Recovery Tasks: An Automated Approach to Failure Recovery

Brian Demsky[1], Jin Zhou[1], and William Montaz[2]

[1] University of California, Irvine
[2] Octo Technology

Abstract. We present a new approach for developing robust software applications that breaks dependences on the failed parts of an application's execution to allow the rest of the application to continue executing. When a failure occurs, the recovery algorithm uses information from a static analysis to characterize the intended behavior of the application had it not failed. It then uses this characterization to recover as much of the application's execution as possible.
We have implemented this approach in the Bristlecone compiler. We have evaluated our implementation on a multiplayer game, a web portal, and a MapReduce framework. We found that in the presence of injected failures, the recovery task version provided substantially better service than the control versions. Moreover, the recovery task version of the game benchmark successfully recovered from a real fault that we accidentally introduced during development, while the same fault caused the two control versions to crash.

1 Introduction

All too often, failures are caused by the propagation of errors through critical components of software applications. Current software development tools actually encourage the introduction of unnecessary dependences between conceptually unrelated components. These dependences introduce new error propagation pathways, which in turn can introduce new vulnerabilities. For example, many programming languages encourage developers to map otherwise independent software components onto the same thread. If one component fails, other components mapped onto the same thread will likely fail even though their only relationship with the original failure is artificially induced via the mapping of components to threads.

Our previous work on Bristlecone introduced a task-based language designed to eliminate artificial dependences that serve to propagate errors [8]. A shortcoming of Bristlecone is that it cannot prevent the propagation of failures through legitimate dependences. If a failure occurs, it can be desirable for tasks that legitimately depend on the failed part of the computation to operate in a degraded manner. For example, if a failure prevents rendering a web page frame, the web browser can still render the web page by simply rendering the frame as an empty box.

This paper extends our previous work on Bristlecone to manage failure propagation through legitimate dependences. The technique is based on the observation that although it is difficult to anticipate how applications may fail, there are often locations in an application in which it is straightforward to break dependences on data that is missing because of a failure. For example, developing recovery routines for all possible failures of a web page rendering engine is likely to be impossible. However, a developer

might reasonably write a rendering engine that can assemble frames into a web page even when some frames are missing because of a failure.

We extend Bristlecone with *recovery tasks*. Recovery tasks serve as software circuit breakers — they break legitimate data dependences in the event of a software error to mitigate the damage caused by that error. More precisely, a recovery task can function even if an error in another part of the computation causes some of the recovery task's input parameters to be unavailable. Note that the exact task that breaks a dependence chain is not important — the system simply needs a point in the dependence chain to halt the propagation of a failure.

Our approach uses static analysis to characterize the *intended behavior* of the failed part of a computation. We use the term intended behavior to refer to the behavior that a failed computation would have had if the failure had not occurred. For each possible failure point, this analysis computes which tasks the computation, had it not failed, would have executed. The analysis then identifies recovery tasks in these sets. The recovery algorithm then uses the recovery tasks to break data dependences on the failure and recover that part of the computation.

A failure will cause the application to skip some tasks. The analysis next determines which data structures these skipped task would have modified. The runtime uses these results to mark any data structures that the skipped part of the application may have modified as damaged. It then uses the recovery tasks to break the execution's dependence on the damaged data structures.

Our approach contains the following key components:

- **Language Extensions:** Developers use annotations to declare a set of recovery tasks that can execute even if a failure causes some of their parameter objects to be unavailable. The developer guards accesses to those parameter with checks that verify that the parameter is available before accessing it.
- **Static Analysis:** The compiler analyzes the application's code and task specifications to construct an abstract state transition graph for each class. These graphs abstract concrete objects' state with nodes that represent abstract states. We have developed a static analysis that reasons about the state transition graphs to characterize the intended behavior of the failed code.
- **Recovery Algorithm:** The runtime system uses static analysis results to reason about the intended behavior of the failed part of the computation. While it is in general impossible to determine the exact intended behavior of the failed part on the objects' states, our analysis can still generate constraints on the possible states of these objects. The recovery algorithm uses the results of the static analysis to determine which recovery tasks should be executed.

1.1 Comparison to Manual Recovery

Many programming languages, including Java, provide exception handling mechanisms that are designed to help applications recover from failures. Exception handling works best when recovery can be performed at a location that syntactically encloses the failure and the recovery action allows the application to return to completely normal execution. Unfortunately, effective error recovery can require addressing a wide range of consequences of an error, which may propagate through both the control and data dependences. In particular, the natural place to recover from an error that prevents the

generation of a data structure can often be after several subsequent operations on the data. Moreover, it may not be possible to completely recover from an error at a single program point — the effects of the error may linger for some time and require that recovery actions be woven throughout the application.

Writing exception handlers can require the developer to write code that propagates failure recovery information to the points at which application can perform recovery. Our approach automatically reasons about an application to characterize the effects of error propagation through both data and control dependences. Our algorithm uses this information to generate a set of recovery actions for the application.

