Containment Domains

From OSKernelWiki

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

This document is intended for both application programmers and compiler/runtime programmers. The first part of the document, through the simple GPU use example, is most relevant to application programmers.

Document History:

- 11/01/2011: Published version 1.0

Contents

Introduction to Containment Domains

Why CDs?

Containment Domains (CDs) were created to allow an application to recover in the face of various types of component failures, including hard (e.g. persistent) and soft (e.g. transient) hardware failures, transient software failures, and complete node failures.

For the initial implementation, we have focused on handling the failures of an attached processor like a GPU. Without CDs (or some other protection), a GPU failure is fatal to an application although not fatal to the compute node itself.

What are CDs?

CDs can be thought of as a mechanism to support fine-grained, hierarchical, application-directed, uncoordinated checkpoint and restore. The application directs the 'data' recovery process through the CD API, which provides the following capabilities:

1. A processing element (PE) can recover in isolation from other PEs.
2. The granularity of recovery can be smaller than a PE, for example within a single function.
3. During execution, the application only needs to save the minimum amount of application data/state needed for re-execution. This data/state is saved in the CD store.
4. The CD store is updated atomically, so a CD is always guaranteed to be consistent.

Currently, the application must direct the 'control' recovery process to ensure that after calling the CD API to restore the data, the application restarts in the correct location. After data restoration is handled by the CD infrastructure, this restoration of control may take different forms. For example, it may be handled via the structuring of the application to start at the top of a loop once the correct loop index has been restored by the CD infrastructure. Alternately, the code may jump to the correct location via a 'goto'-type construct.

Individual PEs can have their own CDs. They can recover in isolation from other PEs in the application. This distinguishes the CD solution from a global Checkpoint Restart (CPR) solution where all of the PEs must restart.

Data within application memory only needs to be saved in a CD if that data is needed to re-execute the application. Typically, data only needs to be saved if the application is going to alter it from its initial state (i.e. READ-WRITE data). (However, READ-ONLY data can also be saved in a CD.) Thus, CDs only need to accumulate necessary data when the application is about to change it. CDs have been designed so that data can be added to a CD in a piecemeal fashion as the application needs, rather than needing to know up front all the data that may be changed. This feature allows a developer to easily incorporate CDs into his application.

All critical CD operations are atomic. This guarantees that, once created, there will always be a CD available to restore a point in time (at least until that CD is intentionally discarded). Even when advancing this CD to a newer point in time, a failure will not cause the CD to be lost.

How do CDs work?

CDs capture two types of state. They save data from application memory at a particular point in time prior to it being changed by the application, and they optionally log the MPI traffic for a processing element (PE). Saving this state allows CDs to restore a piece of an application to a specific point in time, so that it can start re-executing at that point. This point in time corresponds to when the data was saved.

CDs can be updated so that they capture a newer point in time than when they were first created. This ability allows them to accumulate work that the application has already performed. If a failure occurs and a recovery must be done, this means more work has been saved and less must be redone.

CDs can have child CDs. Smaller pieces of work within a PE can be saved within a child CD. Child CDs provide the following benefits. They can reflect a point in time that is newer than their parent. They can do this while still maintaining the parent's ability to recover to its point in time. On a recovery involving just the child CD where the child reflects a later point in time, less work needs to be redone because the point in time is newer. Also, on recovery involving just the child CD, only the data within the child CD needs to be restored. The data contained within the parent does not need to be restored.

Sibling CDs are child CDs with the same parent CD that are operating concurrently.

Child and sibling CDs are optional capabilities that may be used when they are appropriate for the structure and algorithms of an application.

Achieving Efficiency with CDs

It is possible to save data each and every time that data is modified. This could result in overwriting the data multiple times in a CD. There are performance costs (i.e. overhead) associated with storing data into CDs. There are also costs incurred when a failure occurs as some amount of work must be re-executed. Balancing these two costs helps determine the frequency with which data should be saved into a CD. The following example uses a GPU failure to illustrate the trade-offs.

Most HPC applications have an outer serial loop that marks progress of the computation towards a final solution. Usually this is a time step loop or an iterative convergence loop of an algorithm. For simplicity, outer serial iterations will be called 'time steps'.

In this context there are two types of CD data that must be saved: 'compute data' and 'MPI traffic'. The two different types need to be saved with different frequencies with regard to time steps.

The first type of data is 'compute data', consisting of both CPU and GPU application memory. This part of the CD is updated every 'N_CD' time steps, where N_CD is an application-specific parameter chosen to minimize total CD overhead. Every N_CD time steps each GPU updates to host memory sufficient GPU state (termed the 'rootCD') such that a failed and rebooted GPU can be restarted at an earlier point in the computation. Without going into modeling details, it is useful to understand the basic trade-offs made in choosing N_CD. On the one hand, choosing small N_CD incurs the performance penalty of using the PCIe bus to update rootCD in host memory with high frequency. On the other hand, large N_CD implies much redundant computation upon replay, since on average a GPU will fail at N_CD/2 iterations between updates of rootCD. The trade-off of these opposing trends leads to a well-defined optimization problem that can be solved with appropriate modeling of GPU failures.

For the second type of data, MPI traffic, every MPI call is saved for every time step. This is termed 'MPI logging'. The CD infrastructure logs a FIFO of all MPI calls. All MPI RECV and collective calls and associated data (since the last update of rootCD) are retained in host memory for later replay. All other MPI calls -- but not their data -- are retained in MPI logging and are skipped during replay.

Limitations

The API described in this document provides significant infrastructure for implementing CDs. As written, it can be used directly by programmers with some effort, and for CUDA applications is likely a reasonable strategy. For higher level languages (e.g. OpenMP with accelerator directives), this API could also be used directly, but ultimately provisioning additional directives for the higher level languages and better automating the integration of programming model semantics with the CD infrastructure would be preferable. For example, creating a CD API function that would automatically set the point in an application that the CD should restore to would alleviate the programmer from having to structure his program to do this control flow. That programming environment level work is not part of the current effort.

The current approach to CDs is an MPI-centric one. The CD API could be extended in the future to encompass other non-MPI languages, like PGAS, or to address other global-address space models. For the PGAS-like languages, this extension would involve substituting similar functions in the chosen language for the MPI_log functions mentioned in this API. Extending the CD infrastructure to support these additional languages or models will likely be harder and may have higher overhead than for MPI.

This is a first step in an incremental approach to Containment Domains. It provides a framework to start low-level implementation. For example, the current CD API calls add complexity and many additional lines of CD code when added to a program. Compiler support for CD directives, a deferred implementation, would make CDs easier to use and less of an impact to application developers. This support is likely necessary for broad adoption of the CD infrastructure.

While using the CD API can add complexity and additional lines of code, the amount of each depends on how CDs are used. A minimalist approach using one root CD without any child CDs will result in many fewer lines of code than a more complex approach. Application codes with multi-level CD schemes need child CDs. For example, a parent CD may be used for saving state in an outer time-stepping loop, while a child CD may be used for saving state in an inner loop doing conjugate gradient iterations.

This document focuses on transient failures and recoverable errors. For example, the GPU may encounter an error, but it can still be rebooted. This is a recoverable failure. The GPU has not encountered a hardware failure that renders it unable to reboot. A GPU that is unable to reboot is tantamount to a node failure, at least, for applications expecting to use a GPU.

Technical Description

A containment domain (CD) contains a copy of selected application memory and represents a specific "point in time" during application execution. Different CDs can represent different points in time. After a failure, a CD is used to restore application memory to a particular point in time prior to re-executing a piece of the application. The re-execution of one piece of the application can be done independently from other pieces of the application. This differentiates CDs from traditional Global Check Point Restart, which requires that the **entire** application be restarted from the same point in time **on all nodes**. With CDs, all of the pieces of the application on **one** node that fails can be re-executed, or just individual pieces of the application running on the node may be re-executed. These pieces may be running on just one PE on the node.

Terminology

- **Application Memory** -- the current, live contents of memory that the application is using
- **CD store** -- the physical location where the CD stores both the address ranges and their associated data; this storage may be local on the node or remote; this storage may be persistent.
- **Address Range** -- the address and its length where the application memory is stored
- **Data** -- In a CD, the data was copied from application memory to the CD store. Data is always associated with an address range.

- **Root CD** -- a CD that does not have a parent.
- **Parent CD** -- a CD that has a child CD.
- **Child CD** -- a CD that has a parent CD.
- **Sibling CDs** -- child CDs who share the same parent and exist concurrently
- **Recovery** -- Use the restore_cd() call to restore the application memory to the point in time captured by the CD. This allows the application to recover from a failure.
- **Active CD** -- the active CD in the calling thread. When a CD is created, the active CD is saved and the new CD becomes the active CD. When the active CD is destroyed the previously saved CD becomes the active CD (e.g. managed last in, first out). The constant CURRENT_CD can be used to refer to this CD on a CD API call. The active CD has thread scope and will reside in thread local storage.
- **PE** -- Processing Element; an individual process or thread carrying out a task independently of other PEs
- **CD entry** -- It is a data structure that refers collectively to the address range, range type, and the associated data added to a CD. Typically, a CD will contain multiple entries.
- **Restoration type** -- this indicates where a CD will obtain the data used during a restore. The restoration type is one of the following: COPY, PARENT, REGEN; and is determined by how the data was added to the CD: add_to_cd_via_<copy|parent|regen>.

CD Contents

CDs contain several types of information

1. Address ranges -- the address and type (read, write) of the data that the CD is tracking, but not the data itself
2. Data -- the data associated with an address range; the data is copied from application memory to the CD store
3. MPI traffic sent or received by a PE -- this traffic is required to re-execute a PE from a particular point in time independently from other PEs.
    1. For an MPI send, no data is logged, only the fact that the send occurred. For a receive, both the receive and the data are logged.

