Bristlecone: A Language for Robust Software Systems

Brian Demsky and Alokika Dash

University of California, Irvine

Abstract. We present Bristlecone, a programming language for robust software systems. Bristlecone applications have two components: a high-level organization description that specifies how the application's conceptual operations interact, and a low-level operational description that specifies the sequence of instructions that comprise an individual conceptual operation. Bristlecone uses the high-level organization description to recover the software system from an error to a consistent state and to reason how to safely continue the software system's execution after the error.

We have implemented a compiler and runtime for Bristlecone. We have evaluated this implementation on three benchmark applications: a web crawler, a web server, and a multi-room chat server. We developed both a Bristlecone version and a Java version of each benchmark application. We used injected failures to evaluate the robustness of each version of the application. We found that the Bristlecone versions of the benchmark applications more successfully survived the injected failures.

1 Introduction

Software faults pose a significant challenge to developing reliable, robust software systems. The current approach to addressing software faults is to work hard to minimize the number of software faults through development processes, automated tools, and testing. While minimizing the number of software faults is a critical component in the development process for reliable software, it is not sufficient: the faults that inevitably slip through the development and testing processes will still cause deployed systems to fail.

The Lucent 5ESS telephone switch, the Ericsson AXD301 ATM switch, and the IBM MVS operating system are examples of critical systems that use recovery routines to automatically recover from software failures [1, 2]. The software in these systems contains a set of manually coded recovery procedures that detect errors and then take actions to automatically recover from the errors. The reported results indicate that the recover routines can provide an order of magnitude increase in the reliability of these systems [3]. This additional reliability comes at a significant additional development cost — the recovery routines for the Lucent 5ESS telephone switch constitute more than 50% of the switch's software [4]. As a result of these high costs, recovery procedures have been primarily relegated to the domain of critical infrastructure software that can justify the cost. A wide range of other applications including desktop applications such as web browsers, office applications, games, servers, and control systems could potentially benefit from lower-cost automated recovery. The goal of Bristlecone is to provide a lower-cost approach to software recovery that will enable a larger class of applications to benefit from this technique.

The key inspiration for this research is the observation that many software errors propagate through software systems to cause further damage either through data structure corruption or control-flow–induced coupling between conceptual operations. We have developed Bristlecone, a programming language for robust software systems, to address the error propagation problem. The basic idea is to address error propagation by having developers write software systems as a set of decoupled tasks with each task encapsulating an individual conceptual operation. The developer also provides specifications that describe how these decoupled tasks interact and optionally what consistency properties should hold for data structures. The runtime checks for data structure consistency violations and monitors for illegal operations (such as illegal memory accesses or arithmetic errors) to detect software errors. If the runtime detects an error in the execution, the runtime rolls back the data structures to their state at the beginning of the task's execution, and then uses the task specifications to adapt the execution of the software system to avoid re-executing the same error and make forward progress.

Alternatively, we can view Bristlecone as a programming language that allows for a large space of possible execution paths for any given software system with an implicit ordering of how desirable any given path is. If the most desirable path results in an error, the runtime rolls back the execution enough to follow a different path thereby avoiding the error. The result is a robust software system that can continue to successfully provide service even in the presence of errors.

1.1 Bristlecone Language

Figure 1 gives an overview of the components in the Bristlecone system. We can view software systems as a composition of thousands of conceptual operations — in practice, the correct execution of any conceptual operation is likely to be independent of many of the other conceptual operations. However, many traditional programming languages force developers to linearize the conceptual operations of a software system. This linearization tightly couples these conceptual operations: if one conceptual operation fails, it becomes unclear how to safely execute any future conceptual operations.

Bristlecone avoids artificially coupling operations by providing the developer with the *task* program construct. The developer uses a task to encompass a single conceptual operation. Tasks are represented in Figure 1 as rectangles. A set of task specifications loosely couple the tasks together. Each task contains a task specification that the runtime uses to determine (1) when to execute the task, (2) what data the task needs, and (3) how the task changes the role this data plays in the computation. If a task fails, the runtime uses the task specifications to reason how to adapt the future execution of the software system so that the execution does not depend on the failed task.

Bristlecone contains the following components (represented by rounded boxes in the figure):

- **Bristlecone Compiler:** The Bristlecone compiler compiles the tasks and task specifications into C code. Our implementation then uses the gcc C compiler to generate executables. The ellipse labeled `Compiled Tasks` represents the compiled tasks.

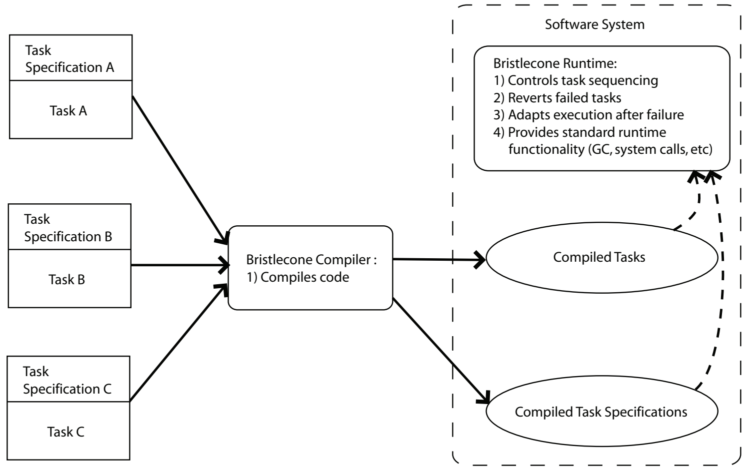

Fig. 1. Overview of the Bristlecone System

- **Runtime:** The runtime uses the compiled code and compiled specifications generated by the compilers (represented by the ellipses in the figure) to execute the software system. It uses the consistency checker to detect errors that silently corrupt data structures. The runtime then uses rollback to recover consistent data structures if it detects a software error. Finally, it uses the task specifications to determine when to execute the tasks and how to recover from errors.

1.2 Scope

Bristlecone is not suitable for all software systems. Certain computations, such as some scientific simulations, are inherently tightly coupled. While Bristlecone may detect errors in such software systems, it is unlikely to enable these systems to recover in any meaningful way. For other computations, it may be desirable for a software system to shut down rather than deviate from a specific designed behavior or produce a partial result.

Bristlecone is designed for software systems that place a premium on continued execution and that can tolerate some degradation from a specific designed behavior. For example, we expect that Bristlecone will be useful for financial server software, e-commerce systems, office applications, web browsers, online game servers, sensor networks, and control systems for physical phenomena. For applications like finance, Bristlecone can be used to develop software systems that only process error-free transactions and back out all changes that corrupt data structures, while still ensuring that cosmetic errors do not cause potentially expensive downtime. Ultimately, the software developer must decide whether using this approach is reasonable for a given software system.

This decision could depend on the environment in which a system is deployed. For example, in systems with redundant backup systems, we expect that developers would design the primary system to fail-fast and the backup system to be robust in the presence of errors.

1.3 Contributions

This paper makes the following contributions:

- **Bristlecone Language:** It presents a programming language which exposes both the conceptual operations and the ordering and data dependences between these conceptual operations to the compiler and runtime system.
- **Recovery Strategy:** It presents a strategy for repairing the damage caused by a software error and adapting the software system's execution in response to the error to enable it to safely continue execution.
- **Experience:** It presents our experience using Bristlecone to develop three robust software systems: a web crawler, a web server, and a multi-room chat server. For each benchmark, we developed both a Bristlecone version and a Java version. We designed the Java versions to be resilient: they use threads to tolerate failures. Our experience indicates that the Bristlecone versions are able to successfully recover from significantly more of the injected failures.

