Introduction

In this project, our objective is to create an application-transparent checkpoint-restore framework which can be integrated with user-level applications. This framework should reliably and efficiently allow a user-level application to be restored to a previously saved state after a crash unrelated to logical issues with the program itself and continue computation as if the crash had never happened. This framework should also be scalable to multithreaded applications as well as cache-intensive applications.

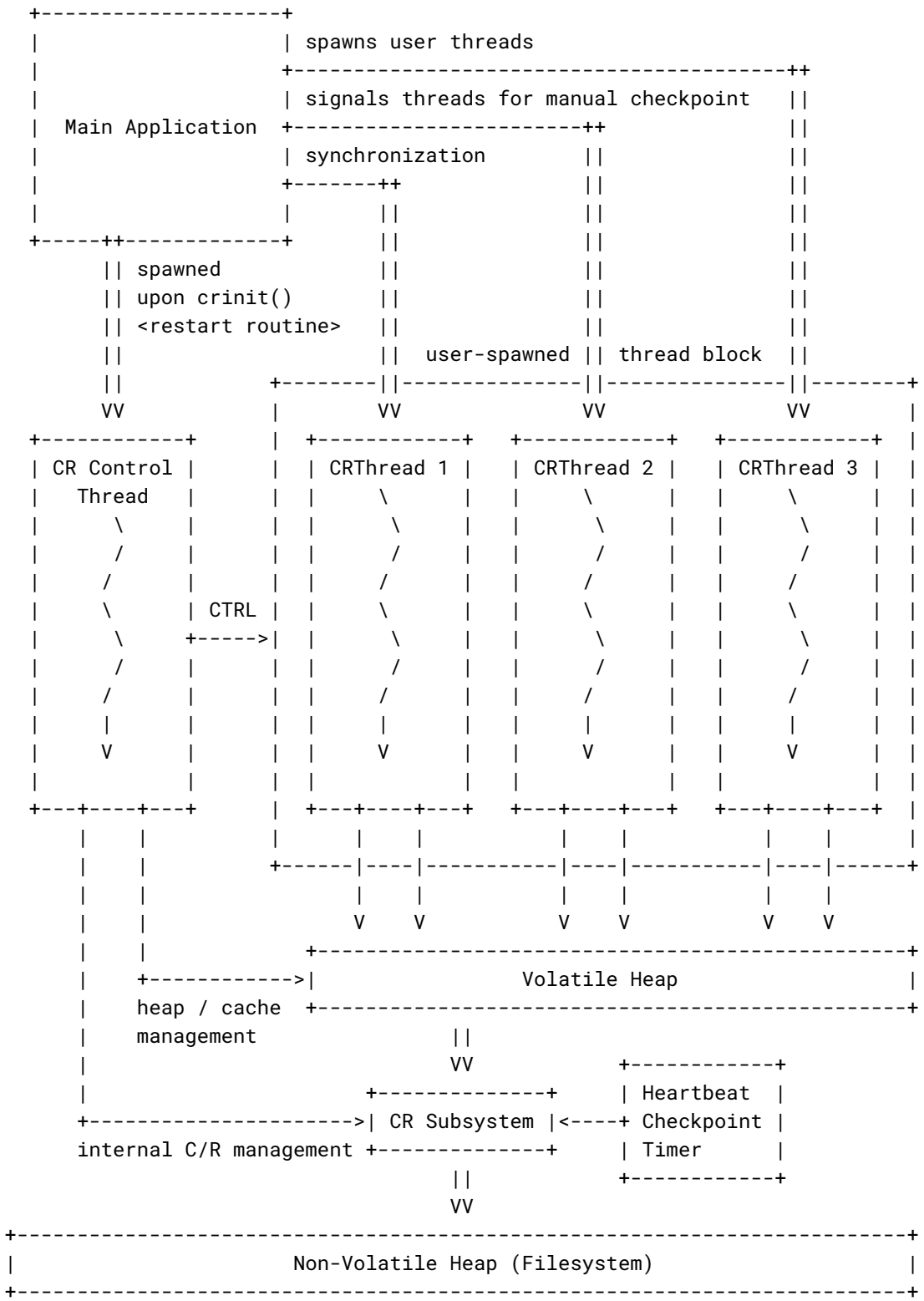
The purpose of this document is to outline the planned implementation of our framework. This includes determining when to checkpoint, what data should be saved at a checkpoint, and how checkpoints will be created. Additionally, we outline the procedure to restore a checkpointed state and a user-interface for our framework.