A CD can track an address range without copying the associated data to the CD store. This will be explained in the APIs.

Persistent Storage

Persistent storage may be located on or off of a node. Because node memory is volatile memory, persistence is a relative term. Volatile node memory is persistent in the face of an attached processor failure where only the attached processor's memory would be lost. A GPU is an example of an attached processor. When a node fails, its volatile memory, by definition, does not persist. For a node failure, non-volatile memory will be persistent. This non-volatile memory may be located off the node or on a non-volatile device attached to the node. Flash memory is an example of attached non-volatile memory. Additionally, in the case of a node failure, the volatile memory of another node can also be considered persistent, so long as that node does not fail.

Simple API Use Example

Here is a simple example GPU application:

```
main (int argc, char* argv) {
   int node_copy_of_result, i, j, error;

   load_gpu_kernel();

   for (i=1,1000) {
         run_gpu_kernel();
         copy_from_gpu(gpu_result, node_copy_of_result);  // Source: gpu_result (on GPU)
                                                          // Destination: node_copy_of_result (on CPU)
         MPI_Send(node_copy_of_result);
         MPI_Recv(....);
   }
}
```

And the example using containment domains with all GPU management shown for clarity and the loop counts set as described in the introduction:

```
#define N_CD 10

main (int argc, char* argv) {
   char *storage_info = NULL;
   int node_copy_of_result, i=1, j=1, error;
   /* The variable gpu_state is saving GPU memory. */
   /* The variables i, j, and node_copy_of_result are saving Node memory. */
   struct cd_addrspec as[4] = {{&gpu_state, sizeof(gpu_state), READ_WRITE, GLOBAL},
                               {&node_copy_of_result, sizeof(node_copy_of_result), READ_WRITE, GLOBAL},
                               {&i, sizeof(i), READ_WRITE, GLOBAL},
                               {&j, sizeof(j), READ_WRITE, GLOBAL}}

   cd = create_cd(NULL, storage_info, COMM_LOGGING_ENABLED, "root", &error);

   load_gpu_kernel();

   for (i=1,100) {

      add_to_cd_via_copy(cd, as, 4);  // before data modified and after each advance_cd_point_in_time
      error = 0;

      for (j=1,N_CD) {
         run_gpu_kernel();
         error = gpu_error();
         if (!error) {
            copy_from_gpu(gpu_result, node_copy_of_result);  // Source: gpu_result (on GPU)
                                                             // Destination: node_copy_of_result (on CPU)
            error = gpu_error();
         }

         error = application_detects_sdc();                  // App. needs to check for silent data corruption.

         if (error) {
            reboot_GPU();
            load_gpu_kernel();
            restore_cd(cd);
            break;
         }

         MPI_Send(node_copy_of_result);
         MPI_Recv(....);
      }

      if (!error) {
        if (advance_cd_point_in_time(cd) == FAILURE) {
           reboot_GPU();
           load_gpu_kernel();
           restore_cd(cd);
        }
      }
   }

   commit_cd(cd);
}
```

Piece by piece explanation:

```
#define N_CD 10

main (int argc, char* argv) {
   char *storage_info = NULL;
   int node_copy_of_result, i=1, j=1, error;
   /* The variable gpu_state is saving GPU memory. */
   /* The variables i, j, and node_copy_of_result are saving Node memory. */
   struct cd_addrspec as[4] = {{&gpu_state, sizeof(gpu_state), READ_WRITE, GLOBAL},
                               {&node_copy_of_result, sizeof(node_copy_of_result), READ_WRITE, GLOBAL},
                               {&i, sizeof(i), READ_WRITE, GLOBAL},
                               {&j, sizeof(j), READ_WRITE, GLOBAL}}
```

Definition of the data structure used in the add_to_cd_via_copy calls.

```
   cd = create_cd(NULL, storage_info, "root", COMM_LOGGING_ENABLED);

   load_gpu_kernel();

   for (i=1,100) {
```

Create the CD, load the GPU, and begin the outer loop. The limits of the inner and outer loop are chosen to balance the containment domain overhead and the amount of work lost on a failure based on the amount of modified data and MPI traffic involved.

```
      add_to_cd_via_copy(cd, as, 4);  // before data modified and after each advance_cd_point_in_time
```

Save application state in the CD. This needs to be done for all data that will be modified before it is modified for the first time or before it is modified again after a call to advance_cd_point_in_time (hence the placement just before the inner loop in this example).

```
      error = 0;
```

Clear error before starting inner loop.

```
     for (j=1,N_CD) {
```

Inner loop (total iterations 100 * 10). On a failure, a max of 10 (average of 5) iterations of the inner loop would have to be re-executed.

```
         run_gpu_kernel();
         error = gpu_error();
         if (!error) {
            copy_from_gpu(gpu_result, node_copy_of_result);  // Source: gpu_result (on GPU)
                                                             // Destination: node_copy_of_result (on CPU)
            error = gpu_error();
         }
```

Run GPU kernel, copy result to node memory. Check for errors.

```
         error = application_detects_sdc();                  // App. needs to check for silent data corruption.
```

The CD infrastructure does not check for Silent Data Corruption (SDC). This is the responsibility of the application.

```
         if (error) {
            reboot_GPU();
            load_gpu_kernel();
            restore_cd(cd);
            break;
         }
```

If there were any GPU failures; reboot the GPU, reload the GPU kernel, restore application memory, and restart execution from last good state. In this case the last good value of i will be restored and the inner loop will be re-executed from the beginning.

```
         MPI_Send(node_copy_of_result);
         MPI_Recv(....);
```

Exchange results with other ranks. This is done after checking for GPU errors to ensure that only good data is sent to other PEs.

```
      }
```

End of inner loop.

```
      if (!error) {
        if (advance_cd_point_in_time(cd) == FAILURE) {
           reboot_GPU();
           load_gpu_kernel();
           restore_cd(cd);
        }
      }
```

If the inner loop completed without error, update the CD with the current application memory (move the CD point in time atomically to "now") and discard MPI log. If this fails, then reboot GPU, restore state, and re-execute.

```
   }
   commit_cd(cd);
}
```

commit_cd discards the CD because it is no longer needed.

A short explanation of the CD APIs used in the example:

create_cd

```
cd_handle create_cd(cd_handle parent_cd,
    char *storage_info,
    enum comm_log log_mpi_traffic,
    char *name,
    int *error);
```

The create_cd API is used to create a containment domain object for the application to use to save and restore state.

add_to_cd_via_copy

```
int add_to_cd_via_copy(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount);
```

The add_to_cd_via_copy API is used to copy data to a containment domain before it is modified, so that it will be available to restore application memory to that value if needed in the future. Other copy variants are described later.

restore_cd

```
int restore_cd(cd_handle cd);
```

The restore_cd API is used to restore application memory to a value it had at a prior point in time.

advance_cd_point_in_time

```
int advance_cd_point_in_time(cd_handle cd);
```

The advance_cd_point_in_time API is used to update the state saved in the containment domain to the values that are in application memory at the point in time advance_cd_point_in_time is called. After advance_cd_point_in_time completes, a restore_cd call will restore application memory to the point in time of the most recent advance_cd_point_in_time call if any have been made.

commit_cd

```
int commit_cd(cd_handle cd);
```

The commit_cd API is used to discard CDs that are no longer needed. It is also used to merge a child CD with its parent as described later..

Containment Domain API for applications

Applications will use the following API functions to utilize Containment Domains. The API functions will use the lowercase abbreviation 'cd' which stands for Containment Domain. This differs from the text of this wiki, which uses uppercase CD.

These definitions will be used throughout the API descriptions.

```
typedef void * cd_handle;
#define CURRENT_CD (cd_handle)-1;  // Implicit CD handle

enum addr_type {READ_ONLY, READ_WRITE};
// READ_ONLY:    memory is used, but not modified
// READ_WRITE: memory is modified, and the initial value of memory is used

enum addr_scope {GLOBAL, CONSTRAINED};
//GLOBAL:  address range is valid for ancestor CDs
//CONSTRAINED:  address range is not valid for ancestor CDs

enum mpi_log {COMM_LOGGING_DISABLED, COMM_LOGGING_ENABLED, COMM_LOGGING_INHERIT};
// COMM_LOGGING_DISABLED:  MPI traffic will not be logged
// COMM_LOGGING_ENABLED: MPI traffic will be logged
// COMM_LOGGING_INHERIT:  MPI traffic logging will be the same as the parent CD

struct cd_addrspec {
   void * address;
   size_t length;
   addr_type  addr_tp; //READ_ONLY, READ_WRITE
   addr_scope addr_scope; //GLOBAL, CONSTRAINED
};
```

The cd_addrspec structure is initialized like this in many of the Use Cases.

```
int x;
struct cd_addrspec as[1] = {&x, sizeof(x), READ_WRITE, GLOBAL};
add_to_cd_via_copy(cd, as, 1);
```

create_cd

```
cd_handle create_cd(cd_handle parent_cd,
    char *storage_info,
    enum comm_log log_communication_traffic,
    char *name,
    int *error);
```