The remainder of the paper is structured as follows. Section 2 presents an example that illustrates our approach. Section 3 presents the Bristlecone languages. Section 4 presents the runtime system. Section 5 presents our experience using Bristlecone to develop several robust software applications. Section 6 discusses related work; we conclude in Section 7.

2 Example

We next present a web server example that illustrates the operation of Bristlecone. This web server has specialized e-commerce functionality and maintains state to track inventory.

As the example web server executes, the conceptual state or role of objects in the computation evolves. This evolution changes the way that the software system uses the object and can change the functionality that the object supports. For example, the Java `connect` method changes the functionality of a `Socket` object in a computation: after the `connect` method is invoked, data can be written to or read from that `Socket` object.

The Bristlecone language provides *flags* to track the conceptual state of an object. The runtime uses the conceptual state of the object as indicated by the object's flag to determine which conceptual operations or *tasks* to invoke on the given object. When a task exits, it can change the values of the flags of its parameter objects.

2.1 Classes

Figure 2 gives part of the `WebRequest` class definition. The web server example uses instances of the `WebRequest` class to manage connections to the web server. The `WebRequest` class definition declares three flags: the `initialized` flag, which indicates whether the connection is in the initial state; the `file_req` flag, which indicates that the server has received a file request from this client connection; and the `write_log` flag, which indicates whether the connection information is available for logging.

```
class WebRequest {
  /* This flag indicates that the WebRequest object is in its
     initial state. */
  flag initialized;

  /* This flag indicates that the system has received a request
     to send a requested file. */
  flag file_req;

  /* This flag indicates that the connection should be logged. */
  flag write_log;
  ...
}
```

Fig. 2. WebRequest Class Declaration

In many cases, the developer may need to invoke a task on multiple objects that are related in some way. Bristlecone provides a tag construct, which the developer can use to group objects together. New tag instances are created using tag allocation statements of the form `tag tagname=new tag(tagtype)`. Such a tag allocation statement allocates a new tag instance of type `tagtype` and assigns the variable `tagname` to this tag instance. The developer can tag multiple objects with a tag instance to group them, and then use that tag instance to ensure that the runtime invokes a task on two or more objects in the group defined by the tag instance. The developer can tag an object by including the statement `add tagname` in an object allocation site to tag the newly allocated object or in a `taskexit` statement to tag a parameter object. The example uses the `connection` tag to group a `WebRequest` object with the corresponding `Socket` object that provides the TCP connection for that web request. Tag instances can be added to objects when the object is allocated, and they can be added or removed to or from a task's parameter objects when the task exits.

2.2 Tasks

Bristlecone software systems consist of a collection of interacting tasks. The key difference between tasks and methods is that the runtime invokes a task when the heap contains objects with the specified flag settings to serve as the task's parameters. Note that while the runtime controls task invocation, tasks can call methods. The runtime uses a task's specification to determine which objects serve as the task's parameters and when to invoke the task.

Each task declaration consists of the keyword `task`, the task's name, the task's parameters, and the body of the task. Figure 3 gives the task declarations for the web server example. We indicate the omission of the Java-like imperative code inside the task declarations with ellipses. The first task declaration declares a task named `startup` that takes a `StartupObject` object as a parameter and points the parameter variable `start` to this object. The declaration also contains a guard that states that the `StartupObject` object must have its `initialstate` flag set before the runtime can invoke this task. The runtime invokes the task when there exist parameter objects in the heap that satisfy the parameters' guard expressions. Before exiting, the `taskexit` statement in the `startup` task resets the `initialstate` flag in the `StartupObject` to false to prevent the runtime from repeatedly invoking the `startup` task.

```
/* This task starts the web server */
task startup(StartupObject start in initialstate) {
  ...
  ServerSocket ss=new ServerSocket(80);
  Logger l=new Logger() (initialized:=true);
  taskexit(start: initialstate:=false);
}

/* This task accepts incoming connection requests and creates a
   Socket object. */
task acceptConnection(ServerSocket ss in pending_socket) {
  ...
  tag t=new tag(connection);
  WebRequest w=new WebRequest(...)(initialized:=true, add t);
  ss.accept(t);
  ...
}

/* This task reads a request from a client. */
task readRequest(WebRequest w in initialized with connection t,
     Socket s in IO_Pending with connection t) {
  ...
  if (received_complete_request)
    taskexit(w: initialized:=false, file_req:=true,
      write_log:=true);
}

/* This task sends the request to the client. */
task sendPage(WebRequest w in file_req with connection t,
     Socket s with connection t) {
  ...
  taskexit(w: file_req:=false);
}

/* This task logs the request. */
task logRequest(WebRequest s in write_log, Logger l in
     initialized) {
  ...
  taskexit(s: write_log:=false);
}
```

Fig. 3. Flag Specifications for Tasks

Task declarations can contain constraints on tag bindings to ensure that the parameter objects are related. A tag binding constraint contains the keyword `with` followed by the type of the tag and the tag variable. For example, the task declaration `task readRequest(WebRequest w in initialized with connection t, Socket s in IO_Pending with connection t)` ensures that the runtime only invokes the `readRequest` task on a set of parameter objects in which the first parameter object is bound to an instance of a `connection` tag and the second parameter object is bound to the same `connection` tag instance. When the task executes, the tag variable `t` is bound to that `connection` tag instance.

2.3 Error-Free Execution

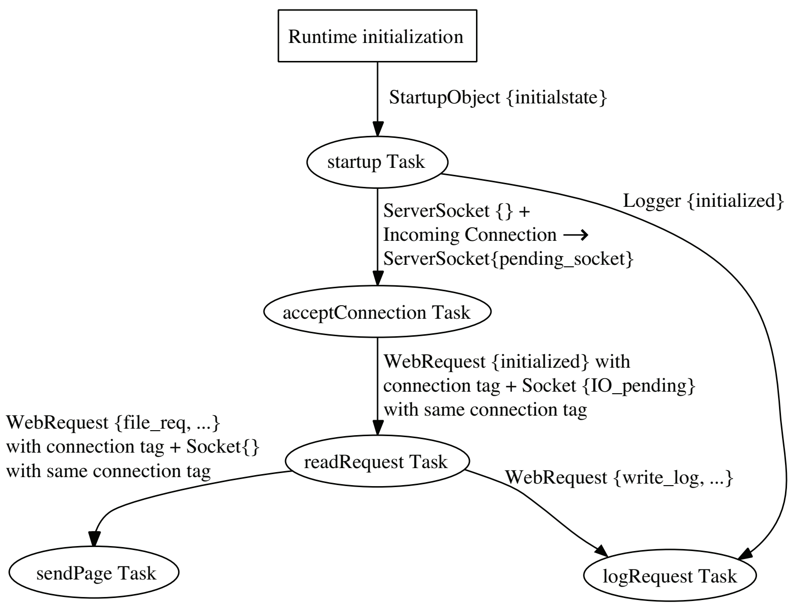

Fig. 4. Task Diagram for the Web Server

Figure 4 gives a diagram of the dependences between tasks in the web server example. The ellipses in the diagram represent tasks and the edges represent the control and data dependences between the tasks. The rectangle labeled `Runtime initialization` represents the initialization performed by the Bristlecone runtime. From this diagram, we can see that the web server performs the following operations in an error-free execution (although not necessarily in this order):

1. **Startup:** When a Bristlecone program is executed, the Bristlecone runtime creates a `StartupObject` object and then sets its `initialstate` flag to true. Setting this flag causes the runtime to invoke the `startup` task in our example. Note that the code never explicitly calls a task. Instead, the runtime keeps track of the status of the flags of objects in the heap and invokes a task when the heap contains objects with the specified flag settings to serve as parameters.

