OCR
The Open Community Runtime Interface

Version 1.0.0, June, 2015

Tim Mattson, Romain Cledat, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Bala Seshasayee, Rob van der Wijngaart, Vivek Sarkar

This page intentionally left blank

Contents

1. Introduction

Extreme scale computers (such as proposed Exascale computers) contain so many components that the aggregate mean-time-between-failure is small compared to the runtime of an application. Programming models (and supporting compilers and runtime systems) must therefore support a variety of features unique to these machines:

- The ability for a programmer to express O(billion) concurrency in an application program.
- The ability of a computation to make progress towards a useful result even as components within the system fail.
- The ability of a computation to dynamically adapt to a high degree of variability in the performance and energy consumption of system components to support efficient execution.
- The ability to either hide overheads behind useful computation or have overheads small enough to allow applications to exhibit strong scaling across the entire exascale system.

There are a number of active research projects to develop a runtime system for extreme scale computers. This specification describes one of these research runtime systems: the *Open Community Runtime* or *OCR*.

The fundamental idea behind OCR is to consider a computation as a dynamically generated directed acyclic graph (DAG) [14, 15, 18]of tasks operating on relocatable blocks of data (Data Blocks). Task execution is managed by events. When the data blocks and events a task depends upon are satisfied, the preconditions for the execution of the task [12] are met and the task will at some later point run on the system. OCR tasks are *non-blocking*. This means that once all preconditions on a task have been met, the task will eventually run to completion regardless of the behavior of any other tasks or events.

Representing a computation in terms of an event-driven DAG of tasks decouples the work of a computation from the "Units of execution" that carry out the computation. The work of a computation is virtualized giving OCR the flexibility to relocate tasks and data to respond to failures in the system [17], achieve a better balance of load among the processing elements of the computer, or to optimize memory and energy consumption [2, 4, 6, 10].

Representing the data in terms of data-blocks decouples the data in a computation from the computer's memory subsystem. This supports transparent scheduling and dynamic migration of data across hardware resources.

OCR is a low level runtime system designed to map onto a wide range of scalable computer systems. It provides the capabilities needed to support a wide range of programming models including data-flow (when events are associated with data-blocks), fork-join (when events enable the execution of post-join continuations), bulk-synchronous processing (when event trees can be used to build scalable barriers and collective operations), and combinations thereof. While some programmers will choose to program directly at the level of the OCR API, we expect most will use higher level programming environments that map onto OCR; hence why we describe OCR as a runtime system rather than an application level programming environment.

1.1. Scope

OCR is a vehicle to support research on programming models and runtime systems for extreme scale computers [8, 9]. This specification defines the state of OCR at a fixed point in its development. There are several limitations in OCR that will be relaxed as it continues to develop.

OCR is a runtime system and collection of low level Application Programming Interfaces (APIs). While some programmers will directly work with the APIs defined by OCR, the most common use of OCR will be to support higher level programming models. Therefore, OCR lacks high level constructs familiar to traditional parallel programmers such as **`reductions`** and **`parallel for`**[1].

All parallelism must be specified explicitly in OCR. It does not extract the concurrency in a program on behalf of a programmer. The OCR execution model is a low level model; abstract enough to support relocation of tasks and data to support resiliency or to minimize the energy consumed by a computation, but low level enough to cleanly map onto the hardware of extreme scale computers.

OCR is designed to handle dynamic task driven algorithms expressed in terms of a directed acyclic graph (DAG). In an OCR DAG, each node is visited only once. This makes irregular problems based on dynamic graphs easier to express. However, it means that OCR may be less effective for regular problems that benefit from static load balancing or for problems that depend on iteration over regular control structures.

OCR is defined in terms of a C library. Programs written in any language that includes an interface to C should be able to work with OCR.

OCR tasks are expressed as event driven tasks (EDTs). The overheads associated with OCR API calls depend on the underlying system software and hardware. On current systems, the overhead of creating and scheduling an event driven task can be fairly high. On system with hardware support for task queues, the overheads can be significantly lower. An OCR programmer should experiment with their implementation of OCR to understand the overheads associated with managing EDTs and assure that the work per EDT is great enough to offset OCR overheads.

[1] Reductions can be supported in OCR using an accumulator/reducer approach [3, 13] and **`parallel for`** can be supported in OCR using a fork-join decomposition similar to the cilk_for construct.

OCR is currently a research runtime system, developed as an open-source community project. It does not as yet have the level of investment needed to develop a production system that can be used for serious application deployment.

1.2. Glossary

Acquired The state of a data block when its chunk of data is accessible to an OCR object. For example, an EDT must acquire a data block before it can read-from or write-to that data block.

Data Block (DB) The data, used by an OCR object such as an EDT, that is intended for access by other OCR objects. A data block specifies a chunk of data that is entirely accessible as an offset from a starting address.

Dependence A dependence is a link between the post-slot of a source event or data-block and the pre-slot of a destination EDT or event. The satisfaction of the source OCR object's post-slot will trigger the satisfaction of the destination OCR object's pre-slot.

Event driven Task (EDT) An OCR object that implements the concept of a task. An EDT with N dependences will have N *pre-slots* numbered from 0 to $N-1$ and one post-slot. Each of the *pre-slots* associated with an EDT connects to a single OCR object, while the EDT's single *post-slot* can connect to multiple OCR objects. An EDT transitions to the *runnable* state when all its pre-slots have been satisfied; the pre-slots determine which data blocks, if any, the EDT may access. Once an EDT is runnable, it is guaranteed to eventually run.

EDT function The function that defines the code to be executed by an EDT. The function takes as arguments the number of parameters, the actual array of parameters, the number of dependences and the actual array of dependences. *Parameters* are static 64-bit values known at EDT creation time and *dependences* are dynamic control or data dependences. The parameter array is copied by value when the EDT is created and enters the *available* state. The dependences (namely the array of dependences) are determined at runtime and are fully resolved only when the EDT is launched and is runnable. The EDT function can optionally return the GUID of a Data Block or event that will be passed along its "post" slot.

EDT template An OCR object from which an EDT instance is created. The EDT template stores meta-data related to the EDT definition, the EDT function, and the number of parameters and dependences available to EDTs instantiated (created) from this template. Multiple EDTs can be created from the same EDT template.

Event

An OCR object used as an indirection mechanism between other OCR objects interested in each other's change of state (unsatisfied to satisfied). Events are the main synchronization mechanism in OCR.

Finish EDT A special class of EDT. As an EDT runs, it may create additional EDTs which may themselves create even more EDTs. For the case of a finish EDT, the EDTs created within its scope (i.e. its child EDTs and further descendants) complete and satisfy their post-slots before the finish EDT can satisfy its post-slot. The result is that any OCR object linked to the post-slot of the finish EDT will by necessity not become runnable (i.e. be scheduled for execution) until the finish EDT and all EDTs created during its execution have completed.

Globally Unique ID (GUID) A value generated by the runtime system that uniquely identifies each OCR object. The GUIDs for the OCR objects reside in a global name space visible to all EDTs.

Latch Event A special type of event that propagates a satisfy signal to its post-slot when it has been satisfied an equal number of times on each of its two pre-slots. In other words, if you imagine a monotonically increasing counter on each of the two pre-slots, the latch event's post-slot will be satisfied if and only if both monotonically increasing counters are non-zero and equal. Note that once the latch event's post-slot is satisfied, satisfaction on the latch event's pre-slots will result in undefined behavior; the latch event will therefore only satisfy its post-slot at most one time.

Link A dependence between OCR objects typically expressed as a connection between the post-slot of one OCR object and the pre-slot of another. Data-Blocks exposed through the pre-slots of an EDT are said to be "linked to the EDT".

OCR object An object managed by OCR. *EDTs*, *events*, *templates*, and *data blocks* are the most frequently encountered examples of OCR objects. Each OCR object has a unique identifier, or GUID.

OCR program A program that is conformant to the OCR specification. Statements in the OCR specification about the OCR program only refer to behaviors associated with the constructs that make up OCR. For example, if an OCR program were to use a parallel programming model outside of OCR, that program is no longer a purely conformant OCR program and its behavior can no longer be defined by OCR.

Released The state of a data block that is no longer accessible by a certain OCR object. For example, after an EDT has finished all of its modification to a data block and it is ready to make those modifications accessible by other EDTs, it must release that data block.

Slot Positional end point for a dependence. An OCR object has one or more slots. Exactly one slot is a *post-slot*. This is used to communicate the state of the OCR object to other OCR objects. The other zero or more slots are *pre-slots*, which are used to manage input dependences of the OCR object. A slot can be:

- Unconnected: There are no links connecting to the slot;

- Connected: a link attaches a source post-slot to a destination pre-slot.

A slot in the *connected* state can be:

- Satisfied: the source of the link has been triggered;
- Unsatisfied: the source of the link has not been triggered.

Task A non-blocking set of instructions that constitute the fundamental "unit of work" in OCR. By "non-blocking" we mean that once all preconditions on a task are met, the task is runnable and it will execute at some point, regardless of what any other task in the system does. The concept of a task is realized by the OCR object "Event Driven Task" or EDT.

Trigger This term is used to describe the action of either a "satisfied" post-slot or of an event whose trigger rule is satisfied. In the former case, when a post slot on an OCR object is satisfied, it triggers any connected pre-slots causing them to become "satisfied". In the latter case, when an event's trigger rule is satisfied (due to satisfaction(s) on its pre-slot(s)), it satisfies its post-slot. Therefore, for most events, when the event's pre-slot becomes satisfied, this will trigger the event and therefore cause it to satisfy its post-slot which will in turn trigger the dependence link and satisfy all pre-slots connected to the event's post-slot. The conjugated form *triggered* is used as an attributive past participle; that is "an EDT that has finished executing the code in its EDT function and released its data blocks will satisfy the event associated with its post-slot and become a triggered EDT".

Unit of Execution A generic term for a process, thread, or any other executable agent that carries out the work associated with a program.

Worker The unit of execution (e.g. a process or a thread) that carries out the sequence of instructions associated with the EDTs in an OCR program. The details of a worker are tied to a particular implementation of an OCR platform and are not defined by OCR.

1.3. OCR objects

An OCR object is a reference counted entity managed by OCR. Every OCR object has a globally unique ID (GUID) used to identify the object. An OCR program is defined in terms of three fundamental objects.

- *Event Driven Task (EDT)*: A non-blocking unit of work in an OCR program.
- *Data block (DB)*: A contiguous block of memory managed by the OCR runtime accessible to any OCR objects to which it is linked.
- *Event*: An object to manage dependences between OCR objects and to define ordering relationships (synchronization) between them.

