The Gozer Workflow System

Jason Madden*, Nicolas G. Grounds*, Jay Sachs*, and John K. Antonio†

*RiskMetrics Group, 201 David L. Boren Blvd, Suite 300, Norman, OK, USA
†School of Computer Science, University of Oklahoma, Norman, OK, USA

Abstract

The Gozer workflow system is a production workflow authoring and execution platform that was developed at RiskMetrics Group. It provides a high-level language and supporting libraries for implementing local and distributed parallel processes. Gozer was developed with an emphasis on distributed processing environments in which workflows may execute for hours or even days. Key features of Gozer include: implicit parallelization that exploits both local and distributed parallel resources; survivability of system faults/shutdowns without losing state; automatic distributed process migration; and implicit resource management and control. The Gozer language is a dialect of Lisp, and the Gozer system is implemented on a service-oriented architecture.

1. Introduction

Gozer names both a platform for developing and executing complex distributed and parallel processes called workflows as well as the high-level language used to encode workflow computations. A primary goal of the platform is to make it easy to take advantage of a distributed system.

The Gozer platform is built on top of the Java platform. It runs within the (proprietary) BlueBox environment, a distributed, message-passing cluster based on a service-oriented architecture. Service instances communicate by placing XML messages on a message queue (the Java Message Service) which distributes the messages to available nodes. Each service describes the operations it offers with an XML document called a WSDL. The Gozer platform exploits the asynchronous nature of the message queue and its ability to load-balance across multiple instances of a single service to achieve distributed concurrency and fault tolerance, i.e., survivability. The underlying BlueBox platform provides monitoring and management features.

The Gozer language is a Lisp dialect that could be described as a "scripting language" due to its support for interactive development, rapid prototyping, and tight integration with existing Java libraries and BlueBox services. Its primary influence is Common Lisp, but it includes elements from other languages such as Clojure [1] and Groovy [2]. Although the Gozer language and BlueBox platform are currently mostly used for the processing of financial data, they can be considered general-purpose. The Gozer language is executed by a custom bytecode-based Gozer virtual machine and runtime layered on top of the JVM (Java Virtual Machine). This virtual machine (the GVM), described in Section 4.1, provides support for the *futures* used to implement the local thread-based parallelism of Section 2 and the *continuations* required by the distributed concurrency of Section 3. Lisp's simple evaluation model and the ease with which a runtime VM could be developed for it drove the decision to use Lisp.

Listing 1 shows example functions for computing the sum of squares in map/reduce style. The function `loc-sum-squares` performs the computation entirely locally using Gozer's sequential `loop` construct for the map step (squaring the numbers) and a sequential application of addition for the reduce step. Section 2 describes how the function `par-sum-squares` allows for a degree of local parallelism in the map step. Finally, as detailed in section 3.5, the function `dist-sum-squares` distributes the map step across available nodes and then performs the reduce step in a single local process, after all the squares have been computed. Notice the similarity among the three variants, which highlights the simplicity of expressing parallel/distributed computations using Gozer.

2. Local Parallelism

Local (shared-memory) parallel operations in Gozer are based upon threads and a thread pool, the native parallel primitives provided by the underlying Java platform. Although the Gozer programmer is free to use these primitives, higher-level operators based upon those from Multilisp [3] are provided by the Gozer language. These operators are all declarative in nature and are designed to allow the programmer to focus on opportunities for achieving concurrency, rather than the implementation details.

A key abstraction provided by Gozer for expressing local parallelism is the *future*. A future represents a computation that may not have completed yet, and represents a promise to deliver the value of that computation when required, at a future point in time. Until a future's computation completes, the future is said to be *undetermined*, after which the future is *determined*. Any value that is not a future is always said to be determined. Futures are used to exploit opportunities for concurrency that exist between the computation of a value and its ultimate use. For example, when transforming a set by applying a function to all members of a set, the earliest time that the transformed value for the *first* transformed member of that set could be used is after the transformation has completed on the *last* member of the set. An opportunity for concurrency exists that can easily be expressed with a future, which in Gozer is declared with the `future` macro. The function `par-sum-squares` from Listing 1 illustrates an example of this usage.