   When the runtime invokes the `startup` task, the `startup` task creates a `ServerSocket` object to accept incoming connections to the web server. Next, it creates a `Logger` object to manage logging web page requests and sets its `initialized` flag to indicate that the object is ready to provide logging functionality. Finally, it resets the `StartupObject` object's `initialstate` flag to false to prevent the runtime from repeatedly invoking the `startup` task.

2. **Accepting an Incoming Connection:** At some point, the web server will receive an incoming connection request from a web browser. This causes the runtime to set the `ServerSocket` object's `pending_socket` flag to true, which in turn causes the runtime to invoke the `acceptConnection` task with this `ServerSocket` object as its parameter. The `acceptConnection` task creates a `WebRequest` object to store the connections state and calls the accept method on the `ServerSocket` to create a `Socket` object to manage communication with the web browser. Note that the `acceptConnection` task creates a new `connection` tag instance to group the `Socket` object and `WebRequest` object together by binding this tag instance to the `WebRequest` object and then passing this tag instance into the `accept` method to bind the newly created `Socket` object.
3. **Reading a Request:** After a connection is established, the client web browser sends a web page request to the server. In response to this incoming web page request, the runtime sets the Socket object's `IO_pending` flag to true[1], which in turn causes the runtime to invoke the `readRequest` task. The `readRequest` task checks whether the server has received the complete request.[2] If it has received the complete request, it sets both the `file_req` flag and the `write_log` flag to true and resets the `initialized` flag to false. These flag changes cause the runtime to eventually invoke both the `sendPage` and the `logRequest` tasks and prevents repeated invocations of the `readRequest` task on the same object.
4. **Sending the Page:** The runtime invokes the `sendPage` task when the `WebRequest` object's `request_processed` flag is set to true. The `sendPage` task then reads the requested file and sends the contents of the file to the client browser. The `sendPage` task then resets the `received_request` flag to false to prevent repeated invocations of the `sendPage` task.
5. **Logging the Request:** The runtime invokes the `logRequest` task when both the `WebRequest` object's `write_log` flag is set to true and the `Logger` object's `initialized` flag is set to true. The `logRequest` task writes a log entry to record which web page was requested. The `logRequest` task then resets the `write_log` flag to false to prevent repeated invocations of the `logRequest` task.

[1] The `IO_pending` flag is declared with the `external` keyword to indicate that the runtime manages setting and clearing this flag. The current runtime implementation of Bristlecone is single-threaded and, therefore, uses non-blocking I/O. Future runtime implementations will support multiple concurrent tasks and (transactional) blocking I/O [5].

[2] Note that it is possible for a client browser to split a long request across multiple packets and therefore it may be necessary to invoke the `readRequest` task multiple times to receive a single request.

2.4 Error Handling

The Bristlecone runtime uses task specifications to automatically recover from errors. For example, suppose that the `logRequest` task fails while updating the `Logger` object. If the web server were written in a traditional programming language, it could be difficult to recover from such a failure. While some traditional languages provide exceptional handling mechanisms, using them effectively is challenging — the developer must both identify which failures are likely to occur and reason about how to recover from those failures. Alternatively, the program could simply ignore the failure. Unfortunately, if the web server were to simply ignore the failure, it could easily leave the `Logger` object in an inconsistent state, possibly eventually causing a catastrophic failure later.

To address this issue Bristlecone tasks have transactional semantics — upon failure, the Bristlecone runtime aborts the enclosing transaction to return the affected objects, including the `Logger` object, to consistent states. The runtime then records that the `logRequest` task failed when invoked on the combination of those specific `WebRequest` and `Logger` objects. The runtime uses this record to avoid re-executing the same specific failure. At this point, the Bristlecone runtime has returned the web server to a known consistent state and must now determine how to safely continue the web server's execution.

The traditional problem with using transactions to recover from deterministic software faults is that after aborting a transaction the software system cannot make forward progress — retrying the same transaction will cause the system to repeat the same failure. Bristlecone solves this problem by using the flags, tags, and task specifications to determine which other tasks are safe to execute after the error. Although the software fault prevents the system from logging this request, since the `file_req` flag is set to true, the task specification for the `sendPage` task allows the runtime to invoke the `sendPage` task. Therefore, the runtime can still safely serve the web page request.

The end result is that the software system is able to safely continue to execute even in the presence of software errors. Bristlecone is able to successfully isolate the effects of the error to a minimal part of the web server's execution — only a single task is aborted and the abort is logged. Without Bristlecone, the web server could potentially leave the `Logger` object in an inconsistent state, possibly causing the web server to fail to log future requests. If the web server written in a conventional language was designed to log request before serving a request, corruption of the log data structure could even cause the server to stop serving requests.

3 Language Design

The Bristlecone language includes a task specification language that describes how to orchestrate task execution. Bristlecone introduces object flags to store the conceptual state of the object. Each task contains a corresponding task specification that describes which objects the task operates on, when the task should execute, and how the task affects the conceptual state of objects.

Bristlecone is an object-oriented, type-safe language with syntax similar to Java. Figure 5 presents the grammar for Bristlecone's task extensions to Java. We omit the Java-like imperative component of Bristlecone from the grammar to save space. The

flagdecl	:=	`flag` *flagname*; | `external flag` *flagname*;
tagdecl	:=	*tagtype tagname*;
taskdecl	:=	`task` *name*(*taskparamlist*)
taskparamlist	:=	*taskparamlist*, *taskparam* | *taskparam*
taskparam	:=	*type name* `in` *flagexp* | *type name* `in` *flagexp* `with` *tagexp*
flagexp	:=	*flagexp* `and` *flagexp* | *flagexp* `or` *flagexp* |!*flagexp* | (*flagexp*) | *flagname* | `true` | `false`
tagexp	:=	*tagexp* `and` *tagtype tagname* | *tagtype tagname*
statements	:=	... | `taskexit`(*flagactionlist*) | `tag` *tagname* = `new tag`(*tagtype*) | `new` *name*(*params*)(*flagortagactions*) | `assert`(*expression*)
flagactionlist	:=	*flagactionlist*; *name* : *flagortagactions* | *name* : *flagortagactions*
params	:=	... | `tag` *tagname*
flagortagactions	:=	*flagortagactions*, *flagortagaction* | *flagortagaction*
flagortagaction	:=	*flagaction* | *tagaction*
flagaction	:=	*flagname* := *boolliteral*
tagaction	:=	`add` *tagname* | `clear` *tagname*

Fig. 5. Task Grammar

developer includes a flag declaration inside a class declaration to declare that objects of that class contain the declared flag. Flag declarations use the `flag` keyword followed by the flag's name. The developer may optionally use the `external` keyword to specify that the flag is set and reset by the runtime system. External flags are intended to handle asynchronous events such as communication over the Internet or mouse clicks. External flags are intended to be declared in library code with the corresponding runtime component setting and clearing the external flag.

The developer can use tags to enforce relations between the parameters of a task. The developer can create new tag instances with the `new tag` statement and a tag type. Note that there may be many instances of a given type of tag. Each different instance of that tag is distinct — objects labeled by two different instances of the same tag type are not grouped together. The developer can bind tags to objects when an object is allocated or bind or unbind tags to or from parameter objects at the task's exit.