1.2 Contributions

This paper makes the following contributions:

- **Recovery Algorithm:** It presents a new recovery algorithm that manages the propagation of errors through legitimate dependences to recover applications from failures.
- **Analysis:** It presents a static analysis and a recovery algorithm that can reason about the intended behavior of the failed part of a computation.
- **Language Extensions:** It presents language extensions that developers can use to express high-level insight into how to modify an application's execution to break dependences that would otherwise serve to propagate failures.
- **Experience:** It presents an evaluation of the technique on several benchmarks. For each application, we report our experience developing the application and evaluate how robust the application is to injected failures relative to control versions.

2 Example

We present a web browser example that illustrates the recovery algorithm.

2.1 Classes

Figure 1 presents parts of the `Page`, `Frame`, and `FrameDescriptor` class declarations. When the example web browser parses a frame, it creates a new `Page` object to store the rendered web page. For each frame, the parser creates a `FrameDescriptor` object that describes where to place the frame and a `Frame` object that contains the information needed to render the frame. The `Frame` object will store the rendered frame.

Class declarations contain declarations for the class's abstract states. Bristlecone's abstract states support orthogonal classifications of objects: an object may simultaneously be in more than one abstract state. The runtime uses the abstract state of an object to determine which *tasks* to invoke on the given object. When a task exits, it can change the values of the abstract states of its parameter objects.

An abstract state is declared with the keyword `flag` followed by a name. The `Frame` class declaration contains three abstract state declarations: the `plugin` state, which indicates that rendering the frame object requires a plugin; the `rendered` state, which indicates that the browser has rendered the frame; and the `processed` state, which indicates that the browser has incorporated the rendered frame into the page.

2.2 Tasks

Figure 2 presents task definitions from the web browser example. A task definition consists of the `task` keyword, the task's name, the task's parameter declarations, and

```
 1 public class Page {
 2   flag rendered;
 3   flag displayed;
 4   ...
 5 }
 6
 7 public class Frame {
 8   flag plugin;
 9   flag rendered;
10   flag processed;
11   ...
12 }
13
14 public class FrameDescriptor {
15   ...
16 }
```

Fig. 1. Class Definitions

```
 1 task ParsePage(...) {
 2   ...
 3   tag pt=new tag(pagetag);
 4   Page p=new Page()(add pt);
 5   ...
 6   while(moreFrames()) {
 7     ...
 8     tag ft=new tag(frametag);
 9     FrameDescriptor fd=new FrameDescriptor() (add ft);
10     Frame f=new Frame()(add pt, add ft);
11     ...
12   }
13 }
14
15 task RenderFrame(Frame f in !rendered && !plugin) {
16   if (needsplugin())
17     taskexit(f: plugin:=true);
18   ...
19   taskexit(f: rendered:=true);
20 }
21
22 task InvokePlugin(Frame f in plugin and !rendered) {
23   ...
24   taskexit(f: rendered:=true);
25 }
26
27 task AddFrameToPage(Page p in !rendered with pagetag pt, FrameDescriptor fd
28      with frametag ft, optional Frame f in rendered and !processed with
29      pagetag pt and frametag ft) {
30   if (isavailable(f)) {
31     //Add Frame to Page
32     ...
33   }
34   if (lastframe)
35     taskexit(f: processed:=true; p: rendered:=true);
36   else
37     taskexit(f: processed:=true);
38 }
39
40 task DisplayPage(Page p in rendered and !displayed) {
41   //Display Page
42   ...
43   taskexit(p: displayed:=true);
44 }
```

Fig. 2. Task Definitions

the task's body. A parameter declaration consists of a type, a parameter variable, and a guard expression. An object can serve as a task's parameter if it satisfies the parameter's

guard expression. The runtime invokes a task when there exist parameter objects in the heap that satisfy all the parameter guard expressions for the task. We discuss some of the example task definitions below:

- **ParsePage Task:** The `ParsePage` task allocates a `Page` object to the web page, splits the page into individual frames, and then generates a `Frame` object and a `FrameDescriptor` object for each frame.
  Note that it is important that the `Frame` objects are associated with both the correct `FrameDescriptor` and `Page` objects. Otherwise, the web browser may place frames in the wrong pages. The `ParsePage` task groups these objects by using tags. It creates a new tag instance of type `pagetag` and then binds this tag to the `Page` and `Frame` objects.
- **RenderFrame Task:** The `RenderFrame` task renders a frame. Its parameter declaration indicates that the runtime can invoke this task on `Frame` objects in the heap and the parameter guard expression `!rendered` indicates that the parameter object must not be in the `rendered` abstract state. When invoked, the task checks whether rendering this frame requires a plugin, and then it either executes a `taskexit` statement that transitions the object into the `rendered` abstract state to indicate that the frame is rendered or a `taskexit` statement that transitions the object into the `plugin` abstract state to indicate that a plugin is required to render the frame.
- **AddFrameToPage Task:** The `AddFrameToPage` task adds a rendered frame to the web page. Even if a software fault prevents a frame from being rendered, it is still possible to display the web page with that frame blanked. Therefore, we use the `optional` keyword to specify that the task can execute even if the `Frame` parameter is unavailable due to a failure. We call tasks that contain optional parameters recovery tasks. Recovery tasks use `isavailable` checks to verify that an optional parameter is available before accessing that parameter.
  Note that it is important that both the `FrameDescriptor` object corresponds to the `Frame` object and the `Frame` object is a frame for this specific `Page` object. The tag guard expression `with pagetag pt` in the first and third parameter declarations ensures that those parameter objects are bound to the same `pagetag` tag instance.

