Containerization methods are becoming vital for conducting reproducible science. Container engines, such as Linux Containers (LXC) [1] and Docker [2], use capabilities in the Linux kernel to sandbox a computation and its dependencies in a host machine. The container engine then re-runs the sandbox in isolation on target machines without the target environment interfering with the computation. Container engines provide user-friendly methods to create, share, and deploy containers, thus contributing to the conduct of reproducible science [3].

Using a container engine to port a computation on a new machine, however, is only the first step toward conduct of reproducible science. To establish reproducibility, in most cases, the ported computation is repeated at least once, but often repeated several times possibly by changing input parameters and arguments. A container runtime isolates each repetition; however it does not verify if results of repeated computations match (or do not match) with results obtained from original execution of the computation on the host machine.

Application provenance can verify if repeated results match (or do not match) and thus aid in reproducible execution. Current container engines, however, do not audit provenance of computations. System call auditing methods such as *ptrace*, Linux audit, and logging tools can audit provenance in containers, similar to their traditional use for auditing provenance in operating systems [4], [5]. But their naive use increases container creation time. Further, current container runtimes are oblivious of provenance data—they do not distinguish between audited provenance obtained from host versus target machine making it difficult to verify reproducible execution.

In this paper, we audit provenance *during container creation* using auditing mechanisms such as *ptrace*, but restrict the amount of auditing in order not to increase container creation time. We include the audited provenance within a container runtime to verify and optimize further container runs. In particular, the container runtime uses audited provenance to optimize iterative runs, especially as input arguments and parameters change. The container runtime, termed `SciInc`, strictly maintains isolation guarantees, and improves the efficiency of reproducing computations within a container. We make the following contributions in developing `SciInc`:

Auditing Provenance in Containers. We describe an algorithm for generating provenance using system call events audited during container creation. Events audited during container creation are a subset of the events audited in general within an operating system. The resulting provenance is minimal and is represented as a data dependency trace at the granularity of file and processes. The trace consists of both causal dependencies, which arise because of data flow, and version dependencies, which arise if previously used data is repeatedly modified.

Performing Incremental Recomputation Using Provenance Trace. SciInc uses audited provenance to improve efficiency of container replay, especially as input arguments change. We describe a change propagation algorithm that selectively re-executes (and re-records) only those versions of processes or files dependent upon changed input. During the repeat step, versions whose input dependencies have not changed are not re-executed; Instead, previous computation results are retrieved from memoization.

Storing Process and File Versions. We create process versions (aka checkpoints) and file versions in minimal space for a container. Our primary contributions are (i) determining the necessary and sufficient versions as determined by the data dependency graph, and (ii) using userspace techniques to checkpoint and version. We present a provenance-aware checkpoint/restore and a de-duplication layer in userspace based on basic kernel capabilities.

Prototype Implementation. We present a prototype implementation of the container runtime. The runtime is easily deployable within existing container technology such as Docker and within emerging containers for the conduct of reproducible science such as Sciunit [6], [7] and ReproZip [8]. Our runtime is application and programming language agnostic and does not alter or restrict the container environment.

We organize the rest of this paper as follows: Section II presents some background knowledge and assumptions. Section III describes the need for creating versioned data dependency traces for re-execution. Section IV describes how to generate a versioned data dependency trace by observing file use in processes. Section V describes how to maintain memoized versions of files and processes. Section VI presents our experiments showing the minimal overheads using the algorithm. Section VII describes related work. We conclude with a discussion and future work in Section VIII.

II. Background

We provide some background and state some assumptions.
Host and target environment: A host environment refers to the environment (dependencies, configuration, shell settings, etc.) of the machine where a user executes their experiment. A target environment refers to the environment (dependencies, configuration, shell settings, etc.) of the machine where a user intends to run their experiment, primarily for establishing reproducibility of an experiment.
Deterministic experiment: A deterministic experiment consists of computations, data and environment (collectively termed computational artifacts), which when repeated at different points in time, on nominally equal hardware configurations, leads to similar results. The discussion in this paper is limited to deterministic non-distributed experiments.
Experimental results: Results of a deterministic experiment are either performance-based or analysis-based. In performance-based results, the hardware configuration is an implicit input to the computation, and the result depends upon the state of the hardware. Analysis-based results produce the same value on each run. Analysis results can still be non-deterministic and involve memory allocations or pseudo-random number generation, but we assume such virtual memory allocations and seeds is tracked and on repetition uses the same values.
Container creation: `SciInc` assumes a container engine that encapsulates computational artifacts of deterministic experiments to create a container image. Container engines internally use system calls in the Linux kernel to maintain environment isolation of artifacts. The details of these methods is out of scope of this paper, but we describe the necessary user-interaction with the engine required to create container images[1]. In particular, container engines [2], [9] differ in the amount of specification required to create a container image. In this paper, we assume minimal specification in that the user provides only the specification for running an application, which includes the entry program and all input arguments. The container engine determines all the necessary dependencies associated with the application. The engine copies the files into a virtual file system that remains isolated from the target environment on which the container reruns.

[1]We refer to container images and containers as the same

Container replay: A container replay refers to repeating a given deterministic experiment with the exact same inputs under the isolation guarantee. Thus an *analysis* experiment in a container will lead to exactly same results if the computation is executed with the same parametric inputs. If a container runtime determines the experiment is deterministic in which the inputs have not changed, the runtime need not re-execute the experiment. Incremental replay or computation refers to container replay under changed input arguments. If some inputs are the same as previous ones, incremental computation may exclude identical sub-computations, and instead, reuse any transient outputs. For container replay, we assume the user only specifies changed input arguments.