When a computation involves futures, the Gozer programmer generally does not need to take special precautions. Futures can freely be mixed with other values, passed to and returned from functions, stored in data structures, and so on. The GVM is responsible for managing the execution and determination of futures.

The GVM does not provide a guarantee about the sequence in which futures are determined. In the case of IO or other side effects, order of operations can be important. Gozer's `touch` and `pcall` operators allow the programmer to control sequencing in these cases. The `touch` operator causes the calling thread to await the determination of a particular value before proceeding, while `pcall` applies a function, but only after all its arguments are determined.

3. Transparently Distributed Workflows

3.1. Overview

Shared-memory thread-based local parallelism has a number of disadvantages for the construction of large, complex, evolving processes. First, it doesn't scale well beyond a single physical machine. The potential for side-effects makes it challenging to evolve a local process over time or to integrate code from multiple authors. Any robustness in the case of machine failure such as the saving and resuming of intermediate states must be programmed explicitly for each process. Finally, especially in the case of long running processes, it may be desirable to "suspend" a process in order to allow a higher-priority process to use scarce resources such as memory; such suspension would similarly require explicit handling in the design of each process. Gozer's distributed workflows are designed to overcome these difficulties by allowing a single process to span multiple machines, by using a fork/join paradigm that prohibits side-effects, by automatically creating and maintaining persistent checkpoints, and by using non-blocking, zero-resource consuming, event driven processing.

Gozer's distribution facilities, and in general its integration with the BlueBox platform, are provided in a separate module known as Vinz. Vinz offers a simplified set of abstractions to workflow authors intended to make writing fully distributed, concurrent workflows as similar to writing local, sequential programs as possible. As with local parallelism,

Listing 1. Sum-of-Squares Variants

```
(defun loc-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (* number number))))

(defun par-sum-squares (numbers)
  (apply #'+
         (loop for number in numbers
           collect (future (* number number)))))

(defun dist-sum-squares (numbers)
  (apply #'+
         (for-each (number in numbers)
           (* number number))))
```

opportunities for distribution are written in a declarative fashion, and the details of implementation are provided by the platform.

Listing 1's function `dist-sum-squares` demonstrates distributed programming with Vinz. Choosing an appropriate level of concurrency, distributing work to available nodes, gathering results, and continuing the computation when all concurrent work is completed are all automatically handled by Vinz. Importantly, waiting for computations to complete is event driven and consumes no resources. Even though conceptually `dist-sum-squares` blocks awaiting the `for-each` results, no actual blocking occurs because the (distributed) process state is saved to persistent storage and is restored only after all necessary results are available. This type of distribution may be nested to an arbitrary depth and the results of each step may be arbitrarily complex.

A distributed workflow begins as a Gozer program. Vinz takes this program and makes it available for running on the nodes of the BlueBox cluster. This is done by wrapping the Gozer program up as a distinct BlueBox service. Operations are the only way to interact with a service in BlueBox and the only way instances of services can interact with each other, so this new service provides a standardized set of operations (see Table 1) that may be invoked on some available instance of the workflow service by placing the appropriate messages in the message queue.

The GVM allows a Gozer program (specifically, a flow of control within the program) to request a *continuation* at any point by executing the `yield` or `push-cc` special forms. A continuation represents the completion of the same flow of control (compare to a future, which represents the completion of a different flow of control). Additionally, the `yield` form causes the GVM to return control to its own caller. Using the operations in Table 1 and GVM continuations requested at opportune moments, Vinz automatically distributes and migrates workflows

Execution of a workflow is typically initiated by invoking the Start operation with a set of workflow-defined parameters. This causes the creation of a *task*, which uniquely

Table 1. Vinz Service Operations

Operation	Description
Start	Asynchronously begin execution of a workflow, returning its id.
Run	Synchronously execute a workflow, returning its id.
Call	Synchronously execute a workflow, returning its last result.
Terminate	Management operation to asynchronsly terminate any running workflow.
RunFiber	Begin execution of a portion of the workflow on this instance.
AwakeFiber	Resume a suspended parent fiber when a child fiber has completed.
ResumeFromCall	Resume a suspended fiber when a remote operation completes.
JoinProcess	Resume a suspended fiber when any arbitrary process has completed.

identifies that particular running instance of the workflow. Every task contains one or more uniquely identified *fibers* (initially one). A fiber encapsulates a Gozer flow of control that may be advancing on only a single node at any given time. A task is somewhat analogous to an operating system process, while a fiber is analogous to a thread within that process.