- input:
  - parent_cd: is NULL for a root CD; for a child CD, it is the handle of the parent CD or CURRENT_CD to indicate the currently active CD.
  - storage_info: describes the attributes of this CD's store (the default may instead be specified via environment variables):
    - These are some examples of the information the storage_info might contain:
      - caching
      - remote storage description
      - storage hierarchy
      - persistence attributes
      - name of a directory under which this CD and its descendants should be stored. It is recommended that all root CDs within an application specify the same directory.
  - log_communication_traffic: COMM_LOGGING_DISABLED, COMM_LOGGING_ENABLED, COMM_LOGGING_INHERIT
    - all CDs in a single parent/child relationship must specify the same value for log_communication_traffic. COMM_LOGGING_INHERIT will always work for children, but the root CD must specify a value either ENABLED or DISABLED. Stated another way, if the root CD logs communication traffic, so must all of its descendants.
      - If the root CD logs the communication traffic, but one of its children does not, then the root CD may not be able to effect a recovery because of the child's missing communication log. During a recovery operation, restore_cd requires the communication logs for the CD being restored. If that CD has any descendants, restore_cd needs the communication logs from those child CDs, too. The communication traffic in these logs will be replayed during the recovery operation. If a child CD did not log the communication traffic that it generated (because it had incorrectly specified COMM_LOGGING_DISABLED), then the recovery operation could not successfully do a replay of the communication traffic because of the missing log.
        - The CD infrastructure can detect a mismatch between a child's COMMUNICATION_LOGGING setting and its parents and flag an error.
  - name: A string containing a unique name to identify a root CD. A name should only be supplied for root CDs. For child CDs, the name string should be NULL. See [#CD Naming Conventions] for more information.
- output:
  - CD handle
  - error

Semantics:

- A new CD is created.
- If it is a child CD, it is associated with its parent CD.
- The new CD becomes the currently active CD, referenced by CURRENT_CD.

commit_cd

```
int commit_cd(cd_handle cd);
```

- input:
  - cd: the CD handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
- output:

    - error
- semantics:
    - Root (has no parent) CD:
        - Free CD store
        - Free all MPI traffic logs
    - Child (has a parent) CD:
        - For CD entries present in both the child CD and parent CD, do not update the entry in the parent CD. (Oldest wins.)
            - The "point in time" reflected by the parent CD does not change.
        - For an entry that is present ONLY in the child CD, but not in the parent CD, copy the entry to the parent CD.
        - The child CDs's MPI logs are handled on a situational basis.
            - If the child CD never had any siblings, its MPI log is merged with its parent
            - If the child CD has (or had) siblings, its MPI log is discarded. This may cause the rollback and re-execution of an ancestor CD to fail (it may be possible to aggregate sibling MPI logs such that they can be replayed correctly but further investigation is required, until that is complete sibling CDs should not use MPI if ancestor re-execution is required).
        - If the child CD contains entries incorrectly labeled as GLOBAL that may go out of scope in the parent CD, the results of commit_cd are undefined and potentially catastrophic. Those entries should be marked with CONSTRAINED to prevent them from being merged with the parent CD.
    - Note: The entries added to a CD should reflect the point in time captured by the CD. Their data will be used during a restore.
    - For entries present in both the child and parent, if the entry was labeled READ-ONLY in the parent but READ-WRITE in the child, then the parent's copy will be promoted to READ-WRITE.
        - The function advance_cd_point_in_time only updates READ-WRITE entries while leaving READ-ONLY entries alone.
        - This is true even if the entry was added to the CD as READ-WRITE using the function add_to_cd_via_parent().
    - A CD cannot be committed if it has any uncommitted children. The children must be committed first. If a CD that had a child was committed, it would orphan that child. This is not supported by the API.
    - The CD and its associated storage are invalidated and discarded. After performing a commit_cd, a restore_cd using this CD can no longer be done. If the CD was the active CD, the previously saved CD will become the active CD.
    - Committing a CD does not cause a recursive commit of the descendant CDs. All descendant CDs should be committed prior to doing a commit of an ancestor.

restore_cd

```
int restore_cd(cd_handle cd);
```

- input:
    - cd: the CD handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
- output:
    - error
- semantics:
    - restore_cd is the only CD operation that modifies application memory; the add_to_cd_via_<copy|parent|regen> and advance_cd_point_in_time operations only read/copy application memory to the CD store.
    - Application memory is restored to the point in time captured by the CD (including its uncommitted children)
        - Only the entries in the CD and in its descendants are used for restoration.
    - The "restored" CD--the one specified in the restore_cd() call--and its CD store continue to exist

after the call to restore_cd.

- If there are multiple CDs involved, restoration happens in this order. The leaf node child CDs are restored into application memory first. Then, the next oldest ancestor is restored. This continues on up the CD hierarchy until finishing with the CD specified in the restore_cd call e.g. restore newest to oldest; "oldest wins").
- Within a given CD, addresses are restored in this order: all addresses added by add_to_cd_via_copy, then all addresses added by add_to_cd_via_parent and finally all addresses added by add_to_cd_via_regen. This allows the regen function to depend on other application memory. Within each of those three groups the order the addresses are restored is not defined.
- Child CDs are treated as follows.
    - Once a child CD is used to complete the restoration, the child CD and its associated storage are invalidated and discarded. If the CD was the active CD, the previously saved CD will become the active CD.
    - While the child CD's entries are used to restore application memory, its entries are never added to its parent CD. Entries are only added on a commit_cd call.
    - The child CDs's MPI logs are handled on a situational basis.
        - When the CDs are strictly hierarchical (no siblings at any level), the MPI logs of children are merged with their ancestor's MPI logs on a restore.
        - In the case of sibling CDs, the MPI logs are discarded. This may cause the rollback and re-execution of an ancestor CD to fail.
- If the same CD entry is contained in multiple descendant CDs, the CD nesting order determines the final restored content ("oldest wins")
    - The same entry should not appear as READ-WRITE in sibling CDs or their descendants. READ-ONLY entries can appear in sibling CDs.
- Application memory is restored based on how the entry was added to the CD.
    - If memory was added via add_to_cd_via_copy, it is copied into application memory from the CD store.
    - If memory was added via add_to_cd_via_regen, it is copied into application memory using data supplied by the regen function.
    - If memory was added via add_to_cd_via_parent, it is copied into application memory using data provide by the parent. The CD entry in the parent provides the data based on how the entry was added (via copy, parent, or regen).
- Prepare MPI logs for replay
    - Quiesce outstanding async operations
- Behavior is undefined if an address range in the CD is deallocated prior to being used by restore_cd. See #Deallocating CD data.

advance_cd_point_in_time

```
int advance_cd_point_in_time(cd_handle cd);
```

- input:
    - cd: the CD handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
- output:
    - error
- semantics:
    - The purpose of advance_cd_point_in_time is to move the point in time captured by the CD to the present time. It accomplishes this by copying data from application memory into the CD store thus "updating" the entries. It only updates the READ-WRITE entries. READ-ONLY entries will **not** be updated. It also advances the file positions of any files that it is monitoring to their current file position.
        - READ-ONLY entries should be identical to application memory because the application memory should not have changed since the entry was originally added to the CD.
    - advance_cd_point_in_time demotes READ-WRITE entries to READ-ONLY entries.
        - The reason for the demotion is that the entry has been updated and should match what is

in application memory. If the application does not modify the corresponding memory before the next advance_cd_point_in_time call, then the entry won't need to be updated by the subsequent advance_cd_point_in_time call.
        - A CD entry must be promoted to READ-WRITE before advance_cd_point_in_time will update it.
        - To promote a READ-ONLY entry to READ-WRITE entry, the entry must be re-added to a CD as READ-WRITE via one of the add_to_cd calls.
            - This only changes the label of the entry in the CD, it does not copy any data into the CD because the data was already present in the CD.
        - This leads to an efficient use of advance_cd_point_in_time. It will only update entries that have changed. See the example.

```
Example:

- add_to_cd_via_copy 1GB of READ-WRITE data
- compute
- advance_cd_point_in_time // Updates all 1 GB and demotes it to READ-ONLY data
- add_to_cd_via_copy 9 bytes of READ-WRITE data
- advance_cd_point_in_time // Updates only 9 bytes

For the second advance only 9 bytes have changed, so only they will be updated.
The other 1 GB does not need to be updated.
```

- semantics (continued):
    - advance_cd_point_in_time does not update READ-WRITE entries that were added to the CD using the add_to_cd_via_regen or add_to_cd_via_parent functions. Only READ-WRITE entries added via add_to_cd_via_copy will be updated.
    - This function may **only** be called on a CD that does not have any children.
        - This restriction exists for two reasons.
            - 1. Uncommitted child CDs might contain entries that were not in the parent, but would still be needed for a recovery at the parent level. Advancing the parent CD without these entries means that a current copy of the data is not added to the parent CD. Thus, a current copy of the data does not exist anywhere in a CD store. It does not exist in the parent. It does not exist in the child CD because the child CD has not been advanced and still contains the old data. This absence of current data would prevent the parent from recovering its new point in time.
            - 2. Even if all the entries in the child are contained in the parent, they may not have the same labels. Some entries may be READ-WRITE in the child, but READ-ONLY in the parent. All of the child CDs must be committed before an advance_cd_point_in_time call to ensure that all of the entries within the parent CD are properly labeled. Committing a child CD with READ-WRITE entries will promote those same entries in the parent CD from READ-ONLY to READ-WRITE. If the entries are not labeled READ-WRITE when they should be, then the advance will not update the data associated with those entries.
    - If the CD being advanced is a child CD, then it will first be implicitly committed to its parent before its data is updated.
        - Prior to the advance, the child may have saved some data that is required to restore the parent's point in time. For this reason, the implicit commit copies entries that are present in the child but not in the parent to the parent. This is done to preserve the parent's ability to restore its point in time.
        - The implicit commit promotes any READ-ONLY entries in the parent, where those same entries are READ-WRITE in the child, to READ-WRITE in the parent as well.
        - The only difference between an explicit commit (i.e. actually calling commit_cd) and the implicit commit done by advance_cd is that the child CD, the one being advanced, is not invalidated.
        - As with commit_cd, the implicit commit will not update the parent CD if the entries in the child already exist in the parent.
            - If advance_cd is called a second time on a child CD and no new entries have been added to that CD, then the implicit commit will not update the parent CD because the

        parent already contains all of the child's entries.
    - As with commit_cd, the implicit commit will follow the same rules for handling MPI logs. In the case of sibling CDs, they are discarded. For only children, they are merged with the parent CD.
    - If the child CD contains entries incorrectly labeled as GLOBAL that may go out of scope in the parent CD, the results of advance_cd_point_in_time are undefined and potentially catastrophic. Those entries should be marked with CONSTRAINED to prevent them from being merged with the parent CD.
  - The advance updates the CD store atomically. Either all of the entries that need to be updated in the CD will be updated or none of them will.
    - In the case where the atomic update fails, this function will return an error.
  - advance_cd_point_in_time is more than the sequence of create_cd/add_to_cd/commit_cd, which are individually atomic but the composition of all of them is not atomic
  - This CD's MPI logs are discarded.
    - There is no point to retaining the MPI logs because they occurred prior to the point in time that this CD now represents.