The developer declares a task using the `task` keyword followed by the task's name, the task's parameters, and the task's code. Each task parameter declaration contains the parameter's name, the parameter's type, a flag guard expression that specifies the state of the parameter's flags, and an (optional) tag guard expression that specifies the tags the object has. The task may be executed when all of its parameters are available. A parameter is available if the heap contains an object of the appropriate type, that object's flags satisfy the parameter's guard expression, and that object contains the tag instances that the parameter's guard expression specifies. Bristlecone adds a modified `new` statement that specifies the initial flag settings and tag bindings for a newly allocated object. These take effect when the task exits. Bristlecone contains a `taskexit` statement that specifies how the task changes the state of the flags or tag bindings of its parameter objects at that task exit point.

Bristlecone contains an `assert` statement that can be used to specify correctness properties that must hold. The goal of assert statements is to provide a mechanism to detect higher-level errors that do not cause low-level exceptions. The compiled application uses the assert statements to detect errors at runtime— if it detects an error, the runtime system will invoke the recovery algorithm. These assertion statements can be used with data structure consistency checking tools [6, 7], JML assertions [8], or design by contract methodologies [9]. In many cases, the assertions can be generated automatically using dynamic invariant detection tools [10–12].

4 Runtime System

The Bristlecone runtime is responsible for dispatching tasks, detecting errors, and recovering from errors.

4.1 Task Execution

Recall that the task specification contains guard expressions for all of the task's parameters and that the runtime executes a task when parameter objects are available that satisfy these guards. We next discuss how our implementation efficiently performs task dispatch. A naive approach to task dispatch could potentially be very inefficient — a parameter's guard expression is quantified over all objects in the heap!

Parameter Sets The runtime maintains a *parameter set* for each parameter of each task. A parameter set contains all of the objects that satisfy the corresponding parameter's guard. For each object type, the runtime precomputes a list of parameter sets that objects of this type can potentially be a member of. When a task exit changes an object's flag settings or tag bindings, the runtime updates that object's membership in the parameter sets by traversing the precomputed list of possible parameter sets for the class and evaluating whether the object satisfies the guard expression to be a member of the parameter set.

Bristlecone also uses the parameter sets as root sets for garbage collection. Objects in Bristlecone are garbage collected if (1) the object is unreachable from any potential parameter objects and (2) the object cannot be a parameter object of any task as determined by membership in a parameter set. Note that it is possible to write incorrect programs that leave objects in task queues (e.g consider a two parameter task with tagged parameters, the program might only change one parameter object's flags leaving the other parameter object in the queue). We have developed a static analysis that the developer can use to automatically identify this type of memory leak [13].

Task Queue A *task invocation* is a tuple that includes both a task and bindings for that task's object parameters and tag parameters. An *active task invocation* is a task invocation that satisfies all of the task specification's guards and can therefore safely be invoked by the runtime. The runtime maintains the *task queue* of all active task invocations and executes task invocations from this task queue.

Our implementation maintains a conservative approximation of the task queue — our implementation's task queue may contain a number of non-active task invocations in addition to all of the active task invocations. When an object is added to a parameter set, the implementation generates all active task invocations that bind that object to the corresponding parameter and then adds these active task invocations to the task queue. When an object is removed from a parameter set, our implementation does not remove task invocations from the task queue. Instead, before the implementation executes a task invocation in the queue, the implementation verifies that the task invocation is still active.

Iterators We next describe how our implementation efficiently generates all active task invocations. Note that tag bindings restrict how parameter objects can be grouped together into a task invocation, and therefore, a naive implementation can needlessly explore many task invocations that do not satisfy tag guards. For example, the `sendPage` task in a web server may require both a `WebRequest` object and a `Socket` object tagged with the same `connection` instance as parameters. An efficient implementation must prune the search space of possible task invocations to avoid the overhead of exploring many task invocations that do not satisfy the tag guards.

Our implementation searches the parameter binding space using a sequence of iterators. It uses two iterator types: *object instance iterators* and *tag instance iterators*. Object instance iterators iterate over the objects in the corresponding parameter set that are compatible with all tag variable bindings made by previous iterators. In general, we expect that relatively few objects will be bound to a given tag instance and relatively few tag instances will be bound to a given object. Our implementation uses this expectation to optimize the object iterators: if the parameter has a tag guard with a tag variable that was bound by a previous tag iterator, the implementation optimizes the object iterator to only iterate over the objects bound to that tag instance. Tag iterators iterate over tag instances that are bound to an object. Tag iterators bind the tag variables in tag guards to tag instances.

As described above, our iterators use the constraints provided by the tag guards to prune the search space. Note that the order of the iterators can affect the size of the search space that the implementation explores to generate all active task invocations. Our implementation precomputes iterator orderings for each parameter of each task. The implementation uses the following ordering priority:

1. Tag iterators for tags bound to parameter objects that have already been iterated over have the highest priority. We expect that the set of iterated tag instances will be small and, therefore, tag bindings will substantially prune subsequent object iterations for parameters bound to the same tag variable.

2. Object iterators for parameters with tags that are bound by previous tag iterators.

3. Object iterators for parameters with tags that have not yet been iterated over.

4. Remaining object iterators have the lowest priority.

Task Execution Semantics Tasks may fail either as a result of software errors, hardware failures, or user errors. If a task fails, it may leave data structures in inconsistent states. Further computation using these inconsistent data structures will likely have unpredictable and potentially catastrophic results. To avoid this problem, tasks in Bristlecone have transactional semantics — if a task fails, the Bristlecone runtime aborts the task's transaction.

Recall that a potential issue with the use of transactions in traditional programming languages is that after the system recovers to the previous point, the system may simply re-execute the same deterministic fault and that fault will cause the system to fail repeatedly in the same way. Bristlecone addresses this issue by using the flexibility provided by the task-based language to avoid re-executing the same failure. The Bristlecone runtime records the combination of task and parameter assignments that caused the failure and uses this record to avoid re-executing the failed combination task and parameter assignments. Instead, the runtime executes other tasks to avoid retriggering the same underlying fault.

4.2 Error Detection

Errors can cause the computation to produce incorrect results and corrupt data structures, potentially eventually causing the software system to perform unacceptably. Bristlecone uses runtime checks to detect errors, enabling the software system to adapt its execution. The Bristlecone runtime uses error detection routines to trigger recovery actions.

Bristlecone uses checks to detect many software errors. For example, the Bristlecone compiler generates array bounds checks. These checks verify that the software system does not read or write past the end of arrays. The Bristlecone compiler also generates the necessary type checks for array operations and cast operations. These checks ensure that the dynamic types of objects do not violate type safety.

The runtime uses hardware page protection to perform null pointer checks. This is implemented by catching the segmentation fault signal from the operating system. These checks ensure that the software system does not attempt to dereference a null pointer or write values to the fields of a null pointer. The runtime also uses hardware exceptions to detect arithmetic errors including division by zero. Native library routines also signal errors to the runtime. For example, if a software systems attempts to send data over a closed network connection, the runtime will signal an error. Software errors can also cause a program to loop. Looping can prevent the software system from providing services. It is straightforward to support developer-provided task time-outs that the runtime can use to detect looping tasks.