Once Start has created a task and fiber, it prepares the environment in which the main fiber will execute, and, with the support of the GVM, saves its initial continuation (state) to persistent storage. The Start operation then issues an asynchronous invocation of the RunFiber operation with a parameter identifying the newly created fiber. Its job complete, Start now returns the task's ID to the caller, who can use it to monitor the progress of the task.

When the message queue delivers a RunFiber request to an instance of a Vinz workflow service, the fiber's continuation is loaded from persistent storage, and the GVM begins executing that continuation. The GVM continues to run until the program has been completed, or until the next continuation is requested, at which point the fiber is halted and its state stored for later execution.

While running, a fiber can create and execute (via RunFiber) other fibers. These fibers are *children* of the first fiber. The `for-each` and `parallel` macros described in Section 3.5 create and manage fibers automatically. The `fork-and-exec` and `join-process` forms of Section 3.4 allow advanced programmers to create and wait for their own fibers.

3.2. Non-Blocking Service Requests

In practice, Vinz workflows largely consist of requests to other BlueBox services. In a traditional synchronous service invocation, the sender is blocked until its request has been delivered by the message queue, the service has completed processing, and the results are returned by the message queue. During this time, the sender is consuming resources (physical memory and a BlueBox request "slot") without making any progress; in essence, those resources are being wasted. An ordinary asynchronous request does no better unless the sender can find other work to do, a situation that is both rare and complex to manage.

Vinz workflows solve the problem of wasted resources by automatically executing a Gozer `yield` statement when

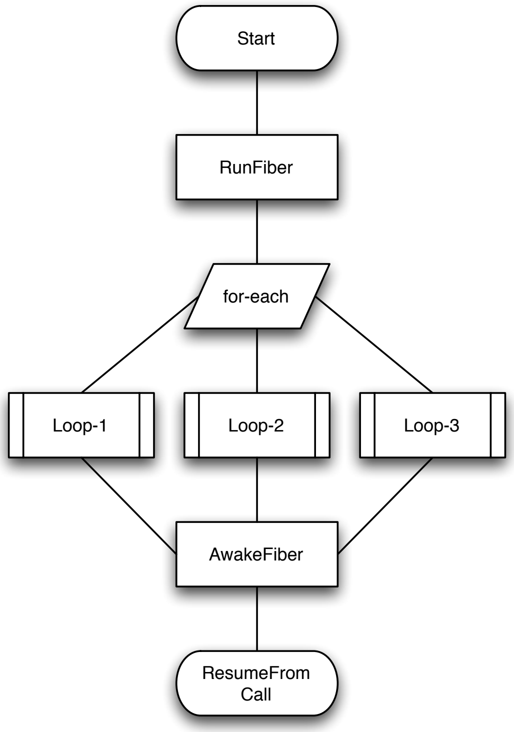

Figure 1. Sample Workflow Lifetime

a service request is made. The request message is sent asynchronously, and the message queue is instructed to deliver the response not to the sending instance (as with a traditional system) but instead to any workflow service instance by means of its ResumeFromCall operation. While the service request is being delivered and processed, the Gozer state is saved to persistent storage. Later, when the results are available, the Gozer state is restored and processing is resumed.

Overall, this allows many more tasks to be in progress at any one time. Wall-clock time, CPU resources and memory that would otherwise have been wasted blocking can now be used by a different task to make progress. Because the message queue is constantly load balancing in-progress requests and making decisions based on priority, this also helps to improve interactive usage (interactive requests are less likely to be held up by individual long-running batch workflows). In addition, together with the entire state of the task being regularly stored to stable storage and the message queue providing buffering and re-delivery of messages in case of instance failure, this makes for a highly robust system, one in which the failure of any instance will result in only minimal delays as other instances automatically compensate.

