ECRAM 0.3.0
Technical Documentation

Kim-Thomas Rehmann, Christian Wolf,
Kevin Beineke and Michael Schöttner
Heinrich-Heine-Universität Düsseldorf, Germany

March 19, 2012

Contents

1 Introduction **1**

2 ECRAM Interface **2**
2.1 Objects 2
2.2 Consistency 3
2.3 Condition variables 4
2.4 Nameservice 4
2.5 Debug Interface 4
2.6 Unstable Interface 5

3 Developing Applications **6**
3.1 Prerequisites 6
3.2 Running ECRAM Applications 7
3.3 Understanding Distributed Objects 7
3.4 Example Applications 9

4 Objects **10**
4.1 Object Allocation 10
4.2 Object Accesses 11
4.3 Naming Objects 14

5 Replication **15**
5.1 Versions and Replicas 15
5.2 Module Interface 15
5.3 Version Comparison 16
5.4 Replica Access 16

6 Consistency **17**
6.1 Call Dispatcher 17
6.2 Speculative Execution 17
6.3 Transaction Information 18

6.4 Transaction Validation 18
6.5 Local Commits 19

7 Messaging 20
7.1 Networking 20
7.2 Node Management 21
7.3 Sending and Receiving Messages 21

8 Debugging and Monitoring 23
8.1 Debugging 23
8.2 Monitoring 24
8.3 Wireshark Packet Dissector 24

9 DTK – Job Management 26
9.1 Preprocessor definitions 26
9.2 Interface functions 26
9.3 Internal functions 28
9.4 Data structures 29
9.5 Debug functions 31
9.6 Code example 31

10 DTK – MapReduce 34
10.1 MapReduce 34
10.2 ECRAM MapReduce Framework 34
10.3 Framework 35
10.4 Preprocessor definitions 41
10.5 Code example 42

Chapter 1

Introduction

This document explains interface, design and high-level implementation of the **ECRAM** library. The best and most accurate documentation for **ECRAM** is probably the source code itself. Therefore, the code contains Doxygen-formatted comments and normal comments. See the **ECRAM** website[1] for the source code archive, which contains an up-to-date version of this document.

To help finding a way through the source code, this document specifies the names of files and functions. Preprocessor definitions that can be set in the configuration dialog (`make menuconfig`) are specified in footnotes.

First, we present the **ECRAM** library's interface. Second, we give a general introduction to developing applications with **ECRAM**. Third, we present the internals of **ECRAM**'s various components and modules.

[1] http://www.cs.uni-duesseldorf.de/AG/BS/english/Research/ECRAM

Chapter 2

ECRAM Interface

The ECRAM interface is defined in file ecram.h. A node participates in a distributed application by using the functions ecram_startup and ecram_shutdown. When a bootstrap node is not specified, the node starts a distributed application as the first node. Otherwise, it tries to join an already running distributed application.

2.1 Objects

An ECRAM object is identified by an object ID (OID of type ecram_object_id_t) that is unique in the scope of a distributed application. The width of OIDs is configurable at compile-time.[1]

ECRAM either supports direct-mapped objects or flexible objects.[2] Direct-mapped objects reside in the CPU's virtual address space, such that their OIDs coincide with their virtual address. Flexible objects are not permanently associated with virtual memory addresses.

The characteristics of direct-mapped objects are as follows:

- OIDs are 64 bit wide.
- Atomic objects are 1 byte large. OIDs of atomic objects are consecutive, i.e. offset 1 from OID x is object x+1.
- An OID is a memory address.

[1] config parameter ECRAM_OBJECT_ID

[2] config parameter ECRAM_IN_MEMORY_OBJECTS

- Accesses are transparently detected via MMU.[3] Alternatively, the application can use `ecram_read/ecram_write` for explicit accesses.[4]

The characteristics of flexible objects are as follows:

- OID width is not restricted.
- Objects have variable size. OIDs of atomic objects are not necessarily consecutive. This is not fully implemented yet.
- An OID is independent of object storage.
- The functions `ecram_read/ecram_write` must be used to access objects.[5] Access detection via MMU is not supported.

Objects are created using `ecram_alloc` and destroyed using `ecram_free`. Files can be mapped as objects with `ecram_mmap` similarly to the `mmap` system call.[6] The `ecram_munmap` function deletes an object that has been mapped using `ecram_mmap`, but currently it does not synchronize the object with the file. The function `ecram_msync` is not implemented yet, because file mappings are not managed globally.

2.2 Consistency

Applications can start transactions with `ecram_bot` and finish them with `ecram_eot`. **ECRAM** executes transactions speculatively. If **ECRAM** detects a conflict with a concurrent transaction, it transparently restarts the transaction.[7] The semantics of non-transparent restart are still undefined. The restart mechanism can optionally restart the CPU's floating-point unit.[8] An extended library interface could allow weakly consistent object accesses.[9]

To allow experimenting with transaction properties, the `ecram_transaction_attributes_t` structure enables setting various attributes in calls to `ecram_bot`. For example, transaction statistics can be exported to the calling application.[10] Access to transaction statistics is also

[3] config parameter `ECRAM_ENABLE_ACCESS_DETECTION`
[4] config parameter `ECRAM_ENABLE_READ_WRITE`
[5] config parameter `ECRAM_ENABLE_READ_WRITE`
[6] config parameter `ECRAM_ENABLE_MMAP`
[7] config parameter `ECRAM_ENABLE_TRANSPARENT_RESTART`
[8] config parameter `ECRAM_ENABLE_FPU`
[9] config parameter `ECRAM_ENABLE_SYNC`
[10] config parameter `ECRAM_ENABLE_TRANSACTION_INFO`

possible using the function `ecram_get_statistics`.[11] The validation phase can optionally be skipped if the developer can preclude or tolerate conflicts.[12]

2.3 Condition variables

ECRAM provides a simple mechanism to avoid busy waiting for object state changes, similar to synchronization with condition variables.[13] The `ecram_wait` call blocks until an object is in a specific state. However, upon returning from the call, an application must check whether the object is still in the requested state. Also, short durations of specific states can remain unnoticed, because, unlike `pthread_cond_wait`, `ecram_wait` is not coupled to a mutex.

2.4 Nameservice

A simple nameservice has been built into **ECRAM** (currently only usable with direct-mapped objects).[14] An application can store an object ID under a name using `ecram_nameservice_get`, and retrieve the object ID for a specified name using `ecram_nameservice_set`.

To explore subtrees in the nameservice, the nameservice contains two functions that apply a passed function to several entries in turn. The function `ecram_nameservice_apply` applies the function recursively to the decendants of a specified nameservice entry. Similarly, the function `ecram_nameservice_list` applies the function non-recursively to the children of an entry.

2.5 Debug Interface

Each **ECRAM** module should have a function `module_debug` taking a pointer to a string, i.e. a `char **`.[15] If a string is supplied, it can be parsed to read additional debug parameters. The updated position in the string should be written back.