III. The SciInc

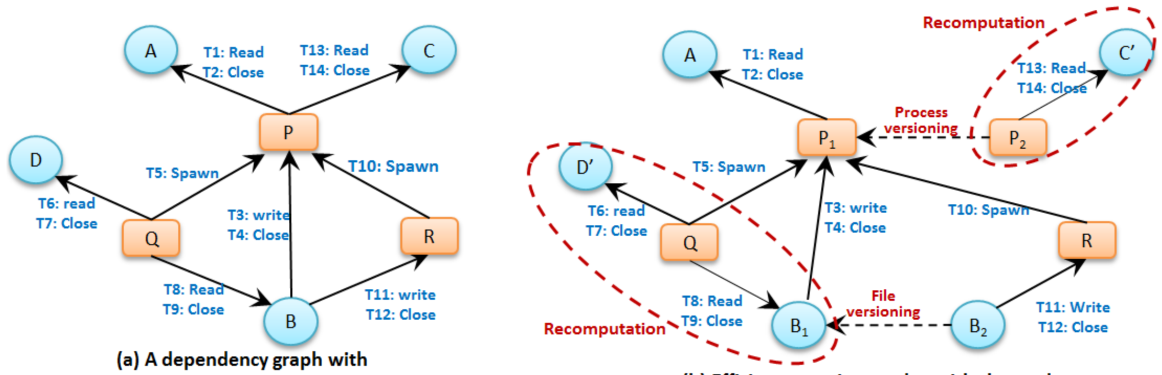
`SciInc` tracks provenance of deterministic experiments during container creation. There are two concerns for tracking provenance during container creation: First how to track data and control flow events within a container engine. Second, the granularity at which provenance should be tracked. Finer granularity (at system call level) improves efficiency of container replay but increases audit overhead. Since we assume minimal user specification, a provenance audit mechanism monitors experiment execution in terms of system calls that were made by the container engine. But we reduce the granularity of tracking by restricting to a few system calls. In particular, only three calls are audited, the system call for opening a file (read/write), the system call for closing a file (read/write), the system call for spawning (fork/exec) a process.

Based on this minimal specification and tracking, we model the execution of a deterministic experiment as a dependency graph representing control and data dependencies in the execution. The nodes of the graph consist of vertices representing files and processes. Edges represent control events, such as a process opens a file for reading or opens a file for writing, or a process spawns another process. We also records time duration for events such as files reads and writes in delta intervals, but process spawns are recorded as instantaneous.

Figure 1 shows an example of graph. The deterministic experiment comprises processes P, Q, and R with input files A, C, and D, and file B as the result of the computation. The figure also shows edge control events recording interactions of a process with files along with a timestamp. For instance, process P reads file A from T_1 to T_2, and P spawns R at time T_{10}.

The recorded provenance is minimal given the three system call events. But is not efficient for container replay. Consider the following scenario in which container is replayed with changed inputs:

- *Inputs A and D remain the same, but the user changes C to C'.* Typically, if a change in input re-executes the entire computation. However, timestamps show that no other graph vertex causally depends on C. If P is versioned at the time C is first opened and read, then `SciInc` can use that version of P to recompute with C'.



(a) A dependency graph with timestamps based on minimal provenance auditing.

(b) Efficient container replay with changed inputs.

Fig. 1: Audited provenance must be verisoned for efficient container replay (Graphs are presented with W3C PROV-DM [10]).

- *Input A and C remain the same, but the user changes D to D'.* Again if an input changes, the entire computation is re-executed. However, D is only causal to Q so partially re-executing Q along with its other inputs should be sufficient. However, Qs other input B has changed since Q used it. Thus `SciInc` either includes Bs lineage (nodes A, C, P, and B), or the version of B read by Q.

Intuitively the scenarios show a process which previously opened a file for writing and closed must be versioned if it reads a new file; thus if the new file changes we can simply recompute from the newer version of the process. Similarly, if a process uses a file before overwriting, the file must be versioned, so that recomputation can use the new version.

The examples above show that an execution trace of a computation consisting of causal dependencies is not sufficient to provide a maximum advantage of incremental recomputation. Process versions and file versions, if appropriately created, improve the efficiency of incremental recomputation. The problem is to determine which nodes must be versioned. Indeed verticies that have been part of a previous causal dependency must be versioned. Thus historical tracking of vertex participation in an *information flow* is necessary. Such information flow must restrict to a minimum the number of events necessary to record the execution of the computation in an application-agnostic way.

We next present an algorithm to create in real-time a dependence graph consisting of both causal and version dependencies based on minimal application events. We then show how to use this dependency graph for container replay with changing inputs.

IV. Versioned Provenance Graph

A. Definitions

We present an algorithm to construct a versioned provenance graph, as shown in Figure 1(b), based on a minimal sequence of system call events used to capture the interaction between processes and files.

For the rest of this paper, we consider a provenance graph $G = \{V, E\}$ composed of a set of vertices $V = \{Activitiy(V_A), Entity(V_E)\}$ and a set of edges $E = \{e = (u, v) \mid u \neq v; u \in V \wedge v \in V\}$. Following the W3C PROV-DM [10], there are two types of vertices: activity vertices, which represent processes, and entity vertices, which represent files in a provenance graph. In our graphs, we use a rectangle to represent an activity and an eclipse to represent an entity.

When an event n occurs, the algorithm establishes an edge $e = (u, v)$ between the vertices. Events are open and close of files for reading and writing and process spawns. An event n labels an edge with a W3C type, such as *wasGeneratedBy*, *used*, *wasDerivedFrom*, and *wasInformedBy*, and informs its direction. An event also associates a time interval to the edge defined as:

Definition 1. ***Association Time Interval.*** *Association time interval $T_e = [t_1, t_2]$ of an edge $e = \{u, v\}$ in a provenance graph G is an interval of time $[t_1, t_2]$ when n occurs ($t_1 \leq t_2$).*

As an example, consider Figure 1(a) where a process P opens a file A for reading. The association time interval $T = [1, 2]$ of the edge between P and A shows that the process P opens file A to read at time $t = 1$ and closes file A at $t = 2$. Similarly, process P spawns Q at time $t = 5$, thus, the association time interval is $T = [5, 5]$. Note that even though process Q runs from $t = 5$ to $t = 9$, the association time interval between P and Q is only $[5, 5]$, since we assume the direct relationship between these two processes occurs instantaneously at $[5, 5]$.

It is possible to use association time intervals to form a relation of causal dependence between file or process vertices.