Of course, there are some cases where the service operation is expected to be very fast, and therefore the relative overhead of the aforementioned task migration might be considerable. In these cases, the programmer can, statically or dynamically, choose to require Vinz to instead make a standard synchronous request. If a service request is attempted from a future's background processing thread, it's generally not possible for process migration to occur (what state should be persisted? what about other futures?) so Vinz detects this and automatically makes a standard synchronous request.

3.3. Deflink

Vinz takes advantage of the dynamic code-generation features of Gozer macros and the published service interfaces to make it easy for a workflow to interact with any service. A macro called `deflink` provides this functionality. This macro requests a service's interface in the form of an XML document, parses it, and then generates a set of functions to invoke each operation the service publishes, together with the appropriate placement of `yield` statements to make the request non-blocking. The XML message structure is flattened into a set of parameters for the function, and the function is capable of coping with complex XML trees by using corresponding Gozer data structures.

As part of the source code for the workflow, each `deflink` is evaluated when the source code for the workflow is loaded. This ensures that the functions generated will be appropriate for the service version currently operating, which helps smooth over minor version incompatibilities. If for some reason an operation cannot be interacted with from a Gozer function, `deflink` instead generates a Gozer macro that signals an error. In this way, if and only if the workflow tried to invoke that operation, a compile-time error will occur and the workflow will not be loaded, thus avoiding runtime errors.

Listing 2 shows an example invocation of the `deflink` macro, and some of the generated functions (edited for size). Notice that the documentation specified in the interface document is preserved for the Gozer programmer. The function `SM-ListSessions-Method` provides the high-level interface using named keyword function arguments. The function `SM-ListSessions` actually invokes the service, handling the case of background threads as well as using Gozer's condition system to provide optional restarts in the event of error (see Section 3.7 for more on error handling).

Listing 2. Deflink

```
(deflink SM :wsdl "urn:security-manager-service"
            :port "SecurityManager") =>

(defun SM-ListSessions-Method
   (&key FilterParams WithinRealm)
 "Returns a list of sessions visible to the..."
 (let ((msg (create-message "SM-ListSessions")))
   (. msg (set "FilterParams" FilterParams))
   (. msg (set "WithinRealm" WithinRealm))
   (SM-ListSessions :message msg)))

(defun SM-ListSessions (&key message)
   "Returns a list of sessions visible to the..."
  (restart-case
      (let ((response
             (cond
               ((%is-fiber-thread)
                (call-wsdl-operation-async
                 :soap-action "...:ListSessions"
                 :message message)
                (yield))
               (otherwise
                (call-wsdl-operation
                 :soap-action "...:ListSessions"
                 :message message)))))
         (parse-wsdl-response response))
    (ignore () (log "Ignoring an exception"))
    (retry  () (SM-ListSessions :message message)))
```

3.4. Forking Fibers

As discussed earlier, the fundamental unit of computation in a Vinz workflow is a fiber. A fiber can be running at most once on one node in the BlueBox cluster, and every workflow begins with a single fiber created by the Start operation. In order for workflow processing to take advantage of multiple nodes in the cluster, then, multiple fibers must be created.

Fiber creation is similar to the creation of processes in the Unix model. A parent fiber, including all its state, is first cloned with a `fork` call. The newly created fiber then continues its execution by executing a different code path than its parent. In practice this different code path is always calling a user-supplied function and so the primitive operation combines this `fork` of a new fiber and `exec` of a function into a single step. The newly created child fiber is scheduled for execution by placing a RunFiber request on the message queue.

Although this combination of the fork and exec operations into a single step is the model commonly used for threads that run within the same process, the initial cloning of the parent fiber means that the analogy to Unix processes is a better fit. Although the variables in the child fiber start out identical to those in the parent, changes either fiber makes will not be visible to its clone. This eliminates any burden of synchronization when mutating variables and values, thus simplifying the programming model both for the workflow author and the Vinz implementation, and it drastically reduces the amount of distributed coordination that Vinz must perform.

Workflow authors have direct access to the ability to create new fibers through the `fork-and-exec` function which returns to the parent fiber the id of the child fiber, while executing a supplied function (often an anonymous `lambda`) in the child. A fiber can wait for any other fiber to terminate using the `join-process` function (analogous to the Unix `wait` function). A fiber that calls `join-process` ultimately invokes `yield` and so relinquishes its resources until such time that the requested fiber terminates. If called from a background processing thread, `join-process` only suspends that thread, leaving the rest of the fiber unaffected.

