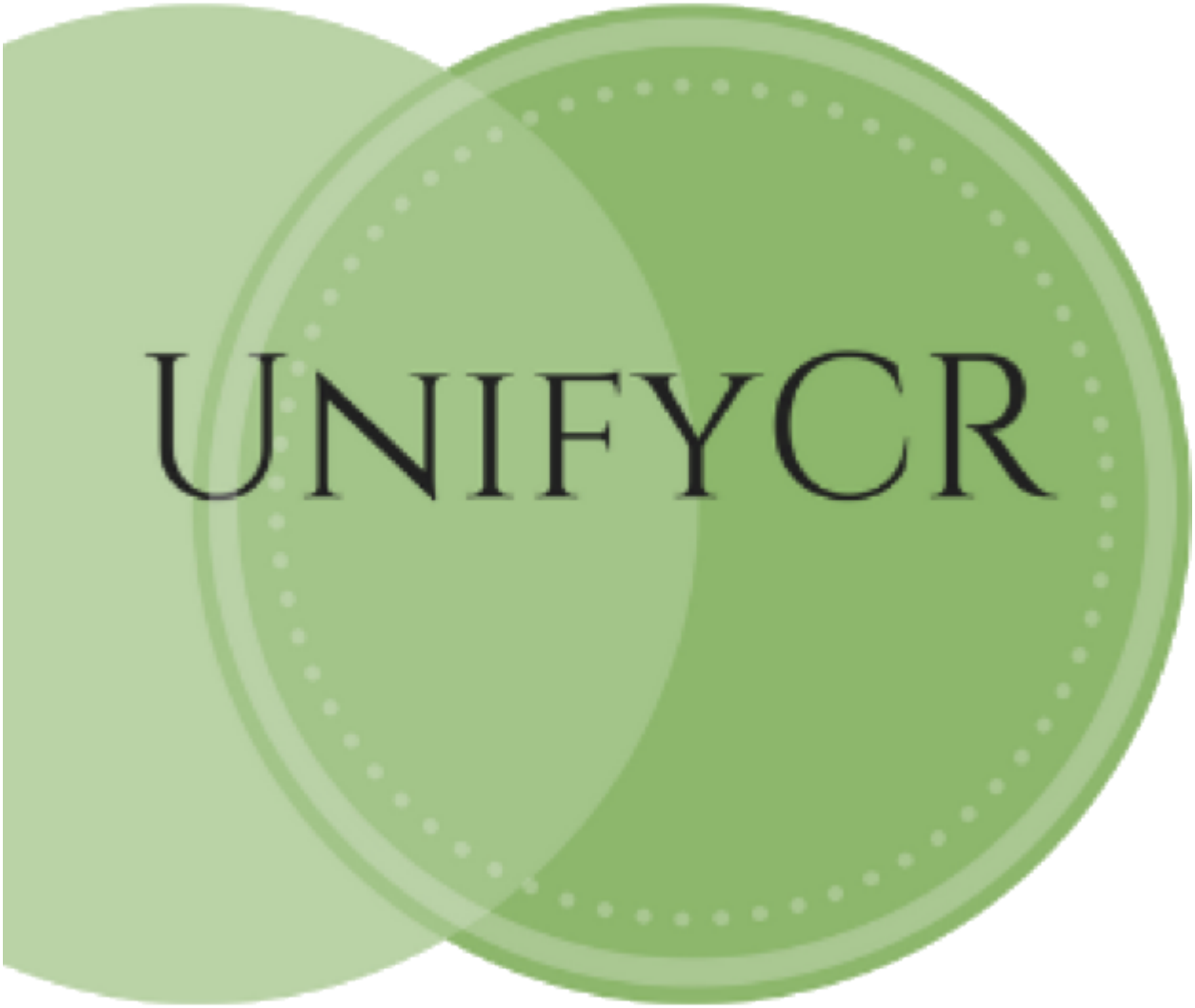


UnifyCR Documentation

Release 0.1

Kathryn Mohror, Adam Moody, Oral Sarp, Feiyi Wang, Hyogi Sim,

Jun 27, 2018

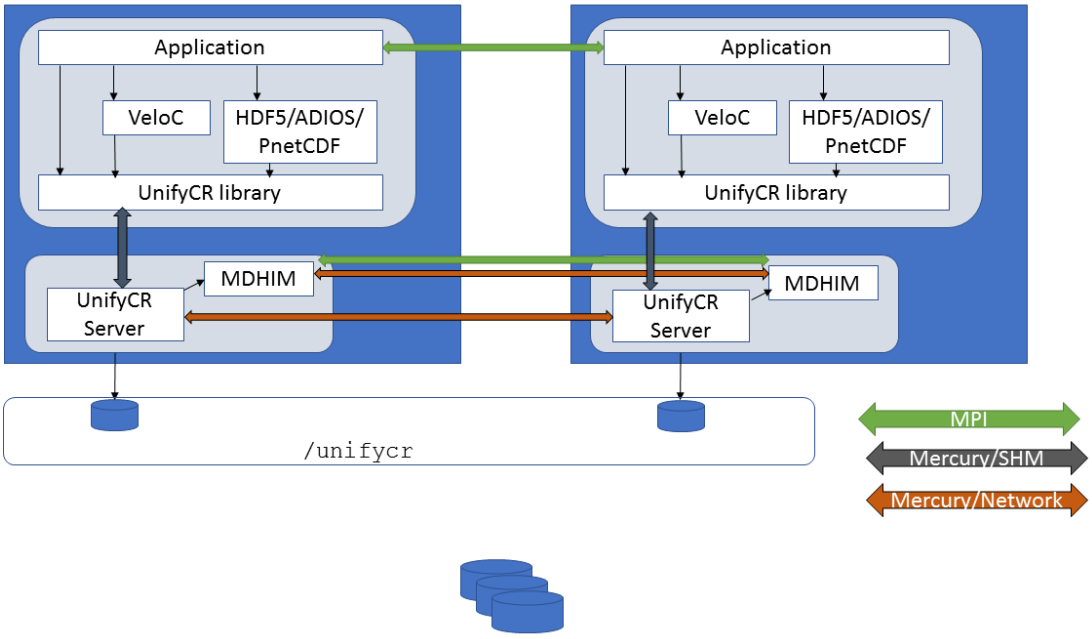
User Guide

CHAPTER 1

Overview

UnifyCR is a user level file system currently under active development. An application can use node-local storage as burst buffers for shared files. UnifyCR is designed to support both checkpoint/restart which is the most important I/O workload for HPC and other common I/O workloads as well. With UnifyCR, applications can write to fast, scalable, node-local burst buffers as easily as they do the parallel file system. This section will provide a high level design of UnifyCR. It will describe the UnifyCR library and the UnifyCR daemon.

1.1 High Level Design



UnifyCR will present a shared namespace (e.g., /unifycr as a mount point) to all compute nodes in a users job allocation. There are two main components of UnifyCR: the UnifyCR library and the UnifyCR daemon. The UnifyCR library (also referred to as the UnifyCR client library) is linked into the user application and is responsible for intercepting I/O calls from the user application and then sending the I/O requests on to a UnifyCR server to be handled. The UnifyCR client library uses the ECP GOTCHA software as its primary mechanism for intercepting I/O calls. Each UnifyCR daemon (also referred to as a UnifyCR server daemon) runs as a daemon on a compute node in the users allocation. The UnifyCR server is responsible for handling the I/O requests from the UnifyCR library. On each compute node, there will be user application processes running as well as tool