Relight: Simple User-Level Checkpointing and Fast-Forward Replay for Distributed Task-Based Systems

ELLIOTT SLAUGHTER✉, SLAC National Accelerator Laboratory, USA
RUPANSHU SOI, Stanford University, USA
MICHAEL BAUER, NVIDIA, USA
ALEX AIKEN, Stanford University, USA

Checkpointing, or periodic saving of program state to storage, is the de facto standard technique used to mitigate risks of nondeterministic bugs, hardware faults, and job wall-time limits in long-running programs. Traditional approaches require users to manually manage the migration of data to and from storage when capturing checkpoints and when resuming execution. However, for task-based programs, where the user has already factored the computation into tasks and the program data into collections, sufficient information is available to automatically capture and resume from checkpoints with minimal code changes. We present Relight, the first framework for automatic, distributed checkpointing of task-based programs that provides an efficient fast-forward replay for full job recovery. On a set of already-optimized benchmarks, we demonstrate that Relight delivers checkpointing performance and scalability comparable to the original, unmodified codes when running on up to 512 nodes of the Piz Daint supercomputer.

ACM Reference Format:
Elliott Slaughter, Rupanshu Soi, Michael Bauer, and Alex Aiken. 2026. Relight: Simple User-Level Checkpointing and Fast-Forward Replay for Distributed Task-Based Systems. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 350 (October 2026), 34 pages. https://doi.org/10.1145/3839482

1 Introduction

Long-running software applications often need to cope with many sources of potential failure: nondeterministic bugs, hardware faults, job wall-time limits, preemption, power outages, etc. While considerable work has been devoted to mitigating partial failures (e.g., hardware failure on a single compute node), the scenario encountered most often in high-performance computing (HPC) today is that long jobs (running days to weeks) must be broken into smaller, separate jobs (often 12–24 hours) to meet job scheduler constraints and share the machine with other users. Users spend significant effort to address this challenge and cannot rely on partial or online recovery solutions. Full job recovery, in which no computational resources persist across job restart, is mandatory.

Checkpointing—saving a program's state so that it can be restored later—is the standard approach for implementing full job recovery. Checkpointing in HPC applications is normally implemented at application level, possibly with the aid of an I/O library [21, 24, 38, 46] or checkpointing framework [8, 14, 26, 34, 37, 50]. Users must decide which data to save, where to save it, and how often to save it as well as performing all necessary synchronization to ensure the consistency of the saved data. Depending on the type of I/O used, clients may be responsible for choosing various details of the underlying storage format as well. Additionally, users must write, test, and maintain an

Authors' Contact Information: Elliott Slaughter, SLAC National Accelerator Laboratory, USA, eslaught@slac.stanford.edu; Rupanshu Soi, Stanford University, USA; Michael Bauer, NVIDIA, USA; Alex Aiken, Stanford University, USA.

ACM 2475-1421/2026/10-ART350
https://doi.org/10.1145/3839482

alternative code path to restore the application state from a checkpoint. A mistake in any of these steps will prevent an application from restarting from its checkpoints, resulting in lost productivity and wasted machine resources. These complications compound as applications grow in complexity, particularly in long-lived codes using multiple, separately developed libraries that must all be harmonized under a single checkpointing scheme while scaling to hundreds or thousands of nodes.

In response, automatic checkpoint-restart frameworks aim to transparently capture checkpoints, thereby relieving users of the burden of implementing manual checkpointing. However, automatic checkpointing introduces additional challenges. Traditional programming models for HPC such as MPI [43] do not capture sufficient information about an application's state to save and reconstruct it; MPI observes only data that is communicated over the network and does not see the application's computation. Existing automatic approaches [2, 20, 25] must stop the entire program, drain any outstanding network traffic, and save the entire heap along with the state of all threads. This approach is inherently blocking and synchronous, and is prohibitively expensive at scale. Furthermore, it is invasive, requiring extensive instrumentation of the underlying runtime systems (e.g., communication layers like MPI and GPU drivers like CUDA) and even other language implementations with which a program interacts (e.g., through third-party libraries). The continued evolution of both networks and architectures in HPC, particularly with the introduction of GPUs, has created insurmountable maintenance burdens for these approaches, which must be extended to support new kinds of hardware and their vendor-supported programming models. No existing automatic approach supports the needs of HPC applications today.

While it is difficult to provide a general-purpose solution for checkpointing that is both simple to use and tractable to implement, we demonstrate a new class of checkpointing methods that can deliver on both fronts for *implicitly parallel task-based* HPC and data analytics programming models [3, 7, 27, 35, 39]. Programs in task-based models are organized into coarse-grained units of computation (*tasks*) and data (*collections*). Implicit parallelism means that while the execution is parallel, the program has sequential semantics. The information encoded in task-based programming models enables our approach where users need only specify the location of checkpoints in the program, but do not need to describe which collections should be captured or how data should be saved. Instead, a dynamic liveness analysis can automatically infer the minimum cut of collection data that must be checkpointed to enable restart from each user-specified checkpoint location. This is important for large applications where it can be challenging to determine the live set accurately and difficult to maintain it as a large code base evolves over time. Additionally, leveraging knowledge of how tasks use collections, our approach can also safely and automatically overlap checkpointing of data with application execution to minimize overhead. Furthermore, since the program is abstracted into tasks, with data movement and synchronization handled by the task-based system, our approach is entirely agnostic to the network and node architecture and can easily compose with other programming systems such as those needed to target GPUs. The combination of both automatically determining the data to be saved and also safely overlapping the saving of that data with further application computation is novel—we know of no approach to checkpointing general parallel and distributed applications that provides this functionality.

In addition to efficient checkpointing, we leverage the structure of task-based programs to facilitate automatic and fast program restart. To resume a program, we must load saved collection data and restore the control state (program counters, local variables, etc.). Control state is difficult to checkpoint without modifying the implementation of the underlying language and runtime system, which we wish to avoid to allow third-party runtimes and languages (e.g., CUDA, HIP) to be used as-is. Rather than attempting to capture and restore this control state, we instead propose to efficiently reconstruct it via *fast-forward replay*, a novel technique in which a program restarts at a checkpoint by running from the beginning, completely eliminating the need for a separate

recovery code path. Fast-forward replay lives up to its name as we skip re-executing any tasks called in the program's original execution up to the checkpoint. Intuitively, eliding task execution on restart is safe because all the side-effects of tasks on collection data are already reflected in the saved checkpoint data that will be reloaded; several edge cases can compromise the safety of fast-forward replay and we discuss the necessary restrictions on programs in Section 3.1. We guarantee safety under these restrictions by giving a semantics for fast-forward replay in a simple tasking language in Section 3 and proving that it correctly reconstructs program state on replay.

To demonstrate our approach, we present Relight, a simple checkpointing framework with fast-forward replay that targets the Legion task-based programming system [7]. Relight is a user-level shim layer that wraps the Legion API, providing an identical interface without requiring any modifications to Legion. Users annotate their programs with the locations where checkpoints should be taken (and how frequently; e.g., every 10th iteration) and link against the Relight library instead of Legion. Developing Relight above a sequential task-based programming model greatly simplifies the implementation, as we can delegate how to asynchronously offload data movement for checkpointing in a safe and performant way to the underlying runtime. We evaluate Relight by running applications on up to 512 nodes of the Piz Daint supercomputer [1]. With reasonable checkpoint intervals, Relight completely overlaps checkpoint I/O with application execution, achieving performance and scalability comparable to the original, uncheckpointed applications. We show checkpoint intervals between 3.67 to 107 seconds, far less than those traditionally employed in HPC, typically in the tens of minutes. In our experiments, the resources used by Relight, in memory and on disk, are constant on a per-node basis while weak scaling, and also constant in the duration of the program. Our approach is the first practical, scalable solution to the problem of whole-job recovery for task-based systems. To our knowledge, there is no widely adopted solution for any programming model that offers such a simple and scalable solution for general HPC applications.

To summarize, we make three novel contributions:

- We present a checkpointing approach for task-based programs that can automatically compute what data needs to be saved and save that data while also overlapping the writing of the checkpoint with further application computation.
- We introduce fast-forward replay, a technique for rapid re-execution that reconstructs the control state of an application without the need for a dedicated recovery code path.
- We demonstrate by example that task-based systems are expressive enough to enable user-level implementation of efficient checkpointing, allowing the checkpointing system to be written in a sequential programming model that simplifies development while enabling an efficient parallel implementation.

The paper is organized as follows: Section 2 presents a motivating example and describes the class of task-based programming models we consider. Section 3 introduces a simple tasking language and discusses restrictions on programs to guarantee safety. We give both a normal and checkpointing semantics, and prove that fast-forward replay in checkpointing semantics is functionally equivalent to the normal semantics. Section 4 describes how Relight wraps a generic task-based system. Section 5 discusses our implementation of Relight for Legion and various optimizations. Section 6 evaluates Relight on a set of applications running on the Piz Daint supercomputer. Section 7 examines related work, Section 8 discusses the limitations of our approach, and Section 9 concludes.

2 Motivation and Task-Based Programming Models

We begin with a motivating example to demonstrate the salient features of the class of task-based programming models we consider and how Relight leverages them. Pennant is a Lagrangian hydrodynamics [19] application written in Regent [42], a language for the task-based Legion

```
1 task continue_simulation(time: double) ... end
2 task calc_dt(dthydro: double) ... end
3 task physics_step_1(zones: region(zone), points: region(point), dt: double)
4   where reads writes(zones, points) do ... end
5 task calc_dthydro(zones: region(zone), dt: double)
6   where reads(zones) do ... end
7 task main()
8   var zones = region(...)
9   var zones_part = partition(disjoint, zones, ...)
10  var points = region(...)
11  var points_part = partition(disjoint, points, ...)
12  var private, shared = points_part[0], points_part[1]
13  var private_part = partition(disjoint, private, ...)
14  var shared_part = partition(disjoint, shared, ...)
15  var ghost_part = partition(aliased, shared, ...)
16  -- initialize regions, declare dt, dthydro, time...
17  while continue_simulation(time) do
18    dt = calc_dt(dthydro) -- future
19    for i = 0, N do -- parallel
20      physics_step_1(zones_part[i], private_part[i], dt)
21    end
22    -- more physics steps...
23    var dthydro = inf
24    for i = 0, N do -- parallel
25      dthydro min= calc_dthydro(zones_part[i], dt) -- future
26    end
27    time += dt -- future
28    __checkpoint()
29  end
30 end
```

Listing 1. Simplified excerpt from the Pennant proxy app showing regions, partitions and timestep loop.

programming model[1]. The main data structure in Pennant is an unstructured, 2D mesh. Figure 10 shows a sample mesh. The mesh consists of cells, called *zones*, and vertices, called *points*. The collections that hold these data structures are created in Listing 1 on lines 8–15. Other variables hold values relevant to the time stepping of the code, such as the current simulation time (lines 18, 27). The program iterates over dynamically-sized time steps (line 17) with function calls dispatched as implicitly asynchronous tasks (lines 17–18, 20, and 25). The system executes these tasks in parallel where possible while still maintaining their sequential semantics.

The term "task" has a variety of meanings in the literature; our focus here is on a specific class of task-based systems. In HPC and systems for large-scale data analytics (such as Spark and Dask), a key property of a task is that its arguments are the only data accessible to that task. There are no global variables or other mechanism allowing a task to name data other than what is passed as a task argument or computed within the task, and there is no mechanism for sending messages or signals of any kind to other tasks beyond calling a task with some arguments. Thus, tasks communicate only through their arguments and results. This design ensures that tasks and data are co-located for execution which is essential for correctness on both distributed memory machines and accelerators with discrete memories as well as enabling much more efficient transfers of bulk data by the system prior to task execution.

Additionally, in most HPC task-based systems, the *effects* of tasks on arguments (read, write, or both) are either declared or easily inferred.[2] Given that tasks communicate only through inputs/outputs and effects are statically known, a task-based runtime can construct an accurate

[1]An implementation of Pennant that directly targets the Legion C++ API is also available, and is considered in our evaluation in Section 6. We present the Regent version here because it is more concise.

[2]In Legion, on which Relight is built, as well as StarPU, effects are explicitly declared.

dependence graph between tasks, showing which tasks produce output that is consumed as the input to other tasks. Such dependence graphs are the basis for the implicit parallelism exploited by these systems: independent tasks (that have no dependencies between them either directly or transitively) can execute in parallel. Depending on the system, these dependence graphs can be constructed statically [11, 18] or dynamically as in the case of Legion, Regent, and others [3, 7, 42, 53].

Regent has two kinds of data types: *regions*, collections for distributed data, and *futures* that hold the results of tasks. Regions are passed by reference, thus `physics_step_1` called on line 20 can modify `zones_part[i]` and `private_part[i]`. Tasks in Regent declare their effects on regions, making it possible to determine, from the call site arguments and task signature, what data is needed and how the task will use it. The underlying task-based runtime system automatically moves data if necessary to the location where the tasks run; the source-level program is not concerned with data movement or coherence. Regions are explicitly *partitioned* so tasks can name subsets of data that they will use; Figure 10 shows the partitions for the sample mesh.