3.5. For-Each and Parallel

The `fork-and-exec` function, in combination with the `join-process` function, is enough to implement many useful distribution strategies. However, these functions are very low-level and as such may be error-prone. Vinz provides two macros that are conceptually layered on top of `fork-and-exec` that capture the most commonly used distribution patterns.

The most frequently used of these macros is `for-each`, which implements the map step of the map/reduce paradigm. Listing 1 provides an example of using `for-each`. This macro takes as input a sequence of values, and for each value in that sequence, executes the same body of statements. The results of these executions are collected and returned to the parent fiber. The parent fiber was "blocked" (as with `yield`) until all the executions were complete. Optionally, `for-each` may group the values into "chunks" which may then be handled in a locally-parallel fashion, for a combination of distributed and local concurrency. The for-each macro completely abstracts away the operations involved in setting up concurrent fibers and awaiting their completion.

Less frequently used is the `parallel` macro. This macro simply executes all the forms in its body in new fibers. The result of each form is collected and returned to the parent fiber (which was again blocked).

If `for-each` or `parallel` is used from a background thread, it cannot yield the fiber for the same reasons that a non-blocking service request cannot. The solution in this case is to have the background thread fork a new fiber which in turn executes the `for-each` or `parallel` code. The background thread synchronously joins this fiber.

The `for-each` macro in particular may result in an arbitrarily large number of new fibers (a number equal to the number of values in the sequence) and their corresponding RunFiber requests on the message queue. In order for cluster resources to be shared among workflows in the desired way (not necessarily fairly), these macros introduce a configurable throttling mechanism. Called the *spawn limit*, this throttle prevents any individual macro invocation from resulting in more than the configured number of concurrently executing fibers at one time. The spawn limit may be dynamically adjusted by the workflow. Listing 3 shows a simplified example[1] of what the macro in Listing 1 might expand to given the numbers from one to five, if the spawn limit was three. The total number of `yield` forms will be equal to the number of child fibers created, but their distribution will differ depending on the spawn limit.

Listing 3. Vinz Spawn Limit

```
(let ((parent-pid (get-process-id))
      (children   (list))
      (func       (lambda (number)
                    (* number number)
                    (awake parent-pid))))
  (append! children (fork-and-exec func :argument 1))
  (append! children (fork-and-exec func :argument 2))
  (append! children (fork-and-exec func :argument 3))
  (yield)
  (append! children (fork-and-exec func :argument 4))
  (yield)
  (append! children (fork-and-exec func :argument 5))
  (yield)
  (yield)
  (yield)
  (collect-child-results child-pids))
```

Listing 3 shows that the parent fiber must wait for each child fiber to awaken it before continuing. The child fiber awakens the parent fiber by placing a message for AwakeFiber on the message queue. This is more efficient than having the parent fiber invoke `join-process` for each child fiber created since the child fibers may finish in any order and in a `for-each` the order does not matter; only that the total number of `yield` operations match the total number of AwakeFiber messages matters. While this is simple and robust, it is also a fairly heavy-handed way of achieving parent/child communication, and can introduce artificial bottlenecks and very bursty behaviour (discussed further in Section 5).

3.6. Task Variables

Because fibers are cloned copies of their parents, side effects like variable or value mutation are not visible between fibers. Child fibers created with the macros discussed in the previous section communicate to their parents by their return values in a very functional fashion, which is a good fit for Gozer's semi-functional nature. In some cases, though, a distributed algorithm can be greatly simplified if some mutable values could be globally shared between all fibers.

Vinz supports this mutable sharing through a concept know as task variables. A task variable is declared at the top-level of the workflow program with the `deftaskvar`

1. In particular, awaking parent fibers is not performed by a function call the child fiber executes. For reliability, it's actually a property of the fiber itself. The fibers created by `fork-and-exec` do not notify their parent of termination, but the fibers created by these macros do.