In addition to these fundamental objects, other OCR objects play a supporting role; making programming more convenient or to provide information the OCR runtime can use to optimize program execution.

- *EDT Template* An OCR object used to manage the resources required to create an EDT.
- *Affinity container* An OCR object used to influence the placement of EDTs in an executing program.

Objects have two well defined states.

1. *Created*: Resources associated with an object and its GUID have been created.
2. *Destroyed*: An object that is destroyed is marked for destruction when the destruction command executes. A destroyed object, its GUID, and any resources associated with the destroyed object are no longer defined. The object is not actually destroyed and the associated resources are not freed until the reference count is zero[2].

Furthermore, for OCR data blocks, we have two additional states:

1. *Acquired*: the data associated with the data block has become accessible to the acquiring OCR object thereby incrementing the acquired objects reference count.
2. *Released*: The object is no longer accessible by the OCR object that had earlier acquired it. The reference count on the released object is decremented.

1.3.1. Dependences, Links and slots

An OCR program is defined as a directed acyclic graph with EDTs and Events as nodes and edges that define *links* between objects. A link defines a dependence between OCR objects. The links are defined in terms of *slots* on the OCR object which define an end point for a dependence. There are two types of slots, *pre-slots* and *post-slots*. A pre-slot defines the prerequisites for an OCR object while the post-slot is used to communicate results from an OCR object.

Event, data blocks and EDTs each have a single *post-slot*. For example, when an EDT finishes the work assigned to it, the EDT sets its post-slot to the state *satisfied*. This triggers any links between that EDT's post-slot and the pre-slot of later OCR objects in the program DAG. The rules defining when a post-slot triggers, the so-called *post-slot trigger rule* depends on the type of OCR object and is discussed in Section 1.3.3.

Some OCR objects (such as EDTs) may also have an optional set of *pre-slots*. A pre-slot defines an incoming dependence or a pre-condition for execution by an EDT. The post-slot of one EDT, for example, can be connected to the pre-slot of another EDT thereby establishing a control dependence between the EDTs. Likewise, the post-slot of a data block can be connected to the pre-slot of an EDT to establish an immediately satisfied data dependence.

[2] As an optimization, the runtime may choose to reuse the same physical object for different logical objects [11, 16].

Slots are used along with data blocks to define data dependences between OCR objects. For example, consider a producer consumer relationships between a pair of EDTs. The post-slot of the producer EDT is connected to the pre-slot of the consumer EDT. When the producer finishes its work and updates the data block it wishes to share, it associates that data block with its post-slot and changes the post-slot's state to "satisfied". This triggers the link between the producer and the consumer making the data block available to the consumer; who can now safely use the data block from the producer.

Slots can take-on the following states.

- Unconnected: There are no links connecting to the slot;
- Connected: a link attaches a source post-slot to a destination pre-slot.

A slot in the *connected* state can be:

- Satisfied: the source of the link has been triggered;
- Unsatisfied: the source of the link has not been triggered.

All slots are initially in the **`unconnected`** state. Data block post-slots are immediately **`satisfied`** as soon as they are connected.

1.3.2. Event Driven Task (EDT)

The fundamental unit of work in OCR is the *Event Driven Task* or *EDT*. When all pre-conditions on an EDT have been met it becomes a runnable EDT. Later when its input data blocks are acquired, the EDT is ready to ready to execute. The OCR runtime guarantees that a runnable EDT will execute at some point and once running, the EDT will progress to its terminal state and cannot be halted by the action of other OCR objects; hence why the execution of an EDT is said to be *non-blocking*.

The work carried out by an EDT is defined by the *EDT function*. The EDT function prototype and return values are defined in the OCR API (see section 2.6).

- The parameters of the EDT function which are copied by value when the EDT is created.
- Dynamic dependences expressed through a dependence array that is formed at runtime from explicit user-specified dependences.
- An optional GUID of a OCR object holding data (a *data block*) that will be used to satisfy the EDT's post-slot. This is the return value of the function.

When the OCR API is used to create an EDT (using the **`ocrEdtCreate()`** function) one or two GUIDs are created. The first (always returned) is the GUID for the EDT itself. The second (returned only on programmer request) is the GUID of the event implied by the post-slot of the

EDT[3] When the OCR function returns a data block, the GUID of that data block is used to satisfy this implied event.

Using a post-slot in a link to another object is just one method to trigger other OCR objects. OCR includes the **`ocrEventSatisfy()`** function to trigger other OCR objects through explicitly created dependence links. The OCR runtime, however, is allowed to defer all event satisfactions to the end of the EDT. This is an important performance optimization designed into OCR. This is also consistent with the intent of OCR to define the state of an evolving computation by the versions of data blocks and a log of the EDTs that have completed. This implies that an OCR programmer should ideally treat EDTs as small units of work that execute with transactional semantics.

OCR defines one special type of EDT; the *finish EDT*. An EDT always executes asynchronously and without blocking once all of its pre-conditions have been met. A finish EDT, however, will not trigger its post-slot until all EDTs launched within its scope (i.e. its child EDTs and EDTs created within its child EDTs) have completed. The finish EDT still executes asynchronously and without blocking. The implied event associated with the post-slot of a finish EDT is a *latch event*, i.e. it is connected to the post-slots of all EDTs created within its scope and does not trigger until they have all finished.

For both normal and finish EDTs, the EDT is created as an instance of an *EDT template*. This template stores metadata about EDTs created from the template, optionally defines the number of dependences and parameters used when creating an instance of an EDT, and is a container for the function that will be executed by an EDT. This function is called the *EDT function*.

1.3.3. Events

An event is an OCR object used to coordinate the activity of other OCR objects. As with any OCR object, events have a single post-slot. Events may also have one or more pre-slots; the actual number of which is determined by the type of event.

The post-slot of an event can be connected to multiple OCR objects by connecting the single post-slot to the pre-slots of other OCR objects. When the conditions are met indicating that the event should trigger (according to the *trigger rule*), the event sets its post-slot to *satisfied* therefore establishing an ordering relationship between the event and the OCR objects linked to the event. Events therefore play a key role in establishing the patterns of synchronization required by a parallel algorithm [7].

When an event is satisfied, it can optionally include an attached data block on its post-slot. Hence, events not only provide synchronization (control dependences) but they are also the mechanism OCR uses to establish data flow dependences. In other words, a classic data flow algorithm defines tasks as waiting until data is “ready”. In OCR this concept is implemented through events with attached data blocks.

[3] It is important to note that although, semantically, an EDT can be the source of a dependence, when adding a dependence, the programmer must use the GUID of the associated event as the source.

Given the diversity of parallel algorithms, OCR has defined several types of events:

1. *Once event*: The event is automatically destroyed on satisfaction. Any object that has the Once event as a pre-condition must already have been created and linked by the time the Once event is satisfied.
2. *Idempotent event*: The event exists until explicitly destroyed by a call to **`ocrEventDestroy()`**. It is satisfied once and subsequent attempts to satisfy (i.e. trigger) the event are ignored.
3. *Sticky event*: The event exists until explicitly destroyed with a call to **`ocrEventDestroy()`**. It is satisfied once and subsequent attempts to satisfy (i.e. trigger) the event result in an error code being returned when trying to satisfy the event.
4. *Latch event*: The latch event has two pre-slots and triggers when the conditions defined by the latch trigger rule are met. The event is automatically destroyed once it triggers; in this regard, it is similar to a *once event*.

Events "trigger" when the appropriate *trigger rule* is met. The default trigger rule for events is when the link on their pre-slot is satisfied, the event triggers and passes the state from the pre-slot to its post-slot. For example, if the pre-slot has an associated data block GUID, that data block GUID will be propagated through the event's post-slot.

The trigger rule for a latch event is somewhat more complicated. The latch event has two pre-slots; an increment slot and a decrement slot. The latch event will trigger its post-slot when the event receives an equal but non-zero number of satisfy notifications on each of the pre-slots. Once a latch event triggers, any subsequent triggers on the pre-slots of the latch event are undefined. For regular events, when it is triggered with a data block, the GUID of that data block is passed along through the post-slot of the event. For a latch event, however, the GUID of a data block that triggers a pre-slot is ignored.

1.3.4. Data Blocks

Data blocks are OCR objects used to hold data in an OCR program. A data block is the only way to store data that persists outside of the scope of a collection of EDTs. Hence, data blocks are the only way to share data between EDTs. The data blocks are identified by their GUIDs and occupy a shared name space of GUIDs. While the name space is shared and globally visible, however, an EDT can only access **a)** data blocks passed into the EDT through a pre-slot or **b)** a data block that is created inside the body of the EDT.

When a data block is created, the default behavior is that the EDT that created the data block will also acquire the data block. This increments the reference counter for the data block and plays a key role in managing the memory of an OCR program. Optionally, an EDT can create a data block on behalf of another EDT. In this case, a programmer can request that the data block is created, but not acquired by the EDT.

Conceptually, data blocks are contiguous chunks of memory that have a start address and a size. They have the following characteristics:

- all memory within the data block is accessible from the start-address using an offset, meaning an EDT can manipulate the contents of a data block through pointers.
- The contents of different Data blocks are guaranteed to not overlap.
- The pointer to the start of a data block is only valid between the acquire of the data block (implicit when the EDT starts) and the corresponding **`ocrDbRelease()`** call (or the end of the acquiring EDT, whichever comes first)

Data blocks can be explicitly connected to other OCR objects through the OCR dependence API (see Chapter 2.7). The more common usage pattern, however, is to attach data blocks to events and pass them through the directed acyclic graph associated with an OCR program to support a data-flow pattern of execution.

Regardless of how the data blocks are exposed among a collection of EDTs, a program may define constraints over how data blocks can be used. This leads to several different modes for how an EDT may access a data block. The mode is set when the OCR dependences API is used to dynamically set dependences between a data block and an EDT. Currently, OCR supports four modes:

1. *Read-Write* (default mode): The EDT may read and write to the data block, but the system will not enforce exclusive access. Multiple EDTs may write to the same data bock at the same time. It is the responsibility of the programmer to assure that when multiple writers are writing to a read-write data block, appropriate synchronization is included to assure that the writes do not conflict. Note: it is legal for an OCR program to contain data races. See section 1.5 for more information.
2. *Exclusive write*: The EDT requires that it is the only EDT writing to a data block at a given time. If multiple EDTs are runnable and want to access the same data block in *exclusive write* mode, the runtime will serialize the execution of these EDTs.
3. *Read only*: The EDT will only read from the Data Block. The OCR Runtime does not restrict the ability of other EDTs to write to the data block, even if the writes from one EDT might overlap with reads by the EDT with read only access. If an EDT writes to a data block it has acquired in constant mode, the results of those writes are undefined should other EDT's later acquire the same data block.
4. *Constant*: The EDT will only read from the data block and the OCR runtime will assure that once the data block is acquired, writes from other EDTs will not be visible. If an EDT writes to a data block it has acquired in constant mode, the results of those writes are undefined should other EDT's later acquire the same data block.