[11] config parameter `ECRAM_ENABLE_STATISTICS`
[12] config parameter `ECRAM_ENABLE_SKIP_VALIDATION`
[13] config parameter `ECRAM_ENABLE_WAIT`
[14] config parameter `ECRAM_ENABLE_NAMESERVICE`
[15] config parameter `ECRAM_ENABLE_DEBUG`

2.6 Unstable Interface

Some functions in ECRAM are declared as unstable, because they do not fit into the clean interface and might be dropped at some point in the future.[16] Examples for such functions are `ecram_set_nodename`, `ecram_is_initial_node` and `ecram_get_own_node_id`. A well-designed application should not rely on these functions to exist or to work as expected.

[16] config parameter `ECRAM_ENABLE_UNSTABLE`

Chapter 3

Developing Applications

First, we describe the prerequisites to building and using ECRAM. Second, we walk through the process of configuring, building and running an ECRAM application step by step. Third, we introduce several example applications that can serve as starting points for developing applications.

3.1 Prerequisites

Before you start with ECRAM, ensure that the following software packages are installed on your system:

- GCC —`gcc`
- Make —`make`
- Libc —`glibc-dev`
- GLib with thread support —`libglib-dev/libgthread-dev >= 2.14`
- readline —`libreadline-dev`
- bfd —`binutils-dev` — only needed for the extended backtrace functionality[1]
- fuse —`libfuse-dev` — only needed for building the FUSE module[2]

The prerequisites will not be a problem on any current Linux distribution. Some distributions have slightly different names for the packages, such as `xyz-devel` instead of `xyz-dev` for development packages.

[1] config parameter `ECRAM_ENABLE_EXTENDED_BACKTRACE`
[2] config parameter `APPS_FUSE`

3.2 Running ECRAM Applications

The following description helps you build and start up an ECRAM application for the first time.

1. Get the ECRAM source code and change to its top-level directory:
   `cd ~/ecram`
2. Configure ECRAM to suit your needs:
   `make configure`
3. Build the ECRAM library and the provided applications:
   `make`
4. Start the first instance of an application:
   `LD_LIBRARY_PATH=build build/apps/basic/basic -a 127.0.0.1` Setting `LD_LIBRARY_PATH` enables your application to find the ECRAM library without installing it system-wide.
5. On another console, start another instance of an application:
   `LD_LIBRARY_PATH=build build/apps/basic/basic -a 127.0.0.2 -b 127.0.0.1`

As a convention, the `-a` parameter specifies the address to listen for incoming connections, and the `-b` parameter specifies the address of the bootstrap node. Type `q <Enter>` to quit the command shell. After having managed to start two instances of an application on the localhost (127.0.0.x), try to start more instances of the application on different computing nodes.

3.3 Understanding Distributed Objects

To get a first understanding of the distributed objects provided by ECRAM, try several commands in the basic application's interactive shell. First, look at the command categories provided by the basic application, and at the commands for object management. Enter the characters after the prompt, and press the *Enter* key.

```
basic >?
basic >?o
```

Second, start a transaction, allocate an object of 20 bytes, register it in the name-service, and end the transaction.

```
basic >tb
BoT
basic >oa20
allocate
allocated 20 bytes at 0x10000d000
basic >ns /hello 0x10000d000
set value for name
/hello <- 0x10000d000
basic >te
EoT
```

Third, switch to the console running the second node and print the name-service entry, outside or inside a transaction.

```
basic >ng /hello
get value for name
/hello -> (nil)
basic >tb
BoT
basic >ng /hello
get value for name
/hello -> 0x10000d000
basic >te
EoT
```

Note that an access outside a transaction not necessarily retrieves the newest version.

Experiment with waiting for an object condition with command `c=42,0x10000d000` and modifying an object with command `ow0x10000d000,42` (in a transaction). Then try to cause a conflict between concurrent transactions, e.g. start two transactions, write to the same object and finish both transactions. You should observe the second transaction fails to commit and is transparently restarted by **ECRAM**, i.e. all objects will be restored to their initial state.

Map a file with `fm README` and retrieve file information on the other node with `fi 0x100010000`, passing the ID of the file object. Finally, dump the mapping with `ddd 0x10000f000,607`, where `0x10000f000` is the object ID of the memory-mapped file data and `607` is the size of the mapping.

3.4 Example Applications

The `src/apps` subdirectory contains several example applications. The *idle* application contains all code needed to start or join a distributed ECRAM application. The key line in the source code is

```
int ret = ecram_startup(address, port,
   bootstrap_address, bootstrap_port);
```

The *simple* application is an example for allocating objects, using transactions and storing and retrieving entries in the name-service. Look at the function `test_transactional_consistency` to understand how to allocate and initialize an object, store it in the name-service, and busy-wait for another node to modify the object. Once you have figured out how the code works, modify it to use `ecram_wait` instead of busy-waiting.

For a more advanced example on using ECRAM, see the *basic* application. The MapReduce applications such as *wordcount* and *raytracer-mr* are explained in a dedicated chapter later in this document.

Chapter 4

Objects

Object management comprises the distributed ID space, heaps of objects, object allocation dispatcher, access management and MMU control.

4.1 Object Allocation

Objects are allocated using a layered approach. The distributed ID space is partitioned into regions. Each region is assigned to one node. To create smaller objects in regions, regions are handled as heaps.

4.1.1 Distributed ID Space

ECRAM partitions the distributed ID space using regions of objects. The ID space management is implemented in `space.c`. For efficient object lookup, the management will eventually use the key-based routing module `kbr.c` (not implemented yet).

4.1.2 Heap Management

A heap is a region of allocatable objects that are bound to a specific node. The heap module sub-allocates in memory regions obtained from the space module.

4.1.3 Object Allocation

Object allocation requests go to the object allocation dispatcher implemented in `object.c`. The dispatcher decides from which heap to allocate the object. The decision depends on the allocation attributes passed to `ecram_alloc`, but can also be based on monitoring of allocation behaviour or heuristics.

Small objects can be allocated with low space overhead using the *mspaces* allocator in `malloc.c`. The *mspaces* allocator allocates heaps using `object_mmap`. Large objects can be allocated as one entire heap using the *page* allocator.

The `object_free` function should give back the object to the heap it has been allocated from. The function is currently not implemented, because freeing storage to a remote heap is difficult.

The function `object_mmap` allocates an entire heap and, if a file descriptor is passed, copies the file data into the object. The munmap functions frees the heap, it does not write back the file data.

4.2 Object Accesses

Object can be accessed by writing directly to the virtual address corresponding to the object ID (for direct-mapped objects only), or by using read/write functions. Accesses are broken down internally to object blocks.

4.2.1 Block Size

The developer can select the minimum size of an object block.[1]

- Object block size sz $<$ 4KB is only possible for flexible objects. Direct-mapped objects require object block size being a multiple of 4KB.
- Backing storage (physical memory mappings) for direct-mapped objects is created on demand.

4.2.2 Backing Storage

The `access.c` module provides backing storage for direct-mapped objects. These mappings merely cache data from the replication module. Only during transactions that contain direct writes to memory, mappings may contain updated (not yet committed) data. Therefore, we can discard mappings in case of memory pressure.