Futures are passed to tasks by value.[3] They typically hold small values such as scalars, and are immutable. There are no effects on futures because they are always read-only. In Regent, futures are implicit[4] and the compiler transparently promotes the return value of a task call, if it has one, to a future. For example, in Listing 1, the return values of the task calls on lines 18 and 25, `dt` and `dhydro`, are implemented as futures, which allows the top-level thread of control to continue executing and launching more parallel work without waiting for the task call to complete. Regent will even promote `time` on line 27 to a future, converting the `+=` to a task call that creates a new future, again to avoid blocking. The only point where the program blocks is line 17, when the value of the `time` future must resolve to determine whether to continue the `while` loop.

While some details of Regent's abstractions are specific to Legion, pass-by-reference collections, partitioning, and pass-by-value futures are found in nearly all HPC and data analytics task-based programming models. Furthermore, most task-based programming models have sequential semantics with parallelism inferred from task data dependencies [3, 7, 9, 11, 18, 35, 39]. Versions of Pennant written in other task-based programming models would be similar, and the same considerations for implementing checkpointing and restart would apply (albeit with different terminology).

To use Relight, usually the only requirement for users is to annotate their program with the locations where checkpoints should be performed. In Regent, this is done with the `__checkpoint()` intrinsic (line 28). The simplicity of this intrinsic is in contrast to most existing checkpointing frameworks that require clients to verbosely specify exactly which data must be saved and how it should be saved. Ensuring only the minimal subset of data is saved is important for maintaining checkpoint efficiency as writing data to disk is costly, but not saving all necessary data will thwart replay. For real applications with lots of data (both futures and regions), determining the minimum required subset of data for a checkpoint can be challenging, both the first time the code is written, as well as during the lifetime (often years) of the application as it is adapted to add and remove functionality. To relieve users of this burden, Relight automatically computes the live set of data at a checkpoint by inspecting the sequence of tasks dispatched along with the usage of futures and regions. In Relight this liveness analysis is performed dynamically (see Section 5) but some task-based programming models could perform it entirely statically [9, 18]. Regardless of how the analysis is performed, checkpoints are guaranteed to save exactly the minimum set of data needed to restart from the specified point in the program. Whenever the program is modified, Relight automatically recomputes what data to save without further input from the user. The only

[3] In the underlying runtime system, futures are represented as a data structure and handles that refer to this data structure that are passed as task arguments. However, because the application can only copy futures or read futures (which will block if the future is unresolved), futures have value semantics at the application level.

[4] Futures are explicit in Legion programs, to which Regent compiles.

limitation is that checkpoints from different versions of the program are incompatible as they store different data structures and may expect different control flow.

To save checkpoint data, Relight leverages the sequential semantics of task-based models by simply inserting data movement operations into the sequence of tasks and then relying on the underlying system to infer implicit parallelism. This approach guarantees that the latency of checkpointing is automatically hidden to the utmost extent possible as the underlying system will discover the maximum amount of available task parallelism and use it to overlap data movement with application-level computations. Relight further uses the partitions of data made by the application to automatically guide the creation of separate files so that parallel I/O can be performed to maximize disk bandwidth. Using application-level partitions ensures that data is stored and can be reloaded in a way that is likely to be reused by the application, obviating the need for expensive reshuffles. All this is performed automatically without any user guidance.

To replay, the user executes the program again from the beginning, the same as when they first ran it. When collections are made that correspond to data stored in the checkpoint, Relight begins asynchronously loading the data for them. In parallel with the data loading, Relight accelerates returning to the checkpoint by eliding task launches while still performing the necessary computations to regenerate the control state of the application. Here we leverage the implicitly parallel nature of task-based systems, which will recreate the same collections and tasks as the original execution using the same code paths and preserve the ordering of data-dependent operations. An analysis shows our approach requires only a restricted form of deterministic execution, which implies fast-forward replay can be used with libraries that are not task-based and/or do not have sequential semantics (e.g. GPU-accelerated libraries) and still correctly restart programs from checkpoints. We further discuss this limited form of determinism in Section 3.1. Once fast-forward execution reaches the checkpoint statement, and all data is loaded, then normal execution resumes.

3 Formalization

In this section we introduce a minimal task-based language to illustrate the restrictions we impose on programs (Section 3.1) and to state and prove the correctness of fast-forward replay (Section 3.2).

Figure 1 defines a simple task-based language with two data types, integers (passed by value) and collections (passed by reference), extended with the operations `checkpoint` and `choice`, which is a placeholder for any non-deterministic program operation. In a program e *where* D we refer to e as the *top-level expression*, the program that calls the tasks in D. A major simplification from a realistic task-based language (besides omitting static types and task effects, which for simplicity we ignore) is that there is no data partitioning or data distribution, which is important for efficient distributed execution but orthogonal to the correctness of fast-forward replay. A second simplification is that we show the correctness of fast-forward replay only for a single checkpoint; our implementation handles multiple checkpoints and also enforces that replay can only be used with successfully saved checkpoints. Our implementation also has a number of optimizations over the basic checkpointing approach treated in our model. These issues are discussed further in Sections 4 and 5.

As is standard in tasking, the language has sequential semantics but the intended implementation is that tasks execute asynchronously whenever possible, limited only by enforcement of data dependencies between tasks [48]. Since Relight is a sequential user-level library incorporated into a broader sequential application that is only implicitly parallelized by the underlying runtime, it suffices to formalize and prove the correctness of fast-forward replay using a sequential semantics.

The sequential operational semantics $\rightarrow_s$ in Figure 2 and the left-hand side of Figure 4 is mostly standard. Judgments have the form:

$$D, E, S, n \vdash e \rightarrow_s v, E', S', m$$

$$
\begin{array}{rcll}
\text{Programs } P & : & \text{e } \textit{where} \text{ D} & \\
\text{Definitions } D & : & [t_1(x_1) = e_1, \ldots, t_n(x_n) = e_n] & \\
\text{Expressions } e & : & i & \mid \texttt{newcollection}(e) \\
& \mid & e[e] & \mid e[e] := e \\
& \mid & \texttt{let } x \texttt{ in } e & \mid x := e \\
& \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e & \mid \texttt{while } e \texttt{ do } e \\
& \mid & e;\ e & \mid t(e) \\
& \mid & \texttt{checkpoint} & \mid \texttt{choice}
\end{array}
$$

Fig. 1. A minimal task-based language with integers, collections, control structures, and task calls.

where D is the environment of task definitions (which is fixed), E is the lexical environment of local variables (a map from variable names to collection references and integer values), and S is the contents of the store (a map from heap locations to collections). The expression `newcollection(`e`)` returns a reference to a newly allocated and initialized collection of size e. While not essential, for convenience we assume the allocation function `next_available` is deterministic in its store argument, which makes all aspects of the semantics deterministic except for `choice`. Collections are accessed by reads $e[e]$ and writes $e[e] := e$ and local variables are also mutable so that there is interesting state to reconstruct on replay—thus judgments produce updated environments E' and stores S' as well as values v, which may be either integers or heap references.

Note that there are no futures in the semantics—since the semantics is sequential there is no asynchrony and no concept of a future. In Section 3.2 we formally define checkpoints using this semantics where we will save the values returned by tasks. Our implementation does the same thing: it is not the future data structure, but the value of resolved futures that are saved at checkpoints.

The semantics is instrumented with counters both in the environment (n in the judgment above) and the conclusion (m in the judgment above). These *value numbers* play no role in the standard semantics given in Figure 2, but will play a role in the semantics of fast-forward replay (Section 3.2).

3.1 Restrictions on Programs

To be used with fast-forward replay programs must adhere to four restrictions: the only side-effects of tasks are on their collection arguments, accesses to elements of collections occur only within tasks and not in the top-level expression e, that e must be *replay deterministic*, which intuitively means that the only source of non-determinism is within tasks and not in the top-level expression, and that checkpointing occurs only in the top-level expression. We discuss each of these in turn.

The requirement that task side-effects are limited to collection arguments is captured directly in the operational semantics of task calls in Figure 4: the only thing in a task's execution environment that could give it access to the store is the value of its task argument. An example of a violation of this requirement would be a task that maintained persistent state outside of the collections that future invocations of the same task could access, like maintaining a counter or other cumulative private state across task calls. In practice, we have seen examples where interoperation with external libraries violates this condition, particularly combining task-based and MPI code where the MPI portion maintains its own internal state. To use Relight in such applications necessitates writing custom checkpointing code for the external library when taking a checkpoint (e.g., code to save and restore the state of the MPI sub-program) while Relight handles the task-based code.

The restriction that the top-level expression e cannot contain store reads and writes is not essential to our approach; there would be no correctness issue in allowing store accesses in e. However, for performance reasons we want to checkpoint collections in bulk by saving and restoring entire

$$\frac{}{D, E_0, S_0, n \vdash i \rightarrow_s i, E_0, S_0, n} \quad \text{[Int]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e \rightarrow_s i, E_1, S_1, m \\ l = \text{next\_available}(S_1) \end{array}}{D, E_0, S_0, n \vdash \texttt{newcollection}(e) \rightarrow_s l, E_1, S_1[(l,1) \leftarrow 0, \ldots, (l,i) \leftarrow 0], m} \quad \text{[New]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s l_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash e_1[e_2] \rightarrow_s S(l_1, v_2), E_2, S_2, p} \quad \text{[Read]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s l_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \\ D, E_2, S_2, p+1 \vdash e_3 \rightarrow_s v_3, E_3, S_3, q \end{array}}{D, E_0, S_0, n \vdash e_1[e_2] := e_3 \rightarrow_s v_3, E_3, S_3[(l_1, v_2) \leftarrow v_3], q} \quad \text{[Write]}$$

$$\frac{D, E_0[x \leftarrow 0], S_0, n+1 \vdash e \rightarrow_s v_1, E_1, S_1, m}{D, E_0, S_0, n \vdash \texttt{let}\ x\ \texttt{in}\ e \rightarrow_s v_1, E_1, S_1, m} \quad \text{[Let]}$$

$$\frac{D, E_0, S_0, n+1 \vdash e \rightarrow_s v_1, E_1, S_1, m}{D, E_0, S_0, n \vdash \texttt{x := e} \rightarrow_s v_1, E_1[x \leftarrow v_1], S_1, m} \quad \text{[Assign]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 \neq 0 \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow_s v_2, E_2, S_2, p} \quad \text{[IfTrue]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 = 0 \\ D, E_1, S_1, m+1 \vdash e_3 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightarrow_s v_2, E_2, S_2, p} \quad \text{[IfFalse]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 \neq 0 \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \\ D, E_2, S_2, p+1 \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 \rightarrow_s v_3, E_3, S_3, q \end{array}}{D, E_0, S_0, n \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 \rightarrow_s v_3, E_3, S_3, q} \quad \text{[WhileTrue]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s i_1, E_1, S_1, m \\ i_1 = 0 \end{array}}{D, E_0, S_0, n \vdash \texttt{while}\ e_1\ \texttt{do}\ e_2 \rightarrow_s 0, E_1, S_1, m} \quad \text{[WhileFalse]}$$

$$\frac{\begin{array}{c} D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s v_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p \end{array}}{D, E_0, S_0, n \vdash e_1; e_2 \rightarrow_s v_2, E_2, S_2, p} \quad \text{[Seq]}$$

$$\frac{}{D, E_0, S_0, n \vdash \texttt{choice} \rightarrow_s 0, E_0, S_0, n} \quad \text{[Choice0]}$$

$$\frac{}{D, E_0, S_0, n \vdash \texttt{choice} \rightarrow_s 1, E_0, S_0, n} \quad \text{[Choice1]}$$

Fig. 2. Standard semantics $\rightarrow_s$ except task calls and checkpointing.

collections rather than individual elements. If we allowed store accesses in e then those individual memory operations would need to be recorded as part of checkpoints to ensure correct fast-forward replay. Disallowing top-level store accesses does not limit the expressiveness of programs, because store operations in the top-level expression can be moved into a task. For example, the program

let a **in** a := **newcollection**(100); a[1] := 42; a[1]

can instead be written

let a **in** a := **newcollection**(100); f(a); g(a) **where** [
f(x) = x[1] := 42,
g(x) = x[1]]

In fact, this second program is an idiomatic use of tasking languages while the first program is not: the access to collection contents should be in asynchronous tasks that can be offloaded to parallel resources, not at the top level. In our experiments in Section 6 all of the programs we experimented with already had all of their collection accesses in tasks and required no changes, but as noted any program can be easily rewritten to satisfy this constraint.

The third restriction is replay determinism. For fast-forward replay to work correctly, Relight must observe the same sequence of environment updates, task launches and collection creations when re-executing a program. Consider the program

let a **in** (**if** choice **then** a := t1(1) **else** a := t2(2)); checkpoint **where** [
t1(x) = ...
t2(x) = ...]

Assume the true branch is taken when this program is executed, so `a` is assigned `t1(1)`. Replay execution may, however, take the false branch of the `if`, so that `a` is assigned `t2(2)`. The replay execution would fail because it executes different task calls than the original execution. We require that non-deterministic operations like `choice` do not occur in the top-level expression.

Figure 3 illustrates several practical scenarios where client programs could similarly invalidate fast forward replay. In Figure 3a, control flow of the top-level expression depends on an unseeded random number generator, leading to different tasks potentially being executed during replay. Similarly, if a branch depends on a timing-dependent value, such as the status of a future, as in Figure 3b, that too can cause a replay to get out of sync with the original execution. In Figure 3c, the order of task launches depends on an unordered set data structure with an iteration order that is not stable across runs (e.g., Python enforces hash randomization across runs for security). For many programs that violate replay determinism, there is again a simple fix: move the non-deterministic behavior into a task body which we demonstrate is safe in Section 4.3. We examined a set of pre-existing task-based programs for our experiments (Section 6) as well as a set of production codes (Section 6.4) and did not find any violations of replay determinism among these.