1.4. Execution Model

OCR is based on an asynchronous task model. The work of an OCR program is defined in terms of a collection of tasks organized into a directed acyclic graph (DAG) [14, 15, 18]. Task execution is managed by the availability of data (the "data blocks") and events; hence why the tasks are called "Event Driven Tasks" or EDTs.

An OCR program executes on an abstract machine called the *OCR Platform*. The OCR platform is a resource that can carry out computations. It consists of:

- A collection of network connected nodes where any two nodes can communicate with each other.
- Each node consists of one or more processing elements each of which has its own private memory[4].
- Workers that run on the processing elements to execute enqueued EDTs.
- A globally accessible shared name space of OCR objects each denoted by a globally unique ID (GUID).

OCR is designed to be portable and scalable, hence, the OCR Platform places minimal constraints on the physical hardware.

The OCR program logically starts as a single EDT called **`mainEDT()`**. In other words, the programmer does not provide a **`main()`** function. The OCR runtime system creates the **`main()`** function on the programmer's behalf to set up the OCR environment and then calls the user provided **`mainEDT()`**. The expected function prototype for the **`mainEDT()`** is described in section 2.2.

The DAG corresponding to the executing program is constructed dynamically and completes when the **`ocrShutdown()`** or **`ocrAbort()`** function is called. This rather simple model can handle a wide range of design patterns including branch and bound, data flow, and divide and conquer.

To understand the execution model of OCR, consider the discrete states and transitions of an executing EDT as defined in figure 1.1. An EDT is created and once its GUID is available for use in API function, the EDT is said to be *Available*. At some point, the dependences are fully defined for the EDT and it becomes *resolved*. Note that the transition from Available to Resolved is not called out as a named transition. This implies that it is not generally possible for the system to set a distinct time-stamp corresponding to when the transition occurred. In this case, the transition is un-named because dependencies may be added dynamically up until the EDT *Launch* transition. At this point the EDT is *Runnable*.

Once an EDT is runnable, it will execute at some point during the normal execution of the OCR program. At some point all data blocks linked to an EDT will be acquired and the EDT becomes *Ready*. The EDT and any resources required to support its execution are then submitted to *workers* [5] which execute the tasks on the processing elements within the OCR platform. The

[4] By "private" we mean a memory region that is not accessible to other processing elements.

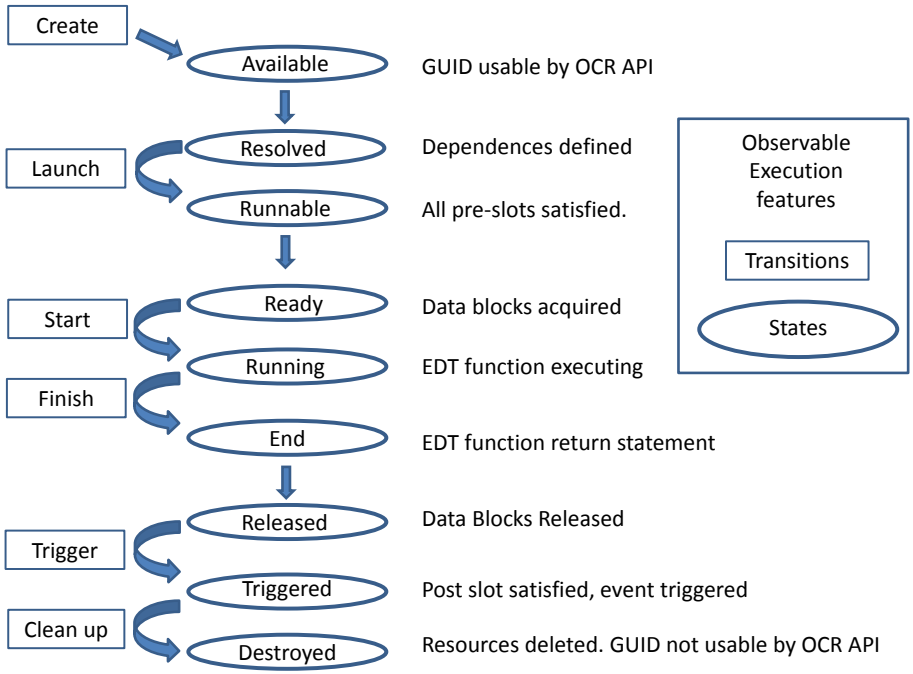

Figure 1.1.: Observable execution features.

workers and the data-structures used to store tasks waiting to execute (i.e. work-pools) are a low level implementation detail not defined by the OCR specification. When reasoning about locality and load balancing, programmers may need to explicitly reason about the behavior of the workers [1], but they do not hold persistent state visible to an OCR program and are logically opaque to OCR constructs. The scheduler inside the implementation of OCR will then schedule the EDT for execution and the EDT *Starts* to execute and becomes a *Running* EDT.

Normal EDT execution continues until the EDT function returns. The EDT undergoes a *Finish* transition and the EDT is in the *End* state. At some point the EDT will release the data blocks associated with the EDTs execution and the EDT enters the *Released* state. At this point, any changes made to data blocks will be available for use by other OCR objects. Later the EDT will mark its post-slot as satisfied to *Trigger* the event associated with the EDT; thereby becoming a *Triggered* EDT. At some later point the system will *Clean-up* resources used by the EDT (including its GUID) and the EDT is Destroyed.

Since an EDT is non-blocking, once it becomes “runnable” it will run on the OCR platform at some point in the future. During its run:

- The EDT can only access data blocks that have been passed in through its pre-slots as well as

any data blocks that the EDT creates internally. This means that before an EDT starts, the OCR runtime knows all the data blocks that will be accessed (minus the ones created within the EDT).

- The EDT can call into the runtime to create and destroy data blocks, EDTs and events.
- The EDT can create *links* or *dependences*. This is accomplished through the **`ocrAddDependence()`** function of the OCR API. The following types of dependences can be created:
  - *Event to Event*: The destination event's pre-slot is chained directly to the source event's post-slot. For all events but the latch event, this means that the triggering of the source event will trigger the destination event.
  - *Event to EDT*: One of the destination EDT's pre-slot is chained directly to the source event's post-slot. When the source event is triggered, this will satisfy the EDT's pre-slot. If a data-block was associated with the triggering of the source event, that data-block will be made available to the EDT in the dependence array in the position of the pre-slot. This is a "control + data" dependence. In the other case, no data-block will be made available and the dependence is akin to a pure control dependence.
  - *Data Block to Event*: Adding a dependence between a data-block and an event is equivalent to satisfying the event with the data-block.
  - *Data Block to EDT*: Directly adding a dependence between a data-block and an EDT (a pure data-dependence) immediately satisfies the EDT's pre-slot and makes the data-block available to the EDT in the dependence array in the position of the pre-slot.
- The EDT cannot perform any synchronization operations that would cause it to block inside the body of the task (i.e. the EDT must be non-blocking). The only mechanism for synchronization within OCR is through the events that link OCR objects, which are explicit to the runtime.

A computation is complete when an EDT terminates the program (e.g. with a call to **`ocrShutdown()`**). Typically, the EDT that terminates the program is the last EDT in the program DAG, and the programmer has assured that all other EDTs in the DAG have completed execution before the function to terminate the program is called.

Since the OCR runtime creates the **`main()`** function, the programmer doesn't need to manage the low level details of initializing and cleanly shutting down OCR.

Links can imply control dependences or, when a data block is associated with an event, they imply data flow between OCR objects. In either case, the events constrain the order of execution of EDTs typically executing the program as a data flow program.

With both data and tasks conceptually decoupled from their realization on a computer system, OCR has the flexibility to relocate tasks and data to respond to failures in the system, achieve a better balance of load among the processing elements of the computer, or to optimize memory and energy consumption [2, 4, 6, 10]. This requires that the state of an OCR program can be defined strictly in terms of which tasks have completed their execution and the history of updates to data blocks. By saving a log of updates to Data Blocks relative to the Tasks that have completed execution, the

system can recover the state of a computation should components of the system fail. This requires, however, that EDTs execute with transactional semantics.

1.5. Memory Model

A memory model defines the values that can be legally observed in memory when multiple units of execution (e.g. processes or threads) access a shared memory system. The memory model provides programmers with the tools they need to understand the state of memory, but it also places restrictions on what a compiler writer can do (e.g. which aggressive optimizations are allowed) and restrictions on what a hardware designer is allowed to do (e.g. the behavior of write buffers).

To construct a memory model for OCR, we need to present a few definitions. The operations inside a task execute in a non-blocking manner. The order of such operations are defined by the *sequenced-before* relations defined by the host C programming language.

When multiple EDTs are running, they execute asynchronously. Usually, a programmer can make few assumptions about the relative orders of operations in two different EDTs. At certain points in the execution of EDTs, however, the OCR program may need to define ordering constraints. These constraints define *synchronized-with* relations.

The "transitive closure" of sequenced-before operations inside each of two EDTs combined with the synchronized-with relations between two EDTs defines a *happens-before* relationship. For example:

- if **`A`** is sequenced-before **`B`** in EDT1
- if **`C`** is sequenced-before **`D`** in EDT2
- and **`B`** is synchronized-with **`C`** in EDT2
- then **`A`** happens-before **`D`**.

These basic concepts are enough to define the memory model for OCR.

OCR provides a relatively simple memory model. Before an EDT can read or write a data block, it must first *acquire* the data block. This is not an exclusive relationship by which we mean it is possible (depending on the mode of the data block in question) for multiple EDTs to acquire the same data block at the same time. When an EDT has finished with a data block and it is ready to expose any modifications to the data block to other EDTs, it must *release* that data block.

> Any function in the OCR runtime that releases a data block must assure that all loads and stores to the data block occur before the data block is released and that the release must complete before the function returns.

The only way to establish a synchronized-with relation is through the behavior of events. If the pre-slot of EDT2 is connected to the post-slot of EDT1, then EDT2 waits for event associated with the post-slot of EDT1 to trigger. Therefore, the satisfy event from EDT1 synchronizes-with the

triggering of the pre-slot of EDT2. We can establish a happens-before relationship between the two EDTs if we define the following rule for OCR.

> An EDT must complete the release of all of its resources before it marks its post-event as satisfied.