Definition 2. ***Causal Dependence.*** *Given a provenance graph G, a vertex V_A is causally dependent upon vertex V_B iff:*

- $\exists e : e \in E, e = (V_A, V_B)$, *or*
- $\exists Path : Path = \{e \in E : e_1 ... e_n\}, \{v \in V : v_1 ... v_{n+1}\}$, $\forall i : 2 \leq i \leq n,\ V_A = v_1,\ V_B = v_{n+1}$, $e_{i-1} = (v_{i-1}, v_i),\ e_i = (v_i, v_{i+1})$, $e_{i-1}^{End} > e_i^{Start}$

The Definition 2 shows that V_A causally depends upon V_B if there is a connected path from V_A to V_B where either the path comprises one edge, or each vertex along the path connects edges such that an incoming edges end time is greater than or equal to an outgoing edges start time. As in Figure 1, this dependence is contingent upon a time interval in which the two vertices form an association. For example, in Figure 1(a) vertex Q causally depends on vertex A. There is a path ($e_1 = \{Q, P\}$ and $e_2 = \{P, A\}$) between Q and A. When traversing from Q to A the end time of e_1 is 5 which is greater than e_2's end time of 2. Conversely, Q is not causally dependent upon C. There is a path ($e_1 = \{Q, P\}$, $e_2 = \{P, C\}$) which connects Q to C, however the end time of e_1 (5) is less than the end time (14) of edge e_2.

The existence of non-causal paths, such as between Q and C in Figure 1(a) indicates need for versioning. We now define a version vertex.

Definition 3. ***Version vertex.*** *Given a provenance graph G, a version of vertex v is a vertex v', which must be added to G such that any path through v has a valid causal dependency.*

- $u, w \in V, u \neq v, e_1, e_2 \in E, e_1 = (u, v) \wedge e_2 = (v, w) \rightarrow$ e_1, e_2 *form a causally dependant path.*

By definition, a version vertex creates a version edge between v and v'. For example, in Figure 1(b) process P and file B have been represented with distinct version vertices; P_1, P_2 and B_1, B_2 respectively. A dependent path from Q to C and from Q to R in Figure 1(a) no longer exists in Figure 1(b). Representing the execution as a version graph renders the association time intervals superfluous; causal dependencies can be identified through a simple traversal of the graph without regard to the association time intervals so long as the invariant that all vertices in the graph are version vertices is maintained.

Given an event and time intervals on all edges, identifying if a vertex needs to be versioned is straightforward: those vertices are versioned if there exists an incoming edge that has an end-time less than the start time of any edge on the outgoing path. In other words, if an event creates a new information flow to a vertex which already has information flowing to its descendants, then snapshot the descendants including the vertex as versions, and create new causal dependency from the incoming edge to all new descendant versions.

If the version graphs are being constructed in real-time many edges will not have an end-time. For example, if a process opens a file to read but has not closed the file then there is no end time. Interestingly to create versions, the graph need not store explicit end-time intervals. To determine which vertices to version, all we need to know if information is flowing on an edge. We identify that state with *active* and *inactive* labels on edges. An *active* edge is an edge whose association time interval has a start time, but whose end time is still undefined. In contrast, an *inactive* edge has both a start time and an end time. Further, when an edge becomes inactive (for instance, due to a close event) its status is *marked*. This implies any attempts to reuse this vertex must use a version of the vertex instead of the old version because the old version is on some other causally dependent path.

Figure 2(a) illustrates this real-time construction in which the algorithm connects C to A_1. Marking A_1, B, and D implies previous use in a causally dependent path. Further, there is an information flow from B to A_1, and A_1 to D. If C connects to A_1, all nodes that causally depend on A_1 must be versioned as in Figure 2(b) to preserve the flow of information from B to D via A_1.

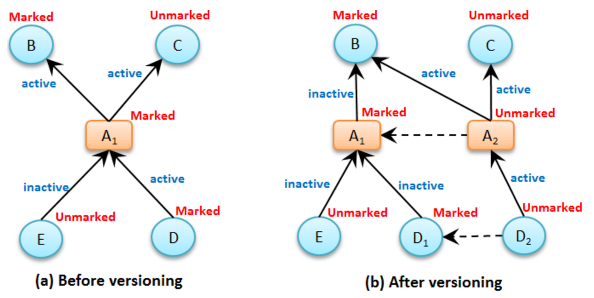


Fig. 2: An example of tracking information flows.

Algorithm 1 and 2 describe the resulting algorithm. In Algorithm 1 we separate the open system call into two events, openRead and openWrite as process and file vertices are swapped and the edge labels are different. For lack of space, we do not provide a line-by-line description of the algorithms. Instead, we refer the reader to the technical report [11]. In this report, we also show that all paths in a version provenance graph generated by Algorithms 1 and 2 are valid causal dependencies.

B. Container replay with changing inputs

Given a versioned graph G generated by Algorithms 1, 2 and a list of input files $\{F\}$, which differ from the original inputs, Algorithm 3 determines which selective processes to re-compute. It does so by selecting all processes that are causally dependent upon $\{F\}$.

Since the input graph, G is a versioned graph, only causally dependent associations exist between vertices (see Definition 3). To determine which processes to re-execute we consider each process u that depends on file $v \in \{F\}$. If this process u is not in P_Φ and it is also not a descendant of any process in P_Φ we add this process u to the list of causally dependent processes P_Φ (lines 4-7). However, before adding process u (Step 2 - line 6), we eliminate all us descendants in P_Φ if any exist (Step 1). Any files dependent upon u are appended to F (Step 3). Once the algorithm determines all dependent processes of altered input files, the processes in P_Φ are re-executed (lines 8-9).