2.3 Error-Free Execution

We next discuss how the runtime would execute the example in an error-free execution:

1. **Parsing the Page:** The browser first executes the `ParsePage` task. This task creates a `Page` object to store the page, parses the web page, and creates both a `Frame` object and a `FrameDescriptor` object for each frame in the page.
2. **Processing Frames:** The browser next processes the frames by performing the following operations:
   - **A. Render the Frame:** The browser executes the `RenderFrame` task to render the frame. If the frame requires a plugin to render, the runtime passes the frame to the `InvokePlugin` task.
   - **B. Optionally Invoke a Plugin:** If a frame requires a plugin, the browser executes the `InvokePlugin` task to invoke the necessary plugin.
   - **C. Add the Frame to the Web Page:** The `AddFrameToPage` task adds a rendered `Frame` object to the `Page` object. Once all frames have been rendered, this task marks the `Page` object as rendered.

3. Displaying the Page: After the `Page` object has been rendered, the `DisplayPage` task displays the page.

2.4 Reasoning About Failures

In this example, the developer has provided a recovery task implementation of the `AddFrameToPage` task that can function even if a failure affects one of its parameter objects. If rendering a web page frame fails, this allows the runtime to break dependences on missing frames at the `AddFrameToPage` task. Breaking these dependences allows the web browser to recover from errors in processing and rendering web page frames and still display the affected web page.

If a failure occurs, the recovery algorithm must characterize how the computation would have proceeded in the absence of the failure. The recovery algorithm can then resume execution of the failed part of the application's execution if it can break the data dependences on failed part of the execution. Therefore, the recovery algorithm computes the set of recovery tasks that the computation was intended to execute.

If the example fails, an important question is whether the computation would have invoked the `AddFrameToPage` recovery task in the absence of a failure. We use static analysis of the abstract state transition graphs to determine the intended behavior of the failed part of the computation. A separate static analysis generates the abstract state transition graphs [8]. Figure 3 presents the abstract state transition graph for the `Frame` class. For every reachable object state, there is a node in this graph with the abstract state component of that state and an abstracted count of the tags of each type that are bound to the object. For example, the node labeled `2. plugin, frametag(1), pagetag(1)` represents objects in the `plugin` abstract state and that are bound to exactly one instance of both a `frametag` tag and a `pagetag` tag. Edges represent the possible transitions an object's state may make during the execution of a task. Double boundaries indicates that new objects can be allocated with that state.

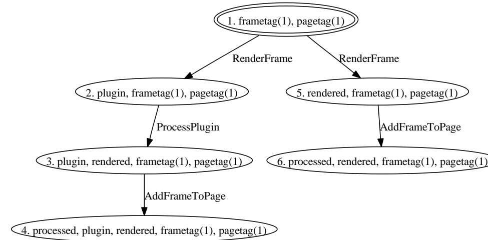

Fig. 3. Abstract State Transition Graph for the Frame Class

The recovery algorithm characterizes the intended behavior of the failed part of the computation. Note that the runtime plays a role in the execution of Bristlecone applications — it non-deterministically selects a task whose parameter guards are satisfied to invoke next. The analysis of the failed part of the execution can suppose that the runtime would have selected whichever schedule for the failed part of the computation that makes recovery easiest. Therefore for each reachable abstract object state, the static analysis computes the set of recovery tasks that the scheduler could cause the application to eventually execute with the object serving as an optional parameter. For each recovery task, it computes the possible states of the object when the task is invoked.

The analysis begins with the recovery tasks and then reasons backwards on the abstract state transition graph. The analysis operates as follows on the example:

1. It first analyzes the two base cases: objects in the state `5. rendered, frametag(1), pagetag(1)` can immediately serve as parameter objects for the optional parameter of the `AddFrameToPage` task. Similarly, objects in the state `3. plugin, rendered, frametag(1), pagetag(1)` can also immediately serve as parameter objects for the optional parameter of the `AddFrameToPage` task.

2. The analysis next reasons backwards and examines the state `2. plugin, frametag(1), pagetag(1)`. If a `FrameObject` reaches this state, the runtime can invoke the `processPlugin` task to place the object in the `3. plugin, rendered, frametag(1), pagetag(1)` state to which it can serve as a parameter object of the `AddFrameToPage` task.