The final restriction is that checkpoints occur only in the top-level expression. Since tasks only communicate through arguments and results, idiomatic task-based programs tend to be structured with frequent returns to the top level to exchange data between different components/libraries of the program. The programs in our experiments and those we examine in Section 6.4 do not contain long-running subtasks that would benefit from checkpointing, and we leave hierarchical checkpoints to future work. Despite this restriction, Relight still allows subtask bodies to be changed or replaced because it will automatically compute what must be saved at each checkpoint.

Our implementation of Relight enforces the restrictions on top-level collection accesses, replay determinism, and checkpoints in subtasks by signaling an error if they are dynamically detected. Because we want to allow task code to be written in whatever programming system is appropriate

```
// Random number generator
// seeded from external source
if random()<0.5:
  run_algorithm0()
else:
  run_algorithm1()
```

(a) Non-deterministic branch.

```
future := launch_task0()
if future.is_ready():
  value := future.get_value()
  run_task1_inline()
else:
  launch_task1(precondition=future)
```

(b) Timing-dependent branch.

```
// Set ordered by hash values
regions := set()
for i in range(num_regions):
  regions.add(create_region())
for region in regions:
  launch_task(region)
```

(c) Unordered data structure.

Fig. 3. Example violations of replay determinism.

$$\frac{\begin{array}{l}\texttt{def t(x)} = \texttt{e}_2 \in D \\ D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s v_1, E_1, S_1, m \\ D, [x \leftarrow v_1], S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p\end{array}}{D, E_0, S_0, n \vdash t(e_1) \rightarrow_s v_2, E_1, S_2, p} \text{[Fun]} \qquad \frac{\begin{array}{l}\texttt{def t(x)} = \texttt{e}_2 \in D \\ D, C, E_0, \_, n+1 \vdash e_1 \rightarrow_r v_1, E_1, \_, m \\ C(n) = (v_2, p)\end{array}}{D, C, E_0, \_, n \vdash t(e_1) \rightarrow_r v_2, E_1, \_, p} \text{[C-Fun]}$$

$$\frac{}{D, E, S, n \vdash \texttt{checkpoint} \rightarrow_s 0, E, S, n} \text{[Check]} \qquad \frac{}{D, C, E, \_, n \vdash \texttt{checkpoint} \rightarrow_r 0, E, C(n), n} \text{[C-Check]}$$

Fig. 4. Standard $\rightarrow_s$ and checkpointing $\rightarrow_r$ semantics of task calls and checkpointing.

to the application and hardware, there is no general method to automatically enforce the restriction on task side-effects for arbitrary code. This property is enforced statically for programs written in Regent, and dynamically for Legion code written in C++.

3.2 Correctness of Fast Forward Replay

We now turn to precisely defining and proving the correctness of fast-forward replay for our small tasking language. We also give a more precise treatment of replay determinism.

We begin by defining checkpoints C using a collecting semantics based on the value numbers. In a proof P of a judgment $D, E, S, n \vdash e \rightarrow_s v, E', S', m$ we rely on the fact that every sub-judgment of P is labeled by a unique integer n in its assumptions.

Consider a program e *where* D. Let P be a proof that $D, E_0, S_0, n \vdash e \rightarrow_s v, E_1, S_1, m$. The *checkpoint environment* C for P is defined to be:

$$C(i) = \begin{cases} (v', j) & \text{if } D, E, S, i \vdash t(e') \rightarrow_s v', E', S', j \text{ appears as a step in } P \\ S & \text{if } D, E, S, i \vdash \texttt{checkpoint} \rightarrow_s 0, E, S, i \text{ appears as a step in } P \end{cases}$$

The checkpoint environment records the results of task calls (both the value and the final value numbering after each call) and the state of the store at every `checkpoint`.

We now give a precise definition of fast-forward replay. The right-hand side of Figure 4 gives the replay semantics $\rightarrow_r$ of task calls and `checkpoint`. Most replay judgments have the form:

$$D, C, E, \_, n \vdash e \rightarrow_r v, E', \_, m$$

Replay takes place in a checkpoint environment C and, except when the actual `checkpoint` operation is reached, the contents of the store are ignored—replay execution is oblivious to the initial and final stores for most $\rightarrow_r$ operations. In particular:

- The replay rules [C-Int], [C-Let], [C-Assign], [C-IfTrue], [C-IfFalse], [C-WhileTrue], [C-WhileFalse], and [C-Seq] for $\rightarrow_r$ are obtained by replacing stores by _ in the judgments for the corresponding rule in $\rightarrow_s$.
- There are no replay rules for [C-Read], [C-Write], [C-Choice0] and [C-Choice1]. Per Section 3.1 these operations cannot appear in top-level expressions. As noted previously, these operations can always be moved into tasks to create an equivalent program that works with fast-forward replay.

- There is also no replay rule [C-New] for allocating a new collection. Without loss of generality, we assume `newcollection` operations in the top-level expression are rewritten as new tasks that return the handle of a newly allocated collection. Our implementation supports `newcollection` at the top level; the purpose of this rewrite is only to simplify the proof of correctness.
- In the standard semantics $\rightarrow_s$ the rule [Check] is a no-op. The corresponding rule [C-Check] for $\rightarrow_r$ restores the store from the checkpoint environment; [C-Check] is the only $\rightarrow_r$ rule that affects the store.

The final and most interesting semantics is for tasks. The $\rightarrow_r$ rule [C-Fun] does not execute task calls during replay but instead looks up the result of the original $\rightarrow_s$ execution in the checkpoint environment C—this is the fast-forward rule where computation is elided on replay.

We can now precisely state the idea behind fast-forward replay: task results are values (either integers or references) and by saving the results of task calls during normal execution we can skip an enormous amount of computation on replay. Furthermore, while our unoptimized checkpointing semantics saves every task call, in fact only the top-level task calls (those not nested inside of any other task call) are used on replay. However, as the $\rightarrow_r$ rules show, re-execution under $\rightarrow_r$ can only reconstruct the environment E and value v. Thus we separately save the store S at `checkpoint` operations; any intermediate states of the store prior to the checkpoint are not needed.

We now prove that a two-step process of fast-forward replay and separately reloading the checkpointed contents of the store correctly restores program state. The following lemma formalizes the correctness of fast forward replay.

Lemma 3.1 (Fast Forward). Consider a program e *where* D where e does not contain `newcollection`, reads or writes of collections, `choice` or `checkpoint`. Let P be a proof that $D, E_0, S_0, n \vdash e \rightarrow_s v, E_1, S_1, m$ and let C be the checkpoint environment for P. Then:

$$D, C, E_0, \_, n \vdash e \rightarrow_r v, E_1, \_, m$$

Proof. By induction on the structure of P. There are two base cases: integer constants and outermost task calls. For integer constants, assume that $D, E_0, S_0, n \vdash i \rightarrow_s i, E_0, S_0, n$. Then $D, C, E_0, \_, n \vdash i \rightarrow_r i, E_0, \_, n$. For task calls, consider any outermost task call in P, i.e., a task call that is not nested inside of any other task call:

$$\frac{\begin{array}{l}\texttt{def t(x)} = \texttt{e}_2 \in D \\ D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s v_1, E_1, S_1, m \\ D, [x \leftarrow v_1], S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p\end{array}}{D, E_0, S_0, n \vdash t(e_1) \rightarrow_s v_2, E_1, S_2, p}$$

Then using the definition of C we have:

$$\frac{\begin{array}{l}\texttt{def t(x)} = \texttt{e}_2 \in D \\ D, E_0, \_, n+1 \vdash e_1 \rightarrow_r v_1, E_1, \_, m \\ C(n) = (v_2, p)\end{array}}{D, E_0, \_, n \vdash t(e_1) \rightarrow_r v_2, E_1, \_, p}$$

The inductive cases are all similar. We show the case for $e_1; e_2$ to illustrate the pattern. Assume:

$$\frac{\begin{array}{l}D, E_0, S_0, n+1 \vdash e_1 \rightarrow_s v_1, E_1, S_1, m \\ D, E_1, S_1, m+1 \vdash e_2 \rightarrow_s v_2, E_2, S_2, p\end{array}}{D, E_0, S_0, n \vdash e_1; e_2 \rightarrow_s v_2, E_2, S_2, p}$$

Then:

$$D, C, E_0, \_, n+1 \vdash e_1 \rightarrow_r v_1, E_1, \_, m$$

by the induction hypothesis. Then it follows that:

$$D, C, E_1, \_, m + 1 \ \vdash \ e_2 \ \rightarrow_r \ v2, E_2, \_, p$$

also by the induction hypothesis. Applying rule [C-Seq], we have:

$$D, C, E_0, \_, n \ \vdash \ e_1; e_2 \ \rightarrow_r \ v_2, E_2, \_, p$$

□

We can now prove our main result, that fast forward replay combined with checkpointed stores can be used to fully reconstruct a program's state.

Theorem 3.2 (Replay). Without loss of generality, consider a program e; `checkpoint` *where* D where e does not contain `newcollection`, reads or writes of collections, `choice` or `checkpoint`. Let P be a proof that $D, E_0, S_0, n \ \vdash \ e$; `checkpoint` $\rightarrow_s \ 0, E_1, S_1, m$, and let C be the checkpoint environment for P. Then

$$D, C, E_0, S_0, n \ \vdash \ e; \texttt{checkpoint} \ \rightarrow_r \ 0, E_1, S_1, m$$

Proof. By rule [Seq] we have

$$\begin{aligned} D, E_0, S_0, n + 1 \ \vdash \ e \quad &\rightarrow_s \quad v, E_1, S_1, m - 1 \\ D, E_1, S_1, m \ \vdash \ \texttt{checkpoint} \quad &\rightarrow_s \quad 0, E_1, S_1, m \end{aligned}$$

Then, by Lemma 3.1, we have

$$D, C, E_0, S_0, n + 1 \ \vdash \ e \ \rightarrow_r \ v, E_1, \_, m - 1$$

and by the definition of the checkpoint environment C and rule [C-Check] we have

$$D, C, E_1, \_, m \ \vdash \ \texttt{checkpoint} \ \rightarrow_r \ 0, E_1, S_1, m$$

□

We can now give a more technical account of replay determinism and how we can justify non-determinism inside of tasks but not in the top-level expression. An expression e is *replay deterministic* if $D, C, E, \_, n \ \vdash \ e \ \rightarrow_r \ v, E', \_, m$ and $D, C, E, \_, n \ \vdash \ e \ \rightarrow_r \ v', E'', \_, m'$ implies that $v = v'$, $E' = E''$, and $m = m'$. Notice that replay determinism is defined with respect to the replay semantics, not the standard semantics, so the determinism requirement does not extend to task bodies because the replay semantics uses only the memoized results of task executions in the checkpoint environment C.

The proof of Lemma 3.1 relies on e and all of its subexpressions being replay deterministic, as the proof assumes there is only one possible result under $\rightarrow_r$ for any expression with given task definitions D, checkpointing environment C and local variable bindings E. It is easy to show by structural induction that any expression satisfying the hypothesis of Lemma 3.1 is replay deterministic, and as we have already seen, adding `choice` makes it possible to write expressions that are not replay deterministic.

4 User-Level Checkpointing

A desirable aspect of Relight is that it is a *user-level* approach that can be implemented as a library that wraps an existing task-based programming system. By working at the user-level, we are able to avoid making invasive changes into the underlying task-based programming system which could compromise correctness or performance. Additionally, being able to depend on sequential semantics and relying on the task-based runtime for parallelization and distribution greatly simplifies the

implementation and maintenance of Relight. In contrast to non-task-based systems that have limited information about a program and must resort to expensive process introspection, our approach is able to leverage the properties of task-based programming systems to efficiently instrument execution and automate checkpointing and restart with minimal overheads. By wrapping an existing task-based programming interface, we gain the capability to: (1) observe all runtime API calls (to see the created collections and tasks), (2) modify or elide certain operations (e.g., to skip task calls on replay), and (3) track the lifetimes of collections by observing their usage by tasks. Note that for most API calls in different task-based systems we can simply pass calls through to the underlying implementation. However, the crucial ones common to all task-based systems, such as for creating collections and launching tasks, require a custom implementation in Relight.

4.1 Tracking Collection Usage

We introspect all API calls that create either *by-value* or *by-reference* data structures, which for convenience we refer to as collections. By-value and by-reference collections encapsulate the two kinds of semantics that collections have in task-based models: pass-by-value and pass-by-reference. Relight must observe the names of these collections and in some cases wrap them. In the case of by-reference collections, most task-based programming models use *handles* to name these collections (e.g., Legion's regions, StarPU's arrays). For by-reference collections, we do not need to wrap handles as their use is observed whenever they are passed as an argument to a task launch. By-value data collections (e.g., futures that represent the return result of tasks in Legion, Dask and Ray) are wrapped to capture all uses of the value, including when they are read directly. To wrap a by-value collection, we decorate it with a new type and store a reference to the original collection internally. We can then observe any uses of the by-value collection through the decorator class.