An EDT can use data blocks to satisfy events in the body of the task in addition to the event associated with its post-slot. We can reason about the behavior of the memory model and establish happens-before relationship if we define the following rule.

> If an EDT uses a Data Block to satisfy an event, all writes to that data block from the EDT must complete before the event is triggered.

Without this rule we can not assume a release operation followed by satisfying an event defines a sequenced-before relationship that can be used to establish a happens-before relation.

The core idea in the OCR memory model is that happens-before relationships are defined in terms of events (the only synchronization operation in OCR) and the release of OCR objects (such as data blocks). This is an instance of a *Release Consistency* memory model which has the advantage of being relatively straightforward to apply to OCR programs.

The safest course for a programmer is to write programs that can be defined strictly in terms of the release consistency rules. OCR, however, lets a programmer write programs in which two or more EDTs can write to a single data block at the same time (or more precisely, the two EDTs can issue writes in an unordered manner). This may result in a data race in that the value that is ultimately stored in memory depends on how a system chooses to schedule operations from the two EDTs.

Most modern parallel programming languages state that a program that has a data race[5] is an illegal program and the results produced by such a program are undefined. These programming models then define a complex set of synchronization constructs and atomic variables so a programmer has the tools needed to write race-free programs. OCR, however, does not provide any synchronization constructs beyond the behavior of events. This is not an oversight. Rather, this restricted synchronization model helps OCR to better scale on a wider range of parallel computers.

OCR, therefore, allows a programmer to write legal programs that may potentially contain data races. OCR deals with this situation by adding two more rules. In both of these rules, we say that address range A and B are non-overlapping if and only if the set A_1 of 8-byte[6] aligned 8-byte words fully covering A and the set B_1 of 8-byte aligned 8-byte words fully covering B do not overlap. For example, addresses $0x0$ and $0x7$ overlap (assuming byte level addressing) whereas $0x0$ and $0x8$ do not. The first rule deals with the situation of multiple EDTs writing to a data block with non-overlapping address ranges.

> If two EDTs write to a single data block without a well defined order, if the address ranges of the writes do not overlap, the correct result of each write operation must

[5] A *data race* occurs when loads and stores by two units of execution operate on overlapping memory regions without a synchronized-with relation to order them

[6] The reference to "8-byte" words assumes the processing elements utilize a 64-bit architecture. For other cases, all references to an 8-byte word in this specification must be adjusted to match the architecture of the processing elements.

appear in memory.

This behavior may seem obvious making it trivial for a system to support. However, when addresses are distinct but happen to share the same cache lines or when aggressive optimization of writes occur through write buffers, an implementation could mask the updates from one of the EDTs if this rule were not defined in the OCR specification.

The last rule addresses the case of overlapping address ranges. Assume that a system writes values to memory at an atomicity of N-bytes. This defines the fundamental store-atomicity for the system.

> If two EDTs write to a single data block without a well defined order, if the address ranges of the writes overlap, the results written to memory must correspond to one of the legal interleavings of statements from the two EDTs at an N-byte aligned granularity. Overlapping writes to non-aligned or smaller than N-byte granularity are not defined.

For systems that do not provide store-atomicity at any level, N would be 0 and the above rule states that unordered writes to overlapping address ranges are undefined. This rule is the well known *sequential consistency* rule. It states that the actual result appearing in memory may be nondeterministic, but it will be well defined and it will correspond to values from one EDT or the other.

Release consistency remains the safest and best approach to use in writing OCR programs. It is conceivable that some of the more difficult rules may be relaxed in future versions of OCR (especially the sequential consistency rule), but the relaxed consistency model will almost assuredly always be supported by OCR.

Any memory in OCR that can be accessed by multiple EDTs resides in data blocks. As discussed in section 1.3.4 there are four access modes for the data blocks in OCR. The modes and how they interact with the OCR memory model are listed as follows.

- *Read-Write* (default mode): The EDT may read and write to the data block. Multiple EDTs may write to the same data block at the same time with values constrained according to the rules in the OCR memory model.
- *Exclusive write*: The EDT requires that it is the only EDT that can commit write operations to a data block at a given time. Writes must follow a sequential total order; i.e. when more than one EDT is writing to a data block in exclusive write mode, all the writes from one EDT must complete before a subsequent EDT can acquire and then write to the data block. This serializes the execution of EDTs that acquire data blocks in exclusive write mode.
- *Read only*: The EDT will only read from the Data Block. The OCR Runtime does not restrict the ability of other EDTs to write to the data block. The visibility of those writes are undefined; i.e. an implementation may choose whether or not to make writes by other EDTs visible.
- *Constant*: The EDT will only read from the data block and the OCR runtime will assure that once the data block is acquired, writes from other EDTs will not be visible.

1.6. Organization of this document

The remainder of this document is structured as follows:

- Chapter 2 defines the OCR Application Programming Interface.
- Appendix A contains a set of pedagogical examples.
- Appendix B contains a set of proposed OCR Extensions.
- Appendix C contains notes specific to current implementations of OCR.
- Appendix D documents the “change history” for OCR and this specification.

2. The OCR API

This chapter describes the syntax and behavior of the OCR API functions, and is divided into the following sections:

- The core types, macros and error codes used in OCR. (Section 2.1 on page 18)
- A description of OCR main entry point: **`mainEdt`**. (Section 2.2 on page 21)
- Supporting functions. (Section 2.3 on page 22)
- Functions to create, destroy, and otherwise manage the contents of OCR data blocks. (Section 2.4 on page 24)
- Functions to manage events in OCR. (Section 2.5 on page 28)
- Functions to create and destroy tasks in OCR. (Section 2.6 on page 32)
- Functions to manage dependences between OCR objects. (Section 2.7 on page 36)

Types, constants, function prototypes and anything else required to use the OCR API are made available through the **`ocr.h`** include file. You do not need to include any other files unless using extended or experimental features (described in the appendices).

2.1. OCR core types, macros, and error codes

The OCR Application Programming Interface (API) is defined in terms of a C language binding. The functions comprising the OCR API make use of a number of basic data types defined in the include file **`ocr.h`**.

Base low-level types The lowest level data types are defined in terms of the following C typedef statements:

- **`typedef uint64_t u64`** 64-bit unsigned integer;
- **`typedef uint32_t u32`** 32-bit unsigned integer;
- **`typedef uint16_t u16`** 16-bit unsigned integer;

- **`typedef uint8_t u8`** 8-bit unsigned integer;
- **`typedef int64_t s64`** 64-bit signed integer;
- **`typedef int32_t s32`** 32-bit signed integer;
- **`typedef int8_t s8`** 8-bit signed integer;
- **`typedef u8 bool`** 8-bit boolean.

OCR opaque types In addition to these low level types, OCR defines a number of opaque data types to manage OCR objects and to interact with the OCR environment:

- **`ocrGuid_t`**: Opaque handle used to reference all OCR objects;
- **`ocrEdtDep_t`**: Type of dependences as passed into the EDTs; it contains both the GUID and the pointer to any data block passed as a dependence.

Other constants The C API for OCR makes use of a set of basic macros defined inside **`ocr.h`**:

- **`#define true 1`**;
- **`#define TRUE 1`**;
- **`#define false 0`**;
- **`#define FALSE 0`**;
- **`#define NULL_GUID`**: A NULL ocrGuid_t;
- **`#define UNINITIALIZED_GUID`**: An Unitialized GUID (ie: never set);
- **`#define ERROR_GUID`**: An invalid GUID.

OCR error codes The OCR error codes are derived from standard error codes. They are defined internally to limit OCR's dependence on standard libraries. Most functions in the OCR API will return a status code where 0 signifies successful completion. A non-zero return code will be one of the following error codes. These codes are found in the **`ocr-errors.h`** include file and described in Table 2.1.

Error code	Generic Interpretation
OCR_EPERM	Operation not permitted
OCR_ENOENT	No such file or directory
OCR_EINTR	Interrupted OCR runtime call
OCR_EIO	I/O error
OCR_ENXIO	No such device or address
OCR_E2BIG	Argument list too long
OCR_ENOEXEC	Exec format error
OCR_EAGAIN	Try again

table continued on next page

table continued from previous page

Error code	Generic Interpretation
OCR_ENOMEM	Out of memory
OCR_EACCES	Permission denied
OCR_EFAULT	Bad address
OCR_EBUSY	Device or resource busy
OCR_ENODEV	No such device
OCR_EINVAL	Invalid argument
OCR_ENOSPC	No space left on device
OCR_ESPIPE	Illegal seek
OCR_EROFS	Read-only file system
OCR_EDOM	Math argument out of domain of func
OCR_ERANGE	Math result not representable
OCR_ENOSYS	Function not implemented
OCR_ENOTSUP	Function is not supported
OCR_EGUIDEXISTS	The objected referred to by GUID already exists
OCR_EACQ	Data block is already acquired
OCR_EPEND	Operation is pending
OCR_ECANCELED	Operation canceled

2.2. OCR entry point: `mainEdt`

An OCR program's single point of entry is the user-defined main EDT (**`mainEdt()`**). This EDT has a function prototype that is identical to any other EDT; its parameters and dependences are however special: the main EDT has no parameters and has a single input data block that encodes the araguments passed to the program from the command line. The arguments can be accessed using **`getArgc()`** and **`getArgv()`**.

ocrGuid_t mainEdt(u32 paramc, u64 paramv, u32 depc, ocrEdtDep_t depv[])*

Parameters

in	**paramc**	Parameters count will always be 0.
in	**paramv**	Parameter version will always be NULL.
in	**depc**	Dependence count will always be 1.
in	**depv**	Dependence vector will have exactly 1 entry containing a data block that encodes the command line arguments.

Returns **`mainEdt`** returns a **`ocrGuid_t`** which is ignored by the runtime. The returned value of an OCR program is set using **`ocrShutdown()`** or **`ocrAbort()`**.

2.3. Supporting functions

OCR provides a set of basic capabilities to support a program execution environment, handled through the following functions.

Functions

- void **ocrShutdown**(void)

  Called by an EDT to indicate the normal end of an OCR program.

- void **ocrAbort**(u8 errorCode)

  Called by an EDT to indicate an abnormal end of an OCR program.

- u64 **getArgc**(void *dbPtr)

  *Retrieves the traditional 'argc' value from **mainEdt()**'s input data block.*

- char * **getArgv**(void *dbPtr, u64 count)

  *Retrieves the 'count' argument from **mainEdt()**'s input data block.*

- u32 PRINTF(const char *format, ...)