3. The analysis finally examines the state `1. frametag(1), pagetag(1)`. The `RenderFrame` task can transition objects from this state into two different abstract states. Because the `AddFrameToPage` task can be executed from both final destination states, the runtime can cause objects in this state to serve as parameters of the `AddFrameToPage` task. Since the `RenderFrame` task decides the initial state transition, there remains uncertainty about the exact state of the recovery task's `Frame` parameter. We represent this uncertainty using the set {`3. plugin, rendered, frametag(1), pagetag(1), 5. rendered, frametag(1), pagetag(1)`} that includes both states.

2.5 Recovering From Failures

We use a hypothetical failure to illustrate the operation of the recovery algorithm. Suppose that the `RenderFrame` task dereferences a null pointer. The runtime first rolls back the `RenderFrame` task to return the heap to a consistent state. Then it performs the following steps to continue past the failure to render the web page:

1. Determine the possible destination states for the failed task: The runtime uses the static analysis results to determine that in the absence of the failure, this task would transition the `Frame` object into either the `2. plugin, frametag(1), pagetag(1)` state or the `5. rendered, frametag(1), pagetag(1)` state. Because the task failed, the runtime cannot determine which of these two states the `Frame` object would have transitioned.

2. Compute the recovery tasks in the intended execution: The runtime uses the static analysis results to determine a set of tasks for each state that the runtime could execute regardless of the application's behavior. The analysis results from the previous section show that the runtime could cause `Frame` objects in the `2. plugin, frametag(1), pagetag(1)` state to transition to the `3. plugin, rendered, frametag(1), pagetag(1)` state. In this state, they can serve as parameter objects for the optional parameter of the `AddFrameToPage` task. `Frame` objects in the `5. rendered, frametag(1), pagetag(1)` state can also serve as parameter objects for the optional parameter of the `AddFrameToPage` task.

3. Compute the intersection: Because the `RenderFrame` task failed, the runtime cannot determine the exact intended execution. However, if a recovery task

appears on all paths, the runtime can still safely execute that task. The analysis computes the intersection of the recovery task results from step 2 to determine that the runtime can cause the `AddFrameToPage` task to be executed. Because the failure prevents the runtime from discovering the path taken by the `RenderFrame` task, the runtime does not know the exact abstract state that the `Frame` object would have been in when the `AddFrameToPage` task executed. So the runtime represents the object's state with the set of possible states {`3. plugin, rendered, frametag(1), pagetag(1), 5. rendered, frametag(1), pagetag(1)`}.

4. **Execute the recovery task:** The runtime next executes the recovery task. Note that the `isavailable` predicate returns false indicating that the `Frame` object is not available because of a failure. The runtime marks the object as a *failed object*. The object's data is now inconsistent with its abstract state. Therefore, the data in that object can never be accessed. This means that the object cannot serve as a non-optional parameter object.
5. **Update the abstract states:** When the recovery task exits, the runtime updates the Frame object's set of states to {`4. processed, plugin, rendered, frametag(1), pagetag(1), 6. processed, rendered, frametag(1), pagetag(1)`}. The execution of tasks on the `Frame` object is now complete. In general, the runtime would compute the intersection of the sets of recovery tasks for all of the possible states that the `Frame` object may be in. The runtime would then execute one of the tasks in the intersection.

3 Static Analysis

The goal of the static analysis is to determine a failed computation's intended behavior.

3.1 Abstract State Transition Graphs

The analysis operates on the abstract state transition graphs that we developed in previous work [8]. A *abstract state node* represents the abstract state and tag components of an object's state — each node contains the states of all the abstracted object's abstract states and a 1-limited count (0, 1, or at least 1) of the number of tag instances of each type that are bound to the object. The abstract state transition graph contains abstract state nodes for each reachable abstract state. The abstract state transition graph contains a set of edges that abstract the actions of tasks on objects. There is an edge between two abstract state nodes if a task can be invoked on an object in the abstract state corresponding to the source node and the task could transition the object into the abstract state corresponding to the destination node.

Abstract state nodes $n \in N$ abstract the reachable abstract states. The set T represents the set of tasks. The set $P \subseteq T \times \mathbb{N}$ represents the set of combinations of tasks and parameter indices for the invocation of a task on an object. The set of edges $E \subseteq N \times P \times N$ represents the possible transactions of an object's abstract state.

3.2 Analysis Abstraction

The analysis computes the *recovery function* $r : N \rightarrow 2^O$ that maps abstract state nodes to their corresponding *recovery set*. A recovery set is the set of recovery tasks invocations for which there exist a scheduling strategy that ensures that the computation

will eventually invoke the task on the object abstracted by the state transition graph. $O \subseteq T \times \mathbb{N} \times 2^N$ is the set of recovery task invocations. Each recovery task invocation $o = \langle t, i, s \rangle \in O$ consists of a task t, the optional parameter index i, and the set s of parameter object abstract states at invocation. These states represent the possible states of the object at task invocation if the recovery task is invoked.