Each time a new collection is made, we assign the collection a *value number* from a monotonically increasing counter and record the association. The counters from our formal operational semantics in Section 3 are one way to assign value numbers by giving a unique integer identifier to every program operation. The important property of value numbers is that they are stable across normal and replay execution, and therefore can be used to give the same collection in both executions the same name (using the value number of the collection's allocation operation as the name).

The unoptimized checkpointing strategy presented in Section 3 saves every collection touched by a program, but in practice it is desirable to be more selective. When a collection is created, it is added to a *live set* of collections that tracks all the collections currently in use by the application–*dead* collections do not need to be checkpointed. Collections also have an associated *dirty bit* that records when modifications have been made to the collection since the last checkpoint.

In addition to introspecting all calls involved in the creation of collections, we must also introspect all task launch calls. In some form or another, every task-based programming model names which by-value and by-reference collections are passed as arguments to each task. For example, Legion has task launcher objects that are configured with region and future arguments and passed to a task dispatch API call, while Dask and Ray invoke tasks as specially decorated functions with future arguments. By intercepting all task launches and observing the collections used by tasks, we can determine which collections are written. Collections that are mutated have their dirty bit set. Newly created by-value collections from a task launch, such as futures, are also marked as dirty.

The complementary part of tracking the live set is observing when collections become dead and can be removed from the live set. Most by-reference collection types tend to have explicit deletion API calls that can be intercepted to record when a collection named by a handle is to be destroyed. (E.g., Legion's regions and StarPU's arrays are explicitly destroyed.) For by-value collections that we've explicitly decorated, we can implement a reference counting scheme to know when no more

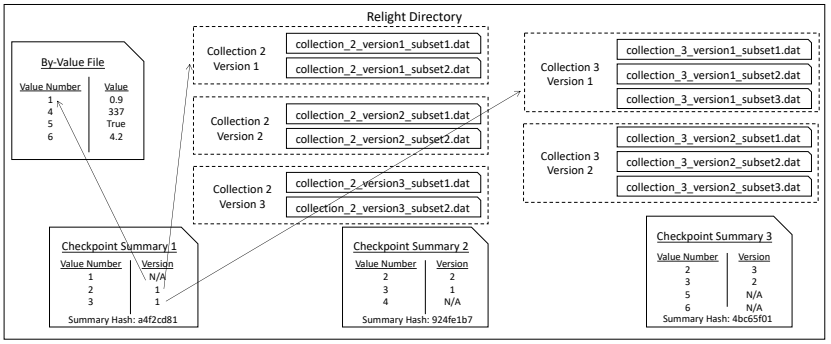


Fig. 5. An example of the structure of a Relight checkpointing directory.

references exist to the value (e.g., futures in Legion, Dask, and Ray) implying the collection is dead and can be removed from the live set.

We now discuss how futures (by-value collections) are handled. First, as just mentioned, any dead futures are discarded. All live futures that are dirty (have not been saved in a previous checkpoint) must be saved. The semantics in Section 3 requires that the state saved at a checkpoint is the state as it would exist in a sequential execution of the program at the point of the checkpoint operation. Thus the checkpoint saves the value (not the future data structure) of any dirty live futures, including waiting on any such futures that are unresolved until the value is available.

An important edge case that must be tracked is when a value is directly used by the top-level task (i.e., the top-level expression in our formal semantics). Consistent with our operational semantics, this can only occur with by-value collections such as futures. An example can be seen on line 17 of Listing 1 where the future result of `continue_simulation` impacts the control flow of the `while` loop. When a value is directly used by the top-level task, we say that it has *escaped*, and we record it in a special set of escaped values. Note that collections that are returned to the top-level task and are passed as arguments to downstream tasks are not considered escaped; only values directly read by the top-level task are considered to have escaped. The set of escaped values must always be recorded as part of the checkpoint, even when they become dead prior to the checkpoint.

To detect violations of replay determinism, Relight also maintains a *summary hash* that compresses the sequence of collection creations/destructions and task launches performed by the top-level task. For each task launch, Relight hashes the name of the task (i.e., its task ID), the value number of each collection used by the task, and how each collection is used by the task (e.g., read-only, read-write) into the summary hash. The operations for collection creation and destruction are also given unique IDs distinct from all other tasks; using these IDs, each creation or destruction of a collection is also incorporated into the summary hash along with the value number of the collection being created or destroyed. The summary hash is therefore a compact representation of the sequence of operations performed by the application that can serve as a unique signature for detecting violations of replay determinism.

4.2 Creating Checkpoints

When Relight is first started, it creates a directory for storing all checkpoint data. The organization of this directory can be seen in Figure 5. A single file is created for storing all by-value data, such as futures, since they tend to be small and are immutable and therefore they only need to be stored once. One or more files will be created for each checkpoint of a by-reference collection using a monotonically incrementing version number associated with the collection. The version number will be embedded in the file name for the collection data so that the system can easily tell which

file(s) correspond to a particular version of a collection. In all cases, collections are stored with their associated value number that was assigned to them when they were created. For by-value collections it is stored directly in the file of by-value data; for by-reference collections it is encoded in the file name. The value number will be essential for associating data with collections upon replay as we discuss in Section 4.3.

Upon reaching a checkpoint statement during normal execution, Relight proceeds to create a *checkpoint summary file* that records the summary hash for verifying replay determinism along with the value numbers of all collections in the live set and their most recent versions. This captures the names of all the collections that would need to be reloaded to resume execution from the checkpoint. The version numbers indicate how to find the data for these collections as shown by the arrows in Figure 5: by-value collections have no version number and can be found in the by-value file, while we can use the value and version numbers of by-reference collections to look for files with a particular name format. The dirty bits in the live set allow Relight to determine which collections need to have their data saved to disk, since they have been written since the last checkpoint (collections without a set dirty bit have already been saved by a prior checkpoint and can be referred to by later checkpoints). After establishing the collections with dirty data that need to be saved, Relight can clear the dirty bits in the live set so future checkpoints can determine which collections have been mutated after this checkpoint.

To ensure maximum overlapping of data movement with the execution of the application, no writing of data to files is performed inline in the top-level task. Instead, we leverage implicit parallelism and the asynchronous nature of task-based systems to hide latency. For each collection with dirty data in the live set, we issue an asynchronous operation to perform the data movement to disk and allow the underlying system to use any task parallelism from the application to overlap data movement with compute. Depending on the base system, asynchronous data movement can take different forms: in systems like Dask and Ray, data movement can be done by launching tasks to write to files, while systems like Legion have dedicated support for issuing asynchronous copies to and from the filesystem [28]. The underlying task-based system is then responsible for ensuring correct sequential semantics of these operations and not allowing later application writes to these collections until the task/copies that save the data are complete. After issuing all asynchronous data movement operations, Relight issues one final asynchronous task that depends on all the data movement operations and writes a *sentinel value* into the checkpoint summary file to detect incomplete checkpoints[5]. The presence of the sentinel value in the checkpoint summary file guarantees that all asynchronous data movement for the checkpoint completed and the checkpoint can be safely used for restart. Our formal semantics of checkpoints in Section 3.2 does not say what happens when saving a checkpoint fails. In our implementation, a checkpoint without a sentinel value is simply not a valid checkpoint; the program must be restarted from a previous successful checkpoint, or from the beginning if all checkpoints lack a sentinel value.

For large by-reference collections, it is often essential that multiple files be created and written to from different nodes to ensure full utilization of disk bandwidth across a parallel filesystem. Each file stores a different subset of the collection data and is written from a different node. Relight can intercept API calls for the creation of partitions of by-reference collections to determine the subsets of a collections that should be saved in each file. For example, a Relight implementation for StarPU would likely create a separate file for each of the tiles of an array to align with how the data is distributed around a multi-node supercomputer. We give a concrete example of how Relight uses Legion partitions to select these subsets when saving by-reference collections in Section 5.2. Figure 5 shows how filenames encode specific subsets of a collection's data.

[5]As an optimization, after writing the sentinel value, this task can also delete prior checkpoints to free up disk space.

The last set of state that must be saved is escaped data. Any time a by-value collection escapes, we save it immediately into the by-value file as it is guaranteed to be needed by any checkpoint to facilitate fast-forward replay. It is safe to save these collections immediately without introducing unnecessary blocking as we know the application already blocked to wait for the value anyway if it has escaped. After saving an escaped by-value collection, its dirty bit is cleared from the live set.

4.3 Fast-Forward Replay

To resume from a checkpoint, the user simply executes the application again. When the Relight library is loaded, it checks for the presence of a Relight checkpointing directory from prior executions. If a directory is found, it looks for the checkpoint summary file with the largest number (i.e., the most recent one) that also contains a sentinel value at the end (recall that the presence of the sentinel value guarantees the checkpoint was fully written). If such a file is not found, execution begins from the start as normal; if such a file is found, it serves as the basis for fast-forwarding back to the checkpoint. Relight counts checkpoints while fast-forwarding until reaching the checkpoint with the same number as the checkpoint summary file, at which point normal execution resumes.

Similar to normal execution, when fast-forwarding to a checkpoint, Relight intercepts calls from the application for creating collections and launching tasks. For each intercepted call, Relight computes the summary hash using an identical procedure as during normal execution. All task launches are safely skipped during fast-forward replay with equivalent semantics to normal execution as proven in Section 3. When a collection is created, Relight checks to see if the collection was live in the checkpoint summary file being used for recovery. To do this we use the value number of the collection because value numbers are stable across replays of the application given the property of replay determinism from Section 3.2. If the value number for the collection is not present in the checkpoint summary file, then we can skip loading the data as the collection isn't live at the restoring checkpoint so loading its data is unnecessary. If it is present, then Relight must load the collection's data. By-value collections are loaded immediately since they are small, while by-reference collections issue asynchronous data movement operations to load the data (similar to how the data was saved). After the asynchronous data movement operations are issued we can resume fast-forward execution without blocking. When by-value collections escape during fast-forward replay, Relight detects this through the decorator object wrapping by-value collections and immediately loads the necessary data from the by-value file. This ensures that the control flow of the top-level task that depends on collection values remains consistent for replay determinism.

Upon reaching the restoration checkpoint, Relight verifies the summary hash matches the one recorded in the checkpoint summary file and reports a replay determinism violation and aborts if they are different[6]. Execution of the application then resumes as normal. Relight adds all the names of the live collections back into the live set, but does not need to block to wait for the asynchronous data movement operations to load data into the collections as the underlying system tracks any dependencies on those collections. When normal execution resumes, Relight still intercepts API calls, but returns to performing the functionality described in Section 4.1 to save future checkpoints. Later checkpoints saved during an execution, even after a fast-forward replay, will further extend the last valid checkpoint and guarantee forward progress of the application.

5 Implementation

We now describe a specific implementation of Relight as a library for the Legion runtime system [7]. Our implementation aims to guarantee three important properties:

[6]Hash collisions can still occur leading to very unlikely false negatives, but all hash differences will be real violations meaning there are no false positives.

(1) the space and time complexity is constant in the number of nodes (i.e., it is scalable),
(2) the space complexity is constant in the number of operations (e.g., tasks) launched, for many programs (including all of our experiments), and
(3) it overlaps I/O and computation to the maximum degree possible.

Legion is most often run on large supercomputers, with heterogeneous accelerators like GPUs, which makes achieving these goals a challenging proposition.

5.1 Sharding Metadata

Legion has several kinds of collections that must be tracked by Relight. Regions and futures were discussed in Section 2; fortunately the metadata required for storing them is independent of machine scale. For example, while the amount of data stored in a region may grow with the size of the machine (when weak scaling), storing the name of a region (of any size) in the live set is constant overhead. However, Legion has one kind of collection that does not have this property: partitions.

Partitions describe logical subsets of regions, called *subregions* (initially discussed in Section 2). Relight must capture partition information for subregions in case their root region is in the live set. In this case, Relight saves the subregions' partition metadata so that partitions can be reconstituted during a fast-forward replay. Many partitions in Legion programs contain $O(N)$ subregions in the number of nodes in the machine, which would create scalability problems if we were to store all the partition metadata for every subregion on a single node as the number of nodes scales up.

To manage partition metadata in a scalable fashion, we rely on Legion's *control replication* [4]. Control replication executes a copy of the top-level task, called a *shard*, on each node while ensuring that all shards collectively execute as a single logical task. Each shard issues the same sequence of tasks in the same order, and since tasks are dispatched to Legion in an $O(1)$ format [44], they are scalable. Note that control replication is one of the implicit parallelization features provided by Legion and not something that user-level code like Relight must control directly.

Relight is aware of Legion's control replication and can detect which shard it is running within. When Relight detects that the top-level task is being executed with control replication, it also shards its collection of the subregion metadata for partitions, with each shard capturing the partitioning metadata for an independent set of subregions of each partition. Upon reaching a checkpoint, each shard will save out its subset of the partitioning metadata into a distinct file. Consequently, each shard performs only $1/N$ of the work for capturing and saving data for each partition. During replay, each shard loads its respective subset of the partitioning metadata, and leverages Legion's collective partitioning interfaces to reconstruct partitions for live regions that are being reloaded.

5.2 Computing Covering Sets

Our implementation of Relight for Legion must checkpoint all region data in a manner that scales with the problem size. Data parallelism in Legion is expressed by partitioning regions into subregions and launching tasks that operate on those subregions. Therefore, we can observe the partitions that are created and used to determine the appropriate subregions to save to disk.