add_to_cd_via_<copy|parent|regen>

There are three functions that can be used to add entries to CDs. They differ only in regard to how they obtain their data. This data is only used during a restore_cd operation. Therefore, a generic description of their functionality is given here with the differences spelled out in subsections.

```
int add_to_cd_via_<copy|parent|regen>(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount, char * storage_info);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
  - addrlist: list of address/length pairs of application memory to be added to the CD. Referred to as address ranges. It also contains the type of each address range, READ-ONLY or READ-WRITE. It also contains a flag that indicates whether this entry should be merged to its parent on a commit_cd or advance_cd_point_in_time call. (A CD entry refers to the address range, it type, and the data associated with the address range.)
  - ascount: number of entries in addrlist
- output:
  - error
- semantics:
  - In a cd_addrspec structure, each address range is labeled as either READ-ONLY or READ-WRITE.
    - These labels are important because the function advance_cd_point_in_time will only update the CD entries labeled READ-WRITE.
    - If a READ-ONLY entry exists in a CD, and that same entry is added to the CD a second time, but this time as a READ-WRITE address, it will be promoted to READ-WRITE. This is how a READ-ONLY entry is promoted to READ-WRITE. Data is not copied on a promotion because it is already contained in the CD entry. If only a portion of an entry is added as READ-WRITE, then only that portion will be promoted.
    - If a READ-ONLY entry exists in a CD, and that same entry is added to the CD a second time, but this time as a READ-WRITE address, it will be demoted to READ-ONLY. This is legal so long as the original data in the CD matches what is in application memory. Otherwise, this is a bug.
    - The advance_cd_point_in_time will demote all READ-WRITE entries to READ-ONLY. After a call to advance_cd_point_in_time, entries must be re-added to the CD as READ-WRITE entries to promote them back to READ-WRITE status.
    - The READ-WRITE label only applies to entries whose restoration type is COPY or to entries whose restoration type is PARENT but who refer to an ancestor whose entry has a restoration type of COPY. On an advance, an entry whose restoration type is COPY and who

        - is labeled as READ-WRITE will be updated. No other entries are updated during an advance. Therefore, the READ-WRITE label is not applicable to an entry whose restoration type is REGEN or to any entries whose restoration type is PARENT that refer to a REGEN entry. Creating a READ-WRITE, REGEN entry is incorrect and will be flagged as an error.
    - In a cd_addrspec structure, each address range is labeled as either GLOBAL or CONSTRAINED
        - GLOBAL indicates the address range is valid for use in ancestor CDs.
        - CONSTRAINED indicates the address range is invalid for use in ancestor CDs, for example a stack variable. An address range marked as CONSTRAINED will not be merged into a parent CD by commit_cd
    - If an address range needs to have its address scope changed, then it must be deleted from the CD and re-added with the new scope.
    - Each address range also has a restoration type (COPY, PARENT, REGEN) which governs where it obtains its data during a recovery operation.
        - An address range's restoration type can be changed by re-adding the address range to the CD with a different restoration type. There are consequences to performing this change. See the Table *Consequences of Changing Restoration Type* below.
            - This implementation is deferred for the prototype.
    - Adding to the CD is an atomic operation. It returns an error if the addition fails.
    - Adding to the CD is thread safe and may be called concurrently.
    - During restore_cd, if a CD contains one or more regen functions, which may have a dependence on other data being restored in the CD, the addresses are restored in the following order: all addresses added by add_to_cd_via_copy, then all addresses added by add_to_cd_via_parent and finally all addresses added by add_to_cd_via_regen. This allows the regen function to depend on other application memory, which must be restored first. Within each of those three groups the order the addresses are restored is not defined. If there are no regen functions present in a CD, then this ordering is not enforced.
- In the future, these functions may be extended, so that the storage location can be specified on a per CD entry basis. Currently, the storage location is specified on a per CD basis.

Table: Consequences of Changing Restoration Type

Original Type	New Type	Consequence to the changed CD entry
PARENT	REGEN	Change the data pointer to point to the regen function.
REGEN	PARENT	Change the data pointer to point to the parent CD.
COPY	PARENT	Change the data pointer to point to the parent CD. Free the data in the CD store.
COPY	REGEN	Change the data pointer to point to the regen function. Free the data in the CD store.
PARENT	COPY	Copy the application data into the CD store. Change the data pointer to point to this data. Copying in this data could cause a point in time problem if the data is different than what would have been added at the time that the entry was first created. If that is the case, then this is a bug in the application and an incorrect usage of this feature rather than a bug in CDs.
REGEN	COPY	Same as PARENT->COPY.

add_to_cd_via_copy

```
int add_to_cd_via_copy(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount, char * storage_info);
```

- semantics:
    - It copies the data at the specified address ranges from application memory into the CD store

according to the following rules #Rules for adding data to CDs.

add_to_cd_via_parent

```
int add_to_cd_via_parent(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount, char * storage_info);
```

- semantics:
  - The address range, but not the data, is tracked in the CD entry.
  - When restore_cd is called, entries created this way will obtain the data from the parent CD.
    - The data does not need to exist in the parent CD's entry. The data may exist in any ancestor.
  - The parent CD must contain this address range; otherwise, this is an error.
  - When an ancestor CD uses a regen function to restore an address range, this address range cannot be referred to by a descendant through the add_to_cd_via_parent API. The reason for this is that the ancestor CD may not be restored during a restore_cd call; only the descendant CD may be restored. If the ancestor is not restored, the regen function may not have all of the necessary state to work correctly. For example, there may a state variable in the ancestor CD that dictates the regen function's behavior. If that variable was not restored, then the regen function may not work. Therefore, this type of reference from a descendant CD is not allowed.

add_to_cd_via_regen

```
int add_to_cd_via_regen(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount, char * storage_info,
   int (*regen)(struct cd_addrspec addrlist[], int ascount));
```

- input:
  - regen: pointer to a function to be called when restore_cd is called. This regen function will be called with the address ranges in addrlist. The regen function (or programming environment runtime) will regenerate the data in application memory. This function should return 0 on success. This function must be idempotent (see semantics below).
- semantics:
  - The address ranges and associated regenerate function are recorded in the CD entry.
  - The specific address range passed to the regen callback function may be different than the addresses originally added to the CD:
    - An address range may have been split into multiple ranges due to changes in the read/write attribute of a sub range
    - Multiple adjacent address ranges may be combined into a single address range
  - During node failures (deferred implementation) the regen function may be called multiple times for the same address range and must produce the same data each time without any side effects (e.g without changing any other state, see http://en.wikipedia.org /wiki/Idempotent#Computer_science_meaning.

add_file_to_cd

```
int add_file_to_cd(cd_handle cd,
   int filedes);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
  - filedes: open file descriptor
- output:
  - error

- semantics:
  - the current file position of the file is added to the CD
    - file must support lseek system call
  - restore_cd will restore the current file position to the same offset
  - file data is **not** saved
  - only appropriate for file i/o that is idempotent (for example sequential writes or reads)

delete_file_from_cd

```
int delete_file_from_cd(cd_handle cd,
   int filedes);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
  - filedes: open file descriptor
- output:
  - error
- semantics:
  - the current file position of the file is deleted from the CD

delete_from_cd

```
int delete_from_cd(cd_handle cd,
   struct cd_addrspec addrlist[],
   int  ascount);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
  - addrlist: list of address/length pairs for memory to be deleted from the CD
  - ascount: number of entries in addrlist
- output:
  - error
- semantics:
  - The CD entry containing the specified address range is deleted from the CD store.
  - Only the CD itself is searched for the address range. Neither its ancestors nor its descendants are searched.
    - Deleting an entry from a parent CD that is referenced by a child CD via an add_to_cd_via_parent results in undefined behavior.
  - This function should be used prior to freeing dynamically allocated application memory memory.
  - This function will improve storage efficiency.
  - This function will improve time efficiency by reducing the time a search takes because of the reduced number of CD entries.
  - This is an atomic operation. It returns an error if the deletion fails or the address in not found.
  - It is thread safe and may be called concurrently.
  - Only remove entries that are no longer needed for recovery. Removing entries needed for a recovery will prevent the recovery from succeeding.
  - Deleting the last entry in a CD will **not** cause the CD to be discarded.

CD Naming Conventions

Each CD will be uniquely identified by an identifier consisting of the following attributes

1. Rank -- rank within the job
2. Name -- a name provided by the application for each root CD. The name is an argument to the create_cd() function. The name distinguishes one root CD from another if there are multiple root CDs

per rank.
3. Depth -- depth within a hierarchical CD chain. This distinguishes ancestors from descendants.
4. Sibling Index -- designator within one level of a CD chain. This distinguishes sibling CDs from one another.