Listing 4. Using A Task Variable

```
(deftaskvar exit-flag
  "A global flag. When this becomes true, stop.")

(defun dist-sum-squares (numbers)
  (for-each (number in numbers)
    (unless ^exit-flag^
      ;; don't do anything if the flag is set
      (if (= -1 number)
          ;; tell everybody to stop working!
          (setf ^exit-flag^ t)
          (* number number)))))
```

Listing 5. Task Variable Reader Macro

```
(set-macro-character
 #\^
 (lambda (the-stream c)
   (declare (ignore c))
   ;; ^foo^ -> (%get-task-var 'foo)
   (let ((var-name (read the-stream t nil t))
         (var-str  (symbol-name var-name)))
     (unless (. var-str (endsWith "^"))
       (error "Task vars must be wrapped in ^"))
    `(%get-task-var ',var-name))
   t)
```

macro, similar to the basic Gozer `defvar` macro for defining global variables. All fibers within a task of that workflow may access and update that variable. Vinz guarantees that each fiber will see a self-consistent value for that variable and will always see the latest value for that variable. Stronger promises such as access order or atomic read-modify-update sequences are not provided.

Gozer global variables are conventionally given names that start and end with the * character ("earmuffs"). Similarly, Vinz task variables are given names that start and end with the ^ character, although this is a requirement, not just a convention. In order to provide the desired semantics for task variable access, Vinz needs to replace each read or write of the variable with a function call that performs the steps of checking for a stale local cache, reading the most recent value from the persistence store, taking out appropriate locks, etc. Vinz does this by hooking into the Gozer source parser (the reader) using a reader macro defined on the ^ character (see Listing 5). Each occurrence of a task variable such as `^exit-flag^` in the source file is read as if it were the form `(%get-task-var '^exit-flag^)`.

3.7. Error Handling

An important part of a robust system is error handling, or, more generally, condition handling. Gozer provides an implementation of the very general Common Lisp condition system which goes above and beyond exception handling by not requiring the stack to unwind to handle conditions [4]. In turn, Vinz provides workflow authors with some convenient extensions built on Gozer's condition system.

The core of these extensions is the concept of a named

Listing 6. Vinz Error Handling

```
(defhandler ignore-handler
  :java ("java.lang.Throwable")
  :action ignore)

(defhandler retry-handler
  :java ("java.net.SocketException")
  :code ("{urn:service}Connet"
         "{urn:service}Transmit")
  :action retry
  :count 5)

(with-handler ignore-handler
  (with-handler retry-handler
    (optional-socket-operation)))
```

handler which is created by the macro `defhandler` and utilized by the `with-handler` macro (Listing 6). A handler associates a list of conditions (whether Java classes or XML QNames [5]) with an action (usually) provided by Vinz, making it possible to centralize condition-handling logic. Instead of repeating the list of conditions every time the programmer wants to take a certain action to handle a condition (in `handler-bind` forms spread throughout the program), the programmer can define a handler once and use it repeatedly in `with-handler`.

Vinz provides four actions (an action is just a function, so the workflow author is free to define additional actions). Two actions, `retry` and `ignore`, invoke an active restart of the same name. The functions created by `deflink` bind these restarts, and the programmer can also bind them. The `retry` restart is intended to be used to deal with possibly transient errors such as network connectivity failures without the programmer being forced to write an explicit loop. Ignoring a condition can be used to allow "optional" operations such as generating debugging data to fail without impacting the workflow.

The two remaining actions are `break` and `terminate`. These actions interact with Vinz fibers and tasks. The former action is named for the Java keyword of the same name. Intended to be used around a `for-each` distributed loop, the `break` action causes the currently executing fiber to immediately terminate cleanly and return `nil` to the parent fiber (other fibers are unaffected). In contrast, `terminate` terminates both the current fiber and the entire task with an error status. Any other fibers that are currently running or are queued by the message queue will notice that the task has terminated in short order and also terminate in error.

Both local and distributed conditions can be handled with these extensions. When a function created by `deflink` invokes a service, the response from the service might be an error, conveniently expressed as an XML QName. The function arranges for this QName to be signaled as an error, thus integrating distributed error conditions into Vinz handling.

4. Implementation

4.1. Futures And Continuations

The futures and continuations described in this paper are a fundamental part of the Gozer programming language. They, and the rest of the runtime semantics of the Gozer language, are implemented by a bytecode interpreter called the Gozer Virtual Machine (GVM), which runs on top of the Java Virtual Machine (JVM).