For a large class of task-based programming models [3, 27, 35, 39], determining how to save distributed collections is straightforward because partitions are always *disjoint* (subsets never overlap) and *complete* (every element must be contained in at least one subset), and each collection can be partitioned exactly once. Thus it is sufficient to traverse the tree of partitions until sufficient subsets are discovered to facilitate enough parallel I/O to fully use disk bandwidth. However, partitions in Legion are more flexible: regions can be partitioned multiple ways (introducing aliasing between subregions of different partitions of a given region) and partitions themselves need not be disjoint nor complete. This makes selecting a set of partitions to save more challenging.

To be correct, we only need to guarantee that every element of every region is saved, which could be done by simply saving all of the root (unpartitioned) regions. However, to efficiently save region data to disk, we need a *covering set* of regions and partitions that maximizes I/O parallelism while minimizing wasted space by avoiding duplicate data. We prefer to save partitions recently used in the program, so that the data is already resident in memory in the regions we request at the point of saving the data to disk, and to avoid triggering communication. A full discussion of the algorithm we use for computing covering sets is in Appendix A.

5.3 Overlapping I/O and Compute

Writing data to disk is both slow and unpredictable: disk access times on shared networked filesystems vary with user load. To hide these unpredictable latencies and ensure that the application continues running while data is written to disk, Relight uses a two-stage copy approach. Region data is first copied into a separate *instance* in a non-disk memory before it is eventually written to disk. (On machines with GPUs, such as in our experiments in Section 6, we use *zero-copy* memory for this purpose: i.e., CPU memory pinned with the GPU driver.) This additional copy, though it takes time and memory, is much faster than writing to disk. Most importantly, it enables the application to run asynchronously with respect to the disk writes that save checkpoint data. Without this copy, Legion must delay application tasks from running to wait for disk writes to complete, and thereby avoid the write-after-read hazard.

In practice, we found that the variance in filesystem access latency was high enough that this basic approach was not sufficient. To further hide latencies, Relight uses a rotating set of N instances. This permits up to N checkpoints to write to disk simultaneously, avoiding situations where a slow disk write blocks subsequent checkpoints and thereby stalls the application. Our experiments used $N = 3$, which we found sufficient to hide disk writes on Piz Daint and offers a reasonable trade-off in memory usage and checkpointing performance. This feature is supported with a custom *mapper* [7] that handles the mapping of tasks and copies generated by Relight and can precisely control which instances each region is mapped to for every operation. The Relight mapper works independently of any application mappers and is therefore transparent to the end user unless they choose to experiment with additional values of N for unusual filesystems.

5.4 Saving Futures

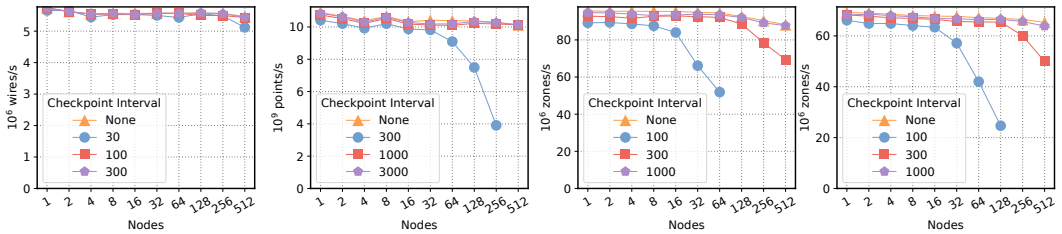
Idiomatic Legion programs tend to minimize blocking to wait for future values because this can delay the launching of additional parallel tasks. However, there are certain situations where blocking is necessary, such as the convergence check seen in Listing 1 line 17. Programs that often wait for future values can create large numbers of escaped futures that need to be saved in the by-value file that stores that data for escaped futures (see Section 4.2).

While the cost of saving such futures is low (about 16 bytes per future), there may be millions of escaped futures that must be saved. We can eliminate a significant amount of storage overhead by observing that the program tends to compute the same value over and over again for these escaped futures (e.g., `true` until the while loop ends). To take advantage of this observation, we have implemented a form of run-length encoding when saving future data in the by-value file. We group together ranges of contiguous value numbers that all share the same value in the by-value file. In our experiments, this often reduced the number of future values saved to a constant number, independent of the number of dynamic futures created during execution of the program.

Finally, a checkpoint requires the values of any unresolved live futures. Our implementation blocks the application until live futures have resolved. Crucially, it does so *after* launching all of the large, asynchronous data copies that transfer region contents first from GPU to CPU memory and then to disk. Thus in our experiments, the time spent blocked on futures is hidden. Appendix B

Application	Language	Checkpoint Size	Timestep	Intensity	Total Futures	Unresolved
Circuit	Regent	0.020 GiB	34.8 ms	0.57 GiB/s	1	0
Stencil	Regent	3.35 GiB	20.7 ms	162 GiB/s	1	0
Pennant	Regent	9.20 GiB	77.1 ms	119 GiB/s	6	1
Pennant	C++	9.97 GiB	107 ms	93.5 GiB/s	7	4

Table 1. Left: Checkpoint size per node, timestep duration (without checkpointing) and intensity (theoretical I/O if checkpointing every timestep with no overhead). Right: total vs. unresolved futures in checkpoint.



(a) Circuit (b) Stencil (c) Pennant (Regent) (d) Pennant (C++)

Fig. 6. Weak scaling vs. checkpoint interval. Higher is better.

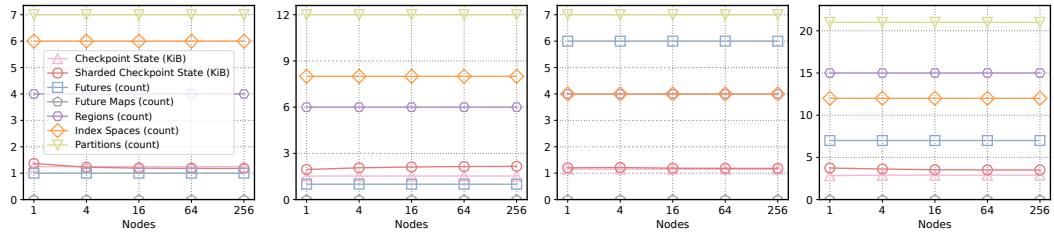
examines a profile from one of our application runs to prove that checkpoints are asynchronous and the cost is dominated by the initial copy from GPU to CPU memory.

6 Evaluation

We evaluate Relight on up to 512 nodes of the Piz Daint supercomputer [1]. Piz Daint is a heterogeneous supercomputer with one Intel Xeon E5-2690 v3 CPU (12 cores) and one NVIDIA Tesla P100 per node. The CPU has 64 GB of RAM, and the GPU has 16GB of HBM2 memory. The supercomputer uses a HPE Cray Aries network. Storage is provided by a Lustre parallel filesystem with a total capacity of 8.8 PB. We used a recent version of Legion and built all software with GCC 10.3 and CUDA 11.2.0. Legion was configured to use GASNet-EX 2022.9.2 as its networking layer [10]. Regent was built with LLVM 13.0 as its code generation backend.

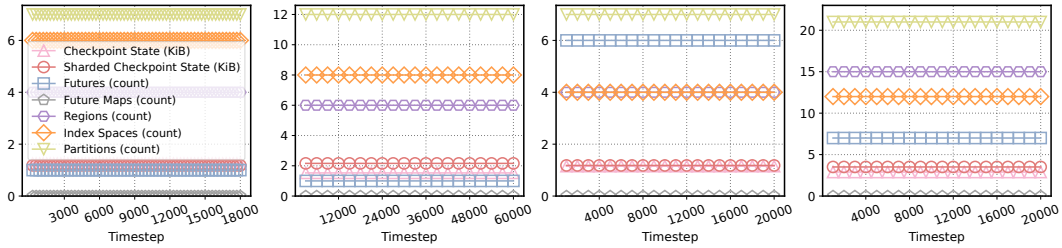
For our evaluation, we use a set of already-tuned benchmarks for Legion and Regent. These include Circuit, a simulation of an electrical circuit on an unstructured graph, Stencil, an implementation of PRK Stencil [51] that uses a star-shaped stencil on a 2D grid, and Pennant, described in Section 2. For Pennant, we used two implementations, in Regent and C++, respectively. Though the codes implement the same physics, they were developed independently by different authors, and thus make different choices about partitioning, mesh numbering, etc., and have slightly different performance and memory usage. All applications are configured similarly to past publications of these benchmarks [6], except for Stencil, where we had to reduce the problem size to keep the total running time within our limited compute-time allocation on Piz Daint. Using the publicly available artifact [5] and artifact description appendix of [6], we confirmed that our benchmarks perform the same or better than previously reported results in our test configurations.

We consider three classes of experiments. First, we perform weak scaling while varying the checkpointing interval, and compare these results with running with checkpointing disabled entirely. Second, we consider the resource usage associated with checkpoint metadata, as measured in the number of bytes saved to disk, and the number of distinct objects represented in those checkpoint files. We scale these tests with both node count and running time (i.e., number of application timesteps) to determine how sensitive these numbers are to those factors. Third, we measure the replay time, varying the number of timesteps to determine how quickly we can replay a large number of timesteps when restarting from a checkpoint.



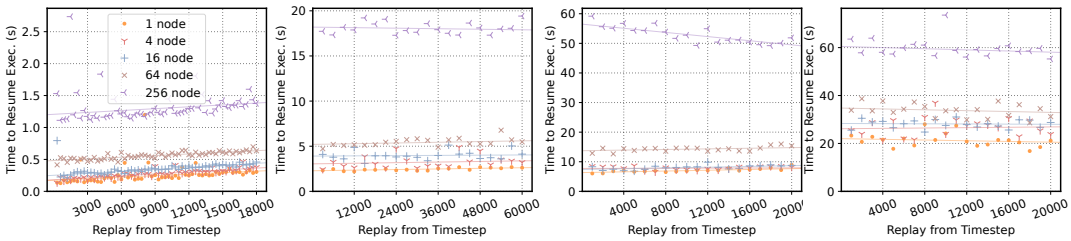
(a) Circuit (b) Stencil (c) Pennant (Regent) (d) Pennant (C++)

Fig. 7. Resource usage vs. node count. Lower is better.



(a) Circuit (b) Stencil (c) Pennant (Regent) (d) Pennant (C++)

Fig. 8. Resource usage vs. timestep. Lower is better.



(a) Circuit (b) Stencil (c) Pennant (Regent) (d) Pennant (C++)

Fig. 9. Replay time vs. timestep. Lower is better.

6.1 Weak Scaling

Figure 6 shows weak scaling performance for the four applications. For each application, we ran with checkpointing disabled, and compared to checkpointing enabled while varying the checkpoint interval. These intervals were determined empirically, by starting at a short interval (frequent checkpoints) and lengthening it until the overhead disappeared. Note that once I/O overhead is overlapped with compute, the I/O cost is fully hidden. The only cost to the application is a single copy of the data from the GPU framebuffer to *zero-copy* memory (CPU memory pinned with the GPU driver); see Appendix B for a representative profile. We found that invoking checkpoints at the top of the main iteration loop every nth iteration works well.

By comparing the performance against a configuration with checkpointing disabled, we demonstrate two important points. First, Relight itself has virtually no baseline overhead. This is in contrast to existing automatic approaches that impose overheads as high as 5% even when no checkpoints are performed at all [20]. Running with Relight is essentially free: for the three Regent benchmarks, at 512 nodes, each application achieved (with the longest reported checkpoint interval) within 0.5% of the performance with checkpointing entirely disabled, while C++ Pennant achieved within 2%. Thus Relight's overhead for tracking value numbers, covering sets, etc. is minimal. Second, for each

application is it possible to find a checkpoint interval such that the cost of checkpointing is entirely hidden, while still performing checkpoints far more frequently than required in actual practice for these applications. In contrast the leading automatic approach [20] reports checkpointing times up to 30 seconds, and must stop the entire application to perform the checkpoint, precluding overlapping computation with I/O. Our checkpointing intervals, though empirically determined, can be explained by the computational properties of the respective applications. Table 1 shows, for each application, the total checkpoint size per node (including both data and metadata). (These are weak scaling experiments, so the per-node checkpoint size remains constant for all node counts.) The table also reports the duration of a timestep, without checkpointing, at one node. Thus for each application we can compute *intensity*, or the theoretical I/O that would be required if we were to checkpoint every timestep with no overhead.

Intensity correlates with the empirical checkpointing intervals we determined. Circuit is the least intense because it has a small working set relative to compute performed. There is less data to write to disk, and it is easier to hide this overhead (which means we can write checkpoints more frequently). Pennant is in the middle: the checkpoints are the largest, but the timestep duration is also high, so it balances out. Empirically we found it necessary to use longer checkpoint intervals to hide the overheads in Pennant. And finally, Stencil has the highest intensity despite a somewhat smaller checkpoint than Pennant, because its kernels are lightweight and run quickly. Of the applications, Stencil must use the longest checkpoint interval to completely hide overhead.

The right side of Table 1 reports total and unresolved futures in each checkpoint. Circuit and Stencil do not use futures in the main loop and thus the unresolved count is zero; yet they straddle the gamut of checkpoint intervals. Pennant contains unresolved futures but sits in the middle. We note that because copies are issued before blocking on futures, intensity is the better predictor of checkpoint interval. See Appendix B for additional profile-based evidence.