daemon processes. The user application is linked with the UnifyCR client library and a high-level I/O library, e.g. HDF5, ADIOS, or PnetCDF. The UnifyCR server daemon also runs on the compute node and is linked with the MDHIM library which is used for metadata services.

CHAPTER 2

Definitions

In this section, we provide some useful definitions for terms used in this document.

2.1 Job

A set of commands that is issued to the resource manager and is allocated a set of nodes for some duration

2.2 Run or Job Step

A single application launch of a group of one or more application processes issued within a job

CHAPTER 3

Assumptions

In this section, we provide assumptions we make about the behavior of applications that use UnifyCR, and about how UnifyCR currently functions.

3.1 Application Behavior

- Workload supported is globally synchronous checkpointing.
- I/O occurs in write and read phases. Files are not read and written at the same time. There is some (good) amount of time between the two phases. For example, files are written during checkpoint phases and only read during recovery or restart.
- Processes on any node can read any byte in the file (not just local data), but the common case will be processes read only their local bytes.
- Assume general parallel I/O concurrency semantics where processes can write to the same offset concurrently. We assume the outcome of concurrent writes to the same offset or other conflicting concurrent accesses is undefined. For example, if a command in the job renames a file while the parallel application is writing to it, the outcome is undefined. It could be a failure or not, depending on timing.