Linux restricts the size and number of memory mappings (virtual memory areas, VMAs) per process. To save mappings, we support allocating multiple objects in one memory region.[2] Mappings are periodically pruned to save virtual memory using the function `access_prune_mappings`.[3]

[1] config parameter `ECRAM_MINIMUM_LOG2_BLOCK_SIZE`
[2] config parameter `ECRAM_NUM_BLOCKS_PER_MMAP`
[3] config parameter `ECRAM_MAX_MAPPING_MEMORY`

4.2.3 Accesses Via Read and Write Functions

All accesses go through the call dispatcher `dispatcher.c`. For direct-mapped objects, the functions `dispatcher_read` and `dispatcher_write` translate OID and offset to block alignment in ECRAM, because OID or offset might exceed `ECRAM_BLOCK_SIZE`. Flexible objects are not implemented yet here.

The functions `read_aligned` and `write_aligned` assume translated OID, offset and size, i.e. OID aligned to `ECRAM_BLOCK_SIZE` and offset plus size less than or equal to `ECRAM_BLOCK_SIZE`. First they retrieve the preferred version of the accessed object from the consistency module. Second, they perform the access by calling `access_read` or `access_write`. Third, they register the access with the consistency module.

The `access_read` and `access_write` functions first prepare the access with `access_prepare` and then transfer data from or to the replication module with `access_export` or `access_import` if a buffer is supplied and the version defined. For flexible objects, data must be transferred directly between input/output buffer and replication module (not implemented yet).

For direct-mapped objects, preparing an access means creating a mapping, loading the replica into the mapping and granting access using the MMU module. Similarly, finishing an access with `access_finish` revokes the access privilege using the MMU module. In case of write accesses, an object may be modified between `access_prepare` and `access_finish`. The version of modifiable objects is set to `undefined_replica_version` to ensure that fresh data will be loaded when `access_prepare` is called again.

The `access_reinit` function is called after restructuring of region allocation. It resets the object to its default state, i.e. zero-filled content.

4.2.4 MMU-based Access Detection

The `mmu.c` module interfaces between MMU-based memory access detection and the dispatcher for read and write functions `dispatcher_read` and `dispatcher_write`. It does not store any state by itself. To catch page faults, the module installs the `segfault_handler` signal handler for SIGSEGV. The functions `mmu_prepare_read`, `mmu_prepare_write` and `mmu_finish` allow to change the access rights of the memory page specified by the OID. They are typically invoked by the access module to allow accesses, or by the rc module to request access detection.

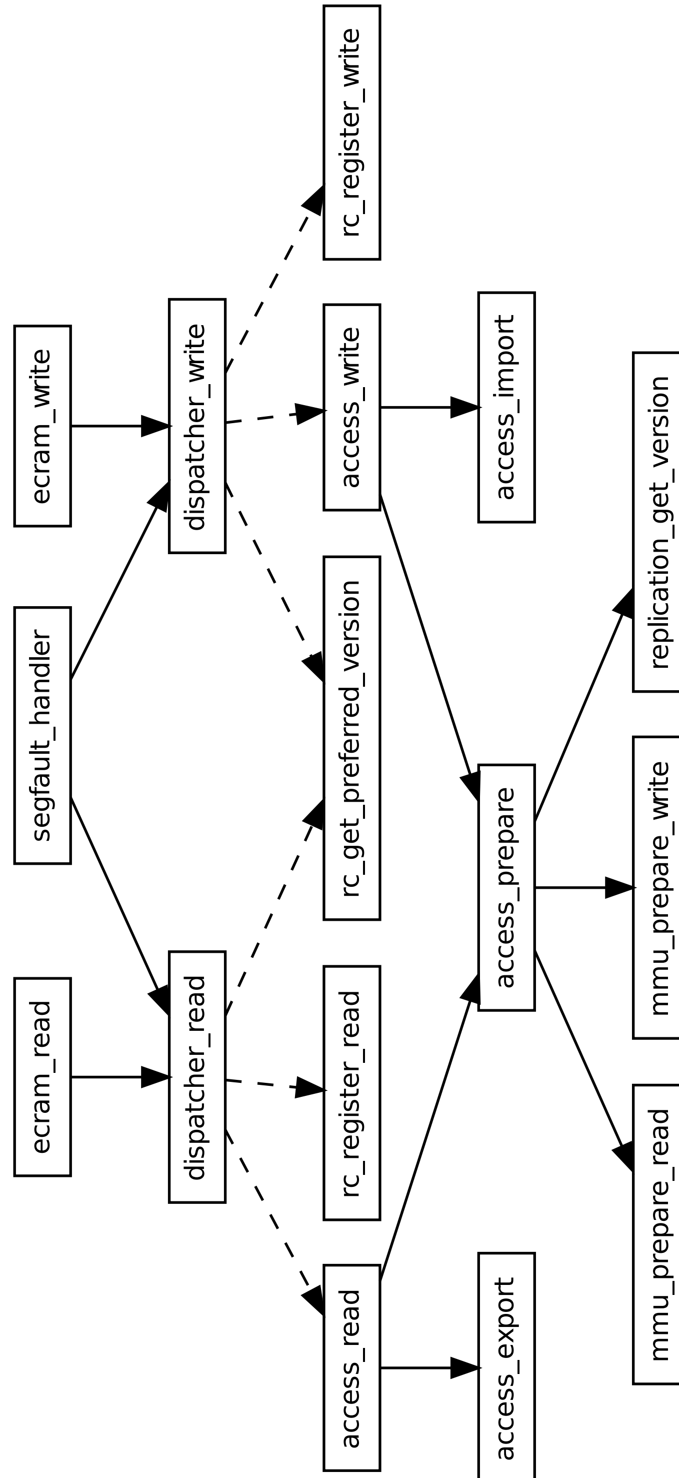

Figure 4.1: Call graph for object accesses

4.2.5 Inter-module Call Structure

Figure 4.1 presents the inter-module function call hierarchy for object accesses. The library interface read/write functions as well as the segfault handler for MMU-based accesses call the function call dispatcher. MMU-based accesses do not call `access_export` or `access_import`, because there is no buffer to transfer data.

4.3 Naming Objects

To enable a distributed application to anchor its data structures, the root module defines a set of root objects. Using `root_set`, an application can register an object ID under an application-defined index. A root object's ID can be retrieved using `root_get`. The module defines at least 256 entries for root objects. The index `ROOT_WORLD` is already predefined, and `ROOT_NAMESERVICE` is the anchor of ECRAM's built-in name-service.

The built-in name-service is implemented by the nameservice module. The functions match their counterparts from the ECRAM interface. The name-service has some predefined entries for the root objects described above, such as `/world` and `/nameservice`. Nameservice entries are organized hierarchically, however, each entry stores at most his first child and one sibling for chaining entries at the same level.

Chapter 5

Replication

The replication module stores the permanent data content of objects. It is also responsible for notification about object changes.

5.1 Versions and Replicas

Object versions are specified using the combination of OID and version number (`replica_version_t`). The replication service is neutral to version numbers, except for `undefined_replica_version`, which acts as a negative result or wildcard, and `initial_replica_version`, which can always be reconstructed in a local operation. A higher version number is considered newer. However, the consistency module defines version IDs and their compatibility.

An object version that exists at a specific node is called a replica. Besides the payload data and the version number, the replica structure can store the previous version number, the version number which invalidated this version, the node who produced this version, and the object's size, which might vary between different object versions. Fields in the structure that are unknown may remain undefined. For example, if the data is currently not available locally, a replica placeholder can be created with `NULL` data to store the `from` node and request the data later.