Bristlecone includes a runtime assertion mechanism to ensure that the execution is consistent with respect to specified properties. The developer can simply write imperative code to check properties or can use the assertion mechanism to call external consistency checking code. This mechanism is intended to be used to ensure data structure consistency or to use techniques such as design by contract to detect higher-level errors. The mechanism can be used in conjunction with JML assertions [8], data structure consistency specifications languages [6, 7], or other runtime checkable specifications.

4.3 Error Recovery

Bristlecone was designed to support reasoning about failures using the high-level task abstraction. In Bristlecone, a task either successfully completes execution or does not execute at all. The Bristlecone runtime uses a straightforward checkpointing-based transaction approach to implement this failure abstraction. Because a task can only access the part of the heap that is reachable from the task's parameter, it suffices to create a snapshot of all objects reachable from the task's parameters.

While the current prototype implementation uses a naive checkpointing-based approach, it is conceptually straightforward for future Bristlecone implementations to leverage the large body of work on efficiently implementing software or hardware transactional memory. A second issue with the current implementation is transactionalizing I/O. One solution is to use a transactional I/O API that delays the effects of I/O operations until a task commits.

If Bristlecone detects an error, it simply fails the entire task and uses this stored checkpoint to rollback the state affected by the failed task. This recovery strategy greatly simplifies reasoning about the state of the software system after a failure. Restoring state from the previous checkpoint ensures that a failure does not leave partially updated data structures in inconsistent states.

Many software errors are deterministic. If Bristlecone re-executes a failed task on the same parameters in the same state, it is likely that the task will fail again due to the same error. Bristlecone addresses this issue by maintaining a record of failures. For each failure, this record contains the combination of the failed task and the parameter assignments that failed. Bristlecone uses this record to avoid re-executing the same failures by checking reference equality of the task's parameters. The Bristlecone runtime then uses the object flags to determine which tasks can be executed even though part of the computation has failed. To better handle non-deterministic failures, the approach can be extended to automatically retry failed task executions a few times. We note that after a failure, a failed object can remain in task queue and never be garbage collected. We expect that in practice, software systems will be mostly correct and therefore a failure will be a rare occurrence and only small amounts of memory will be leaked due to failures.

4.4 Debugging and Error Logging

While it is desirable for deployed Bristlecone software systems to make every effort to avoid failures, during the development phase this behavior can mask failures and therefore complicate the debugging process. To facilitate debugging, Bristlecone can be configured to fail-fast. The fail-fast mode ensures that developers will notice software errors during the development process. Moreover, it would be straightforward to have the runtime record the state of the objects that caused the task failure by using the stored checkpoints. This information could help with debugging many software errors.

Furthermore, both developers and system administrators often want to be aware of failures in deployed systems so that the underlying software faults, if any, can be fixed. Bristlecone contains a logging mechanism that records both the task that failed and the type of error. This log ensures that developers and system administrators are aware of failures in Bristlecone software systems and gives the developers a starting point for diagnosing the cause of the failure. In some cases, developers may wish to create a custom framework to communicate failure data. It would be possible to provide an API that applications could use to query the runtime system about failures.

5 Experience

We next discuss our experiences using Bristlecone to develop three robust software systems: a web crawler, a web server, and a multi-room chat server.

5.1 Methodology

We have implemented the Bristlecone compiler. Our implementation consists of approximately 22,400 lines of Java code and C code for the Bristlecone compiler and runtime system. The Bristlecone compiler generates C code that runs on both Linux and Mac OS X. The Bristlecone runtime uses precise stop-and-copy garbage collection. The source code for our compiler and runtime is available at `http://demsky.eecs.uci.edu/bristlecone/`. We ran the benchmarks on a MacBook with a 2 GHz Intel Core Duo processor, 1 GB of RAM, and Mac OS X version 10.4.8.

For each benchmark, we developed two versions: a Bristlecone version and a Java version. We designed the Java versions to tolerate faults by isolating components of the computation using threads. Without the use of threads to provide fault tolerance, the Java versions would have halted with the first failure.

Our evaluation was designed to evaluate how robust each version of the benchmark applications was to the large class of faults that cause the faulty thread or task to perform an illegal operation. This fault class includes faults that cause null pointer dereferences, out of bound array index errors, failed assertions, failed data structure consistency checks, library usage errors, and arithmetic exceptions. Our evaluation simulated the effects of this fault class by randomly injecting halting failures.

We used the Bristlecone compiler to automatically insert failure injection code after each instruction. We used the Java frontend of our compiler framework to compile and instrument the Java versions. The failure injection code takes three parameters at runtime: the number of instructions to execute before considering injecting a failure, the probability that a failure will be injected, and the total number of failures to inject. For each benchmark, we selected the number of failures and then set the failure probability to ensure that the normal execution of the benchmark would reach the set number of failures.

5.2 Web Crawler

The web crawler takes an initial Uniform Resource Locator (URL) as input, visits the web page referenced by the URL, extracts the hyperlinks from the page, and then repeats this process to visit all of the URLs transitively reachable from the initial URL.

The Bristlecone version contains four tasks. The `Startup` task creates a `Query` object to store the initial URL that was specified on the command line and creates a `QueryList` object to store the list of URLs that the web crawler has extracted. The `requestQuery` task takes a newly created `Query` object as input, contacts the web server specified by the `Query` object, and then requests the URL specified by the `Query` object. The `readResponse` task reads the data that is currently available on the connection and then checks if the task has received the complete web page. The `processPage` task extracts URLs from the web page, checks the `QueryList` object to see if the crawler has seen this URL before, and then creates a `Query` object if the URL has not been seen before.

The Java version uses a pool of three threads to crawl web pages. Each thread dequeues a URL from a global list of pages to visit, downloads the corresponding web page, extracts URLs from the web page, and then stores any URLs it has not seen before into the global list of pages to visit.

We evaluated the robustness of the web crawler by developing both a workload and a failure injection strategy. Our workload consisted of a set of 100 web pages that each contain 3 hyperlinks to other web pages in the set. We used randomized failure injection to inject failures into the executions of the web crawlers. We injected 3 failures into each execution with each instruction having a 1 in 426,000 chance of failing.

We performed 100 trials of the experiment on each of the two versions. For each trial, we measured how many web pages the crawler downloaded. Figure 6 presents the results of the web crawler experiments. Without the injected failures, both versions download 100 web pages. With the injected failures, on average the Bristlecone version downloaded 91 out of 100 web pages and the Java version downloaded 6 out of 100 web pages. While most of the injected failures in the Bristlecone version only affect crawling a single web page, failures that are injected into either the startup task or the processing of the initial web page can affect crawling many web pages. Such failures prevent the Bristlecone version from discovering the URLs of any further pages and significantly lowered the Bristlecone version's average number of crawled pages.

	Java	Bristlecone
Web Pages Crawled (out of 100)	6	91

Fig. 6. Summary of Web Crawler Benchmark Results

5.3 Web Server

The web server benchmark contains features that are intended to model an e-commerce server. The web server maintains an inventory of merchandise and supports requests to perform commercial transactions on this inventory, including adding new items, selling items, and printing the inventory.

The Bristlecone version contains six tasks. The `StartUp` task creates a `ServerSocket` object to accept incoming connections, creates a `Logger` object to log the connections, and creates an `Inventory` object to keep track of the current inventory of merchandise. The `AcceptConnection` task processes incoming connections and creates a `WebSocket` objects to manage each connection. The `ProcessRequest` task reads the data that is currently available from the incoming connection and then checks if the task has received the complete request. When the complete request is available, the `ProcessRequest` task parses the request to determine whether the request is an e-commerce transaction or a simple file request.