Architecture

Overall, our strategy for creating a checkpoint-restore framework requires, at minimum, four robust modules: a non-volatile memory management system, a threading and synchronization system, a checkpoint decisions subsystem, and a restart routine.

Because of the reliance on kernel-level solutions, we will not allow for checkpoint-restore on the main thread. Modern solutions which can checkpoint and restore the main thread each heavily rely on kernel-level solutions. Our solution only allows for the checkpointing and restoration of all user-spawned threads.

Below is an overall system architecture design for how this solution will checkpoint and restore threads.


Main Application
spawns user threads
signals threads for manual checkpoint
synchronization
spawned
upon crinit()
<restart routine>
user-spawned
thread block
CR Control Thread
CTRL
CRThread 1
CRThread 2
CRThread 3
heap / cache management
Volatile Heap
internal C/R management
CR Subsystem
Heartbeat Checkpoint Timer
Non-Volatile Heap (Filesystem)

In essence, what we will be doing is extending the regular user application so that it spawns threads which each can be checkpointed. Initially these threads, spawned in our own user thread block module, run in volatile memory—their stacks will actually be allocated from a heap of our design. We specifically need to reimplement our heap in order to provide for a low level method to decide what memory needs to be cached and when.

The functions `setjmp(jmp_buf env)` and `longjmp(jmp_buf env, int val)` allow us to set jump locations and then jump to them; we could put the jump buffer into some form of non-volatile storage upon checkpoint, `mmap()` the addresses so that they stay consistent between runs, and then `longjmp()` upon restoration.

When restoring the program, we `mmap()` the `jmp_buf[]` and then jump to the currently executing point in our program. We need to make sure that the `mmap()` addresses are coherent with the checkpoint as part of our implementation.

The process specified above will occur for each thread which was checkpointed successfully. The threads will be loaded back into volatile memory, meaning the volatile heap will be restored to its state prior to the crash. Then, each thread's stack pointer will be readjusted to the last selected checkpoint as set by `setjmp()`, and the program should continue execution as normal.

Interface

As a simple C library, we envision this program to have the following interface, included in a single header file - checkpoint_restore.h:

```
/* initializes the entire checkpoint-restore framework */
int crinit(struct crattr *attr);

/* the meat of the checkpoint-restore framework, self explanatory */
int checkpoint(void);
int restore(void);

/* non-volatile memory allocation and management */
void *crmalloc(size_t size);
```

```
void *crrealloc(void *ptr, size_t size);
void crfree(void *ptr);

/* non-volatile thread library, based off of pthreads */
int crthread_create(crthread_t *thread,
                    void *(*start_routine), (void *), void *arg);
int crthread_join(crthread_t thread, void **retval);
int crthread_cancel(crthread_t thread);
int crthread_kill(crthread_t thread, int sig);

/* non-volatile semaphores, based off of POSIX semaphores */
int crsem_init(crsem_t *sem, unsigned int value);
int crsem_post(crsem_t *sem);
int crsem_wait(crsem_t *sem);
```

Design Considerations

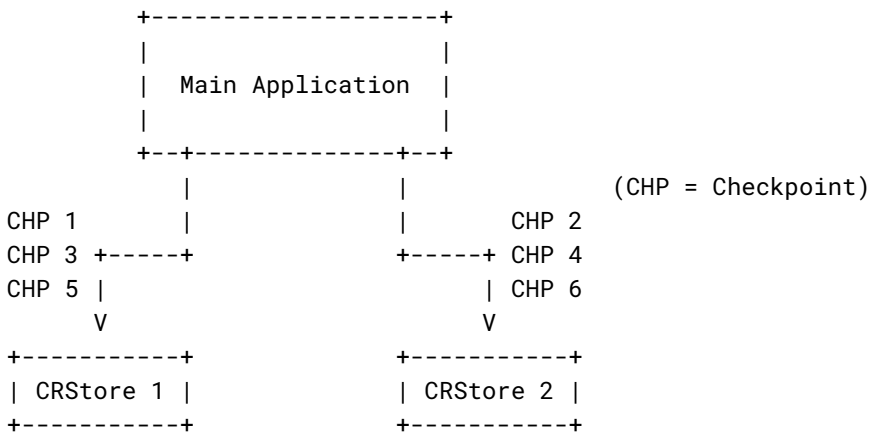
Most implementations of checkpoint-restore for user-programs may be categorized as either application-level or system-level. While application-level implementations have the potential for greater efficiency, it is the responsibility of the user to integrate checkpoint-restore functionality into the source code of each application they intend to checkpoint. System-level implementations have the advantage of application-transparency meaning that no modifications to an application's source code is necessary [1]. This allows the checkpoint-restore process to be abstracted from the user in such a way that the user only needs to know a handful of terminal commands to run an application with checkpointing and restore from a checkpoint.