Continuations are not supported by the JVM (there is no way to capture a call stack and re-enter it later). This implies that the JVM's operand stack and function calling operations could not be used. Instead, the GVM implements its own stack-oriented architecture, in many ways similar to the JVM's architecture. The stack consists of ordinary Java objects representing function calls together with arguments, local variables, etc. These objects are used to create the continuations requested by `yield` and `push-cc`. This is similar to the approach taken by Stackless Python [6]. Compilation to bytecode (as opposed to a tree-walking interpreter) was introduced as an optimization for Vinz persistence.

In contrast with continuations, the JVM does provide the concept of futures through its `ExecutorService`. The challenge for the GVM here was to make them transparent to the programmer, completely managing their execution and determination, while allowing Vinz to integrate them into its distributed workflows (it is problematic to migrate a fiber from one machine to another while some of its futures were still running on the first machine). To do this, the GVM adopts the rule that passing any future to a Java library or a BlueBox service will cause that future to be determined. In addition, when capturing a continuation, futures referenced from that continuation are determined (the continuation doesn't become available until all futures have completed). Finally, the BlueBox platform provides an `ExecutorService` that integrates with its native load balancing heuristics, and Vinz configures futures to be created using this implementation.

4.2. Workflow Distribution

The BlueBox service framework provided many of the tools required to build distributed workflows: a service-oriented architecture, a message queue with addressable services, a global process tracking service, etc. Only a few additional features were required.

One clear need was a way to persist a fiber's state and data so that one instance could write it, and another instance could later read it and resume execution. The Java platform defines a built-in way to externalize in-memory objects called serialization, and so building on this support was the obvious approach. Vinz thus writes a fiber's state and data using Java serialization, with many customizations for efficiency and to broaden what can be successfully serialized. A shared NFS filesystem provides all instances with read and write access to this data. To prevent this filesystem from becoming a single point of failure in the distributed system, very highly-available network attached storage (NAS) servers running on purpose-built enterprise hardware are utilized.

Much time was spent optimizing Vinz serialization for performance. A series of tests determined that compressing the serialized data before writing it to NFS was a net win by reducing IO costs considerably, even though the Java serialization format is computationally expensive to compress with standard deflate-based compression techniques. It was also discovered that plain deflate can be made to perform approximately 30% better than the more robust and space-efficient gzip format for this data. Analysis of serialized data resulted in the introduction of a custom serialization format that stored the most commonly serialized objects more efficiently. Even after all this, reconstituting a fiber from its persisted state is still relatively slow and so a cache of recently seen fibers is maintained in memory on each instance. Because Vinz executes no control over where a fiber will be asked to run (leaving that in the hands of the message queue), the cache is only somewhat effective. Empirical measurements show cache hit rates of about 18% and 66% for mutable and immutable data, respectively.

Another obvious requirement was a way to prevent a single fiber from being run by different JVMs at the same time (for example, in the AwakeFiber case discussed above). Within a single JVM, the usual thread locks suffice for this, but distributed locks would be required. Since the persistence information was being shared using an NFS filesystem, the natural choice was to use file locks on the NFS files. This was simple and, thanks to the enterprise NFS servers in production use, effective, but it was completely opaque. Moreover, the need to be able to test in environments using different NFS servers required the development of code to cope with each implementation's unique quirks. To remedy these problems, a custom distributed lock implementation based on the Apache ZooKeeper distributed coordination system[2] is being developed as a replacement for NFS file locks. This same locking and persistence infrastructure is used to implement task variables.

5. Future Work and Conclusion

The Gozer workflow system is in high-volume production usage at RiskMetrics Group. A typical 24-hour period will see around 10,000 new top-level tasks comprising about 45,000 individual fibers. Tasks during this period may run for as long as 12 hours or as little as 20 milliseconds, with the average being about a minute. If these 10,000 tasks were run back-to-back, they would require about 190 hours to complete. Based on this production usage and

2. http://hadoop.apache.org/zookeeper/

experience, a number of improvements in monitoring and management, development effort, and runtime efficiency have been identified.