The CD infrastructure will provide the rank, depth, and sibling index components. The application must provide the name component.

The CD name will be used to find CDs in the persistent store after a node failure or any other failure where the PE that created the CD has failed.

A rank may have one or more PEs executing on it. Each PE could create one or more CDs. In the case of a node failure, a discovery process must ensue wherein CDs are re-associated with the PEs within a rank. Because there could be multiple PEs and thus multiple CDs per rank, then the name component is used to disambiguate one root CD from another.

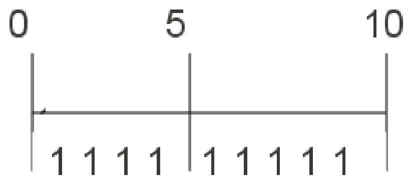
Why is it that the CD infrastructure cannot take care of naming individual CDs? Why does the application need to supply the root CD with names? During recovery after a node failure the application will re-execute. When the application calls the create_cd function, the CD infrastructure will match the persistent CDs, which are identified by name, with the name provided in the create_cd function call. At that point, the CDs can be restored, and the application will begin re-executing from the restored point in time. The names are necessary to guarantee that a CD is associated with the correct thread because different threads of the same rank may execute different code. For example if thread A creates CD A and does algorithm A, while thread B creates CD B and does algorithm B - then during re-execution thread A needs to find use CD A, not CD B. Without names we would have to depend on the order of the calls to create_cd and as A & B are independent threads ordering is not guaranteed.

Note: If there were only one CD per rank, then the rank ID would suffice to re-associate a CD with a rank. This could be handled by the CD infrastructure.

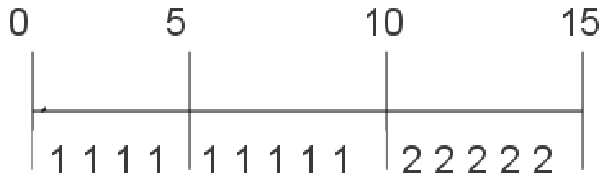
Rules for adding data to CDs

- The application memory--referred to in this subsection as 'data'--that is added to the CD store may be node local memory or may be located on another device accessible from the node.
- Only add_to_cd_via_copy copies data to the CD store. add_to_cd_via_regen and add_to_cd_via_parent provide ways to obtain the data during a recovery operation, but they do not copy data to the CD store.
- If data is copied to a CD and its address range overlaps--either partially or completely--with an address range that was previously added to the CD, then the earlier address range is NOT updated (i.e. not overwritten). Only the non-overlapping part is added to the CD. See the diagram below. In it, the two address ranges contain different data, 1's versus 2's. When the second address ranges is added to the CD, the overlapping portion (5-10) is NOT overwritten.
- The behavior when the same READ-WRITE address range is added to unrelated CDs that are operating concurrently is potentially catastrophic.
  - Unrelated CDs are CDs that are not related. Neither CD is the ancestor or descendant of the other CD.
  - For example, two independent PEs, A and B, each create a CD, CD-A and CD-B, respectively. At time T0, CD-A saves X when X has a value of zero. At time T1, CD-B saves X when X has a value of one. Now, CD-A and CD-B both contain the address range for the variable 'X'. There are two ways that this can go badly. First, if at time T2, both PE A and PE B decide to restore their respective CDs, then it is a race to restore the variable X. The last PE to restore X will determine its value. The other procedure would have been restored with the wrong value. Second, if at any point in time, one of the PEs decides to restore its CD, then the other PE has the variable X modified out from under it.

Persistent Store After Addition of address range 0 – 10; Data: 1's



Persistent Store After Addition of address range 5 – 15; Data: 2's



Deallocating memory referenced by a CD

- The behavior is undefined if the application memory being copied to a CD is deallocated prior to being used by restore_cd, commit_cd or advance_cd_point_in_time call.
  - The application can explicitly deallocate malloc'd memory by freeing it. The application implicitly deallocates memory if the memory was allocated for an automatic variable (stack variable) that went out of scope because the stack frame was popped.
  - Once memory is deallocated, it no longer has the same meaning to the program that it once did. It may be unallocated free memory, or it may have been reallocated for another purpose. If a CD contains a memory address that was deallocated *after* it was added to the CD, and then that CD is restored, this deallocated address is overwritten with the data in the CD.
    - Overwriting memory that was re-allocated corrupts the program's data.

        - Overwriting "free" memory is also inadvisable because it may overwrite meta-data that the allocator stores there although the consequences vary based upon the implementation.
    - Automatic variables can be added to a CD if that CD is the only CD being restored, and it is being restored within the same stack frame that the variables were instantiated. This CD should not be committed to a parent CD that resides outside the current stack frame. Doing so would add these automatic variables to a CD where the variables would be deallocated. Automatic variables should be deleted from the CD via delete_from_cd prior to returning from the stack frame that contains them. The CDs could also be discarded via a commit_cd call, if they are root CDs.

Concurrent Functions

The listed functions are thread safe and may be called concurrently with the same CD handle. Other functions may not be.

1. create_cd
2. add_to_cd_via_copy
3. add_to_cd_via_regen
4. add_to_cd_via_parent
5. add_file_to_cd

The following functions are not thread safe and may not be called concurrently with the same CD handle.

1. commit_cd: Concurrent commits of the same CD are not allowed. A CD may only be committed to its parent one time. After the commit, the CD is invalid.
2. restore_cd: Concurrent restores are not allowed. Concurrent restores of the same CD are redundant because they are restoring the same state to application memory. If two threads share a CD, when one restores the CD, it affects application memory, which both threads are dealing with. Therefore, when one thread does a restore, it is as if the other thread did a restore as well. This is a complication that multi-threaded applications must address.
3. advance_cd_point_in_time: Only one thread can advance a CD at a time.

Containment Domain API for MPI Interposition Layer

As part of the CD infrastructure, an interposition layer will be added between the application and the MPI libraries. This interposition layer will intercept the MPI calls made by the application. These MPI calls will be logged for possible replay. On a recovery, this log will be used to deal with MPI traffic.

MPI log replay allows application re-execution of a PE to occur in isolation from other PEs. During re-execution, all MPI sends and receives executed by the application are handled by the MPI log. This is called **replaying** the MPI log. For receive type calls (recv, barrier, read collectives), the MPI log delivers the data. For send type calls (send, send collectives), if the MPI log has a corresponding send, then the application's send is discarded. It is discarded because the application has already done the send once. It has already been received and cannot be called back. Doing a second send during re-execution would be a mistake. During replay, each application receive call and send call "consumes" the corresponding entry from the MPI log. Once the MPI log has been completely "consumed", then replay is over. From that point on, the application's MPI calls are executed "for real" and are once again logged by the CD infrastructure.

MPI log replay requires that application re-execution produces the same MPI calls in the same order as the original execution. Applications that use conditional execution based on external state (for example current time or a random number generator) may need to change to allow MPI log replay to work.

The interposition layer will allocate space for the MPI logs. It will hand pointers to those logs to the rest of the CD infrastructure. The CD infrastructure will be responsible for the lifetime and disposition of those logs. Specifically, for the MPI logs associated with a CD, the CD infrastructure will

1. Discard the logs after a successful advance_cd_point_in_time call.
2. Merge a child CD's MPI logs with the parent CD's MPI logs (in the strictly hierarchical case) after a successful restore_cd call.

3. Merge a child CD's MPI logs with the parent CD's MPI logs (in the strictly hierarchical case) after a successful commit_cd call.
4. If a child CD has (or had) siblings, discard the child CD's MPI logs on both restore_cd and commit_cd as it is not possible to merge the logs because they were created concurrently without any time order relationship. Discarding the logs can break the ability of an ancestor CD to rollback and re-execute correctly.

The following functions will be called by the interposition layer. They will not be called by the application.

add_MPI_log_to_cd

```
int add_MPI_log_to_cd(cd_handle cd,
    void *logent,
    int  loglen);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
  - logent: pointer to an MPI log entry to be associated with the CD
  - loglen: length of the logent data
- output:
  - error
- semantics:
  - the containment domain must have MPI logging enabled (see create_cd)
  - the logent parameter must be malloc'd by the caller; the CD infrastructure controls the lifetime and disposition of that memory after this call completes without error
  - the logent data must not contain any pointers (because whatever they point to is not saved)
  - the MPI log entry are added to a FIFO queue and will be returned by get_MPI_log_from_cd in FIFO order
  - calling add_MPI_log_to_cd during log replay (e.g. after restore_cd and before get_MPI_log_from_cd has exhausted the log) will return an error

get_MPI_log_from_cd

```
void * get_MPI_log_from_cd(cd_handle cd, int *error);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
- output:
  - error
  - MPI log entry originally provided by add_MPI_log_to_cd
  - NULL (no more log entries)
- semantics:
  - restore_cd will initialize get_MPI_log_from_cd state to the first entry in the log
  - each call to get_MPI_log_from_cd will return the next log entry in FIFO order as originally added by add_MPI_log_to_cd
  - when there are no remaining log entries, get_MPI_log_from_cd will return NULL

delete_MPI_log_from_cd

```
int delete_MPI_log_from_cd(cd_handle cd);
```

- input:
  - cd: handle returned by a prior create_cd or CURRENT_CD to indicate the currently active CD.
- output:
  - error

- semantics:
    - Any MPI log associated with the CD is discarded.

Motivation

Preserving a CD's Ability to Restore its point in time

A primary reason for Containment Domains is to restore an application to a specific point in time. Coupled with this goal is the desire to be able to move a CD forward from its current point in time to a newer point in time.