We must also consider the possibility of a crash happening while in the middle of creating a checkpoint as our program would be in the middle of writing to a file. Our current solution is to use two output files. An output file would have a memory layout with a starting `METADATA` section and a separate `PAYLOAD` section as illustrated below.

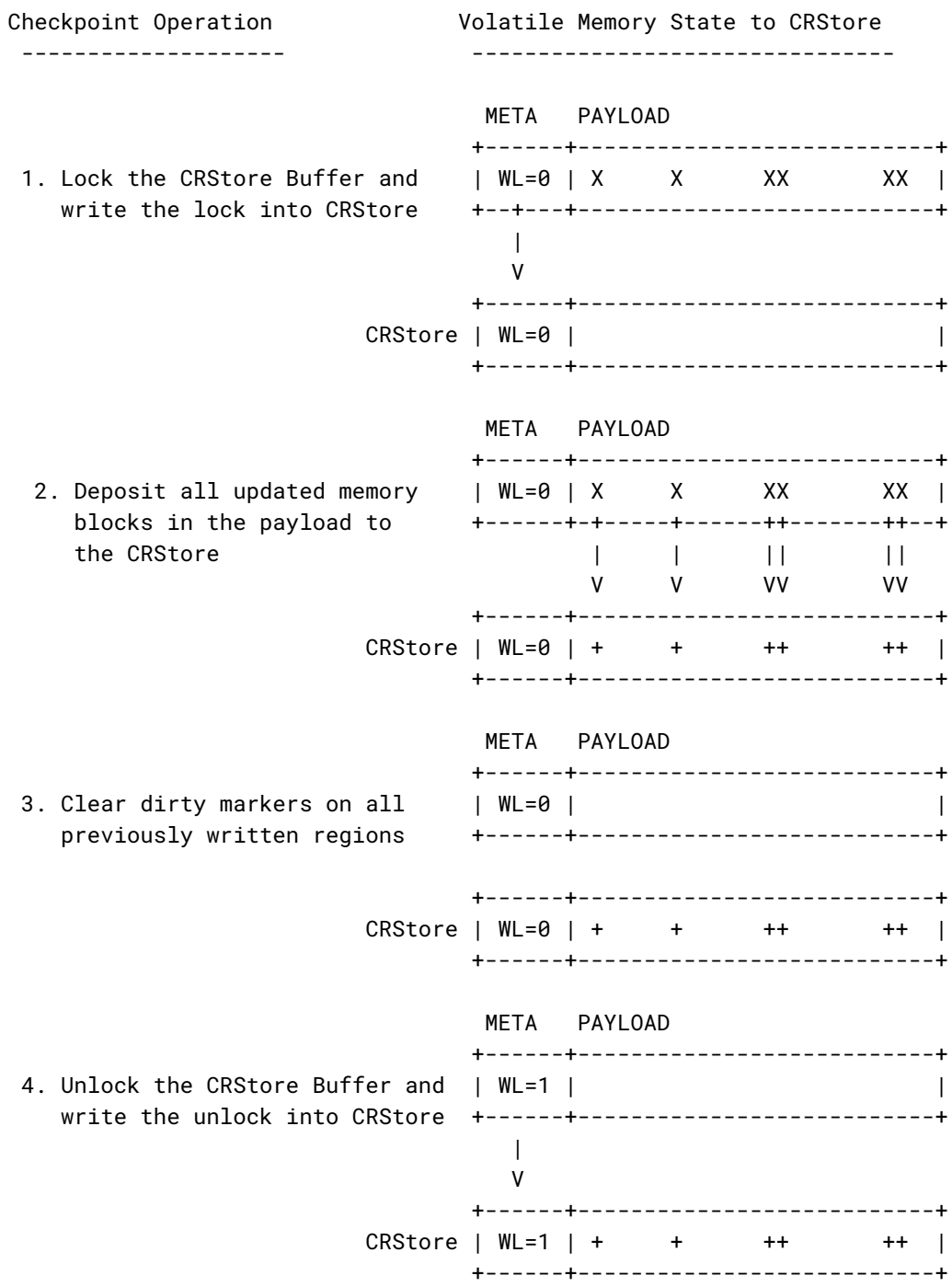
```
+----------+---------+
| METADATA | PAYLOAD |
+----+-----+---------+
     |
     +---> this metadata contains a flag called WRITE_LOCK at some
           predefined offset
```

When we checkpoint, `WRITE_LOCK` is used like a semaphore or lock in that we first set `WRITE_LOCK` to 0, then write all of our data, then set `WRITE_LOCK` back to 1. We know that we had a mid-write crash if `WRITE_LOCK` is still zero when we try to restore that data segment.