The dataflow lattice is the standard lattice for sets: the elements of these sets are sets of recovery task invocations, meet is set union, and the subset relation defines the partial order. The analysis is a fixed-point algorithm on the abstract state transition graph.

3.3 Transfer Function

We next describe the transfer function for computing the set of recovery task invocations for an abstract state node $n \in N$. There are two sources of uncertainty in the abstract state transition graph: (1) there is uncertainty in how a task's execution will change an object's state and (2) there is uncertainty in the task the runtime chooses to invoke. The Section *Results for a Single Task Invocation* describes how we handle the first type of uncertainty in detail. The Section *The Runtime's Choice of Task* describes how we handle the second type of uncertainty. We first describe the basic transfer functions. We later extend the basic analysis to support tags and multiple parameter tasks in Sections 3.4 and 3.5, respectively.

Results for a Single Task Invocation We represent a task invocation on an object using the pair $\langle t, i \rangle \in P$ where t is the task and i is parameter that references the object. For each task invocation $p = \langle t, i \rangle \in P$ the algorithm computes the set of recovery task invocations that can break data dependences if the failed part of the computation includes p. We consider the following two possible cases:

Optional Parameter Case: If parameter i of task t is optional, the set of recovery task invocations for the invocation of the task-parameter pair p on the abstract state n is $\{\langle t, i, \{n\} \rangle\}$.

Normal Case: Otherwise, the algorithm computes the set of possible destination abstract states $N_{dst_n} = \{n_{dst} \mid \langle n, p, n_{dst} \rangle \in E\} = \{n_{dst_1}, ..., n_{dst_m}\}$. Because the runtime does not choose the destination state of a task, the set of recovery task invocations for p can only include combinations of recovery task t_{opt} and optional parameter i_{opt} that appear in the recovery sets of all destination states. The set of recovery task invocations for the task invocation p on n is therefore: $\{\langle t_{opt}, i_{opt}, s_1 \cup ... \cup s_m \rangle \mid \langle t_{opt}, i_{opt}, s_1 \rangle \in r(n_{dst_1}), ..., \langle t_{opt}, i_{opt}, s_m \rangle \in r(n_{dst_m})\}$. The recovery task invocation's set of abstract states is equal to the union of all the component sets of abstract states $\{s_1, ..., s_m\}$ because the analysis cannot determine the destination state of the task invocation p and therefore cannot determine the exact state that an object would be in when it serves as the i_{opt} parameter of the task t_{opt}.

The Runtime's Choice of Task When an abstract state has more than one possible task invocation, the runtime can choose which task to invoke. To compute the set of recovery task invocations for the abstract state n, the analysis first computes the set of recovery task invocations for each pair of task t and parameter i that can be invoked on the abstract state n. The set of recovery task invocations for n is the union of these sets.

3.4 Multiple-Parameter Tasks

Tasks that operate on multiple-parameters pose extra challenges. Because the abstract state transition graph only characterizes the application's behavior with respect to a sin-

gle object, the runtime must ensure that all other parameter object guards for a multiple-parameter task are satisfied. Moreover, a multiple-parameter task could potentially introduce inconsistencies in other object's states if the abstract states of some parameter objects were updated and another parameter object's abstract states were not. For example, if the abstract states of other parameter objects were updated without actually executing the multiple-parameter task, it would likely introduce inconsistencies between the other object's data and the states of its abstract states. If the runtime declared the other objects as failed, the recovery attempt could cause the loss of key data structures. To avoid these issues, the analysis conservatively omits multiple-parameter tasks that change the abstract states or tag bindings of other parameters. Note that omitting these tasks is safe, it simply reduces how much of the computation can be recovered.

We have extended the transfer function for multiple-parameter tasks to add predicates to recovery task invocations. These predicates verify that the heap contains objects that satisfy the guard expressions for the other parameters of the task. The runtime uses these predicates to check whether an execution path involving a multiple-parameter task is feasible, and therefore that the corresponding recovery task can be executed.

3.5 Tag Bindings

Another complication is that a task in the failed part of the execution may bind a new tag instance to an object. Because Bristlecone cannot determine the exact tag instance that would have been bound, the static analysis must conservatively handle this case. We have extended the transfer function to omit recovery task invocations if the current task binds a tag descriptor of the same type as the tag guards that appear either in the recovery task's guard expressions or in any tag guard predicates in the recovery task invocation. Note that omitting these invocations is safe, it simply reduces the set of possible recoveries that the system can generate.