Relight is able to hide the overhead of checkpointing, despite running frequent checkpoints. In Circuit, at 512 nodes and a checkpoint interval of once every 100 timesteps, we achieved a checkpoint rate of once every 1.17 seconds with only modest overhead. When checkpointing every 300 timesteps, or 3.67 seconds, this overhead disappeared. With Stencil at 512 nodes and checkpointing every 1000 timesteps, we hit a rate of a checkpoint every 22.2 seconds. And with Pennant, the largest of the three apps by checkpoint size, we hit a rate of every 84.1 seconds (in Regent) or 107 seconds (in C++) at 512 nodes and checkpointing every 1000 timesteps. Relight achieves these checkpointing rates while maintaining competitive performance at 512 nodes (0.5% or less for Regent benchmarks and 2% for C++ Pennant). By contrast we note that at the checkpointing rates tested here, and with the overheads reported in [20], a double-digit percentage of application time would be spent checkpointing. In practice, most applications running on HPC machines today expect checkpoint intervals on the order of tens of minutes, and Relight is easily able to deliver.

6.2 Resource Usage

Figures 7 and 8 show Relight's resource usage for each application vs. node count and timestep, respectively. For this experiment, we configured the applications to run much longer than needed to measure steady state performance alone. Circuit ran for 18,000 timesteps (about 11 minutes), Stencil ran for 60,000 timesteps (about 22 minutes) and Pennant ran for 20,000 timesteps (27 minutes in Regent, 37 minutes in C++). This experiment demonstrates that all of the applications are capable of running for extended durations, and allows us to compare a meaningful number of checkpoints.

Figure 7 shows that resource usage is flat with node count. The size of the checkpoint state is flat, as is the sharded checkpoint state per node. All specific resources tracked within these checkpoints are also flat. Thus, each node must track (and write to disk) a fixed amount of metadata.

Figure 8 demonstrates that resource usage is also flat with increasing numbers of timesteps. For these applications, we are not limited in the number of timesteps we can capture with this approach; even for long-running simulations, we never hit a point where an application would be unable to capture checkpoints due to space or time constraints.

6.3 Replay Time

To measure the impact of fast-forward replay, we took each checkpoint generated in Section 6.2 and measured the time to replay the application, starting from the call to enable checkpointing (at the beginning of the program) to the completion of the final checkpoint call, just prior to resuming normal execution. (This includes the time to complete all I/O from reading checkpoint data from disk.) Figure 9 shows the result, plotted for each tested node count, along with trendlines. Because of occasional interference from other users on the filesystem, we excluded the single highest and single lowest data points when computing each trendline.

The practical impact of replaying many iterations is negligible, even when running tens of thousands of timesteps. Three of the four applications we tested (Stencil and both Pennants) measure a negative slope on their trendlines at 256 nodes. Although the cost of fast-forward replay is certainly not negative, this shows that the cost is dominated by I/O, which is subject to interference from other jobs on the machine and thus relatively noisy. Even excluding negatively-sloped trendlines, Relight is able to replay about 100,000 timesteps per second (and often more) across our experiments. Thus it would be practical to replay applications even if they ran for many millions of timesteps with this approach. The dominant cost in all of our applications was the I/O to read checkpoint data back from disk, and in practice we note that the restart times measured here are comparable to or better than those previously reported for automatic approaches [20].

6.4 Programming Restrictions

In this section we consider the impact of the four restrictions on programs discussed in Section 3.1: tasks have no side-effects except on collection arguments, access to collections occurs only inside tasks, replay determinism, and checkpoint operations occur only at the top level, not in tasks. In addition to the four benchmarks already discussed, we also examined the source code for four additional production applications: Soleil-X [47] (written in Regent), two versions of the HTR solver [17], written in Regent and C++, respectively, and S3D-Regent [49], written in a combination of Regent, C++, CUDA, and Fortran. The applications have 27K (Soleil-X), 70K (HTR Regent), 120K (HTR C++), and 900K (S3D-Regent) lines of code, respectively.

All of the task bodies in all eight applications adhere to the requirement that all tasks have visible side-effects only on their collection arguments. However, the largest application, S3D-Regent, is a mix of Regent task-based code and MPI, which does maintain its own internal state. While Relight can checkpoint the task-based code, custom checkpointing code would need to be written for the MPI part of the program to enable checkpointing of the entire application.

Task-based systems encourage applications to not access the contents of regions in the top-level task; for example, there are optimizations that can be done only if the top-level task neither reads nor writes the contents of regions, and these optimizations are important enough that compilers like Regent and runtime systems like Legion actually check and notify users if the top-level task does access region contents. Thus, it is unlikely that programmers would violate this condition unknowingly. We confirmed that all eight applications already adhere to this requirement.

To address whether the applications adhere to replay determinism, we consider a similar property already employed by Legion [4]. Every task-based system needs a way to achieve SPMD-like scalability, which is generally done by replicating execution of a single task (e.g., the top-level task) across all nodes to better distribute the control logic for launching many subtasks. But replicating

execution can only work if the code behaves the same in all copies—i.e., is deterministic. Legion enforces *control replication* dynamically and issues a hard error if it is violated. The four programs in our benchmark suite were checked for replay determinism in all of our experiments and the production applications have all been used on at least 10,000 nodes with control replication enabled, making it highly unlikely that any instances of non-determinism remain in their top-level tasks.

Finally, the four experimental benchmarks worked well with top-level checkpoints and there would be no benefit to hierarchical checkpointing within tasks. The story with the production applications is the same; all tasks are short-lived and control frequently returns to the top-level, making global checkpoints the natural choice for these applications.

7 Related Work

Fault tolerance in long-running applications has a significant literature. Approaches are typically distinguished by whether they are capable of withstanding full job failure (in which no computational resources survive the restart) or soft or partial job failure (in which some fraction of computational resources are lost but some survive). In the case of task-based systems, the assumption that at least the root task survives to recover the computation means that partial task recovery methods simply do not apply to the problem of restart from whole job failure.

Approaches with automated support for recovering from full job failure, in contrast to partial job failure, face the task of fully reconstructing an application's execution state. Such approaches typically rely on either runtime mechanisms [2, 20, 25] or compile-time methods [41] of identifying the execution state of the program. Local and heap variables must both be saved, either by inspecting the program state (registers, stack, etc.) or by instrumenting the program through compiler passes (and using instrumented memory allocators for heap memory). In either case, it is necessary to save the contents of outstanding network messages, introducing nontrivial overheads. Because these approaches save the entire virtual address space (or use instrumented memory allocators), without any special knowledge about what application data is necessary for restart, they impose overheads in storage space or running time as well. On modern HPC machines equipped with heterogeneous accelerators (such as GPUs), these approaches must also manage, save and restore the state (including memory contents) of any devices used in the computation, and because the APIs for such devices are not standardized across vendors, the cost grows with the number of vendors supported. In contrast, our approach is oblivious to the underlying hardware and transparently supports GPUs without the use of any GPU-specific code.

A wide variety of frameworks exist for manually integrating checkpointing into applications [8, 14, 26, 34, 37, 50]. While details vary, users are typically required to identify the exact data to be saved, and possibly the storage format. Users are responsible for writing recovery code, which is separate from main application initialization logic, and is often error prone.

Approaches for recovering from soft or partial failures do not provide an ability to fully reconstruct an application's state from disk; therefore many of the challenges faced in fast-forward replay do not exist. Automatic approaches in this space usually focus on saving data either to a buddy node [31] (allowing recovery as long as at least one copy of the data survives), or replicate the execution of the program [36] (to protect from otherwise undetected soft errors). As with fast-forward replay, the structure of task-based programs has been shown to be helpful in making these schemes efficient [13, 31]. These schemes can often be more efficient than full job recovery, because they may be able to reuse the surviving resources [16]. However, because they do not provide the ability to restart from a full job failure, users who require this use case (e.g., for very long-running simulations) must separately (and manually) implement that support.

Less directly related to Relight is recent work in recovery from partial failures in cloud-based systems based on replaying logs [12, 22, 29, 54]. These systems support concurrent transactions

and a persistent log is thus required after failure to reconstruct the order of replayed operations. The key insight in fast forward replay is that replay determinism is sufficient to enable the program state to be reconstructed from only program execution and a checkpoint.

Recovery in systems with transaction semantics and non-deterministic results (depending on the commit order of the atomic transactions) requires keeping logs so that replay can reconstruct a consistent system state [15, 23, 30, 33, 40]. After a failure the log is replayed to be certain that the order of events in the reconstruction is consistent with the order in the original execution. Fast-forward replay has a similar flavor but takes advantage of the task-based runtime's sequential semantics that is not present in systems with concurrent transactions; as a result no logs need to be saved to support fast-forward replay. The replay determinism requirement could be eliminated by keeping logs of the results of any non-deterministic operations in the top-level task; however, this would require knowledge of and instrumentation of all possible non-deterministic actions that could be performed. Given that it is safe to have non-deterministic behavior as long as it is inside of tasks, we have opted to require, and enforce, replay determinism instead.

Recovery from partial failure relies on restricted dependence patterns: either all "jobs" are independent (e.g., as in transaction-based systems) or the computations are data parallel at the top-level, with sequential chains of dependencies between stages of one data parallel worker. Neither of these classes addresses the HPC applications Relight supports.

There is a large amount of work which is more distantly related to our approach. Hardware support for persistent memory can help to enable fault tolerance, but is not yet prevalent in modern HPC machines, and still requires careful manual design of algorithms and data structures to avoid data loss [15]. Actor systems have been shown to integrate with external resources, but require the user to specify if, when, and how to persist the application's state [45]. Previous work in fork-join models with distributed work stealing has shown that automatic checkpoint-restart can be accomplished via a centralized controller or with a distributed-shared data store [32, 52], at the cost of centralized bottlenecks as applications scale.

8 Discussion

Relight provides transparent, efficient, and scalable checkpointing for applications written in a class of HPC and data analytics task-based systems. However, it is not without limitations.

The primary limitation of Relight is programs must be written in a task-based programming model. The runtime system must have knowledge of all tasks that execute, their dependencies, and all the collections used and their partitionings to automatically and correctly manage saving and restoring checkpoints. A core contribution of our work, and the basis for Relight, is the observation that current HPC and data analytics task-based programming models are sufficiently powerful to not just implement Relight, but to implement it as a user-level library, allowing implementations of Relight to rely on the sequential semantics of these systems.

The design of Relight also has some limitations. The replay determinism restriction excludes making randomized choices or having any kind of non-deterministic behavior in the top-level task. This restriction is enforceable through the checking procedure we have described, and there is a simple and general fix, which is to wrap any non-deterministic operations in a subtask, at which point the top-level task itself becomes replay deterministic. It is also notable that none of the codes we have worked with, all of which were written before the development of Relight, required any changes to satisfy replay determinism.

Most interactions with external libraries can be handled similarly, by wrapping the library call in a task. As long as the library call does not maintain persistent state, Relight will correctly checkpoint and restore the results of the library call. If the external library does maintain persistent state outside of the task-based runtime, then a separate checkpointing mechanism is needed to save and

reconstruct that state, while Relight can still handle the task-based portion of the program. While this would require additional work on the part of the user, it is only the non-task-based portion of the application that requires custom support.

Because fast-forward replay takes time proportional to the length of the computation being checkpointed, very long running computations could have noticeable restart times due to the cost of fast-forward replay. In our experience, however, even for the longest-running applications we have examined the time to reload the checkpoint data from disk has generally dominated the replay time—it would require restarting executions with orders of magnitude more task calls at the top-level before fast-forward replay became a noticeable cost of restarting an application.

Similarly, the Relight implementation relies on optimizations that could fail to deliver good performance in certain situations. Relight computes covering sets of partitioned data to ensure all data is saved while maximizing parallelism in writing data to disk. If a region is not partitioned at all, however, Relight will simply write the entire data set from a single processor, which could be very time consuming. Such a situation would almost certainly indicate a poorly designed program, however, as the application can also update that region using only a single task at a time. The compression of escaped future values using run-length encoding depends on the expected behavior that futures at a particular call site will tend to have the same value on repeated calls. This heuristic works extremely well in our experiments, but it is not hard to imagine programs with different behavior. If the run-length encoding of escaped futures does not yield much compression, the cost would be checkpoint files that grow with the length of the computation. Note that this potential issue applies only to futures that escape, which means the contents of the future are actually examined by the top-level task. Because such an action is likely to block progress of the top-level task while the future resolves, idiomatic programs avoid testing futures in the top-level task unless absolutely necessary. Finally, Relight relies on control replication to distribute the work of saving any metadata that is proportional in size to the number of compute nodes. Not every HPC or data analytics task-based system has the equivalent of control replication; some systems, such as Spark and Dask, rely on centralized controllers and additional work would be needed to distribute the saving of metadata. These systems have much simpler metadata than Legion, however, so there may be another approach to scalably saving metadata in those more restricted environments.

As future work, it would be interesting to consider whether checkpoint operations could be placed automatically at program points to minimize the live data saved while also saving often enough to guarantee forward progress (more frequently than the mean time between failures). To date we have found the heuristic of placing checkpoints at the top of the main loop and then empirically adjusting the checkpoint frequency to work very well.