3.2 Consistency Model

In the first version of UnifyCR, lamination will be explicitly initiated by a UnifyCR API call. In subsequent versions, we will support implicit initiation of file lamination. Here, UnifyCR will determine a file to be laminated based on conditions, e.g., texttt{fsync} or texttt{ioctl} calls, or a time out on texttt{close} operations. As part of the UnifyCR project, we will investigate these implicit lamination conditions to determine the best way to enable lamination of files without explicit UnifyCR API calls being made by the application.

In the first version of UnifyCR, eventually, a process declares the file to be laminated through a UnifyCR API call. After a file has been laminated, the contents of the file cannot be changed. The file becomes permanently read-only. After lamination, any process may freely read any part of the file. If the application process group fails before a file has been laminated, UnifyCR may delete the file. An application can delete a laminated file.

We define the laminated consistency model to enable certain optimizations while supporting the perceived requirements of application checkpoints. Since remote processes are not permitted to read arbitrary bytes within the file until lamination, global exchange of file data and/or data index information can be buffered locally on each node until the point of lamination. Since file contents cannot change after lamination, aggressive caching may be used during the read-only phase with minimal locking. Since a file may be lost on application failure unless laminated, data redundancy schemes can be delayed until lamination.

Behavior before lamination:

- open/close: A process may open/close a file multiple times.
- write: A process may write to any part of a file. If two processes write to the same location, the value is undefined.
- read: A process may read bytes it has written. Reading other bytes is invalid.
- rename: A process may rename a file.
- truncate: A process may truncate a file.
- unlink: A process may delete a file.

Behavior after lamination:

- open/close: A process may open/close a file multiple times.
- write: All writes are invalid.
- read: A process may read any byte in the file.
- rename: A process may rename a file.
- truncate: Truncation is invalid (considered to be a write operation).
- unlink: A process may delete a file.

3.3 File System Behavior

- The file system exists on node local storage only and is not persisted to stable storage like a parallel file system (PFS). Can be coupled with
- SymphonyFS or high level I/O or checkpoint library (VeloC) to move data to PFS periodically, or data can be moved manually
- Can be used with checkpointing libraries (VeloC) or I/O libraries to support shared files on burst buffers
- File system starts empty at job start. User job must populate the file system.
- Shared file system namespace across all compute nodes in a job, even if an application process is not running on all compute nodes
- Survives application termination and/or relaunch within a job
- Will transparently intercept system level I/O calls of applications and I/O libraries

3.4 System Characteristics

- There is some storage available for storing file data on a compute node, e.g. SSD or RAM disk
- We can run user-level daemon processes on compute nodes concurrently with a user application

CHAPTER 4

Build & I/O Interception

In this section, we describe how to build UnifyCR with I/O interception.

Note: The current version of UnifyCR adopts the mdhim key-value store, which strictly requires:

"An MPI distribution that supports MPI_THREAD_MULTIPLE and per-object locking of critical sections (this excludes OpenMPI up to version 3.0.1, the current version as of this writing)"

as specified in the project github

4.1 How to build UnifyCR

Download the latest UnifyCR release from the Releases page. UnifyCR requires MPI, LevelDB, and GOTCHA.

4.1.1 Building with Spack

To install leveldb and gotcha and set up your build environment, we recommend using the Spack package manager.

The instructions assume that you do not already have a module system installed such as LMod, Dotkit, or Environment Modules. If your system already has Dotkit or LMod installed then installing the environment-modules package with spack is unnecessary (so you can safely skip that step).