To preserve a CD's ability to restore a particular point in time, it must contain the initial states of the data that have been changed. If it has not captured an initial state before it was changed, it cannot fully restore that point in time. This initial state may be captured in a CD or one of its descendants. If some of the initial state that was captured is overwritten in either the CD itself or in one of its descendants, then the CD has lost its ability to restore that point in time because it would restore incorrect data.

For this reason, the function advance_cd_point_in_time may not be called on a CD with any uncommitted children. All the children should be committed prior to calling advance_cd_point_in_time. This ensures that the parent CD contains all of the address ranges needed to accurately represent the point in time. Uncommitted children could have entries not contained in the parent CD. Advancing_cd_point_in_time on the parent CD while it still had uncommitted children would move the parent CD forwarded in time without updating the entries from the uncommitted child. If the uncommitted child was not advanced as well, it would be no longer be in sync with it parent in regards to the point in time they are both supposed to be representing. Attempting to restore the parent CD would result in undefined, likely incorrect, behavior.

A code example will help illustrate this point why advancing a parent CD while a child CD is uncommitted breaks the parent's ability to restore. The initial state is that x=0 and y=0 at time T0. Later, at time T1, it is x=1 and y=1. To be able to restore time T1, x and y must be restored to their values at time T1.

```
int x=0, y=0;  // Initial state; Time T0

main (int argc, char* argv) {
       int x = 0;
       char *storage_info = NULL;
       struct cd_addrspec addrlist1[1] = {&x, sizeof(x), READ_WRITE, GLOBAL} ;
       struct cd_addrspec addrlist2[1] = {&y, sizeof(x), READ_WRITE, GLOBAL} ;

       parent = create_cd(NULL, storage_info, COMM_LOGGING_ENABLED);
       add_to_cd_via_copy(parent, addrlist1, 1);         // parent contains: x=0
        x += 1;                                           // x=1;

        child = create_cd(parent, NULL, MPI_LOGGING_INHERIT);
        add_to_cd_via_copy(child, addrlist2, 1);          // child contains: y=0

        y += 1;                                           // y=1;

      // At Time T1 application memory contains x=1 and y=1.

       advance_cd_point_in_time(parent);                 // parent contains: x=1, but not y=1. See the Notes.

       commit(child);
       commit(parent);                                   // This commit will discard the CD 'parent'.
        exit(0);
}
```

Notes:

1. Advancing the point in the time of the root CD while child is uncommitted breaks its ability to restore back to time T1. The value of y is only contained in the child CD and is not present in the root CD. Because the child had not committed to root, the root did not contain an entry for y. Without a value for y, the root CD cannot restore back to time T1.

CDs do not need to know the entire state that needs to be saved.

When a CD is created, it does not need to know the entire application memory needed to represent the current point in time. Application memory only needs to be added to a CD before it is modified for the first time. Allowing the function that modifies the memory to add it to a CD supports good software engineering, specifically separation of concerns (http://en.wikipedia.org/wiki/Separation_of_concerns) , abstraction (http://en.wikipedia.org/wiki/Abstraction_principle_%28programming%29) , and modular programming (http://en.wikipedia.org/wiki/Modular_programming) . Application memory can be captured in either a root CD or one of its descendants. Application memory is added to the CD via an add_to_cd_via_<copy|parent|regen> call.

Having to know all of the application memory that was going to change "up front" would put an onerous burden on either the application developer or the compiler. The developer would need to track through all of the functions looking for application memory that is changed, so it could be added to the CD. Alternately, the compiler would need to do similar heavy lifting to automatically, correctly add all of the application memory to the CD that might be changed by the code. Also some application memory may be conditionally added to the CD based on other values of application memory that can only be determined at runtime.

The only requirement placed on the application is that memory be added to a CD before it is modified. So long as that requirement is satisfied the application can structure the CD add_to_ calls any way it chooses.

Data Veracity Checking: Silent Data Corruption Etc.

The CD infrastructure will not provide checks to ensure the veracity of the data in the CD store. It is possible that the CD infrastructure could aid in the detection of silent data corruption (SDC). It could compare read-only data in its store against the data in live memory. Any discrepancies would indicate SDC or some other error. Checks like these could be costly, and thus they would need to be done only at the user's discretion. However, no such veracity checks are planned for the current implementation of CDs.

Strategies for addressing transient and non-transient failures

Using CDs with MPI logging enabled, allows an application to recover from transient failures, like a GPU failure. During the recovery process, the MPI log allows the node to recover by itself in isolation without involving any other nodes because it can replay the MPI traffic.

For non-transient failures, like a node failure, an application will pursue a different strategy. The difficulties presented by outstanding communication traffic, which is lost during a node failure, precluded using the logging strategy that is used for transient failures. The application must coordinate its own recovery amongst its nodes using CDs to restore data on the node that replaces the failing node (either a spare node or the failed node which has been rebooted).

This is the current direction for this project. Should new methods for successfully logging communication traffic during a non-transient failure be discovered, then this may suggest that the non-transient and transient failure cases could be handled similarly with logging.

Use Cases

One CD used to recover from a transient CPU failure

In this use case one CD is used to recover from a non-fatal CPU failure. The CD is used to accumulate the results of completed computations. Each addition of data to the CD moves the point in time captured by the CD forward.

```
int done = 0, compute_done = 0, error, compute_error;
char *storage_info = NULL;
struct cd_addrspec addrlist[2] = {{&data, sizeof(data), READ_WRITE, GLOBAL},
                                  {&compute_done, sizeof(compute_done), READ_WRITE, GLOBAL}}

cd = create_cd(NULL, storage_info, COMM_LOGGING_DISABLED, "root", &error);

while (!done) {
  add_to_cd_via_copy(cd, addrlist, 2);
  compute_error = compute();    // Sets compute_done when finished.
  if (compute_error) {
    restore_cd(cd);
  } else {
    if (compute_done) {
         done = 1;
    } else if (advance_cd_point_in_time(cd) == FAILURE) {
      restore_cd(cd);
    }
  }
}
commit_cd(cd);
```

Step by step explanation:

```
int done = 0, compute_done = 0, error;
char *storage_info = NULL;
struct cd_addrspec addrlist[2] = {{&data, sizeof(data), READ_WRITE, GLOBAL},
                                  {&compute_done, sizeof(compute_done), READ_WRITE, GLOBAL}}

cd = create_cd(NULL, storage_info, COMM_LOGGING_DISABLED, "root", &error);
```

Create a new CD.

```
while (!done) {
```

Loop until global flag indicates complete.

```
add_to_cd_via_copy(cd, addrlist, 2);
```

Add the state to it. The state needs to be added initially as well as after every advance_cd_point_in_time.

```
  compute();
```

Perform computation using saved state.

```
  if (compute_error()) {
    restore_cd(cd)
```

Error found in results of computation, restore application state from CD contents; while loop repeats execution.

```
  } else {
    if (compute_done) {
         done = 1;
```

If we made it here, there was no error. So, check if we are done yet.

```
    } else if (advance_cd_point_in_time(cd) == FAILURE) {
      restore_cd(cd)
    }
```

No error found in computation, and we're not done yet. Update CD with new output state and continue. Each iteration updates the data in the CD to progress the application and allow restoration to a later point in time.

```
  }
}
commit_cd(cd);
```

Loop complete, commit (and discard) CD.

CDs representing static points in time

The following use cases illustrate CDs that always represent the same point in time. They do not "move forward" in time because they are never updated with "changed" data. They are only updated with data representative of that point in time. Thus, while a CD accumulates more state information about its point in time, it is always state information that was unchanged since the CD was created. This illustrates a feature of CDs: they do not have to capture the entire state of the memory that will be changed when the CD is created. Any time memory is going to be changed, then prior to that change, the memory can be added to the current CD or a descendant CD. This eases the burden on the programmer. The programmer does not need to know "up front" all of the variables that may possibly change. Upon either a restore_cd call or a commit_cd call, the data in the child CD is appropriately incorporated with that of its parent CD.

When a parent's data is NOT overwritten with a child's data, this is referred to as "parent wins". When a parent's data IS overwritten with a child's data, this is referred to as "child wins". If a child adds data to a parent CD because the parent does not contain that data, it is referred to as "child wins". During a commit_cd call or a restore_cd call, a child CD's data never overwrites a parents CD's data. During an advance_cd_point_in_time call, the CD's data is updated/overwritten. Advance_cd_point_in_time can only be called on a root CD, so there are no children to worry about.

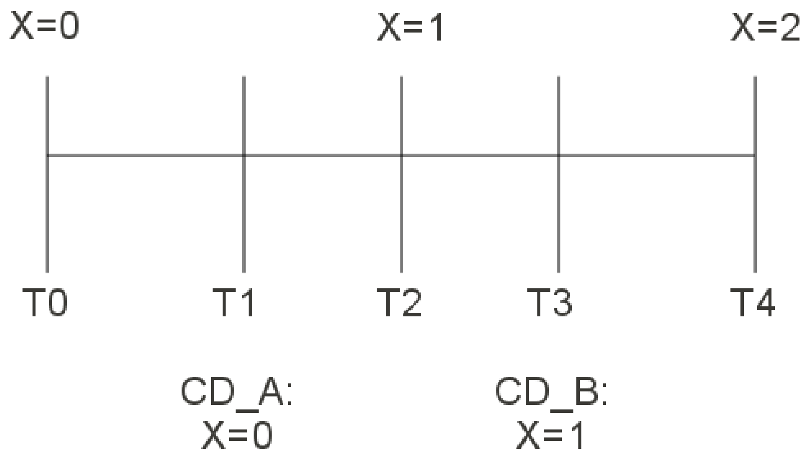
STATIC 1: Child CD stores changed data which is not used during a commit/restore.