  ***printf()** equivalent for OCR.*

2.3.1. void ocrShutdown( )

The user is responsible for indicating the end of an OCR program using an explicit shutdown call (either this function or **ocrAbort()**). Any EDTs which have not reached the *runnable* state will never be executed. EDTs in the *runnable* or *ready* state may or may not execute.

Note

Program behavior after this call is undefined. Specifically:
- The statements in the calling EDT *after* this function call may or may not be executed;
- EDTs in the *runnable* or *ready* state may or may not execute.

Although most programs will choose to call **ocrShutdown()** in the "last" EDT, OCR specifically allows another EDT to call **ocrShutdown()** to support, for example, a computation of the type "find-first-of"; in such a computation, the program can successfully complete even if all EDTs have not executed.

2.3.2. void ocrAbort(u8 *errorCode*)

This function is very similar to **ocrShutdown()** except that it allows for the return of a non-zero value.

The abort error code is passed back to the runtime and its handling is implementation dependent

Parameters

in	**errorCode**	User defined error code returned to the runtime.

Description See notes in Section 2.3.1.

2.3.3. u64 getArgc(void * *dbPtr*)

Returns the number of arguments (traditionally called 'argc') passed to the OCR program. The value is extracted from the unique data block passed to **mainEdt**.

Parameters

in	**dbPtr**	Pointer to the start of the argument data block

Returns The number of arguments passed to the OCR program on the command line

Description When starting, the first EDT (called **mainEdt**) is passed a single input data block which encodes the arguments passed to the main program:

- The first 8 bytes encode 'argc';
- The following 'argc' 8-byte values encode the offset from the start of the data block to the arguments;
- The arguments are then appended as NULL terminated character arrays.

2.3.4. char * getArgv(void * *dbPtr* u64 *count*)

Returns the 'count' argument passed to the OCR program. The value is extracted from the unique data block passed to **mainEdt**.

Parameters

in	**dbPtr**	Pointer to the start of the argument data block
in	**count**	Index of the argument to extract

Returns A NULL terminated string corresponding to **argv[count]**.

Description See Section 2.3.3 for details.

Note

Attempting to extract an argument with **count** greater or equal to the value returned by **getArgc** will result in undefined behavior.

2.3.5. u32 PRINTF(const char * *fmt* ...)

A platform independent limited **printf** functionality.

Parameters

in	**fmt**	NULL terminated C format string containing the format of the output string
in	...	A varaible length list of arguments in agreement with **fmt**

Returns This function returns the number of bytes written out as a **u32**.

Description OCR must support a wide variety of platforms including simulators that emulate real systems. Often, core functionality provided by standard C libraries are not available on all platforms, and, as a result, an OCR program cannot depend on these functions. There was one case, however, where it was felt support was critical: basic printf. This function supports basic printing functionality and supports the following format specifiers:

- Strings using **%s**;
- 32-bit integers using **%d**, **%u**, **%x** and **%X**;
- 64-bit integers using **%ld**, **%lu**, **%lx**, **%lX**, and versions with two 'l';
- 64-bit pointers using **%p**;
- Floating point numbers using **%f**, **%e** and **%E**;
- The '#' flag is supported for **%x**, **%lx** and **%llx**;
- Precision modifiers are also supported for **%f**, **%e** and **%E**.

Note

A conformant implementation may limit the number of characters in the output string.

2.4. Data block management

Data blocks are the only form of non-ephemeral storage and are therefore the only way to "share" data between EDTs. Conceptually, data blocks are contiguous chunks of memory that have a start address and a size. They also have the following characteristics:

- all memory within the data block is accessible from the start-address using an offset (ie: addresses [start-address; start-address+size[uniquely and totally address the entire data-block);
- a data block is non-overlapping with other distinct data blocks
- the pointer to the start of a data block is only valid between the start of the EDT (or the data block creation) and the corresponding **ocrDbRelease** (or the end of the EDT).

The following macros and enums are used with OCR data blocks.

- **enum ocrInDbAllocator_t** containing:
  - **NO_ALLOC** The data block is not used as a heap
- **enum ocrDbAccessMode_t** containing the four access modes. The meaning of the access modes is given in Section 1.3.4:
  - **DB_MODE_RW**
  - **DB_MODE_EW**
  - **DB_MODE_RO**
  - **DB_MODE_CONST**
- **DB_DEFAULT_MODE** which is an alias of **DB_MODE_RW**
- **DB_PROP_NONE** which specifies no special properties on the data block
- **DB_PROP_NO_ACQUIRE** which specifies that the data block should not be acquired on creation

Functions

- u8 **ocrDbCreate**(ocrGuid_t *db, void **addr, u64 len, u16 flags, ocrGuid_t affinity, ocrInDbAllocator_t allocator)

  Request the creation of a data block.
- u8 **ocrDbDestroy**(ocrGuid_t db)

  Request the destruction of a data block.
- u8 **ocrDbRelease**(ocrGuid_t db)

  Release the data block indicating the end of its use by the EDT.

2.4.1. u8 ocrDbCreate(ocrGuid_t * *db,* void ** *addr,* u64 *len,* u16 *flags,* ocrGuid_t *affinity,* ocrInDbAllocator_t *allocator*)

Requests the creation of a data block of the specified size. After a successful call, the runtime will return the GUID for the newly created data block and a pointer to it (if requested).

Parameters

out	**db**	On successful creation, contains the GUID of the data block. If the call fails, the returned value is undefined.
out	**addr**	On successful creation and if the DB_PROP_NO_ACQUIRE is not specified in **flags**, the created data block will be acquired and its starting address will be returned in this parameter. If DB_PROP_NO_ACQUIRE is specified in **flags**, the value returned will be NULL. If the call fails, the returned value is undefined
in	**len**	Size, in bytes, of the datablock to create
in	**flags**	Flags controlling the behavior of the data block creation. The supported flags are: • DB_PROP_NONE: Default behavior (described in this section) • DB_PROP_NO_ACQUIRE: The created data block may not be used by this EDT (NULL will be returned in **addr**). Note that a conforming implementation may delay the creation of the data block until it is acquired by another EDT.
in,out	**affinity**	Reserved for future use. This parameter should be NULL_GUID
in	**allocator**	A data block can be used as the backing memory for malloc/free-like operations. This parameter specifies the allocator to use for these operations inside this data block. Note that if a data block allocator is used, the user should not write directly to the data block as this may overwrite the meta data used by the allocator. The 'NO_ALLOC' allocator is used to indicate that the data block is not used by any allocator and is therefore usable as POD. Supported values for this parameter are given by ocrInDbAllocator_t.

Returns A status code:

- 0: Successful
- OCR_ENOMEM: The runtime could not provide a data-block of size 'len' due to insufficient memory
- OCR_EINVAL: The arguments passed (flags, allocator, etc.) were not valid
- OCR_EBUSY: A resource required for this call was busy. A retry is possible

Description This function is used to create the basic unit of data in OCR: the data block. Unless DB_PROP_NO_ACQUIRE is specified in **flags**, this function also acquires the newly created data block and returns a pointer to the start of the data block in **addr**.

The created data block:

- Will always be 8-byte aligned
- Will not necessarily be zeroed out; the value of the created data block is undefined;
- Will have a GUID that is unique from this call until the user calls **`ocrDbDestroy()`** on this data block.

Note

- Using the DB_PROP_NO_ACQUIRE flag is recommended to allow the runtime to make placement decisions for newly created data blocks. Not specifying this flag may result in a sub-optimal memory placement for the created data block.
- When DB_PROP_NO_ACQUIRE is specified, a conformant implementation may choose not to create the data-block immediately and instead create it lazily before the using EDT runs.
- Like all GUIDs, the uniqueness of a data block's GUID is not necessarily unique throughout the entire program. A data block's GUID is guaranteed unique only as long as the data block exists (between **`ocrDbCreate()`** and **`ocrDbDestroy()`**).

2.4.2. u8 ocrDbDestroy(ocrGuid_t *db*)

Request for the destruction of a data block. All created data blocks should be destroyed when no longer needed to reclaim the space they utilize.

Parameters

in	**db**	GUID of the data block to destroy

Returns A status code:

- 0: successful
- OCR_EPERM: The data block was already destroyed
- OCR_EINVAL: The GUID passed as argument does not refer to a valid data block

Description OCR does not perform automatic garbage collection; all created data blocks therefore need to be explicitly destroyed by the user. This function will request the destruction of a data block but said destruction will be delayed until all EDTs that have acquired the data block have released it (either explicitly with **`ocrDbRelease()`** or by transitioning to the *released* state).

This function does not need to be called by an EDT that has acquired the data block. If the EDT did acquire the data block, however, this function will implicitly release it (equivalent to calling **`ocrDbRelease()`**.

Note

Not all instances of the errors indicated by the error codes can be caught. In other words,

attempting the destroy a data block multiple times may result in the return of an error code but may also result in undefined behavior.

2.4.3. u8 ocrDbRelease(ocrGuid_t *db*)

An EDT acquires a data block either on creation with **`ocrDbCreate()`** or implicitly when it transitions to the *ready* state. All acquired data blocks will be implicitly released by the runtime when the EDT transitions to the *released* state but they can be released earlier using this function. Releasing a data block indicates that the EDT no longer has use for it and also enables other EDTs to "see" the eventual changes to the data block (release consistency).

Parameters

in	**db**	GUID of the data block to release

Returns A status code:

- 0: successful
- OCR_EINVAL: The GUID passed as argument does not refer to a valid data block
- OCR_EACCESS: The calling EDT has not acquired the data block and therefore cannot release it

Description This function is critical in ensuring proper memory ordering in OCR. A data block can be "shared" with another EDT B by satisfying an event that B depends on. B is only guaranteed to see the changes made to the data block by this EDT if this EDT releases the data block before satisfying the event B depends on with this data block.

Note

Once the EDT releases a data block, it can no longer read or write to it (the pointer it had to it should be considered invalid). Violating this rule will result in undefined behavior. A consequence of this is that a data block can only be released at most once by an EDT.

2.5. Event Management

Events are used to coordinate the execution of tasks and to help establish dependences in the directed acyclic graph representing the execution of an OCR program.