Initially, the whole ID space is zero-filled. Also, each allocation of a heap restores the objects in the heap to their initial state. The content of zero-filled versions is encoded as `ZERO_FILLED`.

5.2 Module Interface

The interface functions of the replication module consists of the functions `replication_create_version, replication_get_version` and `replication_wait`. The function `replication_create_version` to

create or update a version created either by the node itself or by a peer node. The complementary function `replication_get_version` writes the specified replica's data to a memory buffer.

5.3 Version Comparison

The module has two different comparator functions: The `replica_version_compare` compares not only versions but also invalidation versions. When searching for `undefined_replica_version` with `replica_version_compare`, any replica that is still valid will do. In contrast, the `replica_version_compare_data` is more exact: It does not accept the `undefined_replica_version` wildcard, and it does not look at invalidation versions.

5.4 Replica Access

The low-level function `lookup_or_create` retrieves or creates a replica. The version parameter specifies which replica is requested. The constant `undefined_replica_version` acts as a wildcard for the highest known version number. Looking up `undefined_replica_version` may create `initial_replica_version` if nothing else known about the version. The `fuzzy` parameter specifies whether a replica with an older version that seems to be valid through version may be returned.

The function `create_or_update_version` works at a higher level. It invalidates any previous version, ensures the replica structure exists using `lookup_or_create` and stores the payload data and other fields in the structure, potentially overwriting values that were undefined so far. Finally, the function checks whether the local node was waiting for a state change for this object. Replica updates during local commits are possible by specifying `undefined_replica_version`.

The function `get_version_from_remote` creates a replica by retrieving the version from a peer node. If the manager node for this version is unknown, it asks the space module for the probable manager. If every other attempt fails, the node contacts its bootstrap node.

Chapter 6

Consistency

All consistency related library calls go through the ecram module and the dispatcher module. Transactional consistency and different variations thereof are implemented in the rc module.

6.1 Call Dispatcher

The consistency dispatcher forwards function calls from the ecram module to the responsible module. It translates function call arguments such as object IDs and attributes to the semantics required by the module.

For transaction management, the dispatcher implements flat nested transactions, transactions may occur inside transactions, but all accesses are attributed to the outermost transaction, which is the only transaction passed through to the rc module. In the `dispatcher_eot` function, `access_prune_mappings` is called allow the access module to save memory by removing old memory mappings.

For direct-mapped objects, the object ID, offset and size parameters to read and write calls are adapted to the block alignment.

6.2 Speculative Execution

The remainder of this chapter is implemented in the rc module. During speculative execution of a transaction, all accesses are recorded in the `accessed_objects` structure. The information stored is the object ID, size, version accessed (previous), and the type of access. If the access is a write, the (current) version field is set to undefined, because it will become defined after validation of the transaction. For a read access, the version field equals the previous field. The rc module

also tracks allocations and frees in order to be able to revert them in case of a transaction failure.

After entering the `rc_eot` function to end a transaction, the information gathered during speculative execution is transformed into a `transaction_t` structure by `build_transaction`.

6.3 Transaction Information

To be able to validate transactions, the rc module stores each object's top-most version number in the `versions` hash-table. It also keeps a history of recent transactions, which serves to update the object version numbers in sequence without omitting or reverting an object. The updating of object versions is done by `update_versions`. The function `insert_transaction` function imports a transaction to the history and object versions.

6.4 Transaction Validation

Transaction validation is currently implemented via a central validator node. A non-validator node offloads transaction validation to the validator by calling `remote_validate_and_commit`, which sends the transaction as a `rc_validate` message to the validator. If the validator finds the transaction to be valid, the originating node receives a defined version number for the transaction, which it stores in the transaction structure.

If the validator node receives a `rc_validate` request, it calls `validate_and_commit` and replies with either the valid transaction's defined version or with `undefined_replica_version`.

Nodes that are not involved in a specific transaction will be notified of it by means of a `rc_commit_notification` message. The notification handler creates replicas or placeholders for the objects modified by the transaction. It also inserts the transaction into the transaction history.

The low-level validation is implemented in the `validate` function. Validation can only run if all transactions are known and contained in the history. Therefore, the validation function waits for the global transaction version `top_version` to equal the version until which the transaction history is complete (`complete_version`). Then the function checks for each object in the transaction's read and write set whether the previous version still equals the current version known for the object from the `versions` table. Any object that has been updated during speculative execution causes the transaction to be invalid.

The optimizations for read-only transactions are optional.[1]

6.5 Local Commits

Local commits can update an object without global validation. They can take place only if all objects accessed by the validating transaction have not been replicated. Therefore, the `validate_and_commit` function needs to ensure that no replicas are handed out during local validation and commit (`local_commit`) using `replication_disable` and `replication_enable`. This is severe inter-module locking and may be considered bad.

[1] config parameter `ECRAM_ENABLE_READONLY_TRANSACTIONS`

Chapter 7

Messaging

ECRAM's communication subsystem consists of TCP-based networking, node management and messaging. Furthermore, a `node_info_block_t` represents each node as an object. The key-based routing module for sending messages in a DHT-like manner over the network is not yet functional.

7.1 Networking

The networking module `net.c` stores connections in two hash tables, one indexed with Node-IDs, the other indexed with socket numbers.

Data is sent over the network with the `net_send` function. Its `message_t` parameter contains all information needed: to which node to send the message, the payload data, the length of the payload etc.

To receive messages from other peers, the module starts a network handler thread. The thread runs an endless loop, blocking on `epoll_wait` until the epoll mechanism signals pending events. For an incoming connection request, the event‘s socket is the `myself.socket`, which results in a call to `epoll_accept_connection` to identify this node by sending a hello message. The `EPOLLOUT` flag signals that a connection has been established, in which case `epoll_established_connection` is called. If a message has been received, the `EPOLLIN` flag has been set, and `epoll_receive` is called.

The `epoll_receive` function prepares the peer‘s receive buffer and reads data from the TCP stream into the buffer with `receive_data` in non-blocking mode. If everything went well until now, `decode_message` extracts ECRAM messages from the buffer. This function will in turn transfer control to `message_handle` in the message module. Finally, `compactify_buffer` is called to ensure that subsequent receive operations will not exceed the buffer‘s capacity despite partial messages remaining in the buffer.

During bootstrap, the function net_update_id enables changing the ID of oneself and of the bootstrap node.

7.2 Node Management

The node.c module contains the functionality to join the network by requesting a node ID from a bootstrap node. It also allows to request connection information about third-party nodes. While bootstrapping, a node uses the undefined_node_id, such that the reply to a bootstrap request must identify the joining node by the socket it is connected to.

7.3 Sending and Receiving Messages

Messages are classified using a type and a subtype. Typically the type corresponds to the module that sends the message, and the subtype is internally defined by the module.

7.3.1 Receiving

The message module handles incoming messages in the network thread in function message_handle. This function looks up the module that will handle the message. A return value of 0 means that the message structure can be deleted by the caller, a return value of 1 means that another thread will free the message, because the message is a reply that has been attached to the corresponding request message, which is identified using the in-reply-to field.

A reply message is passed to process_reply and in turn to the specified module‘s reply handler. The reply handler is called with the original message as argument, such that the reply can be found in the message‘s reply field.