If you use Dotkit then replace `spack load` with `spack use`.

```
$ git clone https://github.com/spack/spack
$ ./spack/bin/spack install leveldb
$ ./spack/bin/spack install gotcha
$ ./spack/bin/spack install environment-modules
$ . spack/share/spack/setup-env.sh
$ spack load leveldb
$ spack load gotcha
```

Then to build UnifyCR:

```
$ ./autogen.sh
$ ./configure --prefix=/path/to/install --enable-debug
$ make
$ make install
```

4.1.2 Building without Spack

For users who cannot use Spack, you may fetch the latest release of GOTCHA

And leveldb (if not already installed on your system): leveldb

If you installed leveldb from source then you may have to add the pkgconfig file for leveldb manually. This is assuming your install of leveldb does not contain a .pc file (it usually doesn't). Then, add the path to that file to PKG_CONFIG_PATH.

```
$ cat leveldb.pc
#leveldb.pc
prefix=/path/to/leveldb/install
exec_prefix=/path/to/leveldb/install
libdir=/path/to/leveldb/install/lib64
includedir=/path/to/leveldb/install/include
Name: leveldb
Description: a fast key-value storage library
Version: 1.20
Cflags: -I${includedir}
Libs: -L${libdir} -lleveldb

$ export PKG_CONFIG_PATH=/path/to/leveldb/pkgconfig
```

Leave out the path to leveldb in your configure line if you didn't install it from source.

```
$ ./configure --prefix=/path/to/install --with-gotcha=/path/to/gotcha --enable-debug -
↪-with-leveldb=/path/to/leveldb
$ make
$ make install
```

Note: You may need to add the following to your configure line if it is not in your default path on a linux machine:

`--with-numa=$PATH_TO_NUMA`

This is needed to enable NUMA-aware memory allocation on Linux machines. Set the NUMA policy at runtime with `UNIFYCR_NUMA_POLICY = local | interleaved`, or set NUMA nodes explicitly with `UNIFYCR_USE_NUMA_BANK = <node no.>`

4.2 I/O Interception

POSIX calls can be intercepted via the methods described below.

4.2.1 Statically

Steps for static linking using –wrap:

To intercept I/O calls using a static link, you must add flags to your link line. UnifyCR installs a unifycr-config script that returns those flags, e.g.,

```
$ mpicc -o test_write \
      `<unifycr>/bin/unifycr-config --pre-ld-flags` \
      test_write.c \
      `<unifycr>/bin/unifycr-config --post-ld-flags`
```

4.2.2 Dynamically

Steps for dynamic linking using gotcha:

To intercept I/O calls using gotcha, use the following syntax to link an application.

```
$ mpicc -o test_write test_write.c \
    -I<unifycr>/include -L<unifycy>/lib -lunifycr_gotcha \
    -L<gotcha>/lib64 -lgotcha
```

CHAPTER 5

Mounting UnifyCR

In this section, we describe how to use the UnifyCR API in an application.

5.1 Mounting

To use the UnifyCR filesystem a user will have to provide a path prefix. All file operations under the path prefix will be intercepted by the UnifyCR filesystem. For instance, to use UnifyCR on all path prefixes that begin with /tmp this would require a:

```
unifycr_mount('/tmp', rank, rank_num, 0);
```

Where /tmp is the path prefix you want UnifyCR to intercept. The rank and rank number is the rank you are currently on, and the number of tasks you have running in your job. Lastly, the zero corresponds to the app id.

5.2 Unmounting

When you are finished using UnifyCR in your application, you should unmount.

```
if (rank == 0) {
    unifycr_unmount();
}
```

It is only necessary to call unmount once on rank zero.

CHAPTER 6

UnifyCR Configuration

Here, we explain how users can customize the runtime behavior of UnifyCR. In particular, UnifyCR provides the following ways to configure:

- System-wide configuration file: `/etc/unifycr/unifycr.conf`
- Environment variables
- Command line arguments

For the duplicated entries, the command line arguments have the highest priority, overriding any configuration options from `unifycr.conf` and environment variables. Similarily, environment variables have higher priority than options in the `unifycr.conf` file. `unifycr` command line utility creates the final configuration file (`unifycr-runstate.conf`) based on all forementioned configuration options.

6.1 unifycr.conf

unifycr.conf specifies the system-wide configuration options. The file is written in TOML language format. The unifycr.conf file has four different sections, i.e., global, filesystem, server, and client sections.

- **[global] section**
  - runstatedir: a directory where the final configuration file (`unifycr-runstate.conf`) has to be created
  - unifycrd_path: path to unifycrd server daemon process
- **[filesystem] section**
  - mountpoint: unifycr file system mountpoint
  - **consistency: consistency model to be used, one of:**
    * `none`:
    * `laminated`:
    * `posix`:

- **[server] section**
  - meta_server_ratio: the ratio between the number of unifycrd daemon and the number of metadata key-value storage instance
  - meta_db_name: name of the database file to store unifycr file system metadata
  - meta_db_path: the pathname of the metadata database file will be created
  - server_debug_log_path: the debug log file of the unifycrd daemon
- **[client] section**
  - chunk_mem: allocation chunk size for unifycr file system memory storage

6.2 Environment Variables

The following is the list of the environment variables that UnifyCR supports.