To mitigate this issue, we should have a double-buffered output. For now, let's say that we're just considering a single-threaded application. We would have a layout that looks like this:



Checkpoints can be made to opposite buffers at each write. Only one buffer is ever written to at a checkpoint. While this approach can certainly have issues with many slower writes, it does solve the crash-on-checkpoint problem, in which we would otherwise have no way to recover from a crash while our framework was performing a background checkpoint. Observe the diagram and explanation below for a brief overview of the checkpointing process:



The theory behind why we're doing this is to be able to detect if a crash which occurs mid-checkpoint has corrupted our data. We define a data corruption to have occurred when a crash happens anywhere within steps 2 and 3, inclusive.

When restoring the program state, we can observe the states of each of the non-volatile stores. If both of the stores have a `WRITE_LOCK` of 1, we choose the store with the most recent timestamp to restore the program. If one of the stores has a `WRITE_LOCK` of 0, however, we can safely assume that a crash-on-write has occurred for that store which means that the only safe restoration point is the store with a `WRITE_LOCK` of 1.

Testing and Evaluation

We will test our framework by running it with sample programs and simulating a crash mid-execution. It is yet to be decided how we may guarantee a crash at specific stages of execution which would be necessary to test the crash-on-checkpoint case. The success of a restore will be determined by confirming the correctness of the result of a program. In addition to checking for correctness, we will also time both the checkpointing process and the restore process.

Scalability Testing

Our framework should be scalable so that it is compatible with applications that use cache-intensive algorithms. To confirm scalability in this sense, we will stress-test our framework using well-known algorithms such as TSP, K-Means, and gradient descent with large problem sizes. We will also consider our framework scalable if it can successfully be run with highly parallelized programs and will test this by using our checkpoint-restore solution on many-threaded programs.

Milestones

Milestone 1: Implementing and Testing Non-Volatile Memory Management
In this first milestone, we aim to create a non-volatile memory module which can dynamically allocate and free memory to and from a file. This is more or less implementing `crmalloc()`, `crfree()`, and `crrealloc()`. Testing for this milestone will entail a stress test which allocates and frees a variety of different payloads - some tests will entail allocation of large memory blocks and others will entail a large number of smaller allocations.

Milestone 2: Implementing Non-Volatile Thread and Semaphore Libraries
In this second milestone, we aim to create a thread and synchronization module which is built using the memory allocator which we created in Milestone 1. The threads and semaphores should function just as the pthread library does.

Milestone 3: Develop Checkpointing Process and Subsystem
In this third milestone, we plan to implement the checkpointing process and subsystem. This will likely overlap our restoration process development since to test our checkpointing, we will need to verify that our checkpoints can actually be used to restore a program state.

Milestone 4: Baseline Correctness Testing
In this fourth milestone, we will test our checkpoint-restart solution on simple, single-thread applications. These tests will verify that upon restore from a checkpoint, a given program returns the expected results.

Milestone 5: Implement Multi-Thread Scalability
In this milestone, we will expand our base checkpoint-restore solution to work for any number of threads.

Milestone 6: Scalability Testing
In this milestone, we will test the scalability of our solution by running it with applications using various numbers of threads. Additionally, we will stress-test using cache-intensive algorithms such as TSP, K-means, and gradient descent.

Milestone 7: Performance Measurements
In this milestone, we take measurements of the time efficiency and throughput of our solution. We may fine-tune our solution further to improve performance.

Milestone 8: Prepare Demo
In this milestone, we will prepare a demonstration of our checkpoint-restore solution which will include examples showing its scalability, highlighting the correctness of our results, and an introduction to the user interface we create.

References

[1] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over Infiniband. In *2006 International Conference on Parallel Processing (ICPP '06).* Columbus, Ohio. IEEE. https://ieeexplore.ieee.org/document/1690651