4 Recovery Algorithm

The runtime should only invoke a recovery task on a failed object when the intended execution would have executed that task. Because the failure prevented part of the computation from executing, the analysis may not be able to determine the exact abstract state of the failed object, but only that the object's abstract state satisfies the recovery task's guard. There are two sources of uncertainty in the abstract state of a failed object:

- **Uncertainty from Failed Tasks:** A task can have multiple exits and therefore potentially transition its parameter objects into different abstract states. Because the runtime cannot determine which exit a failed task would have taken had it not failed, the runtime must conservatively assume that the task could take any of the exits. The recovery algorithm represents this uncertainty using a *possible abstract state set* $S_F = \{n_1, ..., n_j\} \subseteq 2^N$ that contains all possible abstract states that the tasks could have transitioned the parameter objects into. A recovery task can only be invoked on a possible abstract state set if it can be invoked on all of its component abstract states.
- **Uncertainty from the Runtime:** If the runtime would have had a choice between multiple tasks to invoke on an object in a failure-free execution, the runtime can use the same freedom to make recovery easier. Because the choice of which failed task the runtime executes does not have an immediate side-effect, the runtime can

delay this choice. This delay gives the runtime extra flexibility in recovery and provides a beneficial source of uncertainty in an object's state. The recovery algorithm represents this uncertainty source with a *choice set* $C = \{S_{F_1}, ..., S_{F_m}\} \subseteq 2^{2^N}$ of choices between many possible abstract state sets. A recovery task can be invoked on a choice set if it could be invoked on at least one of the component possible abstract state sets. When a recovery task is invoked, its guards constrain the abstract states of the parameter objects and may force the runtime to commit to a specific choice of task scheduling for the failed part of the computation.

4.1 Task Invocation

Task invocation during normal execution is conceptually straightforward — the runtime maintains the current state of the objects and invokes tasks on these objects when the objects satisfy the task's guards. Our previous work describes efficient runtime techniques for task invocation. In this section, we extend this work to support recovery tasks by tracking the states of failed objects. We first describe how the runtime uses static analysis to compute the set of recovery tasks that can be executed on a failed task's parameter objects. We then describe how, after a recovery task completes execution on a failed object, the runtime uses the static analysis results to compute the set of recovery tasks it can execute next on the failed object.

Failed Tasks This section describes the actions taken by the runtime when task t fails with its ith parameter object o in the state given by the choice set $C = \{\{n_{11}, ..., n_{1k_1}\}, ..., \{n_{j1}, ..., n_{jk_j}\}\}$.[3] The runtime first computes which recovery tasks could have been executed had task t not failed. It also characterizes the possible states of the object o at the time these recovery tasks would have been invoked.

The runtime computes the function $R \subseteq T \times \mathbb{N} \rightarrow 2^{2^N}$ that characterizes the set of possible recovery task invocations. We define the function $\mathcal{O} = \mathcal{T}(t, i, n)$ to return the set of recovery task invocations $\mathcal{O}$ for a failure of task t on the ith parameter object in state n. The runtime uses the procedure described in Section 3.3 to compute $\mathcal{T}$ from the static analysis results.

The operator $\diamond$ models the effects of the uncertainty of the failed task's execution by conservatively combining the sets of recovery task invocations – a recovery task is in the combination only if it appears in both sets. Formally, we define $\mathcal{O}_1 \diamond \mathcal{O}_2 = \{\langle t', i', S\rangle \mid \langle t', i', S_1\rangle \in \mathcal{O}_1 \wedge \langle t', i', S_2\rangle \in \mathcal{O}_2, S = S_1 \cup S_2\}$. We use the $\diamond$ operator to compute the set of possible recovery task invocations for an object in the possible abstract state $S_F = \{n_1, ..., n_j\} \subseteq 2^N$ that served as parameter i during a failure of task t as $\mathcal{T}(t, i, n_1) \diamond ... \diamond \mathcal{T}(t, i, n_j)$. We use the set union operator to extend this computation to choice sets — the algorithm computes the set of possible recovery task invocations $\mathcal{C} = \big(\mathcal{T}(t, i, n_{11}) \diamond ... \diamond \mathcal{T}(t, i, n_{1k_1})\big) \cup ... \cup \big(\mathcal{T}(t, i, n_{j1}) \diamond ... \diamond \mathcal{T}(t, i, n_{jk_j})\big)$. We define $R(t, i) = \{S \mid \langle t, i, S\rangle \in \mathcal{C}\}$. The function R gives for each possible recovery task invocation $\langle t, i\rangle$ that can be enqueued, the choice set that characterizes the failed object's state. Note that the object remains enqueued in any previous task queues.

[3] A non-trivial choice set can appear after a failure of a recovery task invocation during the process of recovery. The recovery algorithm continues to try to break other data dependences at future recovery tasks that access the object. Note that the parameter objects of a normal failed task will be in a trivial choice set $C = \{\{n_{11}\}\}$.