7.3.2 Sending

There are several slightly different functions for sending messages:

- The message_send function sends an asynchronous, i.e. one-way and non-blocking, message.
- The message_send_sync function send a synchronous message, which blocks until the corresponding reply has been received. To deal with node failures, the function should be extended with a timeout mechanism and error handling.

- The `message_reply` function sends a reply to a specified request message. Sending a reply is non-blocking.
- The `message_multicast` function sends a message to a list of peers. The current implementation assumes that multicast messages are one-way. Otherwise, the reply processing needs to be extended to work with multiple replies to one message.

Chapter 8

Debugging and Monitoring

The facilities described in this chapter assist the developers in improving ECRAM.

8.1 Debugging

ECRAM's debug output depends on the global debug level[1] and on the per-module debug levels, whatever value is higher. The default debug level is zero, which disables most debug output, but keeps the code compiled in. The debug level can be changed during run-time by calling `set_debug_level_<modulename>` which is an assembler alias to the `set_debug_level` function. Severe errors are output unless the debug level is set to -1.

Debug output is produced using the `dbg_printf, dbg_warn, TODO, dbg_perror and PANIC` macros. The first macro takes the minimum debug level when to print the output, the other macros print the output unconditionally.

ECRAM developers should catch all potential error cases by placing assertions in the code. The `ASSERT(expr)` macro evaluates the argument and, if non-zero, prints on the debug output that the assertion does not hold.

The function `debug_dump_config` prints the config data which is embedded in the ecram library's binary. The function `debug_dump_memory` prints the content of the specified memory in hexadecimal and string format.

[1] config parameter `GLOBAL_DEBUG_LEVEL`

8.2 Monitoring

The monitoring service is designed to be minimally intrusive: It can be turned off completely.[2] The `monitor.h` header file avoids naming collisions by prefixing all symbols with `monitor_`.

Monitoring in ECRAM works by marking entities in the source code. Every time the code reaches the entity, a monitoring event is generated, which causes a handler function to be called. For each entity to be monitored, the monitoring subsystem adds control information to a module's data (`monitor_control_t`).

To monitor an entity, declare it using `MONITOR_DECLARE_EVENT(entity, handler)` or using `MONITOR(entity)` if the `ECRAM_MONITOR_entity` has been declared in the configuration. Use `monitor_trace_event(entity, user_data)` etc. to weave monitoring events in the source code. Alternatively, `monitor_begin_event` and `monitor_end_event` allow to record the entry and exit into a piece of code. The function `monitor_dump_all` causes all monitor entities to be printed by their specific handlers. The developer can modify handler functions during run-time by calling `monitor_set_handler(char *control, char *handler)`. The file `handler.c` defines various handler functions for immediate output, time measurements, collecting user-supplied data and printing call backtraces.

8.3 Wireshark Packet Dissector

The Wireshark network protocol analyzer provides a graphical frontend to record, sort and filter network traffic. As described above, ECRAM messages have a fixed-size PDU header that contains the overall length of the packet. ECRAM network traffic usually comes from or goes to ECRAM's default IP port 2001.

Starting up the ECRAM dissector in `plugin_register` registers two structures: The `hf_register_info hf` registered using `proto_register_field_array` describes the primitive data fields in the ECRAM protocol. The `gint *ett[]` array registered using `proto_register_subtree_array` holds the expansion states of the subtrees.

The dissection of ECRAM packets starts in the function `dissect_ecram`, which reassembles message fragments from the TCP data stream, because data chunks received from sockets need not correspond to ECRAM messages. On each ECRAM message found in the TCP stream, Wireshark calls the function `dissect_ecram_message`, which takes as arguments the `tvbuff_t *tvb`

[2] config parameter `ECRAM_ENABLE_MONITORING`

containing the message data, the `packet_info *pinfo` describing what to display, and the root `proto_tree *tree` of the protocol tree to build. First, the dissector function extracts the elements of the message header and inserts them into the tree. Then it extracts and inserts the specific payload data depending on the message's type and subtype.

Chapter 9

DTK – Job Management

The job management can be used to let nodes execute custom job functions. The node that assigns the jobs is the master, and the other nodes take the role of workers. The communication between the master and the workers is based on job queues. Depending on the preprocessor definitions there may be one global job queue or many private job queues, one for each worker. In general, the master just needs to add a job to a job queue (global or private) to make sure, that it is taken care of. As long as there are jobs in the job queue a worker continues to get jobs from the queue and executes them. Once the job queue is empty, the worker remains in a standby state, waiting for new jobs to arrive by using an `ecram_wait` call on the number of jobs in the job queue. Depending on the preprocessor definitions, a worker may try to steal a job from another job queue before he enters the standby state.

9.1 Preprocessor definitions

It is possible to *run mapreduce jobs from private queues for each worker instead of a global queue*.[1] If this option is enabled, it is also possible to *enable job stealing from local job queues*.[2]

9.2 Interface functions

9.2.1 `job_startup` function

The `job_startup` function initializes the job module. Both the master node and the worker nodes have to call this function at the beginning. The first node

[1]config parameter `MAPREDUCE_RUN_LOCAL`
[2]config parameter `MAPREDUCE_JOB_STEALING`

that calls this function creates the global worker queue and the global job queue. Then it registers both with the nameservice. All other nodes retrieve these queues from the nameservice.

9.2.2 `job_wait_for_workers` function

This function can be used by the master to wait for a certain number of workers to be online before submitting job functions.

9.2.3 `job_submit` function

The purpose of this function is to assign a job to the workers. If no job queue is specified the job queue gets choosen internal. In case of a global job queue all jobs are added to this queue and the FCFS policy is used, to assign the jobs to the workers. In case of private job queues, a round robin scheduling is used to distribute the jobs evenly among the workers. The job object can be set over the parameters of the function. The movable variable determins if the job may be stolen by another worker. The completion variable can be used by the master to check, if the job has finished. So it is possible to assign a group of jobs with the same completion variable and then check for the whole group of jobs, if it has finished (see code example).

9.2.4 `job_run` function

This function is the entry point for the workers. First, the worker registers himself in the global worker queue with help of the `register_worker` function. Then the worker fetches the job queue (`get_job_queue` function). Next the worker begins with the execution of the job functions (`job_loop` function). If the job queue is empty, the worker waits until a job is added to the queue (`wait_for_job` function). The worker continues executing and waiting for jobs until he receives a "terminate" function from the job queue. Then the worker unregisters himself (`unregister_worker` function).

9.2.5 `job_terminate` function

This function adds a "terminate" function to the job queue.

9.2.6 `job_terminate_all` function

This function calls the `job_terminate` function n times, where n is the number of registered workers.

9.2.7 job_get_workers function

This function returns the number of registered workers.

9.3 Internal functions

9.3.1 register_worker function

The worker allocates and sets a worker_node_t object and adds it to the global worker queue.

9.3.2 get_job_queue function

This function returns the global or the local job queue, depending on the preprocessor definition.

9.3.3 job_loop function

This is the main function of the job module, where workers loop waiting for jobs and running them until the terminate variable in the worker_node_t object is set to 1. Basically, in one iteration, the functions wait_for_job, get_job, run_job and finish_job are called, but there is also some logic for the job stealing at the begin of the function: The function idle_nexttime gets called to check if there are jobs left in the current job queue which wait for execution. If there aren't any jobs left, the function steal_work gets called to steal jobs from other job queues.