The `Transaction` task processes e-commerce transaction requests. It first inspects the request to determine whether the request is to add new items to the inventory, to make a purchase, or to display inventory and then performs the requested operation. For example, after receiving a purchase request the task looks up the price of the item in the `Inventory` object, verifies that the item is available, and if so, decrements the inventory count for the item and adds the price of the item to the sales figure.

The `SendFile` task processes file requests. It opens the requested file, reads the file's contents, and writes the file's contents to the socket. The `LogRequest` task logs all of the requests to the log file.

The Java version of the web server uses a thread to monitor for incoming connections. When a new connection arrives, the server spawns a separate connection thread for that incoming connection. The server uses a global object to store the inventory values. We used this design to isolate failures in connection threads to that specific request as much as possible. Note that failures can potentially corrupt the shared state. Note that unlike the Bristlecone version of the web server, a failure in a connection thread will prevent the server from performing any further operations for that connection including logging the request.

We evaluated the robustness of both versions of the web server by developing both a workload and a failure injection strategy. Our workload simulated web traffic to the server. Our workload consisted of a sequence of 4,400 transaction requests. Our failure injection strategy utilized the failure injection code described in the previous section.

We used failure injection to randomly inject 50 failures into the execution with a probability of injecting a failure after a given instruction of 1 in 2,100,000. We performed 200 trials on each of the two versions. For each trial we recorded whether the final inventory request was served, whether the final inventory was consistent, how many requests each version failed to serve, and how many request each version failed to log.

Figure 7 summarizes the results of the fault injection experiments with the web server. The Java version failed to serve the inventory request in 4.5% of the trials while the Bristlecone version failed to serve the inventory request in 1.5% representing a three-fold reduction in the number of failures to serve inventory requests. More importantly, while the Java version served correct inventory responses only 68.6% of the time, the Bristlecone version served the correct inventory response 100% of the time. The Java version failed to serve 3.8% of the web requests and Bristlecone version failed to serve 2.2% of the web requests, representing a 42% reduction in the failure rate. The Java version failed to log 3.9% of the web requests and Bristlecone version failed to log 2.6% of the web requests, representing a 33% reduction in the failure rate.

	Java	Bristlecone
Failures to serve Inventory Responses	4.5%	1.5%
Correct Inventory Responses	68.6%	100%
Failures to Serve Request	3.8%	2.2%
Failures to Log Request	3.9%	2.6%

Fig. 7. Summary of Web Server Benchmark Results

5.4 Chat Server

The multi-room chat server benchmark accepts incoming connections, asks the user to create a new room or select an existing room, and then allows users to chat with other users in the same chat room. The Bristlecone version contains six tasks. The `StartUp` task creates a `ServerSocket` object to accept incoming connections and a `RoomObject` to manage the chat rooms. The `AcceptConnection` task processes incoming chat connections. It creates a `ChatSocket` object to manage this connection and then sends a message to ask the user to select a chat room.

The `ReadRequest` task reads the user's chat room selection. It reads the currently available data from the incoming connection and checks if the chat server has received the complete chat room selection. When the complete room request has been received, the `ProcessRoom` task processes the request. If the requested room does not exist, it creates the requested chat room. It then adds the user to the requested chat room. The chat server stores the mapping of chat room names to the set of chat room participants and for each room, maintains a list of participants in the corresponding room.

The `Message` task processes incoming chat messages and stores these message in a `Message` object. The `SendMessage` task then reads these `Message` objects, parses the messages, and then sends the messages to all of the participants in the chat room. Note that a problematic message or other error condition that causes the `SendMessage` task to fail will not prevent the server from processing future messages from the same connection.

The Java version of the chat server uses a thread to monitor for incoming connections. When a new connection arrives, the server spawns a separate connection thread for that incoming connection. The server uses a global object to store the set of chat rooms. Unless a failure corrupts the room list, this design isolates failures in connection threads to the specific connection. Note that unlike the Bristlecone version of the chat server, a single failure in a connection thread will prevent the server from relaying any further messages from that connection.

We evaluated the robustness of both versions by developing both a workload and a failure injection strategy. Our workload simulated multiple users chatting on the server. Our workload sent a total of 800 messages. Our failure injection strategy utilized the failure injection code described in the previous section.

We used failure injection to randomly inject 10 failures into the execution with a probability of injecting a failure after a given instruction of 1 in 270,000. We performed 100 trials on each of the two versions. For each trial we recorded how many messages were successfully transmitted.

In the presence of the injected failures, the Java version failed to deliver 39.9% of the messages and the Bristlecone version failed to deliver 19.3% of the messages, representing a factor of two reduction in the failure rate.

5.5 Experiences Writing Bristlecone Applications

We have developed Bristlecone and Java versions of three different benchmark applications. In general, we found writing Bristlecone applications to be straightforward. Typically, writing the Bristlecone version of an application simply requires reorganizing the application's code.

The Bristlecone versions of the benchmarks were approximately the same size as the Java versions. The Bristlecone version of the web crawler contained 20% fewer lines of code than the Java version, the Bristlecone version of the web server contained 2% more lines of code than the Java version, and the Bristlecone version of the chat server contained 5% more lines of code. The Bristlecone version of the web crawler was shorter because it did not require an auxiliary data structure to store queries.

One potential concern with Bristlecone is that developers may make mistakes writing the high-level task specifications that Bristlecone requires. In our experience, we have found that task declarations were in general simpler than the lower-level imperative code and therefore easy to write correctly. However, we have developed an analysis that can analyze the task specification to extract state transition diagrams for each class [13]. Developers can use these state transition diagrams to quickly visually verify that their task specifications have the desired behaviors.

5.6 Performance

Although Bristlecone uses standard compilation techniques for the body of methods and tasks, it incurs extra overheads supporting transactions and task invocation. Our current runtime implements transactions using a combination of checkpointing and single-threaded execution. We have measured the current implementation's checkpointing and task invocation overhead to be 4.7 microseconds per task invocation on a 3 GHz Pentium-D machine for a microbenchmark. Researchers have developed efficient hardware or software transactional memory implementations [14–21] that could be used to lower the transaction overhead. Static task scheduling could also be used to statically schedule a sequence of task invocations to further reduce the task invocation overhead.

5.7 Discussion

Our experience indicates that software systems developed using Bristlecone can recover from many otherwise fatal failures. The Bristlecone versions of all three benchmarks were able to recover from many more injected failures and provided a higher of quality of service than the hand-designed Java versions.

Note that these results only hold for software faults that can be automatically detected. These results can be generalized to include faults that cause the application to silently perform an incorrect action, if the developer provides Bristlecone with a runtime-checkable correctness specification the detects the error. Examples of such specifications include runtime assertions or data structure consistency specifications.

6 Related Work

We survey related work in testing, static analysis, exception mechanisms, fault tolerance, programming languages, and software architectures.

6.1 Approaches to Reliable Software

The standard approach to dealing with software failures is to work hard to find and eliminate software faults. Approaches such as extensive testing [22], static analysis [23–25], software model checking [26], error correction codes [27], and software isolation mechanisms [28] are all designed, in part, to eliminate as many potential errors as possible. We expect that Bristlecone will complement these other techniques: Bristlecone will enable software systems to recover from software errors that the other techniques miss.

Many programming languages, including Java, provide an exception handling mechanism [29]. Writing exception handlers requires developers to reason about what parts of the computation are effected by the failure and how to recover the computation from a failure — note that the failed operation may leave critical data structures in inconsistent, partially updated states.