- `UNIFYCR_META_SERVER_RATIO`: the ratio between the number of unifycrd daemon and the number of metadata key-value storage instance
- `UNIFYCR_META_DB_NAME`: the name of the database file to store unifycr file system metadata
- `UNIFYCR_META_DB_PATH`: the pathname of the metadata database file will be created
- `UNIFYCR_SERVER_DEBUG_LOG`: the debug log file of the unifycrd daemon
- `UNIFYCR_CHUNK_MEM`: allocation chunk size for unifycr file system memory storage

6.3 Command Line arguments

Lastly, unifycr command line utility accepts arguments to configure the runtime options.

```
Usage: unifycr <command> [options...]

<command> should be one of the following:
  start       start the unifycr server daemon
  terminate   terminate the unifycr server daemon

Available options for "start":
  -C, --consistency=<model> consistency model (none, laminated, or posix)
  -m, --mount=<path>        mount unifycr at <path>
  -i, --transfer-in=<path>  stage in file(s) at <path>
  -o, --transfer-out=<path> transfer file(s) to <path> on termination

Available options for "terminate":
  -c, --cleanup             clean up the unifycr storage on termination
```

CHAPTER 7

Starting & Stopping

In this section, we describe the mechanisms for starting and stopping UnifyCR in a user's allocation. The important features to consider are:

- Initialization of UnifyCR file system instance across compute nodes

7.1 Initialization Example

First, we need to start the UnifyCR daemon (`unifycrd`) on the nodes in your allocation. UnifyCR provides the `unifycr` command line utility for this purpose. The specific paths and job launch command will depend on your installation and system **configuration_**.

```
user@ unifycr --help

Usage: unifycr <command> [options...]

<command> should be one of the following:
  start       start the unifycr server daemon
  terminate   terminate the unifycr server daemon

Available options for "start":
  -C, --consistency=<model> consistency model (none, laminated, or posix)
  -m, --mount=<path>        mount unifycr at <path>
  -i, --transfer-in=<path>  stage in file(s) at <path>
  -o, --transfer-out=<path> transfer file(s) to <path> on termination

Available options for "terminate":
  -c, --cleanup             clean up the unifycr storage on termination
```

For instance, the following script will launch the unifycrd daemon.

```
#!/bin/bash

```

(continues on next page)

(continued from previous page)

```
    export UNIFYCR_META_SERVER_RATIO=1
    export UNIFYCR_META_DB_NAME=unifycr_db
    export UNIFYCR_CHUNK_MEM=0
    export UNIFYCR_META_DB_PATH=/mnt/ssd
    export UNIFYCR_SERVER_DEBUG_LOG=/tmp/unifycrd_debug.$$

    unifycr start --mount=/mnt/unifycr
```

Note that the `unifycr` utility automatically detects the allocated nodes and launches the daemon on each of the allocated node. In addition, the above environment variables will override any configurations in `/etc/unifycr/unifycr.conf`. See **configurations_** for further details about the configuration.

Next, we can start run our application with UnifyCR in the following manner:

```
    #!/bin/bash

    export UNIFYCR_EXTERNAL_META_DIR=/mnt/ssd
    export UNIFYCR_EXTERNAL_DATA_DIR=/mnt/ssd

    NODES=1
    PROCS=1

    mpirun -nodes ${NODES} -np ${PROCS} /path/to/my/app
```

So, overall the steps taken to run an application with UnifyCR include:

1. Allocate Nodes
2. Update any desired configuration variables in the bash scripts
3. Start the UnifyCR server daemons on each node with the `unifycr` utility.
4. Run your application with UnifyCR

7.2 Stopping

Currently, the UnifyCR server daemon runs throughout a user's job allocation after it is started. Even if the UnifyCR daemon is running in a user's job the UnifyCR file system will only be utilized if the user has mounted a path for UnifyCR to intercept. The UnifyCR daemon is stopped when the user's allocation is exited.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search