One area for future work is to have Vinz automatically learn which requests and loops do or do not benefit from task migration and to enable or disable it appropriately, instead of requiring the programmer to decide, and often guess. The `for-each` chunking function should also dynamically optimize chunk sizes based on the processing time of the body.

The cache effectiveness mentioned in the previous section is less than ideal. Further work is needed in this area, perhaps by devising a way to move the processing work to the last location of the data as is done, for example, in the Swarm system presented at IEEE P2P '09 in Seattle[3].

Task variables, although useful, have a very high synchronization overhead for mutation. Workflow authors have requested lighter-weight cross-process communication mechanisms.

Although the spawn limit solves an operational problem, its implementation currently is sub-optimal. Consider the case where the spawn limit is absent or very high relative to the number of workflow service instances available, n, and suppose that each child fiber is going to execute in approximately the same amount of time. Initially, n child fibers will be executing concurrently. When they finish, n AwakeFiber messages will be placed on the message queue and delivered for execution. Since a fiber can be executing on at most one instance at a time, $n-1$ of those AwakeFiber operations will be forced to wait while a single instance reads and updates the persistence information. Each AwakeFiber instance will proceed in turn, but for some period of time all n instances will be unavailable to process other activity such as other RunFiber requests (and because instances are often shared across services, even unrelated service operations may be blocked, something that Vinz seeks to avoid). To partially counteract this problem, AwakeFiber requests are specified to be low-priority, and a running AwakeFiber places a strict limit on how long it will wait for its turn to execute the fiber before giving up and placing itself back on the message queue for later delivery.

When the spawn limit is low (and especially if the spawn limit is low but the number of child fibers is high), the overhead of sending an AwakeFiber message for permission to spawn the next child seems high. It would be better if, as the child fiber died, it could simply spawn whatever sibling fiber is next without involving the parent. The difficulty here is synchronizing access to the parent so that it continues only when all of its children ultimately complete. Further work is needed in this area.

Finally, the enterprise message queue is currently completely responsible for load balancing and prioritizing messages. Each task starts and runs to completion independently of any other tasks that may be operating, subject only to the capacity limits of the system. In effect, task scheduling is first-come-first-serve, which has been shown to be suboptimal in the presence of deadlines [7]. Work is in progress to develop more efficient and proactive scheduling policies utilizing historical and current information about the state of the entire cluster (shared using ZooKeeper) and tasks in process based on scheduling research presented in [8].

Although still evolving to meet upcoming needs and extend current capabilities, Gozer addresses key requirements associated with processing a vast number and type of workflow computations submitted by, or on behalf of, numerous clients. The workflow requirements vary greatly in terms of computational complexity, overall duration, and terms defined by service-level agreements. Gozer's ability to easily exploit local and distributed resources through implicit parallelization together with its high-level language approach to workflow authoring have allowed the rapid development of dozens of production workflows.

3. http://vimeo.com/6614042

References

[1] S. Halloway, *Programming Clojure*. Pragmatic Bookshelf, 2009.

[2] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in Action*, 1st ed. Manning, 2007.

[3] R. H. Halstead, Jr., "Multilisp: a language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, 1985.

[4] P. Siebel, *Practical Common Lisp*. Berkeley, CA: Apress, 2005, ch. 19. Beyond Exception Handling: Conditions and Restarts.

[5] T. Bray, D. Hollander, A. Layman, and R. Tobin, "Namespaces in xml 1.0," W3C (World Wide Web Consortium), Recommendation, August 2006. [Online]. Available: http://www.w3.org/TR/2006/REC-xml-names-20060816

[6] C. Tismer, "Continuations and stackless python," in *Proceedings Of The Eighth International Python Conference*, Arlington, Virginia, January 2000.

[7] H. K. Shrestha, N. Grounds, J. Madden, M. Martin, J. K. Antonio, J. Sachs, J. Zuech, and C. Sanchez, "Scheduling workflows on a cluster of memory managed multicore machines," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '09)*, July 2009.

[8] N. G. Grounds, J. K. Antonio, and J. Muehring, "Cost-minimizing scheduling of workflows on a cloud of memory managed multicore machines," in *Proceedings of the 1st International Conference on Cloud Computing (CloudCom 2009), in Lecture Notes in Computer Science 5931*, December 2009, pp. 435–450.