The main idea of using the versioned provenance graph in Algorithm 1 and Algorithm 3 is to balance the costs of versioning (e.g. version graph traversal, process check-pointing, and file versioning), and the gain in incremental recomputation. We note that a process will be versioned if and only if it is marked (i.e., spawns a process or opens/closes an output file) and then opens an input file. Naturally, many

Algorithm 1: Dependence Graph

```
input  : Application Events
output: $G$ a dependence graph
1  $G$ = an empty version graph
2  latest($x$) = a function to return the latest version vertex of $x$
3  foreach (Event ae in ApplicationEvents) do
4      switch $ae.Syscall$ do
5          case openRead($ae.Process$, $ae.File$)
6              connect(latest($ae.File$), latest($ae.Process$))
7          case closeRead($ae.Process$, $ae.File$)
8              disconnect(latest($ae.File$), latest($ae.Process$))
9          case openWrite($ae.Process$, $ae.File$)
10             connect(latest($ae.Process$), latest($ae.File$))
11         case closeWrite($ae.Process$, $ae.File$)
12             disconnect(latest($ae.Process$), latest($ae.File$))
13         case spawn($ae.Parent$, $ae.Child$)
14             connect(latest($ae.Parent$), latest($ae.Child$))
15             disconnect(latest($ae.Parent$), latest($ae.Child$))
16 return $G$
```

processes in an application can qualify these criteria and incur a versioning overhead, but the potential for reuse in incremental recomputation may not be clear. In our evaluation section (Section VI) we experiment with both our versioning algorithm as planned and with the frequency of versioning described in the paragraph above.

V. Implementation

We describe the implementation of our container runtime. The author creates the container by executing the application under the context of the runtime. This audits the application with *ptrace*, which intercepts application system calls. The application events are an input to the algorithm in Section IV, which creates in real-time the versioned provenance graph for that execution. While creating the versioned provenance graph, the runtime also physically generates the process and file versions and stores them in the container. We create process checkpoints using the Linux utility checkpoint/restore-in-user space (CRIU) [12], and use a de-duplication layer to maintain file versions. The synchronized creation of the version graph, and the physical process checkpoints and file versions allows for mapping the conceptual file/process version in the graph with their physical serializations.

When the user re-executes an application, the runtime again observes it using *ptrace*, but since a versioned provenance graph exists for the application, it compares the new execution with the old execution using recomputation algorithm in Section IV-B. The algorithm retrieves necessary process and file versions to perform the recomputation. We now describe how to create and store process and file versions.

Algorithm 2: Connect and disconnect functions

```
1  connect (a, b):
2      $\{M\}$ = reverse traversal of incoming active edges of $G$ starting at $b$ and stopping the traversal of each branch upon encountering an unmarked vertex. Return all encountered marked version vertices.
3      foreach (marked vertex $m$ in $\{M\}$) do
4          $m'$ = new version of $m$
5          $G.V$ += $m'$
6      foreach (marked vertex $m$ in $\{M\}$) do
7          $m'$ = latest($m$)
8          $G.E$ += new inactive edge ($m' \to m$)
9          foreach (outgoing active edge $e$ ($m \to u$) in $G.E$) do
10             label $e$ as inactive
11             $G.E$ += new active edge ($m' \to$ latest($u$) )
12         foreach (incoming active edge $e$ ($v \to m$) in $G.E$) do
13             label $e$ as inactive
14             $G.E$ += new active edge(latest($v$) $\to m'$ )
15     $G.E$ += new active edge (latest($b$) $\to$ latest($a$))
16 disconnect (a, b):
17     $\{UM\}$ = traverse all active outgoing edges of $a$ stopping the traversal of each branch upon encountering a marked vertex. Return encountered unmarked version vertices including $a$
18     foreach ($u$ in $\{UM\}$) do
19         mark($u$)
20     label the edge ($b \to a$) as inactive
```

Algorithm 3: Incremental Recomputation

```
input  : Dependence graph $G$ and list of modified input files $\{F\}$
output: Execute all causal dependent processes
1  $P_\Phi$ = Empty
2  foreach (file $v$ in $\{F\}$) do
3      foreach (edge $e = \{u, v\}$ in $G.E$) do
4          if (process $u$ is not in $P_\Phi$) && ($u$ is not directly or indirectly spawned by any process in $P_\Phi$) then
5              Step 1: If any $p \in P_\Phi$ and $p$ is a descendant of $u$, remove $p$ from $P_\Phi$
6              Step 2: $P_\Phi \longleftarrow u$
7              Step 3: if $u$ or a descendant of $u$ writes to any file $f \in G.V$, add $f$ to $F$
8  foreach (process $p$ in $P_\Phi$) do
9      Execute $p$
```

Process Checkpoints To checkpoint process versions we use functionality provided by Checkpoint and Restore in Userspace (CRIU) [12]. CRIU snapshots the state of a computation (which may consist of multiple processes) and then later restore the computation to a running state. For each piece of state that the kernel records about a process, checkpoint-restore queries the kernel twice: first about the value of the process state, to prepare for dumping the state during a checkpoint, and second to pass that state back to the kernel when the process is restored. CRIU defines this process state recursively, i.e., the process state consists of virtual memory mappings, open files, credentials, timers, process ID of the parent process, and all its children. Technically, a straightforward integration of `SciInc` with CRIU is not possible. CRIU also collects state of a running process by freezing the process using *ptrace*, and copies the state into a file. Since `SciInc` also relies on *ptrace* as a system call interposition method for tracking interactions between files and processes, it creates a circular dependency problem: if a process is being provenance-tracked with *ptrace*, then it cannot be checkpointed also with *ptrace*, as no operating system kernel allows double tracing of a process. Consequently `SciInc` cannot integrate with CRIU API but must incorporate explicit checkpoint/restore functionality as part of the provenance tracking.

To checkpoint, the `SciInc` runtime only snapshots the process, which Algorithm 3 determines to version. Since CRIU assumes process state is recursively defined, and Algorithm 3 only checkpoints a specific process, we have changed the checkpoint method to only checkpoint a specific processes as requested by the runtime. This significantly reduces the overhead of checkpointing and makes it efficient for restoring processes for container replay.

File Versioning Versioning of files requires a method to de-duplicate content. Our current method is based on creating an archive of the container and using content-defined chunking [13] to divide the content of the containers into small chunks identified by a hash value. We compare new chunks to stored chunks, and when-ever matches occur, we replace redundant chunks with small references that point to stored chunks. We use *rsync*'s algorithm for content-defined de-duplication. However, unlike *rsync*, we use a combination of fixed-size and rolling hashes. Once the algorithm computes rolling hashes for a file and detects a different block, it stores the difference itself as a delta to get a specific version of a file. [6] describes a more detailed description of rolling hashes and computed deltas.