9 Conclusion

We have presented Relight, a simple user-level checkpointing library for HPC task-based runtimes. Relight only requires users to annotate where checkpoints should be taken while the system automatically determines the minimal set of data to be saved and how best to store it. A novel and provably safe fast-forward replay mechanism rapidly resumes execution without requiring specialized code paths for recovery. Finally, Relight is implemented at the user level, meaning that it is itself a sequential program that is parallelized by the underlying runtime, underscoring the expressiveness of task-based systems. In our experiments, Relight performance scales comparable to the original, uncheckpointed programs, limited only by the achievable I/O bandwidth in the underlying filesystem. Relight therefore represents a new class of checkpointing algorithms that deliver both ease-of-use and high performance for general programs written in task-based systems.

10 Data-Availability Statement

Relight is open-source software, publicly available under the Apache Software License, version 2. The public repository for Relight is available at: https://github.com/StanfordLegion/resilience/

The artifact for this paper includes the complete source code for Relight, including all benchmarks, scripts, and environment/build configuration. The artifact also includes dependencies for Relight, including the specific version of Legion used for our experiments. We include raw measurements for the applications and processing steps required to generate graphs for the paper. The artifact is available at: https://doi.org/10.5281/zenodo.21479558

11 Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration; and the National Science Foundation under Grant No. 2216964. Experiments on the Piz Daint supercomputer were supported by the Swiss National Supercomputing Centre (CSCS) under project ID d108.

References

[1] 2016. Piz Daint - CSCS. http://www.cscs.ch/computers/piz_daint.

[2] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *International Parallel and Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2009.5161063

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23 (Feb. 2011), 187–198. Issue 2. doi:10.1002/cpe.1631

[4] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. 2021. Scaling Implicit Parallelism via Dynamic Control Replication. In *Principles and Practice of Parallel Programming (PPoPP)*. Association for Computing Machinery, New York, NY, USA, 105–118. doi:10.1145/3437801.3441587

[5] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2022. *Artifact for Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence*. doi:10.5281/zenodo.7332228

[6] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. 2023. Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence. In *Principles and Practice of Parallel Programming (PPoPP)*. Association for Computing Machinery, New York, NY, USA, 218–231. doi:10.1145/3572848.3577515

[7] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1109/SC.2012.71

[8] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. 2011. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/2063384.2063427

[9] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*. doi:10.1145/1640089.1640097

[10] Dan Bonachea and Paul H. Hargrove. 2019. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Languages and Compilers for Parallel Computing (LCPC'18)*. Springer, 138–158. doi:10.1007/978-3-030-34627-0_11

[11] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45. doi:10.1109/MCSE.2013.98

[12] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. 2021. Durable functions: Semantics for stateful serverless. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27. doi:10.1145/3485510

[13] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. 2015. Design for a Soft Error Resilient Dynamic Task-based Runtime. In *International Parallel and Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2015.81

[14] Andrew Chien, Pavan Balaji, Peter Beckman, Nan Dun, Aiman Fang, Hajime Fujita, Kamil Iskra, Zachary Rubenstein, Ziming Zheng, Rob Schreiber, Jeff Hammond, James Dinan, Ignacio Laguna, David Richards, Anshu Dubey, Brian van Straalen, Mark Hoemmen, Michael Heroux, Keita Teranishi, and Andrew Siegel. 2015. Versioned Distributed Arrays for Resilience in Scientific Applications: Global View Resilience. *Procedia Computer Science* 51 (2015), 29–38. International Conference On Computational Science (ICCS). doi:10.1016/j.procs.2015.05.187

[15] Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023. Memento: A Framework for Detectable Recoverability in Persistent Memory. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 292–317. doi:10.1145/3591232

[16] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. 2012. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1109/SC.2012.36

[17] Mario Di Renzo. 2022. HTR-1.3 Solver: Predicting Electrified Combustion Using the Hypersonic Task-based Research Solver. *Computer Physics Communications* 272 (2022), 108247. doi:10.1016/j.cpc.2021.108247

[18] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *SC*. doi:10.1145/1188455.1188543

[19] Charles R. Ferenbaugh. 2014. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* (2014). doi:10.1002/cpe.3422

[20] Rohan Garg, Gregory Price, and Gene Cooperman. 2019. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *High-Performance Parallel and Distributed Computing (HPDC)*. doi:10.1145/3307681.3325962

[21] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561. doi:10.1016/j.softx.2020.100561

[22] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, et al. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proceedings of the VLDB Endowment* 13, 5 (2020), 588–601. doi:10.14778/3377369.3377370

[23] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. 1981. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys (CSUR)* 13, 2 (1981), 223–242. doi:10.1145/356842.356847

[24] William Gropp, Steven Huss-Lederman, and Marc Snir. 1998. *MPI: The Complete Reference. The MPI-2 Extensions*. Vol. 2. MIT press.

[25] Paul H. Hargrove and Jason C. Duell. 2006. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In *Journal of Physics: Conference Series*, Vol. 46. IOP Publishing, 494. doi:10.1088/1742-6596/46/1/067

[26] Amin Hassani, Anthony Skjellum, and Ron Brightwell. 2014. Design and Evaluation of FA-MPI, A Transactional Resilience Scheme for Non-blocking MPI. In *Dependable Systems and Networks (DSN)*. doi:10.1109/DSN.2014.78

[27] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017. Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Denver, Colorado) *(ScalA '17)*. ACM, New York, NY, USA, Article 6, 8 pages. doi:10.1145/3148226.3148233

[28] Zhihao Jia, Sean Treichler, Galen Shipman, Michael Bauer, Noah Watkins, Carlos Maltzahn, Patrick McCormick, and Alex Aiken. 2017. Integrating External Resources with a Task-Based Programming Model. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 307–316. doi:10.1109/HiPC.2017.00043

[29] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing Microservice Applications on Serverless, Correctly. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 367–395. doi:10.1145/3571206

[30] Butler W Lampson and Howard E Sturgis. 1979. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California.

[31] Romain Lion and Samuel Thibault. 2020. From Tasks Graphs to Asynchronous Distributed Checkpointing with Local Restart. In *Workshop on Fault Tolerance for HPC At EXtreme Scale (FTXS)*. IEEE, 31–40. doi:10.1109/FTXS51974.2020.00009

[32] Wenjing Ma and Sriram Krishnamoorthy. 2012. Data-Driven Fault Tolerance for Work Stealing Computations. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*. 79–90. doi:10.1145/2304576.2304589

[33] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30. doi:10.1145/3133920

[34] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. 2010. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1109/SC.2010.18

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A Distributed Framework for Emerging AI applications. In *Operating Systems Design and Implementation (OSDI)*. 561–577.

[36] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V. Kalé. 2013. ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/2503210.2503266

[37] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *International Parallel and Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2019.00099

[38] Russ Rew and Glenn Davis. 1990. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications* 10, 4 (1990), 76–82. doi:10.1109/38.56302

[39] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Python in Science Conference (SciPy)*. Citeseer.

[40] Mendel Rosenblum and John K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52. doi:10.1145/146941.146943

[41] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. 2004. Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1109/SC.2004.29

[42] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-Productivity Programming Language for HPC with Logical Regions. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/2807591.2807629

[43] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David Walker, and Jack J. Dongarra. 1998. *MPI-The Complete Reference*. MIT Press.

[44] Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. 2021. Index Launches: Scalable, Flexible Representation of Parallel Task Groups. In *High Performance Computing, Networking, Storage and Analysis (SC)*. doi:10.1145/3458817.3476175

[45] Olivier Tardieu, David Grove, Gheorghe-Teodor Bercea, Paul Castro, Jaroslaw Cwiklik, and Edward Epstein. 2023. Reliable Actors with Retry Orchestration. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1293–1316. doi:10.1145/3591273

[46] The HDF Group. 1997-2023. *Hierarchical Data Format, version 5*. https://www.hdfgroup.org/HDF5/.

[47] Hilario Torres and Gianluca Iaccarino. 2018. Soleil-X: Turbulence, Particles, and Radiation in the Regent Programming Language. *Bulletin of the American Physical Society* 63 (2018).

[48] Sean Treichler, Michael Bauer, and Alex Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/2544173.2509545

[49] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick S. McCormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, and Jacqueline Chen. 2017. S3D-Legion: An Exascale Software for Direct Numerical Simulation of Turbulent Combustion with Complex Multicomponent Chemistry. In *Exascale Scientific Applications*. Chapman and Hall/CRC, 257–278. doi:10.1201/b21930-12

[50] Rob F. Van Der Wijngaart, Marc Gamell, Keita Teranishi, Eric Valenzuela, Michael A Heroux, and Manish Parashaar. 2016. Fenix, A Portable Flexible Fault Tolerance Programming Framework for MPI Applications. In *Workshop on Exascale MPI (ExaMPI)*.

[51] Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels. In *HPEC*. 1–6. doi:10.1109/HPEC.2014.7040972

[52] Gosia Wrzesinska, Ana-Maria Oprescu, Thilo Kielmann, and Henri Bal. 2007. Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*. Springer, 425–436. doi:10.1007/978-3-540-74466-5_46

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10 (2010), 10–10.

[54] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1187–1204.

Supplemental Material

A Covering Set Algorithm

Listing 2 shows the algorithm for computing covering sets. The algorithm is divided into two phases. The first phase runs eagerly on every task that accesses a region. If that task uses a partition, which is particularly likely for group task launches, `track_partition` updates `recent_partitions` to store the partition. We save every partition, as long as it is disjoint and complete and all parent partitions recursively up the tree are also complete (but may or may not be disjoint). The result of `is_eligible` can be cached for efficient access because partitions are immutable in Legion. We only track the one most recent partition in `recent_partitions`, because we are interested in the partitions most recently used.

A second phase runs at each checkpoint to compute the covering sets. For every root region we need to save at a checkpoint, we call the function `compute_covering_set`. This function calls two mutually recursive functions for regions and partitions, respectively. In the region case, we want to identify the best partition, since regions can be partitioned multiple times. Since we know that all eligible partitions will be disjoint and complete, it is sufficient to simply pick the deepest subtree, which maximizes the amount of parallelism in our I/O operations. In the partition case, we start by attempting to cover each region in turn. If we're successful with at least one region (i.e., if that region is itself partitioned), then we keep those covering sets and augment it with any regions that were not covered. Otherwise, as a base case, we return the partition itself.

Figure 11 shows the result of running the algorithm in Listing 2 on the Pennant example from Section 2. In the full Pennant code, three partitions are accessed repeatedly: `zones_part`, the partition of zones, and `private_part` and `shared_part` partitions of points. The algorithm ignores `ghost_part` because it is aliased, and chooses to skip `point_part` because it can find deeper partitions. In practice, we find that this algorithm does a good job of identifying relevant partitions in the program, maximizing parallelism while minimizing data movement and duplication.

B Execution Profile for Regent Pennant

In this section we provide screenshots from a profile taken during a run of the Regent Pennant application to demonstrate several important aspects of Relight's performance that are described in the body of the paper. The profile was taken during a run of Regent Pennant on 128 nodes of Piz Daint, with checkpoints taken every 300 timesteps. When requested, Legion dynamically captures timing information of various aspects of an application as it executes with minimal overhead. A profile consists of a log file from each node containing timing information about tasks and copies (e.g., between GPU memories, from GPU to CPU, or to disk). For each task, profile logs record the:

- creation time (when the task is issued),
- ready time (when all dependencies are satisfied and the task is queued for execution),
- start time (when the task actually begins execution, which can be much later than the ready time if the designated processor for the task is busy),
- and stop time for the task.

Note that Legion's profiler is based on tracing, not sampling, which means that every task and every copy is represented in the profile with perfect fidelity. We can therefore be confident that if the profile appears empty at a given point in time on a given processor, no task is running. Because profiling introduces a small but non-zero overhead (usually in the range of 5–10% depending on the application), we did not have profiling enabled during the experimental runs in Section 6, but only in a separate run to verify the performance properties of Relight.

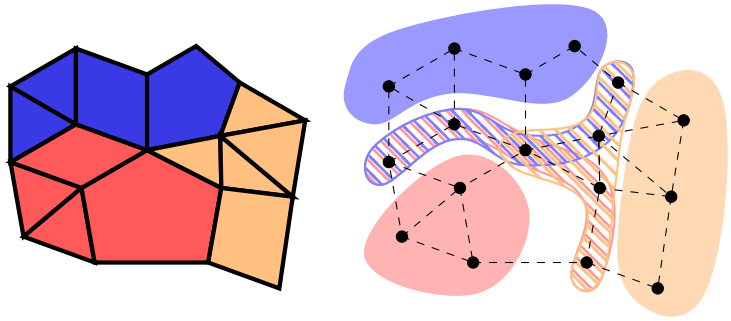


Fig. 10. The Pennant mesh consists of zones (left) and points (right). Partitions of the data are shown in color. Zones are partitioned once, corresponding to the Listing 1 line 9. Points are partitioned hierarchically, and multiple times: first, into private vs. shared (lines 11–12), represented as the solid and hatched regions, respectively. Then each are partitioned again (lines 13–15). The overlapping hatched colors show the ghost partition, while a second disjoint partition is also made of these same points (not shown).