Recovery Tasks on Failed Objects This section describes the actions the runtime takes to execute a recovery task on a failed parameter object. The runtime starts the task's execution with the object in the state computed in the previous section for the task invocation. When the task exits, the runtime updates each of the object's possible states with the abstract states and tag changes from the `taskexit` statement. The runtime then removes the parameter objects from all task queues. If the parameter object is in a non-failed state, the runtime enqueues the object in the task queues. Otherwise, for a failed parameter object in the state given by the choice set $C = \{\{n_{11}, ..., n_{1k_1}\}, ..., \{n_{j1}, ..., n_{jk_j}\}\}$ the algorithm uses the recovery function r to compute $\mathcal{C} = \big(r(n_{11}) \diamond ... \diamond r(n_{1k_1})\big) \cup ... \cup \big(r(n_{j1}) \diamond ... \diamond r(n_{jk_j})\big)$. We define $R(t,i) = \{S \mid \langle t, i, S\rangle \in \mathcal{C}\}$. The algorithm then uses R to determine, for each possible recovery task invocation $\langle t, i\rangle$ that can be enqueued, the corresponding choice set.

5 Experience

We next discuss our experiences using recovery tasks to develop three robust software applications: a multiplayer game, a web portal, and a simplified MapReduce framework. We have implemented the enhanced recovery algorithm with support for recovery tasks in the Bristlecone compiler and runtime. The source code for the compiler, runtime, and benchmarks is available at `http://demsky.eecs.uci.edu/software.php`.

For each benchmark, we developed three versions: a recovery task version, a standard Bristlecone version without recovery tasks, and a Java version.

We used randomized failure injection to simulate the effects of software faults. The compiler inserts failure injection code after every instruction in the generated code. We inject exactly one failure into each execution at a random instruction. The failures we injected simulate the entire class of software faults that cause failures in the same task that contains the fault. This fault class includes illegal memory accesses, failed assertions, failed data structure consistency checks, library errors, and arithmetic exceptions.

We developed this randomized failure injection strategy to avoid biases that hand-selected faults may introduce. Note that our randomized failure injection strategy likely represents an unrealistically harsh metric — it may inject faults that are extremely difficult to recover from, but are unlikely to occur in practice. For example, it sometimes injects failures into simple, completely deterministic startup code. While such injected failures cause the Bristlecone versions to fail to recover because the entire application depends on the startup code, they are unlikely to occur in practice as they would have been caught the first time the application was executed.

5.1 Multiplayer Game

The multiplayer game benchmark is a simplified version of larger scale multiplayer online games. Software bugs have been a recurring problem for many of these games. Our game consists of a world with both humans and monsters. Humans try to escape through exits while monsters try to capture the humans. The game contains AI components that use search algorithms to plan the moves for both monsters and humans. The recovery task version uses a recovery task to collect the players' moves and update the map.

In the process of developing the AI code, which is shared across all versions, we made an unintentional coding mistake that could cause an out-of-bounds array access

under certain circumstances. The recovery task version recovered from this bug while the other two versions crashed. While this experience is only a single anecdote, we found it to be an encouraging validation of the approach.

Our workload was running the game with all players controlled by the AI. We performed 100 trials of the experiment on each of the three versions. We found that using recovery tasks enabled the recovery task version of the game to survive the injected failure in all 100 trials. We found that in the presence of errors, the standard Bristlecone and Java versions were unable to continue the game.

5.2 Web Portal

The web portal models a category of applications that perform independent computations, combine the results, and then display some aggregation to the user. When a web browser requests the portal page, the web portal generates requests for the current weather conditions, stock prices, and the Google home page. Finally, the web portal combines the results from the individual responses into a single page and serves this page to the browser. The recovery task version enhances the data combination phase to enable recovery from failures that make parts of the information unavailable.

Our workload consisted of using a web browser to view the portal web page. We performed 20 trials of the experiment on each of the three versions. We found that using recovery tasks enabled the web portal to serve the unaffected parts of the web portal page in 17 of the 20 trials. We found that in the presence of errors, the standard Bristlecone and Java versions were unable to serve the portal web page. However, all three versions were able to isolate errors to a single request — all versions of the web portal were able to serve future page requests after a failure.

5.3 MapReduce Framework

MapReduce provides an abstract programming model for parallel computations on large data sets [7]. Users specify the computation in terms of a map function and a reduce function and MapReduce automatically parallelizes the computation across machines.

We implemented a simplified MapReduce framework. The implementation partitions the input, invokes the map function, aggregates intermediate results, invokes the reduce function, and aggregates the final results. The recovery task version uses recovery tasks for aggregating the map and reduce results.

Our workload counts the occurrences of each word in a text file. We performed 100 trials on each of the three versions. For each trial, we recorded whether the final output was generated. Without failure injection, all of the versions generated the final output. With the injected failures, the recovery task version produced the final output in 93 trials while the other two versions failed in all trials. When the recovery task version failed, it warned the user that the word counts could potentially be low. We expect that users will often find the output useful as it represents a lower bound on word counts.