VI. EVALUATION

In this section, we describe how the algorithms implemented within `SciInc` are evaluated in comparison with other candidates with three usecases.

To the best of our knowledge, this `SciInc` is the first tool that supports reuse execution or incremental execution at the container level. We evaluate on two criteria: containerization execution (Section VI-A) and incremental execution (Section VI-B). In each section, we select an appropriate competitor with which to make a comparison. To show the effectiveness of containerization in `SciInc` (Section VI-A) we compare with Docker [2]. In VI-B we show the differences in incremental recomputation between `SciInc` and IncPy [14], [15]. IncPy is an enhanced Python interpreter that speeds up script execution times by automatically memoizing function calls.

System settings. Our experiment was conducted on a desktop computer with an Intel Core i7-3770 3.4Ghz (8 cores), 20GB of main memory, 1 TB SATA HDD, and running an Ubuntu 18.04 64-bit operating system. Since our experiment aims to show the efficiency of incremental execution, we conduct all experiment in the same platform to reveal the benefit of incremental execution between executions without any impact of platform differences. This does not mean our proposed method cannot be used to repeat across platforms. Indeed, similar to Docker, `SciInc` can easily be used to repeat its containers in any platform as long as `SciInc` gets installed there (for more details see [9], [6], [7], [19]. Meanwhile, IncPy can only repeat its packages in the same environment.

Usecases. We selected three usecases for our evaluation: (i) Chicago Food Inspections Evaluation (FIE) [16], (ii) the Variable Infiltration Capacity (VIC) [17], and (iii) Incremental Query Execution (IQE) [18]. The detailed descriptions of FIE, VIC and IQE are shown in Table I.

A. *Containerizing effectiveness*

This comparison measures the effectiveness of containerizing an application in terms of storage, creation time (i.e., containerizing or building time), and re-execution time. We examine the differences between `SciInc` and Docker with the selected projects from Table I.

Container size. Figure 3 shows the container sizes in `SciInc` (*Sci. container*) and Docker (*Doc. image*) with different projects having original application sizes (App. Size) shown in Table I.

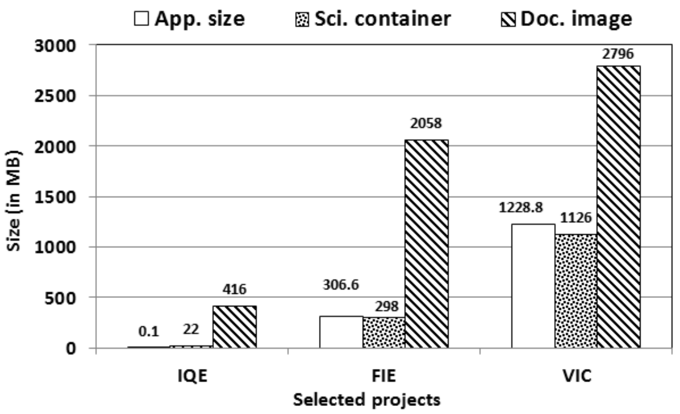


Fig. 3: Container sizes in `SciInc` and Docker with different projects.

As shown, in Figure 3, Docker images are 19X, 7X and 2.5X larger than `SciInc` containers for IQE, FIE and VIC

TABLE I: Usecases descriptions.

	FIE [16]	VIC [17]	IQE [18]
Source code languages	R, Bash	C, C++, Python, C shell script, Fortran	Python
Source code files	29	97	5
Data files	14	11,481	5
Dependency files	659	357	112
Size of all files	306.6 MB	1.2 GB	22 MB
Normal run time	286.756 s	40.259 s	5.226 s

respectively. These larger sizes can be explained by Docker's need to add into the images all information about the linux base kernel as well as libraries, dependencies, and input parameters. `SciInc` only containerizes the digital artifacts (libraries, dependencies or files) that are touched during the execution, significantly reducing the container sizes. Moreover, `SciInc` uses deduplication techniques when storing its containers. As a result, in the case of FIE and VIC, the sizes of `SciInc` containers are even slightly smaller than the original application package sizes (*App. size*, in Figure 3).

Auditing and re-execution times. Figure 4 presents the normal run-times, auditing and re-execution times of different projects with `SciInc` and Docker.

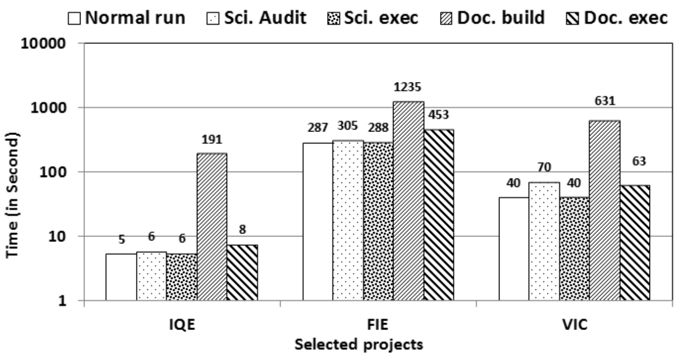


Fig. 4: Normal runs, auditing and re-execution times in `SciInc` and Docker with different projects.

A first observation is that auditing (*Sci. Audit*, in Figure 4) with `SciInc` only takes slightly longer time than a normal run (Normal run, in Figure 4). `SciInc` spends 0.6, 18 and 29 seconds of extra-time to build the containers for IQE (84KB), FIE (307MB) and VIC (1.2GB) respectively. The increase in time during auditing comes from the fact that `SciInc` needs to copy all digital artifacts required for the application execution into the container and to commit the containers into its database; the larger the application, the longer it may take to build a container.

Docker takes longer to build an image than the original run and longer than a `SciInc` containerization (see *Doc. build*, in Figure 4). For example, Docker spends more than 4X longer time to build an image for FIE and 9X longer time to build an image for VIC. As previously mentioned, to build an image, Docker must add all information about the linux kernel, libraries, dependencies, input files and so on. Most of the time, in comparison with `SciInc`, the image built by Docker contains more digital artifacts than it needs for its re-execution. Furthermore, the process of building an image includes installing all required libraries, which can require a long time to complete.