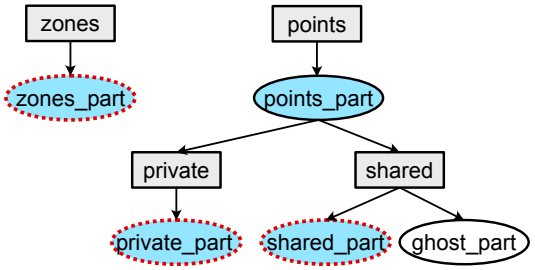


Fig. 11. Pennant region tree with regions in rectangles and partitions in ellipses. Eligible partitions are shaded blue. The covering set is shown in dashed red.

Before introducing the profile, we remind readers of some aspects of Relight described in the main paper that are relevant to the profile. First, as noted in Section 5.3, Relight uses a two-stage copy approach where data is first copied from GPU framebuffer to *zero-copy* memory (CPU memory pinned with the GPU driver) and then from zero-copy memory to disk. While Legion supports copies directly from GPU framebuffer to disk, our approach allows Legion to break write-after-read dependencies that would otherwise stall the execution of subsequent application tasks until the checkpoint is written to disk. Second, as noted in Section 5.2 (and elaborated in Appendix A), we seek to identify as much parallelism as possible by selecting a covering set of subregions, which means that multiple copies can proceed at each stage in parallel. We can now inspect the profile to verify several key properties of Relight:

(1) Application pauses during checkpointing are attributable entirely to copies that must be made from GPU framebuffer to zero-copy memory for performing checkpointing, and not to CPU time spent in checkpoint preparation, Legion runtime overheads, or copying checkpoint data to disk. This ensures that application tasks are delayed as little as possible when performing a checkpoint.
(2) The critical gap from the last application GPU task to the first copy from framebuffer to zero-copy memory is only 15 microseconds, and does not involve the CPU at all. This shows

```
def is_eligible(part):
  if is_disjoint(part) and is_complete(part):
    region := get_partition_parent_region(part)
    while has_region_parent_partition(region):
      part := get_region_parent_partition(region)
      if not is_complete(part):
        return false
      region := get_partition_parent_region(part)
    return true
  return false

def track_partition(part, priv):
  if priv ≠ reduce and is_eligible(part):
    parent := get_partition_parent_region(part)
    recent_partitions[parent] := part

def make_region_tree():
  region_tree := {}
  for region, part in recent_partitions:
    region_tree[region] := part
    while has_region_parent_partition(region):
      part := get_region_parent_partition(region)
      region := get_partition_parent_region(part)
      region_tree[region] += part
  return region_tree

def compute_covering_set_region(region, depth, region_tree):
  best_regions, best_partitions, best_depth := ∅ , ∅ , 0
  for part in region_tree[region]:
    try_regions, try_partitions, try_depth :=
        covering_set_partition(part, depth+1, region_tree)
    if try_depth > best_depth:
      best_regions, best_partitions, best_depth :=
          try_regions, try_partitions, try_depth
  return best_regions, best_partitions, best_depth

def compute_covering_set_partition(part, depth, region_tree):
  regions, partitions := ∅ , ∅
  uncovered := ∅
  recurse := false
  for region in get_partition_subregions(part):
    subregions, subpartitions, subdepth :=
        compute_covering_set_region(region, depth+1, region_tree)
    if subregions = ∅ and subpartitions = ∅ :
      uncovered += region
    else:
      regions += subregions
      partitions += subpartitions
      depth = max(depth, subdepth)
      recurse := true
  if not recurse:
    partitions += part
  else:
    regions += uncovered

def compute_covering_set(root):
  region_tree := make_region_tree(root)
  regions, partitions := compute_covering_set_region(root, 0, region_tree)
  if regions = ∅ and partitions = ∅ :
    regions += root
  return regions, partitions
```

Listing 2. Algorithm for computing covering sets.

that neither Relight nor Legion is on the critical path of these checkpointing copies; instead the latency is purely the time for synchronizing GPU kernels and running GPU copies.

(3) Despite a substantial decrease in GPU utilization, the application actually never fully pauses during checkpoints and a handful of application GPU tasks begin to run even before all copies from GPU framebuffer to zero-copy memory for checkpointing complete. This shows how Relight leverages Legion's dependence tracking to enable application tasks to resume as rapidly as possible, long before the checkpoint completes.

(4) Copies from zero-copy memory to disk run concurrently with both the device-to-host copies and the application itself, illustrating that data movement for checkpointing is automatically pipelined, and continues as the application resumes normal execution.

Figure 12 shows an overview of a representative time slice from the profile, showing the application running with an interruption for checkpointing in the middle. This view shows the average utilization over time of each of the key resources: GPU, CPU, utility (Legion's internal analysis), and channels (copies between pairs of memories, e.g. from GPU to CPU memory, etc.). On either side of this view we see the steady state of the application running with high GPU utilization. In the center,

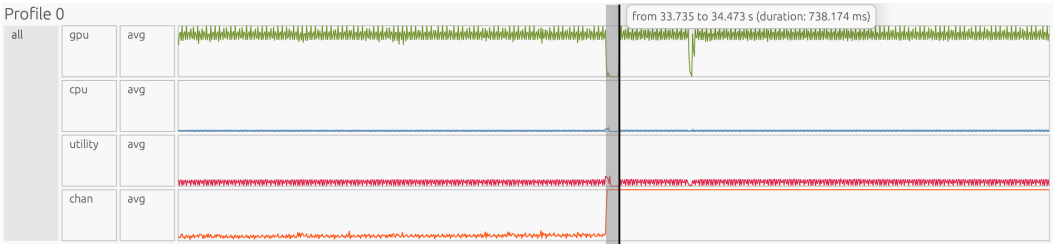


Fig. 12. Profile overview from a Regent Pennant run on 128 nodes, showing average utilization over time for GPU, CPU, utility (internal Legion work), and channels (copies between different pairs of memories, e.g. from GPU to CPU memory, etc.). The cursor (shown as a shaded region with a black line on one side) highlights an interval of approximately 700 ms where GPU execution is interrupted as the checkpoint begins to copy data from the GPU.

GPU utilization drops when the checkpoint is issued, while channel utilization saturates[7], first with copies from GPU framebuffer to zero-copy memory and then subsequently from zero-copy memory to disk. Channel utilization remains high as copies to disk proceed while the application resumes execution. The gap here is approximately 700 ms.

Zooming into the profile we can see the copies occurring in greater detail. Figure 13 shows an expanded view of the first node in the profile. The GPU, CPU and channels have been expanded to show the specific tasks or copies that are running. Each colored rectangle is a task or copy. In particular, we see a pattern of repeated execution on the GPU prior to the checkpoint, which pauses as copies are made from GPU framebuffer to zero-copy memory, and then resumes. The cursor (shown as a shaded region with a black line on one side) highlights the interval from t=33.748 to 34.447 seconds while the copies to zero-copy memory are proceeding, which perfectly aligns with the GPU execution gap.

We make three key observations about the execution patterns shown in Figure 13. First, disk copies (shown toward the bottom of the screenshot) begin as soon as the first copies to zero-copy memory complete and then proceed concurrently with both subsequent copies into zero-copy memory and also GPU execution (once the application resumes). Disk copies do not block execution.

Second, the CPU is largely idle. The long-running task visible on the CPU is the top-level task, and is shown in a lightly shaded color to indicate that it is blocked, either on data or because it has run sufficiently far enough ahead that Legion has chosen to stall execution. The checkpoint is not a compute-heavy operation and the costs of checkpointing are by far dominated by the copies being made from GPU framebuffer to zero-copy memory.

Third, the interruption created by the checkpoint is not total. We see a handful of GPU tasks begin to execute even while these copies are still ongoing. This is because Legion computes the data dependencies between individual pairs of tasks and copies. In this case, the dependencies are write-after-read dependencies: subsequent tasks cannot begin execution until the copy to zero-copy memory has completed (we cannot overwrite data while it is in the process of being copied out of the GPU). The transition from checkpoint to normal execution looks like a handoff because most application tasks write most of the data; but to the extent that tasks use a limited subset of data elements, they may proceed earlier. This demonstrates that Legion is achieving the maximum overlap between checkpoint and application execution, subject to the inherent data dependencies that must be respected.

Finally, we zoom in even further to examine the handoff from the last application GPU task to the first copy issued as part of the checkpoint (from GPU framebuffer to zero-copy memory). Figure 14

[7]Note that, for the purposes of computing utilization of channels, a channel is considered to be utilized if any data copy is active. Legion's profiler does not model the relative bandwidths of channels and copies.

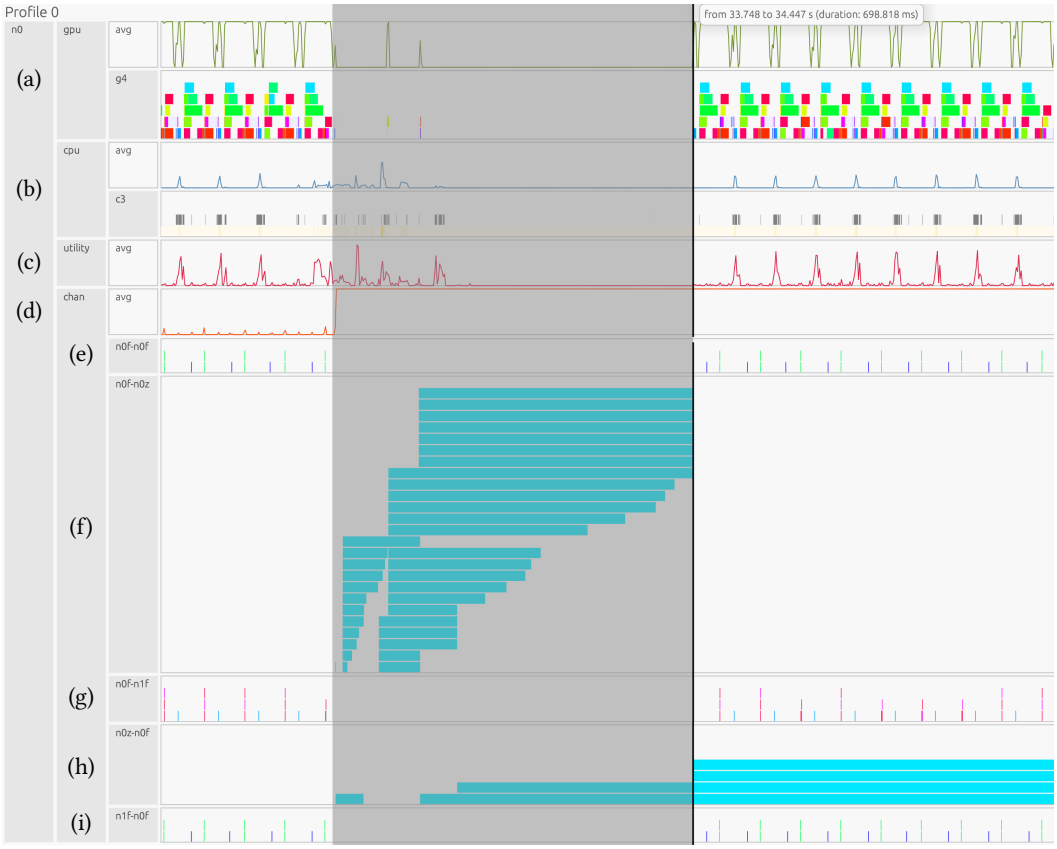


Fig. 13. Expanded view showing the checkpoint pause in detail. From top to bottom, we see node 0's (a) GPU, (b) CPU, (c) utility processor, and (d) channels. All but the utility processor have been expanded to show the specific tasks or copies that are running. Within the channels we see (from top to bottom) copies (e) from node 0 framebuffer to itself (i.e., between allocations within that memory; n0f-n0f), (f) from node 0 framebuffer to zero-copy memory (n0f-n0z), (g) from node 0 to node 1 framebuffer (n0f-n1f), (h) from node 0 zero-copy memory to file (i.e., disk; n0z-n0f), and (i) from node 1 to node 0 framebuffer (n1f-n0f). The cursor again highlights the region of approximately 700 ms where GPU execution is largely paused (though a few GPU tasks are visible at this magnification).

shows this zoomed view, with the cursor highlights the gap of about 15 microseconds. Notably, the CPU is not active during this time at all. Relight is designed to issue copies first (which then run asynchronously) before it blocks on futures. Thus all of the copies from GPU framebuffer to zero-copy memory (as well as to disk) have already been issued, analyzed by Legion and scheduled while the main application is still running. The handoff from application execution to checkpoint copy therefore has very low latency, invoking only Legion's low-level mechanisms for triggering dependencies and initiating copies. All of the copies are in flight prior to the point where Relight blocks on futures and prepares the contents of the checkpoint. As we saw in previous views, the CPU remains largely idle during the checkpoint delay and does not inhibit the application from resuming execution as soon as the relevant data dependencies are satisfied.

Received 2026-03-12; accepted 2026-08-06

Fig. 14. This view is further expanded to highlight the handoff from the last application GPU task (shown in magenta) to the first checkpoint copy from GPU framebuffer to zero-copy memory (shown in teal). Lightly shaded boxes represent tasks that are blocked, medium-shaded portions represent tasks that are ready but not yet executing, and brightly-colored boxes represent running tasks. The small, pink boxes on the CPU are Legion internal meta-tasks used to collect profiling data, and would not appear in a production run of this application.