Fault tolerance researchers have developed many methods to address software failures. Recovery blocks allow a developer to provide multiple implementations of an algorithm and an acceptance test for these implementations [30]. This technique requires the developer to expend the effort to develop multiple implementations and acceptance tests. Furthermore, the recovery block technique may fail if the algorithms share a common defect or if there is an error in the acceptance test.

Backward recovery uses a combination of checkpointing and acceptance tests (or error detection) to prevent a software system from entering an incorrect state [31–34]. Unfortunately, it can be difficult to handle deterministic failures using backward recovery as the same software error will likely cause the software system to repeatedly fail. Forward recovery uses multiple copies of a computation to recover from transient errors [35]. Forward recovery is designed to handle intermittent failures — it cannot help deterministic errors that affect all copies of the computation.

Databases utilize transactions to ensure that the database is never left in a half-updated state by a partially completed sequences of operations [3].

In N-version programming, the developer constructs a software system out of multiple, independent implementations and a decision algorithm to decide which result to use in the event of a disagreement [36]. However, N-version programming may be prohibitively expensive — it requires developers to perform the difficult task of implementing multiple versions that are independent enough to not share failure modes but similar enough to be comparable.

The Recovery-Oriented Computing project has explored integrating an undo operation into software systems [37] and constructing systems out of a set of individually rebootable components [38]. Failure oblivious computing is designed to address memory errors in C programs [39]. It detects erroneous memory operations and discards illegal write operations and manufactures values for invalid read operations. DieHard handles similar memory errors by using replication and randomization of the memory layout [40]. Randomization probabilistically ensures that illegal memory operations can only damage data structures in one of the replicants.

Specification-based data structure repair automatically generates repair algorithms from declarative specifications [7] and imperative consistency checking code [41]. This technique enables software systems to recover from data structure consistency errors.

Researchers have used meta-languages to decompose numerical computations into parallelizable tasks [42]. This technique is applicable to parallelizable numerical computations that compute many subproblems and then combine the subproblem results to compute an overall result. If one of the subcomputations executes slowly, this approach can ignore the subcomputation. Bristlecone is designed to handle a broader class of software systems including servers, control systems, and office applications. Bristlecone can provide stronger correctness guarantees.

6.2 Related Languages

A key component of Bristlecone is decoupling unrelated conceptual operations and tracking data dependences between these operations. Bristlecone's approach contains common elements with many parallel programming paradigms [43]. Dataflow computation was one of the earlier computational models that keeps track of data dependences between operations so that the operations can be parallelized [44]. Note that dataflow languages are not designed to handle failures — a failure in a data flow program will likely cause an operation to fail to place a value in a queue, which would likely cause the application to fail catastrophically because operations that operate on multiple queues would pair the wrong values for the rest of the computation. Bristlecone ensures that failures cannot cause the wrong parameter objects to be paired together or prevent a task from operating on parameter objects that were not affected by the error.

Tuple-space languages, such as Linda [45], decouple computations to enable parallelization. The threads of execution communicate through a set of primitives that manipulate a global tuple space. While these systems were not designed to address software errors as errors in these systems can permanently halt the execution of threads, Bristlecone implements a similar technique to decouple the execution of its tasks.

The orchestration language Orc [46] specifies how work flows between tasks. Orc is designed to decouple operations and expose parallelism. Note that if an operation fails, any work (and any corresponding data) flowing through the task may be lost. Since the goal of Orc is not failure recovery, it was not designed to contain mechanisms to recover data from failed tasks. Therefore, errors can cause critical information to disappear, eventually causing the software system to fail. Bristlecone uses flags to keep track of the conceptual states (or roles) that objects are in, enabling software systems to recover data from software errors and to continue to execute successfully.

Actors communicate through messages [47, 48]. Actors were originally designed as a concurrent programming paradigm. Failures may cause actors to drop messages and corrupt or lose their state. Bristlecone's objects persist across task failures and can still be used by other tasks. Moreover, state corruption in actors can cause actors to permanently crash. Since Bristlecone's tasks are stateless, a previous failure of a task does not affect future invocations of that task on different inputs.

Argus is a distributed programming language that organizes processes under guardians and isolates a process failure to the guardian under which it executes [49]. Inconsistencies could potentially cause the enclosing guardian to shut down. Argus supports failure recover through an exception handling mechanism. This approach is complementary to Bristlecone: a developer can write exception handlers for anticipated failures and Bristlecone can be used to recover from unexpected failures.

Oz is a concurrent, functional language that organizes computations as a set of tasks [50, 51]. Tasks are created and destroyed by the program. A task becomes reducible (executable) once the constraint store satisfied the task's guard. Task reducibility is monotonic — once a task is reducible it is always reducible. Task activation in Bristlecone is not monotonic — the developer can temporarily disable a task when other tasks have placed objects into states that are incompatible with the task or when the effect of a task is no longer desirable. Non-monotonicity makes it straightforward for a Bristlecone application to use multiple implementations of the same functionality for redundancy. Moreover, since task creation is controlled by the program in Oz, it is more difficult to reason statically about tasks.

Concurrent Prolog is logic-based language that uses unification to prove a goal [52, 53]. The proof corresponds to the execution of the program. Concurrent Prolog's guarded notation is similar to Bristlecone's flag expressions, but Concurrent Prolog's evaluation strategy starts from an end goal and reasons backwards. Concurrent Prolog programs may be able to recover from some failures by finding a different execution that reaches the same end goal. The downside is that if a failure prevents the program from completely achieving its end goal, the program will be unable to make partial progress. Bristlecone works forward and therefore can make progress even if a failure prevents the system from completely achieving its goal.

Erlang has been used to implement robust systems using a set of supervisors and a hierarchy of increasingly simple implementations of the same functionality [54]. The supervisors monitor the computation for errors. If an error is detected, the system falls back to a simpler implementation in the hierarchy. Ericsson has taken this approach in their telephone switches. Bristlecone is complementary to the supervisor approach — while the supervisor approach gives the developer complete control of the recovery process, the downside of this approach is that it requires the developer to manually develop multiple implementations of the same functionality. Bristlecone requires minimal development effort and could potentially make recovery cost effective for a larger set of applications. Furthermore, while a shared but minor fault could cause the entire Erlang implementation hierarchy to fail, in many cases Bristlecone may be able to execute around the fault and still provide nearly complete functionality.

Several research projects use type state-based approaches to automatically check that an API is used correctly [55, 56]. Puntigam proposes tokens as a synchronization mechanism for object-oriented languages [57]. Bristlecone flags are similar to these mechanisms with one significant difference — Bristlecone uses flags to determine the execution of a program while these mechanisms only check (or synchronize) the actions of traditional imperative programs.

6.3 Related Software Architectures

The staged event-driven architecture (SEDA) pushes events through stages [58]. Note that this architecture was been designed for high-performance computation and not fault tolerance. An error in a stage can prevent relaying the event and cause information to be lost. Stages also have local state, therefore, corruption of this state will cause that stage to shutdown until reboot. It appears difficult to specify that an application should either execute one sequence of operations or a second sequence, but not both.