It is also important to emphasize that the image building times that we reported in this experiment are only the time to run the Docker build command (i.e., *docker build*); a step which occurs after all the Docker configurations are set. Normally, it takes more time to create a *Dockerfile* and to verify the image than to run the build command. Creating a *Dockerfile* requires some knowledge about the application to correctly specify all the information about the linux kernel, libraries, input files, source, etc. Any missing materials will result in an error and require extra time to rebuild the image. In our case, the actual time to build and verify Docker images for IQE, FIE and VIC measured in hours. It may take much more time or even be impossible to build a Docker image if one does not have enough knowledge about the application.

In contrast, `SciInc` requires no extra-time for installation or configuration. `SciInc` automatically builds the container when the application runs and guarantees a successful re-execution of its container.

As shown in Figure 4, `SciInc` also outperforms Docker in terms of re-execution time (see *Sci. exec* and *Doc. exec*). In all evaluated cases, re-execution times in `SciInc` are slightly higher than normal runs and smaller than those in Docker. To re-execute applications, Docker will run processes in isolated containers which run on a host machine. The additional virtualization layer increases the time of execution. `SciInc` re-executes the application directly on the host machine with *ptrace* and redirects all system call paths to paths within the special root path of the container; there is no virtualization layer.

B. *Incremental recomputation*

In this section we first present the overhead of automatic checkpoint creation and then discuss the speed-up (gain) which may be realized through the restoration of a checkpointed process.

Since Docker does not support incremental recomputation we did not select Docker as a candidate in this section. Instead we consider IncPy [15], [14], a known tool for incremental execution with Python. However, there is a fundamental difference between IncPy and `SciInc`. IncPy is an enhanced Python interpreter that speeds up script execution times by automatically memoizing function calls, whereas `SciInc` supports incremental recomputation at the container level through checkpoint restore. The experiments in this section aim to show the differences and tradeoffs in incremental recomputation between these methods. We show both the overhead and the gain of two versioning methods of `SciInc` (*Sci. Ver1* and *Sci. Ver2*) and IncPy (*IncPy*).

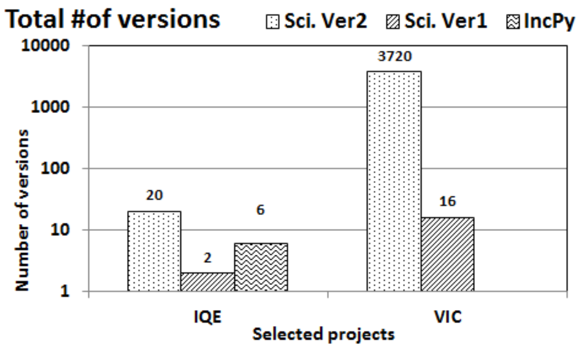


Fig. 5: Total number of versions created with two versioning methods in `SciInc` and IncPy with different projects.

As IncPy is a modification of a python 2.6 interpreter, it is only meaningful for use with python programs. Our IQE example is the only computation in our experiment suite which is comprised solely of python code and will therefore be used for comparison in this section. We note that although IncPy only works with the Python interpreter, it is possible for function level memoization to be ported to other runtimes or compilers (e.g., C, C++, Java), though it is not clear how effective the technique may be in other coding languages (for more details see [15], [14]).

Figure 5 shows the number of functions memoized by IncPy and the number of checkpoint versions created considering two versioning methods in `SciInc`. The second method of `SciInc` (*Sci. Ver2*) creates version whenever a process opens a file, resulting in a larger number of versions. The first method of `SciInc` (*Sci. Ver1*) reduces the number of versions created by only creating version as dictated by Algorithms 1 and 2. Increasing the number of versions created allows for increased granularity of checkpoint restore, but also increases overhead.

Overhead. The total overheads of running applications with versioning methods in `SciInc` and with IncPy in terms of space and time are shown in Figures 6 (a) and (b) [2]; whereas the Figures 6 (c) and (d) present the average costs (in space and time) of creating one image/snapshot with versioning methods in `SciInc` and with IncPy.

As expected, the overheads (in space and time) of *Sci. Ver2* are much larger than those of *Sci. Ver1*. This trend is expected since there are many more versions created in *Sci. Ver2* than in *Sci. Ver1*. The second observation is that, IncPy has very small overheads (in both space and time) in comparison with `SciInc`. IncPy records the bytecodes of a memoized function and its dependencies as well as the function arguments and the return values. This is a smaller number of bytes than recording the bytes which make up the state of the process.

Speed-up/Gain. The motivation behind partial recomputation is ultimately to save time when repeating a computation.

[2]Note that we use logarithmic scale in y axes of those figures for the better readability

TABLE II: Incremental recomputation in `SciInc` and IncPy with IQE.

	No change	**File input changed**	
		master.db	*usercomment.txt*
Sci. Ver2	no repeat (100%)	Step 2 (5%)	Step 5 (95%)
Sci. Ver1	no repeat (100%)	Step 1 (0%)	Step 4 (80%)
IncPy	no repeat (100%)	Step 2 (5%)	Step 2 (5%)

TABLE III: Incremental execution in `SciInc` with VIC.

	No change	**File input changed**		
		stationinfo.txt	*prec.input*	*prcp.inf*
Sci. Ver2	100%	0.39%	95.34%	99.10%
Sci. Ver1	100%	0%	95.34%	99.10%
IncPy	NA	NA	NA	NA

Demonstrating speed-up is hard to do objectively. The authors could present a test case where tremendous gains are achieved; a severe critic could point to cases where no benefit could be realized. To the best of our knowledge, no generalized benchmarks to measure the effectiveness of a recomputaition method exist. Nonetheless, we supply Tables II and III to show the speed-up which may be achieved on our IQE and VIC test cases.

Figure 7 illustrates the workflow of IQE which consists of 6 steps and input files that may be changed at re-execution time. In this test case, *Sci. Ver2* creates checkpoints in each step, while *Sci. Ver1* creates only one checkpoint at step 4. *IncPy* memoizes all the function calls at each step.