The following macros and enums are used with OCR events:

- **`enum ocrEventTypes_t`** containing the types of supported events. The meaning of these event types is given in Section 1.3.3:

  - **OCR_EVENT_ONCE_T**
  - **OCR_EVENT_IDEM_T**
  - **OCR_EVENT_STICKY_T**
  - **OCR_EVENT_LATCH_T**
- **enum ocrLatchEventSlots_t** containing constants to identify the two pre-slots of the latch event type:
  - **OCR_EVENT_LATCH_DECR_SLOT** identifying the decrement slot of the latch event
  - **OCR_EVENT_LATCH_INCR_SLOT** identifying the increment slot of the latch event

Functions

- u8 **ocrEventCreate**(ocrGuid_t *guid, ocrEventTypes_t eventType, u16 flags)

  Request the creation of an event.
- u8 **ocrEventDestroy**(ocrGuid_t guid)

  Explicitly destroys an event.
- u8 **ocrEventSatisfy**(ocrGuid_t eventGuid, ocrGuid_t dataGuid)

  Satisfy the first pre-slot of an event and optionally pass a data block to the event.
- u8 **ocrEventSatisfySlot**(ocrGuid_t eventGuid, ocrGuid_t dataGuid, u32 slot)

  Satisfy the specified pre-slot of an event and optionally pass a data block to the event.

2.5.1. u8 ocrEventCreate(ocrGuid_t * *guid*, ocrEventTypes_t *eventType*, u16 *flags*)

Requests the creation of an event of the specified type. After a successful call, the runtime will return the GUID for the newly created event. The returned GUID is immediately usable.

Parameters

out	**guid**	On successful creation, contains the GUID of the event. If the call fails, the returned value is undefined.
in	**eventType**	The type of event to create. See .
in	*flags*	Flags impacting the creation of the event. Currently, the following flags are supported: • EVT_PROP_NONE: Default behavior • EVT_PROP_TAKES_ARG: The created event will potentially carry a data block on satisfaction.

Returns A status code:

- 0: successful
- OCR_ENOMEM: The runtime could not create the event due to insufficient memory
- OCR_EINVAL: The **`eventType`** argument is invalid or incompatible with **`flags`**

Description This function is used to create the basic synchronization mechanism is OCR: the event. The lifetime of the created event is dependent on its type. See Section 1.3.3 for more detail.

2.5.2. u8 ocrEventDestroy(ocrGuid_t *guid*)

Certain event types, specifically *sticky* or *idempotent* events do not get automatically destroyed when they are satisfied. The user must explicitly destroy these events when they are no longer needed.

Parameters

in	**`guid`**	GUID of the event to destroy.

Returns A status code:

- 0: Successful
- OCR_EINVAL: The GUID passed as argument does not refer to a valid event

Description *Once* and *latch* events are automatically destroyed by the runtime when they trigger and propagate their satisfaction to the objects connected to their post-slots; those events should not be destroyed using this function. Other events, however, need to be destroyed when they are no longer needed.

Note

If, before this call, the event has EDTs waiting on it that are not yet in the *ready* state, those EDTs will never start unless their dependences are reset to another event. If the waiting EDTs are in the *runnable* state, the behavior is undefined.
Using the GUID of an event after it has been destroyed using this call will result in undefined behavior.

2.5.3. u8 ocrEventSatisfy(ocrGuid_t *eventGuid,* ocrGuid_t *dataGuid*)

Equivalent to **`ocrEventSatisfySlot(eventGuid, dataGuid, 0)`**. See Section 2.5.4 for more detail.

Parameters

in	**eventGuid**	GUID of the event to satisfy
in	**dataGuid**	GUID of the data block to pass to the event or NULL_GUID if this event does not take any data blocks (pure control dependence)

Returns A status code

- 0: successful
- OCR_ENOMEM: The runtime could not satisfy the event due to insufficent memory
- OCR_EINVAL: **eventGuid** or **dataGuid** do not refer to valid event or data block GUIDs respectively
- OCR_EPERM: The event has already been satisfied and does not support multiple satisfactions or a data block was passed to an event which does not take arguments

Description See Section 2.5.4 for a detailed discussion of this function.

2.5.4. u8 ocrEventSatisfySlot(ocrGuid_t *eventGuid,* ocrGuid_t *dataGuid,* u32 *slot*)

Satisfy the specified pre-slot of an event thereby potentially causing waiting EDTs to become *runnable*. This function is the primary method of synchronization in OCR.

Parameters

in	**eventGuid**	GUID of the event to satisfy
in	**dataGuid**	GUID of the data block to pass to the event or NULL_GUID if this event does not take any data blocks (pure control dependence)
in	**slot**	Pre-slot on the destination event to satisfy

Returns A status code

- 0: successful
- OCR_ENOMEM: The runtime could not satisfy the event due to insufficent memory
- OCR_EINVAL: **eventGuid** or **dataGuid** do not refer to valid event or data block GUIDs respectively
- OCR_EPERM: The event has already been satisfied and does not support multiple satisfactions or a data block was passed to an event which does not take arguments

Description Satisfying the pre-slot of an event will potentially trigger the satisfaction of its post-slot depending on its trigger rule:

- *Once*, *idempotentent* and *sticky* events will satisfy their post-slot upon satisfaction of their pre-slot
- *Latch* events will trigger if and only if the number of satisfactions on both their pre-slots is equal. See Section 1.3.3 for more detail.

The **dataGuid** argument is used to associate a data block with the event. *Once*, *idempotent* and *sticky* events will pass this data block down their post slot. An EDT connected to the post-slot of the event (or the post-slot of a chain of events connected to this event) will acquire the data block associated with this event when it transitions to the *ready* state. A data block passed to a *latch* event is ignored.

Note

OCR's memory model (see Section 1.5) imposes that, to guarantee the visibility of the writes to a data block passed to an event (and therefore potentially immediately acquired by an EDT), data blocks need to be *released* with **ocrDbRelease** prior to the satisfaction of the event. Failure to follow this rule will result in undefined behavior. Note that data blocks written to by a preceding EDT will already have been released when that EDT finished and moved to the *release* stage.

2.6. Task management

Event driven tasks – EDTs – act as the task abstraction in OCR, and all program computations are expressed using EDTs.

A support type for EDTs is the EDT template which factor out some information about EDTs. To create an EDT, an EDT template first needs to be created. The EDT template can be reused for all instances of an EDT of the same type.

The following macros and enums are used with OCR tasks.

- **EDT_PROP_NONE** which specifies no special properties for the creation of an EDT
- **EDT_PROP_FINISH** which specifies that the created EDT is a *finish* EDT
- **EDT_PARAM_UNK** which specifies that the number of parameters or dependences to an EDT or EDT template is unknown at this time
- **EDT_PARAM_DEF** which specifies that the number of parameters or dependences to an EDT is the same as the one specified in its template

The prototype of an EDT function is given by ocrGuid_t (∗**ocrEdt_t**)(u32 **paramc**, u64 ∗**paramv**, u32 **depc**, ocrEdtDep_t **depv[]**)

Functions

- u8 **ocrEdtTemplateCreate**(ocrGuid_t guid, ocrEdt_t funcPtr, u32 paramc, u32 depc)

  Request the creation of an EDT template.

- u8 **ocrEdtTemplateDestroy**(ocrGuid_t guid)

  Request the destruction of an EDT template.

- u8 **ocrEdtCreate**(ocrGuid_t *guid, ocrGuid_t templateGuid, u32 paramc, u64 *paramv, u32 depc, ocrGuid_t *depv, u16 flags, ocrGuid_t affinity, ocrGuid_t *outputEvent)

  Request the creation of an EDT instance using the specified EDT template

- u8 **ocrEdtDestroy**(ocrGuid_t guid)

  Request the explicit destruction of an EDT.

2.6.1. u8ocrEdtTemplateCreate(ocrGuid_t * *guid*, ocrEdt_t *funcPtr*, u32 *paramc*, u32 *depc*,)

The EDT template encapsulates information concerning the basic signature and behavior of all EDTs created based on the template. This function creats an EDT template.

Parameters

out	**guid**	On successful creation, contains the GUID of the EDT template. If the call fails, the returned value is undefined.
in	**funcPtr**	The function the EDT will execute when it runs. This function must be of type **ocrEdt_t**.
in	**paramc**	The number of parameters EDTs created based on this template will take. If EDTs created based on this template can take a variable number of arguments, the constant EDT_PARAM_UNK can be used.
in	**depc**	The number of pre-slots that EDTs created based on this template will take. If EDTs created based on this template can take a varaible number of arguments, the constant EDT_PARAM_UNK can be used.

Returns A status code:

- 0: Successful
- OCR_ENOMEM: The runtime could not allocate the metadata for the template

Description An EDT template encapsulates the EDT function and, optionally, the number of parameters and arguments that EDTs instantiated from this template will use. It needs to be created only once for each function that will serve as an EDT.

Note

If the runtime is compiled with OCR_ENABLE_EDT_NAMING, the name of the function will also be stored in the EDT template object to aid in debugging.

2.6.2. u8 ocrEdtTemplateDestroy(ocrGuid_t *guid*)

Destroy an EDT template.

Parameters

in	**guid**	GUID of the EDT template to destroy

Returns A status code:

- 0: Successful
- OCR_EINVAL: The GUID passed as argument does not refer to a valid EDT template

Description This function will destroy the EDT template object.

Note

The destruction of the EDT template can occur even if all EDTs created from it have not run to completion. The EDT template cannot, however, be used to create new EDTs once it has been destroyed.

2.6.3. u8 ocrEdtCreate(ocrGuid_t * *guid*, ocrGuid_t *templateGuid*, u32 *paramc*, u64 * *paramv*, u32 *depc*, ocrGuid_t * *depv*, u16 *flags*, ocrGuid_t *affinity*, ocrGuid_t * *outputEvent*)

Creates a new instance of an EDT based on the specified EDT template. In OCR, an EDT will run at most once and be automatically destroyed once it completes execution.

Parameters

out	**guid**	On successful creation, contains the GUID of the EDT. If the call fails, the returned value is undefined.
in	**templateGuid**	GUID of the template to use to create the EDT.

in	**paramc**	Number of parameters (64-bit values) contained in the **paramv** array. Use **EDT_PARAM_DEF** to use the value specified for the EDT template.
in	**paramv**	Pointer to an array of **paramc** u64 values. The values are copied in and can therefore be freed/reused after this call returns. If **paramc** is 0, this parameter must be set to NULL.
in	**depc**	Number of pre-slots for this EDT. Use **EDT_PARAM_DEF** to use the value specified for the EDT template.
in	**depv**	Pointer to an array of **depc** ocrGuid_t values or NULL. All pre-slots specified in this way will be acquired using **DB_DEFAULT_MODE**. Use **ocrAddDependence()** to specify an alternate mode. If you only want to specify only some of the pre-slots use **UNINITIALIZED_GUID** for the unkown ones.
in	**flags**	Flags controlling the behavior of EDT creation. The supported flags are: • EDT_PROP_NONE: Regular EDT • EDT_PROP_FINISH: Create a *finish* EDT
in	**affinity**	Unsupported and reserved for future use. Set to NULL_GUID.
in,out	**outputEvent**	If non-NULL on input, on successful creation, contains the GUID of the event associated with the post-slot of the EDT. For a *finish* EDT, the post-slot of this event will be satisfied when the EDT and all of its descendent EDTs (the closure of all EDTs created in this EDT and its children) have completed execution. For a non finish EDT, the post-slot of this event will be satisfied when the EDT completes execution and will carry the data block returned by the EDT. If NULL, no event will be associated with the EDT's post-slot. Note that in all cases, the event returned is an event that is automatically destroyed on satisfaction. It is therefore crucial to ensure that all waiters on this event are set up properly *before* the EDT transitions to the *runnable* state.