We divide the injected failures into three categories: (1) failures that affect map workers, (2) failures that affect reduce workers, and (3) failures that affect the tasks that coordinate the computation. We observed 86 executions in the first category. The effect of these errors was to cause word counts to be low or missing — in these executions the counts were low by an average of 5%. We observed 7 executions in the second category. The effect of these errors was to cause word counts for some words to be missing. We

observed that 86 words out of 6,213 total words were missing on average from these executions. The 7 failed executions fall in the third category.

5.4 Discussion

We measured the execution time of both the recovery task version and the Java version of the multiplayer game and MapReduce benchmarks. We omit a performance evaluation for the web portal because of the difficulty of measuring its performance given that the portal accesses remote web servers. The Bristlecone version of the MapReduce benchmark running on a RAM disk took 0.63 seconds to execute while the Java version took 0.58 seconds. The recovery task version of the multiplayer game benchmark took 0.94 seconds to execute while the Java version took only 0.63 seconds.

In general, we have found writing Bristlecone applications to be straightforward — most of the code was shared with the Java version. The Bristlecone versions of the benchmarks were comparable in length to the Java versions. The recovery task version of MapReduce framework contains 20% fewer lines of code than the Java version, the recovery task version of multiplayer game contains 5% more lines of code, and the recovery task version of web portal contains 6% more lines of code of which about one third were simply abstract state declarations.

6 Related Work

Recovery blocks [1] and N-version programming [3] are two classic approaches to fault tolerance. These approaches add significant software development costs. Bristlecone is designed to provide fault tolerance for applications that cannot afford the development costs associated with these classic techniques.

Backward recovery uses a combination of checkpointing and acceptance tests to prevent a software system from entering an incorrect state [5]. Forward recovery uses multiple copies of a computation to recover from transient errors [11]. Unfortunately, it can be difficult to handle deterministic failures with these methods as the same error will likely cause the software system to repeatedly fail.

The Recovery-Oriented Computing project has explored systems out of a set of individually rebootable components [4]. Researchers have used retry with reconfiguration to address configuration issues [13]. Contract-based data structure repair [15] is an alternative approach to tolerating failed components.

A key component of Bristlecone is decoupling unrelated conceptual operations and tracking data dependences between these operations. Dataflow computations also keep track of data dependences between operations so that the operations can be parallelized [12]. Errors in a dataflow computation could easily cause key data structures to be lost. Bristlecone's abstract state and tag constructs allow data structures to passively persist across failures.

Tuple-space languages, such as Linda [9], decouple computations to enable parallelization. The threads of execution communicate through primitives that manipulate a global tuple space. However, these language were not designed to address software errors — software errors can permanently halt threads of execution in these languages causing the system to eventually fail.

Orc [6] and Oz [14] are other examples of task-based languages. This work is largely orthogonal as they are not designed for fault tolerance. Actors are a concurrent programming paradigm in which applications are architected as several actors that communicate

through messages [10]. Actors are note designed for fault tolerance and failures may cause actors to drop messages and corrupt or lose their state.

Erlang has been used to implement robust systems using a software architecture containing a set of supervisors and a hierarchy of increasingly simple implementations of the same functionality [2]. Bristlecone is complementary to the supervisor approach — while the supervisor approach gives the developer complete control over recovery, it requires the developer to manually develop multiple implementations of the same functionality. Bristlecone requires only minimal additional development effort.

7 Conclusion

We have presented an analysis that reasons about the effects of potential failures and a recovery algorithm that uses the analysis results to determine how to recover the application from the failure. Our experience shows that the new technique recovers significantly better from failures for our benchmarks. Moreover, we found it straightforward to use this technique to develop applications, and that it did not significantly affect either the complexity or length of the benchmarks.

Acknowledgments. This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments.

References

1. T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *ICSE*, 1976.
2. J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Swedish Institute of Computer Science, November 2003.
3. A. Avizienis. The methodology of N-version programming, 1995.
4. G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *HotOS-VIII*, 2001.
5. K. M. Chandy and C. Ramamoorthy. Rollback and recovery strategies. *IEEE Transactions on Computers*, C-21(2):137–146, 1972.
6. W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Coordination*, 2006.
7. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.
8. B. Demsky and S. Sundaramurthy. Bristlecone: Language support for robust software applications. *To Appear in TSE*, 2010.
9. D. Gelernter. Generative communication in Linda. *TOPLAS*, 7(1):80–112, 1985.
10. C. Hewitt and H. G. Baker. Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.
11. K. Huang, J. Wu, and E. B. Fernandez. A generalized forward recovery checkpointing scheme. In *FTPDS*, April 1998.
12. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1), 2004.
13. F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *SOSP*, 2005.
14. G. Smolka. The Oz programming model. In *JELIA*, page 251, 1996.
15. R. N. Zaeem and S. Khurshid. Contract-based data structure repair using alloy. In *ECOOP*, 2010.