Table II shows the different gains made by `SciInc` and IncPy with the IQE test case. If no inputs change in a subsequent run, there is no repeated computation when using any of the methods; 100% of the run-times are skipped. However, if changes were made to *master.db* (step 2) before the re-execution, then both *Sci. Ver2* and *IncPy* repeat the exectution from the Step 2; 5% of the run-times are saved. In contrast, since *Sci. Ver1* has not created any checkpoints before Step 4, it needs to re-execute the container from the beginning (Step 1).

If the change was made in *usercomment.txt* file (Step 5) (see last column in Table II), *Sci. Ver2* re-executes the container at Step 5 and saves around 95% of the original run-time. *Sci. Ver1* executes the container at Step 4 and saves around 80% of run-time. *IncPy* executes the program from Step 2 and saves only 5% of run-time. *IncPy* needs to execute from Step 2, because it memoizes the function calls by monitoring the status of functions' dependencies, and does not capture the internal versions of dependencies.

Similarly, Table III presents the incremental execution in `SciInc` with VIC when the changes were made at *stationinfo.txt*, *prec.input* and *prcp.inf* at 0.39%, 95.34% and 99.10% of run-times respectively. Since *Sci. Ver2* creates a checkpoint in every open file operation, it maximizes the incremental execution gain. Meanwhile, *Sci. Ver1* creates fewer checkpionts and thus may save less execution time than *Sci. Ver2*. For instance, if a change was made to *stationinfo.txt*, then *Sci. Ver2* can save around 0.39% of run-time, while *Sci. Ver1* can

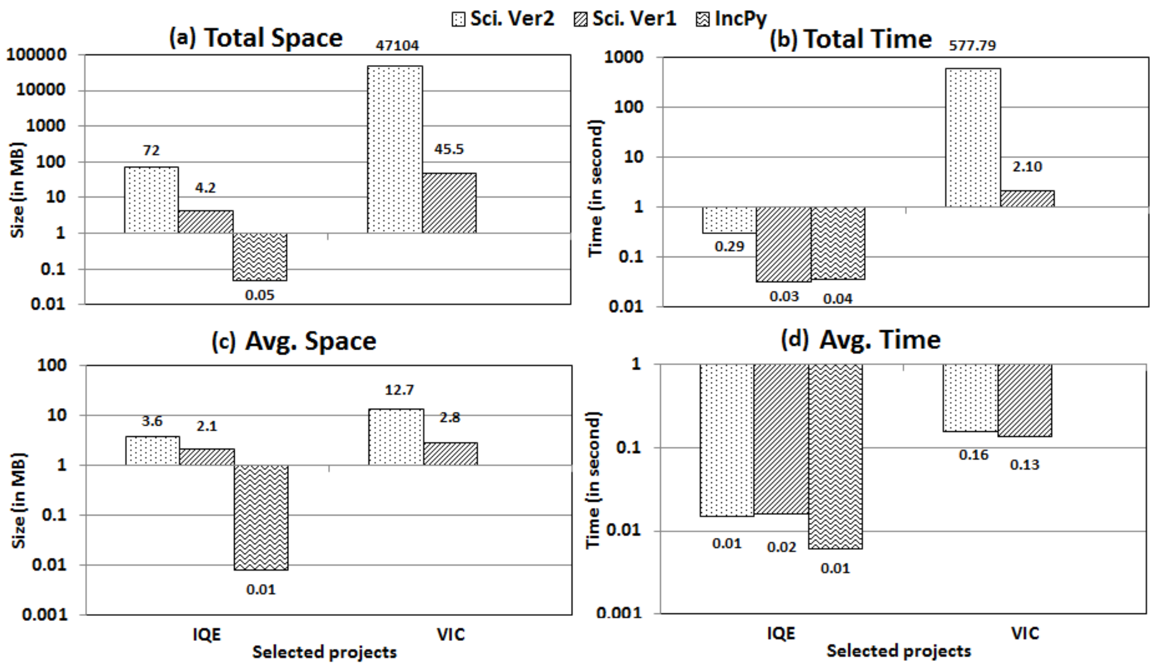


Fig. 6: The overhead of two versioning methods in `SciInc` and IncPy with different projects.

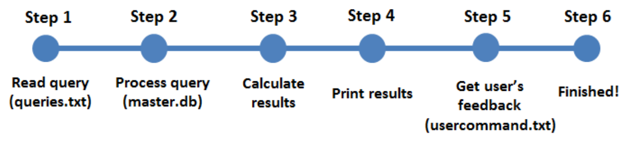


Fig. 7: Processing steps in IQE.

save nothing.

Overall, *Sci. Ver1* and *Sci. Ver2* show the trade-offs between two different granularities of checkpoint creation methods. One can increase gain but suffer larger overhead (*Sci. Ver2*), or achieve less gain while reducing overhead (*Sci. Ver1*). IncPy offers incremental execution at the function level with smaller overhead, but does not provide containerization and is limited to a particular programming language.

VII. Related Work

We review work related to the use of containers for conducting reproducible science, issues with auditing provenance in containers, and incremental recomputation using provenance.

Containers Keahey et al. initially proposed using virtual machines (VMs) to encapsulate large, complicated stacks of scientific software so they can be deployed across supercomputing centers without the need to install each software package individually in every new environment. While VMs are still the unit of sharing and deployment on the cloud, several note that VMs are space inefficient for computational reproducibility [20].

Containers enable light-weight sharing of computational results and improve reproducibility by making computations portable across computing environments with Docker [3]. However, by default Docker does not include capabilities to monitor applications or optimize re-execution of iterative applications.

The UNIX *ptrace* utility can be used to monitor execution of applications. Tools such as CDE [21], ReproZip [22], [8], Sciunit [6], [7] use the UNIX *ptrace* utility to monitor system calls that an application makes. Monitored system calls are used to create a Docker-like container consisting of application binaries, data, and all static and dynamic software dependencies that can be traced during program execution. In this paper, we also have used the UNIX *ptrace* utility to monitor applications, and create a data dependency trace, but we show that a trace without versions is insufficient for incremental re-execution.