Returns A status code:

- 0: Successful
- OCR_ENOMEM: The runtime could not create the metadata required for the EDT
- OCR_EINVAL: The GUID specifying the template is invalid or the number of parameters or pre-slots is not fully resolved

Description The EDT is created based on the function provided during the creation of its template. It is required that the number of parameters (**paramc**) and number of pre-slots (**depc**) be resolved at this time. In other words, if **EDT_PARAM_UNK** was specified for either **paramc** or **depc** when creating the template, this function may not specify **EDT_PARAM_DEF**.

Note

If the EDT is created with all its pre-slots specified and resolved, it may execute immediately which means that the GUIDs returned (**guid** and **outputEvent**) may be invalid by the time this function returns. It is the responsability of the programmer to ensure, if he needs to use these returned values, that the EDT cannot start until a later time (by inserting a "fake" pre-slot for example).

2.6.4. u8 ocrEdtDestroy(ocrGuid_t *guid*)

EDTs are automatically destroyed after they execute. This call provides a way to explicitly destroy a created EDT if the programmer later realizes that it will never become *runnable*.

Parameters

in	**guid**	GUID of the EDT to destroy

Description Most programmers will not have use for this function as OCR implicitly destroys all EDTs that execute. There are some cases, however, where an EDT is created and the programmer later realizes that the task will never execute. This function allows the programmer to reclaim the memory used by the EDT

Note

Destroying an EDT that is in the *runnable* state or later will result in undefined behavior. If the EDT had an associated **outputEvent**, that event will also be explictly destroyed.

Returns A status code

- 0: Successful

2.7. Dependence management

At the core of OCR is the concept of a directed acyclic graph that represents the evolving state of an OCR program. EDTs become runnable once their dependences are met. These dependences can be set explicitly when creating the EDT or dynamically using the functions from this section of the API.

Functions

- u8 **ocrAddDependence**(ocrGuid_t source, ocrGuid_t destination, u32 slot, ocrDbAccessMode_t mode)

  Adds a dependence between OCR objects

2.7.1. u8 ocrAddDependence(ocrGuid_t *source*, ocrGuid_t *destination*, u32 *slot*, ocrDbAccessMode_t *mode*)

Adds a dependence between two OCR objects. Concretely, this will link the post-slot of the **source** object to the **slot**th pre-slot of **destination**. When a dependence exists between post-slot *A* and pre-slot *B*, when *A* becomes satisfied, *B* will also become satisfied.

Parameters

in	**source**	GUID of the source OCR object. The source can be a data block or an event or NULL_GUID
in	**destination**	GUID of the destination OCR object. The destination can be an event or an EDT
in	**slot**	Index of the pre-slot on **destination**
in	**mode**	If **destination** is an EDT, access mode with which the EDT will access the data block. If **destination** is an event, this value is ignored.

Returns A status code

- 0: successful
- OCR_EINVAL: **slot** is invalid
- OCR_ENOPERM: **source** and **destination** cannot be linked with a dependence (for example, if **destination** is a data block)

Description The following dependences can be added:

- Event to event: The destination event's pre-slot will become satisfied upon satisfaction of the source event's post-slot. Any data block associated with the source event's post-slot will be associated with the sink event's pre-slot. This allows the formation of event chains.
- Event to EDT: Upon satisfaction of the source event's post-slot, the EDT's pre-slot will be satisfied. When the runtime transitions the EDT from the *runnable* to *ready* state, the data block associated with the post-slot of the event will be acquired using the **mode** specified.

- Data block to event: Equivalent to **ocrEventSatisfySlot**:**ocrAddDependence(db, evt, slot)** is equivalent to **ocrEventSatisfySlot(evt, db, slot)**.
- Data block to EDT: This represents a pure data dependence. Adding a dependence between a data block and an EDT immediately satisfies the pre-slot of the EDT. When the runtime transitions the EDT from the *runnable* to the *ready* state, the data block will be acquired using the **mode** specified.
- NULL_GUID to event or EDT: This is equivalent to "data block to event" or "data block to EDT" where the data is non-existant.

Bibliography

[1] S. Chatterjee. *Runtime Systems for Extreme Scale Platforms*. PhD thesis, Rice University, Dec 2013.

[2] S. Chatterjee, S. Tasirlar, Z. Budimlić, V. Cavé, M. Chabbi, M. Grossman, Y. Yan, and V. Sarkar. Integrating Asynchronous Task Parallelism with MPI. In *IPDPS '13: Proceedings of the 2013 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, 2013.

[3] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[4] Y. Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010.

[5] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, May 2009. IEEE Computer Society.

[6] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, Apr 2010. IEEE Computer Society.

[7] S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *28th European Conference on Object-Oriented Programming (ECOOP)*, Jul 2014.

[8] V. Sarkar et al. DARPA Exascale Software Study report, September 2009.

[9] V. Sarkar, W. Harrod, and A. E. Snavely. Software Challenges in Extreme Scale Systems. January 2010. Special Issue on Advanced Computing: The Roadmap to Exascale.

[10] D. Sbîrlea, Z. Budimlić, and V. Sarkar. Bounded memory scheduling of dynamic task graphs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 343–356, New York, NY, USA, 2014. ACM.

[11] D. Sbîrlea, K. Knobe, and V. Sarkar. Folding of tagged single assignment values for memory-efficient parallelism. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 601–613, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] D. Sbîrlea, A. Sbîrlea, K. B. Wheeler, and V. Sarkar. The Flexible Preconditions Model for Macro-Dataflow Execution. In *The 3rd Data-Flow Execution Models for Extreme Scale Computing Workshop (DFM)*, Sep 2013.

[13] J. Shirako, V. Cave, J. Zhao, and V. Sarkar. Finish Accumulators: a Deterministic Reduction Construct for Dynamic Task Parallelism. In *The 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, March 2013.

[14] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.

[15] S. Tasirlar. Scheduling Macro-Dataflow Programs on Task-Parallel Runtime Systems, Apr 2011.

[16] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar. A practical approach to doacross parallelization. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 219–231, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] N. Vrvilo. Asynchronous Checkpoint/Restart for the Concurrent Collections Model, Aug 2014. MS thesis.

[18] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.

A. OCR Examples

This chapter demonstrates the use of OCR through a series of examples. The examples are ordered from the most basic to the most complicated and frequently make use of previous examples. They are meant to guide the reader in understanding the fundamental concepts of the OCR programming model and API.

A.1. OCR's "Hello World!"

This example illustrates the most basic OCR program: a single function that prints the message "Hello World!" on the screen and exits.

A.1.1. Code example

The following code will print the string "Hello World!" to the standard output and exit. Note that this program is fully functional (ie: there is no need for a `main` function).

```
1 #include <ocr.h>

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("Hello World!\n");
    ocrShutdown();
6   return NULL_GUID;
}
```

A.1.1.1. Details

The `ocr.h` file included on Line 1 contains all of the main OCR APIs. Other more experimental or extended APIs are also located in the `extensions/` folder of the include directory.

EDT's signature is shown on Line 3. A special EDT, named `mainEdt` is called by the runtime if the programmer does not provide a `main` function[1].

[1] Note that if the programmer *does* provide a `main` function, it is the responsability of the programmer to properly initialize the runtime, call the first EDT to execute and properly shutdown the runtime. This method is detailed in TODO and is not recommended as it is not platform portable.

The `ocrShutdown` function called on Line 5 should be called once and only once by all OCR programs to indicate that the program has terminated. The runtime will then shutdown and any non-executed EDTs at that time are not guaranteed to execute.

A.2. Expressing a Fork-Join pattern

This example illustrates the creation of a fork-join pattern in OCR.

A.2.1. Code example

```
/* Example of a "fork-join" pattern in OCR
 *
 * Implements the following dependence graph:
 *
 *   mainEdt
 *   /     \
 * fun1    fun2
 *   \     /
 * shutdownEdt
 *
 */

#include "ocr.h"

ocrGuid_t fun1(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid,(void **) &k, sizeof(int), 0, NULL_GUID, NO_ALLOC);
    k[0]=1;
    PRINTF("Hello from fun1, sending k = %lu\n",*k);
    return db_guid;
}

ocrGuid_t fun2(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    int* k;
    ocrGuid_t db_guid;
    ocrDbCreate(&db_guid,(void **) &k, sizeof(int), 0, NULL_GUID, NO_ALLOC);
    k[0]=2;
    PRINTF("Hello from fun2, sending k = %lu\n",*k);
    return db_guid;
}

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("Hello from shutdownEdt\n");
    int* data1 = (int*) depv[0].ptr;
    int* data2 = (int*) depv[1].ptr;
    PRINTF("Received data1 = %lu, data2 = %lu\n", *data1, *data2);
    ocrDbDestroy(depv[0].guid);
    ocrDbDestroy(depv[1].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    PRINTF("Starting mainEdt\n");
    ocrGuid_t edt1_template, edt2_template, edt3_template;
```

```
    ocrGuid_t edt1, edt2, edt3, outputEvent1, outputEvent2;

    //Create templates for the EDTs
    ocrEdtTemplateCreate(&edt1_template, fun1, 0, 1);
    ocrEdtTemplateCreate(&edt2_template, fun2, 0, 1);
    ocrEdtTemplateCreate(&edt3_template, shutdownEdt, 0, 2);

    //Create the EDTs
    ocrEdtCreate(&edt1, edt1_template, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &outputEvent1);
    ocrEdtCreate(&edt2, edt2_template, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &outputEvent2);
    ocrEdtCreate(&edt3, edt3_template, EDT_PARAM_DEF, NULL, 2, NULL, EDT_PROP_NONE,
        NULL_GUID, NULL);

    //Setup dependences for the shutdown EDT
    ocrAddDependence(outputEvent1, edt3, 0, DB_MODE_CONST);
    ocrAddDependence(outputEvent2, edt3, 1, DB_MODE_CONST);

    //Start execution of the parallel EDTs
    ocrAddDependence(NULL_GUID, edt1, 0, DB_DEFAULT_MODE);
    ocrAddDependence(NULL_GUID, edt2, 0, DB_DEFAULT_MODE);
    return NULL_GUID;
}
```

A.2.1.1. Details

The `ocr.h` file included on Line 13 contains all of the main OCR APIs. The `mainEdt` is shown on Line 44. It is called by the runtime as a `main` function is not provided (more details in `hello.c`).