In this use case, during a commit_cd or a restore_cd, data from a child CD does not overwrite data in the parent CD. Overwriting the data is undesirable because it would move the point in time represented by the parent CD forward to that of the child's point in time. Here, the parent CD preserves its point in time because its data is not overwritten by the child's. During a commit_cd, data that is present in the child, but not in the parent, is merged into the parent CD. To demonstrate this point, the root CD in main was left intentionally vacant of any data. Normally, it is nonsensical to create a CD and not populate it with any data or use it in any way (like restore_cd or advance_cd_point_in_time).

```
int x=0;

main (int argc, char* argv) {
        char *storage_info = NULL;
        int error;
        root = create_cd(NULL, storage_info, COMM_LOGGING_ENABLED, "root", &error);
        a(root);
                                 // Upon return from function a(), the CD 'root' contains x=0
        commit_cd(root);         // This commit will discard the CD 'root'.
        exit(0);
}

void a(cd_handle parent) {
       int error, compute_error;
       struct cd_addrspec addrlist[1] = { {&x, sizeof(x), READ_WRITE, GLOBAL} };
        cd_a = create_cd(parent, NULL, MPI_LOGGING_INHERIT, NULL, &error);
        add_to_cd_via_copy(cd_a, addrslist, 1); // cd_a contains: x=0
   TOP:
        x += 1;                                    // x=1;
       compute_error = compute_stuff();

        if (compute_error) {
                restore_cd(cd_a);               // Restore app memory to: x=0
                goto TOP;
       }

        //Call another function.
        function_error = b(cd_a);

        if (function_error) {
                restore_cd(cd_a);               // Restore app memory to: x=0
               goto TOP;
        } else {
                commit_cd(cd_a);                // Commit to parent(root): x=0 (child wins)
        }

        return 0;
}

void b(cd_handle parent) {
       int error, compute_error, function_error;
       struct cd_addrspec addrlist[1] = { {&x, sizeof(x), READ_WRITE, GLOBAL} };
        cd_b = create_cd(parent, NULL, MPI_LOGGING_INHERIT, NULL, &error);
        add_to_cd_via_copy(cd_b, addrlist, 1); // cd_b contains: x=1
   TOP:
        x += 1;                                    // x=2;
        compute_error = compute_stuff();

        if (compute_error) {
               restore_cd(cd_b);                // Restore app memory to: x=1;
               goto TOP;
        } else {
               commit_cd(cd_b);         // Commit to parent(cd_a): x=0 (parent wins)
        }

       function_error = do_some_other_stuff_that_may_error();
       return function_error;
}
```

The following timing diagram walks through the program above. The table below the timing diagram shows the different values x assumes under different circumstances--restores of different CDs and a commit-- at time T4.

Timing Diagram



X's Value under different circumstances at time T4

State	Normal	Restore CD_B	Restore CD_A	Commit CD_B
Application Memory	2	1	0 (Parent wins)	2
CD_B Contents	1	1	DISCARDED	DISCARDED
CD_A Contents	0	0	0	0 (Parent wins)

STATIC 2: Child CD stores new, unchanged data which is used during a commit/restore.

During a commit_cd or restore_cd, data from a child CD that is not present in the parent CD is used. During a restore_cd, this data is restored in application memory. During a commit_cd, the data is added to the parent CD. This accumulates additional data in the parent CD. This is data that was present at the time the parent CD was created, but has since been overwritten in application memory.

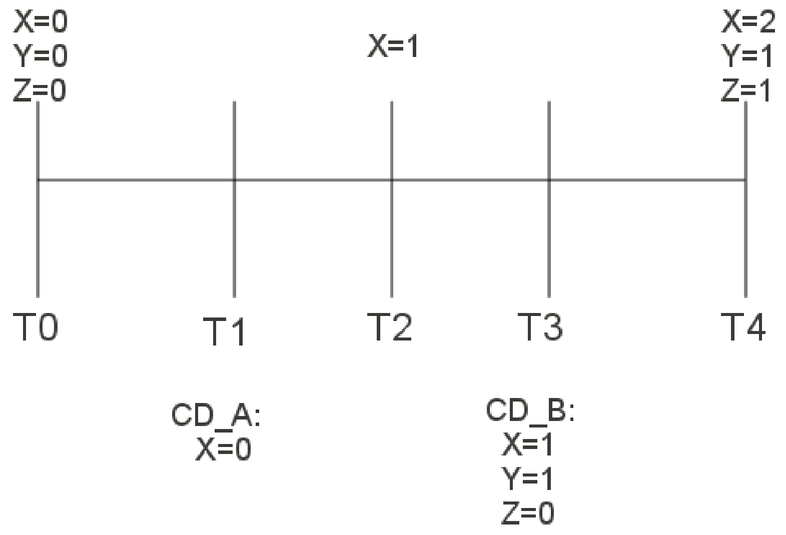
This use case demonstrates what happens to data (variables y and z) that is present in a child CD but not the parent. It also demonstrates that data can be conditionally added to a CD (variable z). The word **Conditionally** in the following use case only applies to the variable z.

Again, as in Use Case 1 the root CD in main was left intentionally vacant of any data. Normally, it is nonsensical to create a CD and not populate it with any data or use it in any way (like restore_cd or advance_cd_point_in_time).

```
int x=0, y=0, z=0;

main (int argc, char* argv) {
        char *storage_info = NULL;
        int error;

        root = create_cd(NULL, storage_info, COMM_LOGGING_ENABLED, "root", &error);
        a(root);
                                 // Upon return from function a(), the CD 'root' contains x=0, y=0, z=0 (Conditionally)
        commit(root);            // This commit will discard the CD 'root'.
        exit(0);
}

void a(cd_handle parent) {
       int error, compute_error;
       struct cd_addrspec addrlist[1] = { {&x, sizeof(x), READ_WRITE, GLOBAL} };
        cd_a = create_cd(parent, NULL, MPI_LOGGING_INHERIT, NULL, &error);

        add_to_cd_via_copy(cd_a, addrlist, 1); // cd_a contains: x=0
   TOP:
        x += 1;                                    // x=1;
       compute_error = compute_stuff();

        if (compute_error) {
                restore_cd(cd_a);               // Restore app memory to:
                                               // x=0,
                                               // y=0 (y is not restored, but this is its value in application memory)
                goto TOP;
       }

        //Call another function.
        function_error = b(cd_a);

        if (function_error) {
                restore_cd(cd_a);               // Restore app memory to: x=0, y=0
               goto TOP;
        } else {
                commit(cd_a);                   // Commit to parent(root): x=0 (child wins), y=0 (child wins)
        }

        return 0;
}

void b(cd_handle parent) {
       int error, compute_error, function_error;
       struct cd_addrspec addrlist[2] = {{&x, sizeof(x), READ_WRITE, GLOBAL},
                                         {&y, sizeof(y), READ_WRITE, GLOBAL}};
        cd_b = create_cd(parent, NULL, MPI_LOGGING_INHERIT, NULL, &error);
        add_to_cd_via_copy(cd_b, addrlist, 2); // cd_b contains: x=1 and y=0
       if (cray_is_profitable()) {
           // Note: this is a single variable and not an array.
           struct cd_addrspec addrlist1 = {&z, sizeof(z), READ_WRITE, GLOBAL};

             add_to_cd_via_copy(cd_b, &addrlist1 , 1); // cd_b contains: x=1, y=0, z=0
       }
   TOP:
       x += 1;                             // x=2;
       y += 1;                             // y=1;
       if (cray_is_profitable()) {
            z += 1;                              // z=1;
       }

       compute_error = compute_stuff();
        if (compute_error) {
                restore_cd(cd_b);               // Restore app memory to: x=1, y=0, and z=0 (Conditionally) ;
                goto TOP;
        } else {
                commit(cd_b);                   // Commit to parent(cd_a):
                                               // x=0 (parent wins)
                                               // y=0 (child wins)
                                               // z=0 (child wins.)
                                               // This last line is conditional upon whether z was ever added to cd_b.
        }

       function_error = do_some_other_stuff_that_may_error();
       return function_error;
}
```

The following timing diagram walks through the program above. The table below the timing diagram shows the different values x and y assume under different circumstances--restores of different CDs and a commit-- at time T4.

Timing Diagram



States under different circumstances at time T4

State	Normal	Restore CD_B	Restore CD_A	Commit CD_B
Application Memory	X=2, Y=1, Z=1	X=1, Y=0, Z=0	X=0 (Parent wins), Y=0 (Child wins), Z=0 (Child wins)	X=2, Y=1, Z=1
CD_B Contents	X=1, Y=0, Z=0	X=1,Y=0, Z=0	DISCARDED	DISCARDED
CD_A Contents	X=0	X=0	X=0	X=0 (Parent wins), Y=0 (Child wins), Z=0 (Child wins)

Note:Z is only present in Containment Domain CD_B if the 'if (cray_is_profitable)' branch in the code was taken in function b().

Delete_from_cd

This use case demonstrates using the delete_from_cd function.

Automatic variables are deallocated as soon as their stack frame is popped. While they can be added to and used in a CD for recovery purposes while they are still allocated, they should be deleted from a CD prior to being deallocated. If a function adds any automatic variables to a CD, it is responsible for deleting those variables prior to returning.

The function calculator() should never commit the CD because it is still in use by main (or more correctly by its caller), it should only use it for add_to, delete_from and restore.