9.3.4 wait_for_job function

The function makes an ecram_wait call with the condition jobs->nwaiting != 0, which means that there are jobs in the queue waiting for execution.

9.3.5 get_job function

This function moves a job from the inner job waiting queue to the inner job processing queue of a job queue and sets the process flag of the job to a given worker. There is also some logic for the job stealing in this function: The steal_work function calls the get_job function to steal a job from another worker. There

are some jobs that may not be stolen, e.g. a finish job. These jobs have the "movable" flag set to 0. If a worker tries to steal such a job with help of the `get_job` function, the function will return `undefined_object_id`.

9.3.6 `run_job` function

The `run_job` function runs a job by calling one of the custom job functions. These functions need to be declared in a job function object. In case of a terminate function the terminate flag of the worker is set to 1 and the function returns.

9.3.7 `finish_job` function

This function removes a finished job from the inner job procssing queue of the job queue.

9.3.8 `idle_nexttime` function

This function checks whether there are jobs left in a given job queue. If there are'nt any jobs left there are two options: If job stealing is disabled, the worker gets blocked until new jobs have arrived. Elsewise the `steal_work` function gets called to steal a job from another job queue.

9.3.9 `steal_work` function

First, the function checks if there are jobs in the global job queue. If this is the case the `get_job` function gets called for the global job queue. If there is no job in the global job queue the function randomly determins a worker and checks his job queue for waiting jobs. If there are no jobs waiting, the function repeats the last to steps for a maximum of n times, where n is the number of registered workers. If a non-empty job queue was found, the `get_job` function for this job queue gets called.

9.4 Data structures

9.4.1 `job` struct

The `job` struct contains all information of a job:

- the name of the function to execute
- the input of the job

- the output of the job
- the variable movable which specifies if the job may be stolen by another worker
- the variable completion which is set after the job has finished
- the worker, which executes the job
- the timestamp of the start of the execution

9.4.2 `job_queue` struct

Depending on the preprocessor definition there is a global job queue for all jobs or each worker has it's own job queue. A `job_queue` stores the following information:

- the total number of jobs in the queue
- a queue for the jobs, which wait for execution
- a queue for the jobs, which are executed at the moment
- the number of the jobs, which are waiting
- the number of the jobs, which are executed at the moment

9.4.3 `worker_node` struct

The `worker_node` struct contains the following information of a worker:

- the variable terminate, which is set if the worker should terminate
- a pointer to the local job queue of the worker
- the id of the worker
- the variable starting which is set to 1 during the starting phase
- the variable is_thief, which is set to 1 while the worker is stealing jobs from other workers
- a pointer to the other node's queue, where jobs get stolen from

9.4.4 `worker_queue` struct

The `worker_queue` contains the following information:

- the number of workers in the queue
- the variable `starting_phase`, which also contains the number of workers in the queue and is used for the distribution of jobs to the job queues in the starting phase, if `private_queues` are enabled (see preprocessor definitions).
- the queue of the workers

9.4.5 `job_function` struct

The `job_function` object contains the following information:

- the name of the function
- a function pointer to a custom job function

9.5 Debug functions

With help of the debug functions it is possible to print information about a job (`job_debug_job` function), to print information about a job queue (`job_debug_queue` function), to print information about a worker (`job_debug_worker` function) or to print all these information (`job_info` function).

9.6 Code example

```
/*the job function object */

job_function_t example_functions[] =
{
        { "example_map", example_map },
        { "example_reduce", example_reduce },
        JOB_END_OF_FUNCTIONS
};

job_startup(ecram_is_initial_node());
```

```
if (ecram_is_initial_node())
{
  ...

  /*wait for nworkers to be online*/

  job_wait_for_workers(nworkers);

  ...

  /*submit a group of jobs with the same specific custom
     job function (app->map_function="example_map",
     queue=ecram_undefined_object_id)*/

  for (job = 0; job < nmaps; job ++)
  {
    job_submit(queue, input_split, app->map_function,
        intermediate, &app->ncompleted_maps, 1);
  }

  /*wait for this group of jobs to be finished*/

  ecram_wait(&app->ncompleted_maps, 0, ecram_wait_equal,
     nmaps);

  ...

  /*submit another group of jobs with the same specific
     custom job function
     app->reduce_function="example_reduce",
     queue=ecram_undefined_object_id*/

  for (job = 0; job < nreduces; job ++)
  {
    job_submit(queue, reduce_input, app->reduce_function,
        final_result, &app->ncompleted_reduces, 1);
  }

  /*wait for this group of jobs to be finished*/

  ecram_wait(&app->ncompleted_reduces, 0,
     ecram_wait_equal, nreduces);
```

```
  /*terminate*/

  job_terminate_all();

}
else
{
  /*entry point for workers*/

  job_run(job_functions, 0); //0: use local queues, 1: use
     the global queue
}
```

Chapter 10

DTK – MapReduce

10.1 MapReduce

MapReduce is a computing model which has been suggested by the Google employees Dean and Ghemawat in 2004. MapReduce restricts the execution flow and data access of applications to achieve a high degree of parallelism. An application that adheres to the MapReduce model consists of two phases: The map phase splits input data such that several worker nodes can compute intermediate results in parallel. The reduce phase transforms the intermediate results into the final result, again in parallel. A dedicated master node splits, shuffles and merges data and assigns jobs to worker nodes. In the original MapReduce model, data dependencies occur only between input and intermediate data respectively between intermediate and output data, such that both phases are embarassingly parallel, which means that in the map and the reduce phase there is nearly no communication between workers necessary. Thus, MapReduce simplifies synchronization at the expense of restraining data dependencies and control flow.

10.2 ECRAM MapReduce Framework

The ECRAM MapReduce Framework is an in-memory, extended MapReduce framework. The framework stores shared input, output and intermediate data in ECRAM. This enables data orientated communication. Also the framework itself stores data in ECRAM. The framework also supports iterative and on-line data processing.

10.3 Framework

10.3.1 Interface

The interface is defined in `lib/dtk/ecram.h`. The `mapreduce_run` function is the entry point of the mapreduce framework. As parameters it takes the name of the application, the master function and a pointer to the job functions. It acts as an dispatcher. The master function is assigned to the initial node, which takes the roll of the master. All other nodes become workers and take care of the job functions. After configurating the mapreduce framework, the master function calls the `mapreduce` function and the application starts.