7 Conclusion

We have successfully developed several robust software systems using Bristlecone. Bristlecone software systems consist of a set of interacting tasks with each task implementing one of the conceptual operations in the software system. The developer specifies how these tasks interact using task specifications. Bristlecone uses transaction to recover data structures from task failures. Bristlecone then uses task specifications to reason about how to continue execution in the presence of a failed task. The key results in this paper include the Bristlecone language, the Bristlecone compiler and runtime, and our experience using the Bristlecone language. Our experience indicates that the task-based approach used in Bristlecone can effectively enable software systems to recover from otherwise fatal errors. Bristlecone promises to increase the robustness of software systems and to decrease the cost of developing many classes of robust software systems.

Acknowledgments. We would like to thank the anonymous referees for their insightful feedback on our paper. This work was funded in part by NSF Grant CCF-0725350 and NSF Grant CNS-0720854.

References

1. Haugk, G., Lax, F., Royer, R., Williams, J.: The 5ESS(TM) switching system: Maintenance capabilities. AT&T Technical Journal **64(6 part 2)** (July-August 1985) 1385–1416
2. Mourad, S., Andrews, D.: On the reliability of the IBM MVS/XA operating system. IEEE Transactions on Software Engineering (September 1987)
3. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
4. Baker, W.O., Ross, I.M., Mayo, J.S., Stanzione, D.C.: Bell labs innovations in recent decades. Bell Labs Technical Journal **5**(1) (January-March 2000) 3–16
5. Harris, T.: Exceptions and side-effects in atomic blocks. Science of Computer Programming **58**(3) (2005) 325–343
6. Demsky, B., Cadar, C., Roy, D., Rinard, M.C.: Efficient specification-assisted error localization. In: Proceedings of the Second International Workshop on Dynamic Analysis. (2004)
7. Demsky, B., Rinard, M.: Data structure repair using goal-directed reasoning. In: Proceedings of the 2005 International Conference on Software Engineering. (May 2005)
8. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: In OOPSLA '00 Companion. (200) 105–106
9. Meyer, B.: Applying "Design by Contact". Computer **23**(10) (1992) 40–51
10. Demsky, B., Ernst, M.D., Guo, P.J., McCamant, S., Perkins, J.H., Rinard, M.: Inference and enforcement of data structure consistency specifications. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. (2006)
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer **7**(3) (June 2005) 212–232
12. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: Proceedings of the 22nd International Conference on Software Engineering. (June 2000)
13. Demsky, B., Sundaramurthy, S.: Static analysis of task interactions in bristlecone for program understanding. Technical Report UCI-ISR-07-7, Institute for Software Research, University of California, Irvine (October 2007)
14. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing. (August 1995)
15. Ananian, C.S., Asanović, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: 11th International Symposium on High Performance Computer Architecture. (February 2005)
16. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: Proceedings of the 2006 Conference on Programming Language Design and Implementation. (June 2006)
17. Spear, M.F., Marathe, V.J., Schereer, W.N., Scott, M.L.: Conflict detection and validation strategies for software transactional memory. In: Proceedings of the Twentieth International Symposium on Distributed Computing. (2006)

18. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation, New York, NY, USA, ACM Press (2006) 14–25
19. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the Twentieth Annual International Symposium on Computer Architecture. (1993)
20. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of the Eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming. (2006)
21. Hammond, L., Wong, V., Chen, M., Hertzberg, B., Carlstrom, B., Prabhu, M., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency (tcc). In: Proceedings of the 11th Intl. Symposium on Computer Architecture. (June 2004)
22. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on java predicates (2002)
23. Ghiya, R., Hendren, L.J.: Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. (1996)
24. Wies, T., Kuncak, V., Lam, P., Podelski, A., Rinard, M.: Field constraint analysis. In: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation. (2006)
25. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: Symposium on Principles of Programming Languages. (1999) 105–118
26. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., Robby, Zheng, H.: Bandera : Extracting finite-state models from Java source code. In: Proceedings of the 2000 International Conference on Software Engineering. (2000)
27. Shirvani, P.P., Saxena, N.R., McCluskey, E.J.: Software-implemented EDAC protection against SEUs. IEEE Transactions on Reliability **49**(3) (September 2000) 273–284
28. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M.: Mach: A new kernel foundation for UNIX development. In: Proceedings of the USENIX Summer Conference. (1986)
29. Goodenough, J.B.: Structured exception handling. In: POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages. (1975)
30. Anderson, T., Kerr, R.: Recovery blocks in action: A system supporting high reliability. In: Proceedings of the 2nd International Conference on Software Engineering. (1976) 447–457
31. Zhang, Y., Wong, D., Zheng, W.: User-level checkpoint and recovery for LAM/MPI. ACM SIGOPS Operating Systems Review **39**(3) (2005) 72–81
32. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent checkpointing under Unix. In: Usenix Winter Technical Conference. (January 1995) 213–223
33. Chandy, K.M., Ramamoorthy, C.: Rollback and recovery strategies. IEEE Transactions on Computers **C-21**(2) (1972) 137–146
34. Young, J.W.: A first order approximation to the optimum checkpoint interval. Communications of the ACM **17**(9) (1974) 530–531
35. Huang, K., Wu, J., Fernandez, E.B.: A generalized forward recovery checkpointing scheme. In: Proceedings of the 1998 Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems. (April 1998)
36. Avizienis, A.: The methodology of n-version programming (1995)
37. Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kcman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N.: Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science (March 15, 2002)

38. Candea, G., Fox, A.: Recursive restartability: Turning the reboot sledgehammer into a scalpel. In: HotOS-VIII. (May 2001) 110–115
39. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William S. Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation. (December 2004)
40. Berger, E., Zorn, B.: Diehard: Probabilistic memory safety for unsafe languages. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation. (June 2006)
41. Khurshid, S., García, I., Suen, Y.L.: Repairing structurally complex data. In: Proceedings of the 12th International SPIN Workshop on Model Checking of Software. (August 2005)
42. Rinard, M.: Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In: Proceedings of the 20th ACM International Conference on Supercomputing. (2006)
43. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In: Proceedings of the 16th European Conference on Object-Oriented Programming. (2002)
44. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**(1) (2004)
45. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 80–112
46. Cook, W.R., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Proceedings of the 2006 International Conference on Coordination Models and Languages. (2006)
47. Hewitt, C., Baker, H.G.: Actors and continuous functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
48. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7**(1) (1997) 1–72
49. Liskov, B., Day, M., Herlihy, M., Johnson, P., Leavens, G., Scheifler, R., Weihl, W.: Argus reference manual. Technical Report MIT-LCS-TR-400, Massachusetts Institute of Technology (November 1987)
50. Smolka, G.: The Oz programming model. In: Proceedings of the European Workshop on Logics in Artificial Intelligence, London, UK, Springer-Verlag (1996) 251
51. Mehl, M.: The Oz Virtual Machine - Records, Transients, and Deep Guards. PhD thesis, Technische Fakultät der Universität des Saarlandes (1999)
52. Shapiro, E.: The family of concurrent logic programming languages. ACM Computing Surveys **21**(3) (1989) 413–510
53. Shapiro, E.: Concurrent Prolog: A progress report. Computer **19**(8) (1986) 44–58
54. Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, Swedish Institute of Computer Science (November 2003)
55. DeLine, R., Fahndrich, M.: Typestates for objects. In: Proceedings of the 18th European Conference on Object-Oriented Programming. (2004)
56. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications. (2007) 301–320
57. Puntigam, F.: Internal and external token-based synchronization in object-oriented languages. In: Modular Programming Languages, Proceedings of the 7th Joint Modular Languages Conference. (2006) 251–270
58. Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services. In: Proceedings of the Eighteenth Symposium on Operating Systems Principles. (October 2001)