Main never needs to delete the automatic variable 'x' from the CD for two reasons. First, 'x' is never deallocated until the program exits. Second, the CD will be discarded anyway when main commits it.

```
main (int argc, char* argv) {
   char *storage_info = NULL;
   int cd_error, comp_error;
   int x = 0, answer;
   root = create_cd(NULL, storage_info, COMM_LOGGING_DISABLED, "root", &cd_error);
   add_to_cd_via_copy(root, {&x, sizeof(x), READ_WRITE, GLOBAL}, 1); // root contains: x=0
TOP:
   answer = calculator(root);
   x += 1;
   comp_error = compute_stuff(x, answer);
   if (comp_error) {
      restore_cd(root);     // Restores x = 0;
      goto TOP;
   }
   commit_cd(root);            // This commit will discard the CD 'root'.
   exit(0);
}

int calculator(cd_handle cd) {                  // Main passed in the cd handle 'root'
   int comp_error;
   int y = 0;                           // Automatic variable
   add_to_cd_via_copy(cd, {&y, sizeof(y), READ_WRITE, GLOBAL}, 1); // cd contains: x=0 and y=0
TOP:
   y += 1;                               // y=1;
   comp_error = compute_stuff(y);
   if (comp_error) {
      restore_cd(cd);                   // Restore app memory to: x=0 and y=0
      goto TOP;
   } else {
      delete_from_cd(cd, {&y, sizeof(y), READ_WRITE, GLOBAL}, 1); // cd contains: x=0
   }

   return y;
}
```

Sibling CDs

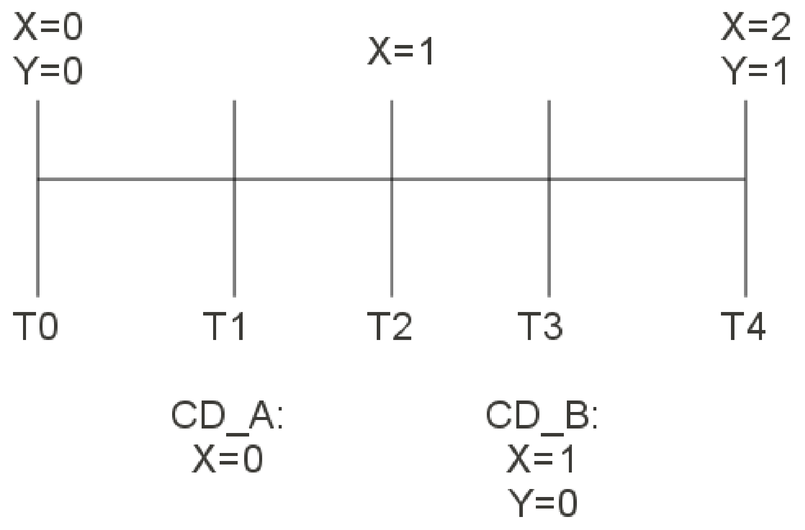
Sibling CDs are CDs with the same parent, who exist concurrently. Each sibling can recover on its own in isolation if it encounters an error unique to itself. The controlling parent program can execute a top-level recovery when it encounters a control error such as a child process terminating abnormally.

```
int main(int argc, char *argv[]) {
        int cd_error, cal_error = 0, status;
        int num_rows = 100, i=0, cpid;
        int data[num_rows];

       /* Some initial internal state */
        int x = 0, y = 0, z = 0;
       struct cd_addrspec addrlist[3] = {{&x, sizeof(x), READ_WRITE, GLOBAL},
                                          {&y, sizeof(y), READ_WRITE, GLOBAL},
                                          {&z, sizeof(z), READ_ONLY, GLOBAL}};

        cd_handle root =  create_cd( NULL, NULL, COMM_LOGGING_ENABLED, “root”, &cd_error);
        add_to_cd_via_copy(root, addrlist, 3);
TOP:
        data[num_rows] = initialize_data();

        for(i = 0; i < num_ranks; i++) {
                cpid = fork();   // using fork for simplicity, would typically be an OMP thread
                if (!cpid) {
                        /* Child */
                        calculator(root, &data[i]);
                        exit(0);
                }
        }
        /* Parent waits for all of its children. */
        while (wait(&status) != -1) {
                /* If child terminated abnormally, clean up and recover. */
                if (!WIFEXITED(status)) {
                       kill_all_children();
                        restore_cd(root);
                        goto TOP;
                }
        }

       /* Discard the root CD via commit_cd. */
       commit_cd(root);

        return 0;
}

int calculator(cd_handle parent, int *data_ptr) {
        int cd_error, calc_error;
        cd_handle child =  create_cd(parent, NULL, MPI_LOGGING_INHERIT, NULL, &cd_error);
        add_to_cd_via_copy(child, {data_ptr, sizeof(int), READ_WRITE, GLOBAL}, 1);
        /* Recover when encountering an error. */
        while (calc_error = compute(data_ptr)) {
                restore_cd(child);
        }

       /* This commit updates the parent CD with initial data, prior to changing that data.*/
       /* This initial data can later be used for recovery, if needed. */
       commit_cd(child);

        return 0;
}
```

Committing CDs

Let's examine committing CDs.

Timing Diagram



States under different circumstances at time T4

State	Normal	Commit CD_B	Commit CD_A	Commit Root
Application Memory	X=2, Y=1	X=2, Y=1	X=2, Y=1	X=2, Y=1
CD_B Contents	X=1, Y=0	DISCARDED	DISCARDED	DISCARDED
CD_A Contents	X=0	X=0 (Parent wins), Y=0 (Child wins)	DISCARDED	DISCARDED
Root CD	EMPTY	EMPTY	X=0 (Child wins), Y=0 (Child wins)	DISCARDED

Recovery from a node failure

```
TBD.
```

Acknowledgement

This research was funded in part by DARPA contract HR0011-10-9-0008.

Bibliography

Containment Domains: a Full System Approach to Computational Resiliency (http://engmgmtwiki.us.cray.com/pmwiki/uploads/CATteam2/containment.pdf) ; Michael Sullivan, Doe Hyun Yoon, Mattan Erez; UT Austin Tech Report TR-LPH-2010-001, December 2010

Assessing an Implementation of Containment Domains for Random GPU Failures (http://inside.us.cray.com/depts/CTO/CTB/Resiliency/Shared%20Documents/CD.lbm.pdf) ; Cray GPU Resiliency Team, Cray Inc., August, 2011; Unpublished

Appendix A: Fortran API

```
module Containment_Domains
  use, intrinsic :: iso_c_binding

  type, bind(C) :: cd_addrspec
      type(c_ptr)       :: address
      integer(c_size_t) :: length
      integer(c_int)    :: addr_tp
  end type cd_addrspec

  ! for use with cd_addrspec field addr_tp.
  integer(c_int), parameter :: READ_ONLY  = 0
  integer(c_int), parameter :: READ_WRITE = 1

  ! for use with cd_addrspec field addr_scope.
  integer(c_int), parameter :: GLOBAL  = 0
  integer(c_int), parameter :: CONSTRAINED = 1

  interface

      function create_cd( parent_cd,        &
                          storage_info,     &
                          log_mpi_traffic,  &
                          name,             &
                          error             &
                         ) bind(C)
             import c_ptr, c_int32_t
             type(c_ptr)        :: create_cd    ! returns cd_handle
             type(c_ptr), value :: parent_cd    ! cd_handle
             type(c_ptr), value :: storage_info ! char *
             type(c_ptr), value :: name         ! char *
             integer(c_int32_t) :: error        ! int * (out)
        end function create_cd

        function commit_cd( cd ) bind (C)
          import c_ptr
          type(c_ptr)        :: commit_cd ! returns cd_handle
          type(c_ptr), value :: cd        ! cd_handle
        end function commit_cd

        function restore_cd( cd ) bind (C)
          import c_ptr, c_int32_t
          integer(c_int32_t) :: restore_cd ! returns int
          type(c_ptr), value :: cd         ! cd_handle
        end function restore_cd

        function advance_cd_point_in_time( cd ) bind (C)
          import c_ptr, c_int32_t
          integer(c_int32_t) :: advance_cd_point_in_time ! returns int
          type(c_ptr), value :: cd         ! cd_handle
        end function advance_cd_point_in_time

        function add_to_cd_via_copy( cd, addrlist, ascount ) bind (C)
          import c_ptr, c_int32_t, cd_addrspec
          integer(c_int32_t)        :: add_to_cd_via_copy ! returns int
          type(c_ptr), value        :: cd            ! cd_handle
          type(cd_addrspec)         :: addrlist      ! by reference
          integer(c_int32_t), value :: ascount
        end function add_to_cd_via_copy

        function add_to_cd_via_regen( cd, addrlist, ascount, regen ) bind (C)
          import c_ptr, c_int32_t, cd_addrspec, c_funptr
          integer(c_int32_t)        :: add_to_cd_via_regen ! returns int
          type(c_ptr), value        :: cd        ! cd_handle
          type(cd_addrspec)         :: addrlist  ! by reference
          integer(c_int32_t), value :: ascount
          type(c_funptr), value     :: regen     ! int (*)(cd_addr_spec*, int)
        end function add_to_cd_via_regen

        function add_to_cd_via_parent( cd, addrlist, ascount ) bind (C)
          import c_ptr, c_int32_t, cd_addrspec
          integer(c_int32_t)        :: add_to_cd_via_parent ! returns int
          type(c_ptr), value        :: cd       ! cd_handle
          type(cd_addrspec)         :: addrlist ! by reference
          integer(c_int32_t), value :: ascount
        end function add_to_cd_via_parent

        function add_file_to_cd( cd, filedes ) bind (C)
          import c_ptr, c_int32_t
          integer(c_int32_t)        :: add_file_to_cd ! returns int
          type(c_ptr), value        :: cd             ! cd_handle
```