10.3.2 User Defined Functions

Only the master function is obligatory. All other functions are optional. The master, pre, post, shuffle functions are executed on the master. All other functions are executed by workers. Depending on the number of iterations, all functions with the exeption of the master function are repeated several times. The chronological order of the functions is equivalent to the order in this document.

master function

All of the configuration takes place in the master function with help of the `mapreduce_application_t app` object. Also parsing of user-defined input data can be implemented in the master function. Therefore the `mapreduce_storage_t app->input` object can be used.

pre function

The pre function takes place after preparing the input data and before splitting it. It has access to the `app->config` object, the `input->data` object and the variables `input->length` and `iteration`. Pre-processing of the input data before each iteration is the purpose of this function.

prepare map functions

After splitting the input and preparing storage for the intermediate results (results of the map functions), the prepare map functions are called. The number of prepare map functions is identical to the number of map functions. These functions have also access to the same objects as the map functions. These are a `mapreduce_storage_t` object and an intermediate result. The first contains

several information of the input data and also some meta information to split the input and the latter is a pointer to a block of allocated memory.

map functions

The map functions usually process the input data and save the intermediate results, so that the reduce functions can use these results to get the final results. Before the map functions are called, the input data gets splitted into equal pieces, one for each map function. Therefore offset and length are calculated, stored in an `mapreduce_storage_t` object, which contains also a reference where to find the input data, and then passed to the map functions. Because there are no dependencies, each map function can process its input split independently. However, the results need to be merged by the reduce functions to gain the final results. To save the intermediate results from the map functions, a block of allocated memory is divided into equal pieces, one for each map function. A pointer to this block is passed to the reduce functions, so that they have access to all intermediate results.

shuffle function

The shuffle function can be used to prepare the intermediate results for the prepare reduce and reduce functions after the map jobs have finished. It therefore has access to all intermediate results.

prepare reduce functions

After preparing the final results, the prepare reduce functions are called. Like the prepare map functions the number of prepare reduce functions equals the number of reduce functions. They also have access to the same objects than the reduce functions, which are a `mapreduce_reduce_input_t` object which contains a reference to all intermediate results and a pointer to the final results. The functions are used for prefetching, preparing statistics etc..

reduce functions

The reduce functions process the intermediate results to gain the final results. Each reduce function usually processes only a part from each intermediate result. The `mapreduce_reduce_input t` object contains a reference to all intermediate results and also some meta information to determine which part of the intermediate results is of interest. The functions also have access to a pointer to the final results.

post function

After the reduce phase has finished, the post function is called, which has access to the app->config object and also the final results, so that it can post-process them after each iteration.

10.3.3 Objects

mapreduce_application_t app object

The mapreduce_application_t app object represents a map reduce application. In the master function all of the configuration can be done using the app object. In general the configuration settings are optional. If a configuration parameter is not specified the default value is used instead. As a result of this the developer needs only to take care of the configuration parameters which are in his interest.

functions The functions can be set with the variables app->pre_function, app->prepare_map_function, app->map_function, app->shuffle_function, app->prepare_reduce_function, app->reduce_function and app->post_function. Note that the functions which are executed by workers need to be defined in an job_function_t object which is a parameter of the mapreduce_run function. Only the name of these functions is then assigned to the app object. The other function variables in the app->object are direct function pointers. Only the functions which are defined in the app object are executed.

number of iterations There are three variables which control the number of iterations: app->max_iterations, app->iteration and app->iterate. Usually the number of iterations is set in app->max_iterations, after each iteration the app->iteration variable is increased and as long as app->max_iterations > app->iteration the iteration continues. A second condition for continuing the iteration is app->iterate != 0, but if the app->max_iterations is set to a value greater than 0, app->iterate is automatically overwritten with 1, even if another value is defined in the master function. A local variable in the pre function ist set to the value of app->iteration, so that the actual number of iterations can be seen. However, in the reduce functions there is acces to the app->iteration and app->iterate variable, so that the usual behaviour of the iterations can be influenced. If for example the

app->iterate variable is set to 0, the iteration stops. Figure 10.1 presents the flowchart for MapReduce iterations.

input The preparation of the input data depends on the app->input object and the app->input_descriptor variable. An input file can be mapped automatically with the map reduce framework, if the app->input object is null and an input descriptor is specified in app->input_descriptor. The prepare_map and map functions then have access to the input data with the ecram_file_t mapping, ecram_object_id_t data and size_t length variables within the maprecuce_storage_t object. For a user-defined input the mapreduce_storage_t app->input object can be set and no automatic mapping will occur. The access to the user-defined input in the prepare map and map functions remains the same. In case of an automatic mapping of the input data the internal variable input->length is set to the length of the file. In case no input descriptor is specified and the app->input object is null, it is set to 0. In these cases it can be overwritten with the app->input_length variable. In case of a user-defined input the variable is defined in the mapreduce_storage_t app->input object. This may have some consequences for the later split phase.

splits (number of map jobs) The app->split_size variable determines the size of one split and also the number of splits. If not defined, the number of splits can be directly set with the app->nsplits variable and the split size is calculated automatically by dividing the input->length variable by the app->nsplits variable. Otherwise, the number of splits is calculated by the formula (input->length + split size - 1)/ split size. For each split an offset and a length are set. The length is for all other than the last split the split size. The length of the last split may be shorter if the division of the input length by the split size doesn't come out even. Both values can be accessed in the map phase with the mapreduce_storage_t object. Also the variables npartitions and id within the mapreduce_storage_t object are set in the split phase. The first contains the number of splits and the second the number of the map job. These variables can be used to determine offset and length within the map phase, if for example a user defined input is used. If neither app->nsplits nor app->split_size are specified, app->nsplits is set to app->nworkers * MAPREDUCE_CHUNK_FACTOR. The default value of app->nworkers is the number of worker nodes.

intermediate size (size of memory for the intermediate results of the map phase) The calculation of the intermediate size depends on the vari-