Using Provenance for Recomputation [23] considers a computation as a data dependency graph (DDG), and *self-adjusts* the computation for input changes by developing a memoized change propagation algorithm. They also analyze the stability of the re-computed data dependency graph. In their case, the DDG is established at the granularity of function calls. Further, all function calls are assumed to have $O(1)$ execution in time for stability analysis. Auditing at the granularity of function calls can impose a significant overhead. Moreover, processes cannot be assumed to be $O(1)$ execution in time. Thus we do not provide stability guarantees but use the self-adjusting principle to guarantee the maximum possible re-execution given changed inputs.

Practical recomputation is also proposed in provenance tracking systems, such as PASS [5] and SPADE [4]. These systems audit provenance at the granularity of files and processes in an application agnostic way using auditing or *ptrace* mechanisms. Provenance traces are used to either avoid redundant computation [24] or detect errors [25]. In `SciInc` we capture provenance in application agnostic way similar to SPADE and PASS, but differ from these systems in two specific ways: First, unlike these systems we do not audit all system calls. In particular application read/writes are not audited, which impose a heavy application overhead and reduce the advantage of memoization. Avoiding reads and writes changes the recomputation guarantees. Our algorithm errs on the side of increased recomputation in exchange for decreased audit overhead. Second, unlike these systems we consider a versioned provenance graph. Causal relationships must be versioned to determine which versions contributed to recomputation.

Recently several methods have been proposed to improve the performance of iterative applications. Data-intensive frameworks, such as Spark [26], Pregel [27], and Nectar [28], and programming-language frameworks such as IncPy [15], [14] cache intermediate state and incrementally recompute subsequent iterations. There is no known use of such frameworks to improve container replay for conduct of reproducible science.

VIII. Conclusions

Several conferences, including this one, are adopting artifact description/artifact evaluation process. Authors are increasingly adopting containers to encapsulate code, data, and environment and conduct reproducible science. In this paper, we have examined how a container runtime can support reviewer requirements. We presented `SciInc`, a container runtime system, that efficiently supports incremental recomputation of data analysis experiments with the aim of improving their review time. Using a versioned provenance graph we show how incremental recomputation can be achieved within the isolation requirements of the container without modifying programs or introducing new software stacks. Our experiments show `SciInc` has better performance than current off-the-shelf containers used for reproducible science and compiler specific incremental recomputation techniques.

Acknowledgements

This work is supported by National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918.

References

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[2] "Docker," https://www.docker.com/, 2019, [Online; accessed 8-Jan-2019].

[3] C. Boettiger, "An introduction to docker for reproducible research," *SIGOPS Oper. Syst. Rev.*, vol. 49, no. 1, pp. 71–79, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2723872.2723882

[4] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 101–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=2442626.2442634

[5] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267363

[6] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, "Sciunits: Reusable research objects," in *IEEE eScience*, Auckland, New Zealand, 2017.

[7] Z. Yuan, D. H. Ton That, S. Kothari, G. Fils, and T. Malik, "Utilizing provenance in reusable research objects," *Informatics*, vol. 5, no. 1, 2018.

[8] F. Chirigati, R. Rampin, D. Shasha, and J. Freire, "ReproZip: Computational reproducibility with ease," in *SIGMOD'16*, 2016, pp. 2085–2088.

[9] "Sciunit-i," https://sciunit.run/, 2017, [Online; accessed 10-Sep-2017].

[10] W3C, "W3C PROV-DM: The PROV data model," 2013. [Online]. Available: https://www.w3.org/TR/prov-dm/

[11] A. Youngdahl, D. H. Ton That, and T. Malik, "Sciinc technical report," https://github.com/ayoungdahl/SciInc-Technical-Report, 2019, [Online; accessed 20-May-2019].

[12] "Checkpoint/restore in userspace," https://criu.org/, 2019, [Online; accessed 8-Jan-2019].

[13] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.

[14] P. Guo and D. Engler, "Using automatic persistent memoization to facilitate data analysis scripting," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 287–297.

[15] P. J. Guo and D. Engler, "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*, ser. TAPP'10. Berkeley, CA, USA: USENIX Association, 2010.

[16] City of Chicago, "Food Inspection Evaluation," https://chicago.github.io/food-inspections-evaluation/, 2017, [Online; accessed 7-May-2017].

[17] M. M. Billah, J. L. Goodall *et al.*, "Using a data grid to automate data preparation pipelines required for regional-scale hydrologic modeling," *Environmental Modelling & Software*, vol. 78, 2016.

[18] D. DBGroup, "Incremental Query Execution," 2019, [Online; accessed 3-April-2019]. [Online]. Available: https://TonHai@bitbucket.org/TonHai/iqe.git

[19] Q. Pham, T. Malik, and I. Foster, "Using provenance for repeatability," in *TaPP'13*. Berkeley, CA, USA: USENIX Association, 2013, pp. 2:1–2:4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482949.2482952

[20] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[21] P. J. Guo and D. Engler, "CDE: Using system call interposition to automatically create portable software packages," in *USENIX'11*. Berkeley, CA, USA: USENIX Association, 2011.

[22] F. Chirigati, D. Shasha, and J. Freire, "Reprozip: Using provenance to support computational reproducibility," in *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, ser. TaPP '13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1:1–1:4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482949.2482951

[23] U. A. Acar, "Self-adjusting computation: (an overview)," in *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '09. New York, NY, USA: ACM, 2009, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/1480945.1480946

[24] H. Lakhani, R. Tahir, A. Aqil, F. Zaffar, D. Tariq, and A. Gehani, "Optimized rollback and re-computation," in *2013 46th Hawaii International Conference on System Sciences*, Jan 2013, pp. 4930–4937.

[25] K.-K. Muniswamy-Reddy and D. A. Holland, "Causality-based versioning," *Trans. Storage*, vol. 5, no. 4, pp. 13:1–13:28, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1629080.1629083

[26] Spark, "Spark Overview," 2019, [Online; accessed 3-April-2019]. [Online]. Available: https://spark.apache.org/docs/latest/index.html

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146.

[28] P. K. Gunda, L. Ravindranath, C. Thekkath, and and, "Nectar: Automatic management of data and computation in datacenters," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.