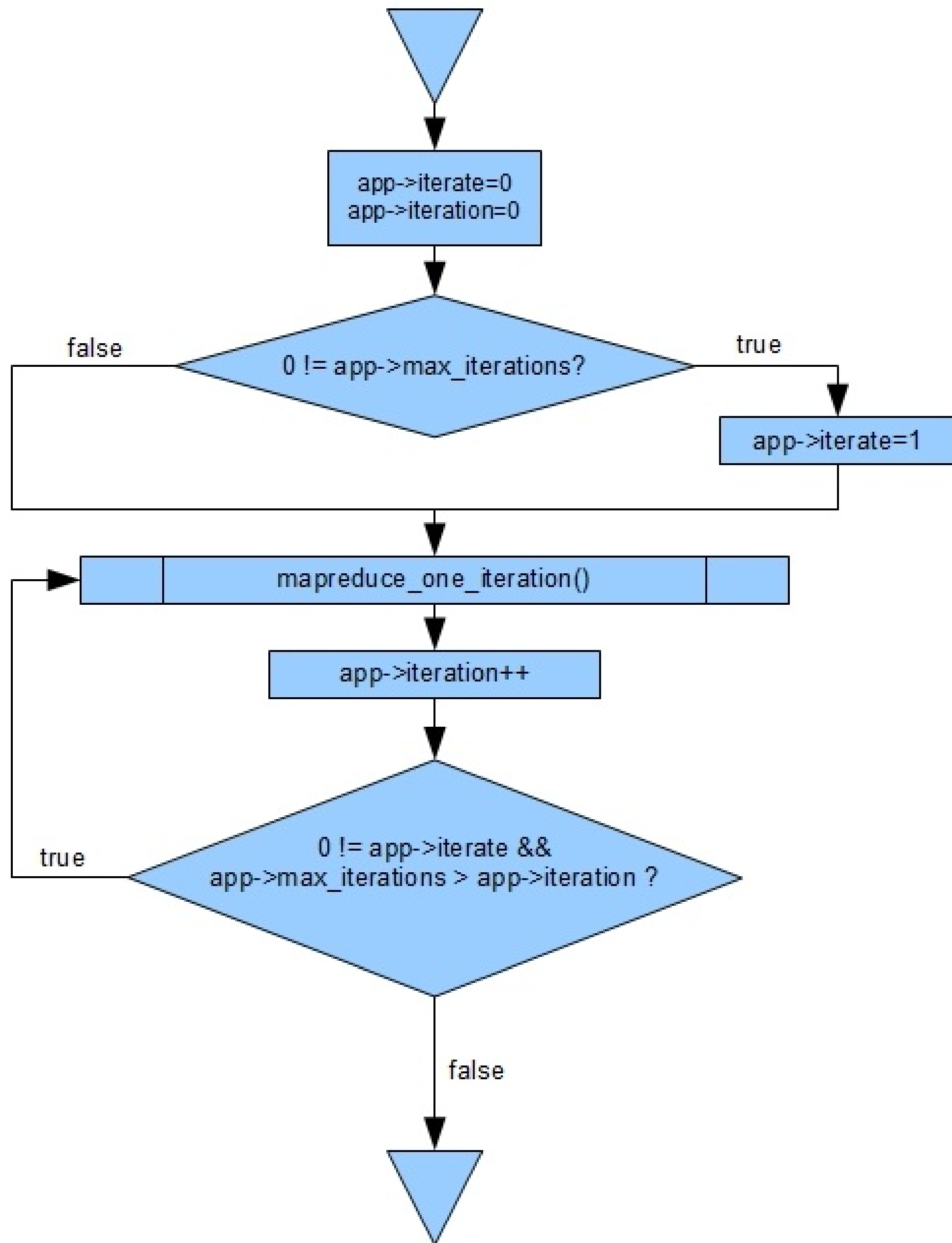

Figure 10.1: Flowchart for MapReduce iterations

able app->total_intermediate_size. If specified, the intermediate size is calculated by dividing app->total_intermediate_size by the number of map jobs. Otherwise, it can directly be set in the app->intermediate_size variable. If neither app->total_intermediate_size nor app->intermediate_size are specified, app->intermediate_size is set to the default value MAPREDUCE_INTERMEDIATE_SIZE.

number of reduce jobs The number of reduce jobs can be determined with the variable app->nreduces. If not specified, there will be no reduce job.

number and size of final results The size of memory for the final results can be determined with app->final_size. If not specified, the size is set to app->nfinals * sizeof(ecram_object_id_t).

storing the output If the app->output_descriptor is specified, the final results are written to a file after each iteration.

mapreduce_storage_t objects

The mapreduce_storage_t objects are the input objects for the map phase. They contain the following information: The objects mapping and data which are set during the preparation of the input and described in the earlier *input* paragraph. The variables offset, length, id and npartitions which are set in the split phase and described in the earlier *split* parapgraph. The config object which can be set in the master function with the app->object parameter. Each map job has its own offset, id and also the length may differ whereas the mapping, data and config parameters are references to the same objects respectively.

mapreduce_reduce_input_t objects

The mapreduce_reduce_input_t objects are the input objects for the reduce phase. They contain the following information: All intermediate results can be accessed with the intermediate_results reference. intermediate_size is a copy of app->intermediate_size as described in the earlier *intermediate size* parapgraph. final_size is a copy of app->final_size as described in the earlier *number and size of final results* paragraph. nintermediates and nreduces are identical and a copy of app->nreduces which is specified in the master function. id is the number

of the reduce job. The two pointers `iteration` and `iterate` are a reference to `app->iteration` and `app->iterate` as described in the *number of iterations* paragraph and can influence the behaviour of the iterations. `config` is a reference to the `app->config` object, which can be specified in the master function. The last three variables are only accessible in the reduce functions and not in the prepare reduce functions.

10.4 Preprocessor definitions

The size of the intermediate results can be specified in the config option *size of intermediate data block in MapReduce*[1]. However, if the `app->total_intermediate_size` is specified, it has no influence at all. If monitoring is enabled, it is possible to enable or disable monitoring of the job functions with help of the config option *enable monitoring of transactions and conflicts in map and reduce functions*[2]. There is the possability to *run mapreduce jobs from private queues for each worker instead of a global queue*[3]. If this is enabled it is possible to set the config option *enable job stealing from local job queues*[4]. The config option: *factor by which the number of workers is multiplied to get the number of map/reduce jobs*[5] is only relevant, if neither `app->nsplits` nor `app->split_size` are specified, as described earlier in the *splits* paragraph. The number of map/prepare map jobs is then determined with the formula `app->nworkers * MAPREDUCE_CHUNK_FAKTOR`.

[1] config parameter `MAPREDUCE_INTERMEDIATE_SIZE`
[2] config parameter `MAPREDUCE_ENABLE_MONITORING`
[3] config parameter `MAPREDECE_RUN_LOCAL`
[4] config parameter `MAPREDUCE_JOB_STEALING`
[5] config parameter `MAPREDUCE_CHUNK_FACTOR`

10.5 Code example

The following code example is copied from the file `/apps/mapreduce/maprreduce.c`. It can be used as a matrix for developing MapReduce applications.

```
/**
 * example map function
 * @param input          mapreduce_storage_t input_split
 * @param output         reference to ?
 */
void example_map(ecram_object_id_t input,
   ecram_object_id_t output)
{
        printf(">> example_map input "PRIo" output
           "PRIo"\n", input, output);
        ecram_bot(0, NULL);
        mapreduce_storage_t *storage =
           (mapreduce_storage_t *)input;
        ecram_object_id_t *intermediate =
           (ecram_object_id_t *)output;
        // allocate and register intermediate data
           structure
        *intermediate = storage;
        ecram_eot(0);
        sleep(5);
}

/**
 * example reduce function
 * @param input          array of intermediate results
 * @param output         reference to ?
 */
void example_reduce(ecram_object_id_t input,
   ecram_object_id_t output)
{
        printf(">> example_reduce input "PRIo" output
           "PRIo"\n", input, output);
        sleep(5);
}
```

This is the master function which configurates the MapReduce Framework using the app object

```
/**
 * configure and run mapreduce on master
 */

void example(ecram_object_id_t infile, ecram_object_id_t
    outfile)
{
        assert(NULL != infile);
        assert(NULL != outfile);
        // create and configure application description
        ecram_bot(0, NULL);
        mapreduce_application_t *app =
            ecram_alloc(sizeof(mapreduce_application_t),
            NULL);
        memset(app, 0, sizeof(mapreduce_application_t));
        strcpy(app->application, "wordcount");
        app->shuffle_function = NULL;
        app->nworkers = job_get_workers(); // NOTE
            initialize with approximate/current number of
            workers
        app->split_size = 0; // NOTE example value
        strcpy(app->map_function, "example_map");
        strcpy(app->reduce_function, "example_reduce");
        app->ncompleted_maps = 0;
        app->ncompleted_reduces = 0;
        ecram_eot(0);
        // run mapreduce
        mapreduce(app);
}
```

```
/**
*  declare job functions
*/

job_function_t example_functions[] =
{
        { "example_map", example_map },
        { "example_reduce", example_reduce },
        JOB_END_OF_FUNCTIONS
};

const char *appname = "example";

/**
 * main function of mapreduce application
 */
int main(int argc, char *argv[])
{
        mapreduce_run(argc, argv, appname, example,
           example_functions);
        exit(EXIT_SUCCESS);
}
```