The `mainEdt` creates three templates (Lines 50, 51 and 52), respectively for three different EDTs (Lines 55, 56 and 57). An EDT is created as an instance of an EDT template. This template stores metadata about EDT, optionally defines the number of dependences and parameters used when creating an instance of an EDT, and is a container for the function that will be executed by an EDT. This function is called the EDT function. For the EDTs, `edt1`, `edt2` and `edt3`, the EDT functions are, `fun1`, `fun2` and `shutdownEdt`, respectively. The last parameter to `ocrEdtTemplateCreate` is the total number of data blocks on which the EDTs depends. The signature of EDT creation API, `ocrEdtCreate`, is shown in Lines 55, 56 and 57. When `edt1` and `edt2` will complete, they will satisfy the output events `outputEvent1` and `outputEvent2` repectively. This is not required for `edt3`. However, `edt3` should execute only when the events `outputEvent1` and `outputEvent2` are satisfied. This is done by setting up dependencies on `edt3` by using the API `ocrAddDependence`, as shown in Lines 60 and 61. This spawns `edt3` but it will not execute until both the events are satisfied. Finally, the EDTs `edt1` and `edt2` are spawned in Lines 64 and 65 respectively. As they do not have any dependencies, they execute the associated EDT functions in parallel. These functions (`fun1` and `fun2`) creates data-blocks using the API `ocrDbCreate` (Lines 18 and 27). The data is written to the data-blocks and the GUID is returned (Lines 21 and 30). This will satisfy the events on which the `edt3` is waiting. The EDT function `shutdownEdt` executes and calls `ocrShutdown` after reading and destroying the two data-blocks.

A.3. Expressing unstructured parallelism

A.3.1. Code example

This example illustrates several aspect of the OCR API with regards to the creation of an irregular task graph. Specifically, it illustrates:

1. Adding dependences between **a)** events and EDTs, **b)** data-blocks and EDTs, and **c)** the NULL_GUID and EDTs;
2. The use of an EDT's post-slot and how a "producer" EDT can pass a data-block to a "consumer" EDT using this post-slot;
3. Several methods of satisfying an EDT's pre-slot: **a)** through the use of an explicit dependence array at creation time, **b)** through the use of another EDT's post-slot and **c)** through the use of an explictly added dependence followed by an `ocrEventSatisfy` call.

```
/* Example of a pattern that highlights the
 * expressiveness of task dependences
 *
 * Implements the following dependence graph:
 *
 * mainEdt
 * |      \
 * stage1a stage1b
 * |     \       |
 * |      \      |
 * |       \     |
 * stage2a  stage2b
 *     \      /
 *     shutdownEdt
 */
#include "ocr.h"

#define NB_ELEM_DB 20

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ASSERT(depc == 2);
    u64* data0 = (u64*)depv[0].ptr;
    u64* data1 = (u64*)depv[1].ptr;

    ASSERT(*data0 == 3ULL);
    ASSERT(*data1 == 4ULL);
    PRINTF("Got a DB (GUID 0x%lx) containing %lu on slot 0\n", depv[0].guid, *data0);
    PRINTF("Got a DB (GUID 0x%lx) containing %lu on slot 1\n", depv[1].guid, *data1);

    // Free the data-blocks that were passed in
    ocrDbDestroy(depv[0].guid);
    ocrDbDestroy(depv[1].guid);

    // Shutdown the runtime
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t stage2a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]);

ocrGuid_t stage1a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ASSERT(depc == 1);
```

```
    ASSERT(paramc == 1);
    // paramv[0] is the event that the child EDT has to satisfy
    // when it is done

    // We create a data-block for one u64 and put data in it
    ocrGuid_t dbGuid = NULL_GUID, stage2aTemplateGuid = NULL_GUID,
        stage2aEdtGuid = NULL_GUID;
    u64* dbPtr = NULL;
    ocrDbCreate(&dbGuid, (void**)&dbPtr, sizeof(u64), 0, NULL_GUID, NO_ALLOC);
    *dbPtr = 1ULL;

    // Create an EDT and pass it the data-block we just created
    // The EDT is immediately ready to execute
    ocrEdtTemplateCreate(&stage2aTemplateGuid, stage2a, 1, 1);
    ocrEdtCreate(&stage2aEdtGuid, stage2aTemplateGuid, EDT_PARAM_DEF,
                 paramv, EDT_PARAM_DEF, &dbGuid, EDT_PROP_NONE, NULL_GUID, NULL);

    // Pass the same data-block created to stage2b (links setup in mainEdt)
    return dbGuid;
}

ocrGuid_t stage1b(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ASSERT(depc == 1);
    ASSERT(paramc == 0);

    // We create a data-block for one u64 and put data in it
    ocrGuid_t dbGuid = NULL_GUID;
    u64* dbPtr = NULL;
    ocrDbCreate(&dbGuid, (void**)&dbPtr, sizeof(u64), 0, NULL_GUID, NO_ALLOC);
    *dbPtr = 2ULL;

    // Pass the created data-block created to stage2b (links setup in mainEdt)
    return dbGuid;
}

ocrGuid_t stage2a(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ASSERT(depc == 1);
    ASSERT(paramc == 1);

    u64 *dbPtr = (u64*)depv[0].ptr;
    ASSERT(*dbPtr == 1ULL); // We got this from stage1a

    *dbPtr = 3ULL; // Update the value

    // Pass the modified data-block to shutdown
    ocrEventSatisfy((ocrGuid_t)paramv[0], depv[0].guid);

    return NULL_GUID;
}

ocrGuid_t stage2b(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ASSERT(depc == 2);
    ASSERT(paramc == 0);

    u64 *dbPtr = (u64*)depv[1].ptr;
    // Here, we can run concurrently to stage2a which modifies the value
    // we see in depv[0].ptr. We should see either 1ULL or 3ULL

    // On depv[1], we get the value from stage1b and it should be 2
    ASSERT(*dbPtr == 2ULL); // We got this from stage2a

    *dbPtr = 4ULL; // Update the value

    return depv[1].guid; // Pass this to the shudown EDT
}
```

```
108
ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {

    // Create the shutdown EDT
113 ocrGuid_t stage1aTemplateGuid = NULL_GUID, stage1bTemplateGuid = NULL_GUID,
        stage2bTemplateGuid = NULL_GUID, shutdownEdtTemplateGuid = NULL_GUID;
    ocrGuid_t shutdownEdtGuid = NULL_GUID, stage1aEdtGuid = NULL_GUID,
        stage1bEdtGuid = NULL_GUID, stage2bEdtGuid = NULL_GUID,
        evtGuid = NULL_GUID, stage1aOut = NULL_GUID, stage1bOut = NULL_GUID,
118     stage2bOut = NULL_GUID;

    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0, 2);
    ocrEdtCreate(&shutdownEdtGuid, shutdownEdtTemplateGuid, 0, NULL, EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_GUID, NULL);
123
    // Create the event to satisfy shutdownEdt by stage 2a
    // (stage 2a is created by 1a)
    ocrEventCreate(&evtGuid, OCR_EVENT_ONCE_T, true);

128 // Create stages 1a, 1b and 2b
    // For 1a and 1b, add a "fake" dependence to avoid races between
    // setting up the event links and running the EDT
    ocrEdtTemplateCreate(&stage1aTemplateGuid, stage1a, 1, 1);
    ocrEdtCreate(&stage1aEdtGuid, stage1aTemplateGuid, EDT_PARAM_DEF, &evtGuid,
133                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_GUID, &stage1aOut);

    ocrEdtTemplateCreate(&stage1bTemplateGuid, stage1b, 0, 1);
    ocrEdtCreate(&stage1bEdtGuid, stage1bTemplateGuid, EDT_PARAM_DEF, NULL,
                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_GUID, &stage1bOut);
138
    ocrEdtTemplateCreate(&stage2bTemplateGuid, stage2b, 0, 2);
    ocrEdtCreate(&stage2bEdtGuid, stage2bTemplateGuid, EDT_PARAM_DEF, NULL,
                 EDT_PARAM_DEF, NULL, EDT_PROP_NONE, NULL_GUID, &stage2bOut);

143 // Set up all the links
    // 1a -> 2b
    ocrAddDependence(stage1aOut, stage2bEdtGuid, 0, DB_DEFAULT_MODE);

    // 1b -> 2b
148 ocrAddDependence(stage1bOut, stage2bEdtGuid, 1, DB_DEFAULT_MODE);

    // Event satisfied by 2a -> shutdown
    ocrAddDependence(evtGuid, shutdownEdtGuid, 0, DB_DEFAULT_MODE);
    // 2b -> shutdown
153 ocrAddDependence(stage2bOut, shutdownEdtGuid, 1, DB_DEFAULT_MODE);

    // Start 1a and 1b
    ocrAddDependence(NULL_GUID, stage1aEdtGuid, 0, DB_DEFAULT_MODE);
    ocrAddDependence(NULL_GUID, stage1bEdtGuid, 0, DB_DEFAULT_MODE);
158
    return NULL_GUID;
}
```

A.3.1.1. Details

The snippet of code shows one possible way to construct the irregular task-graph shown starting on Line 5. `mainEdt` will create **a)** `stage1a` and `stage1b` as they are the next things that need to execute but also **b)** `stage2b` and `shutdownEdt` because it is the immediate dominator of those

EDTs. In general, it is easiest to create an EDT in its immediate dominator because that allows any other EDTs who need to feed it information (necessarily between its dominator and the EDT in question) to be able to know the value of the opaque GUID created for hte EDT. `stage2a`, on the other hand, can be created by `stage1a` as no-one else needs to feed information to it.

Most of the "edges" in the dependence graph are also created in `mainEdt` starting at Line 145. These are either between the post-slot (output event) of a source EDT and an EDT or between a regular event and an EDT. Note also the use of NULL_GUID as a source for two dependences starting at Line 156. A NULL_GUID as a source for a dependence immediately satisfies the destination slot; in this case, it satisfies the unique dependence of `stage1a` and `stage1b` and makes them runable. These two dependences do not exist in the graph shown starting at Line 5 but are crucial to avoid a potential race in the program: the output events of EDTs are similar to ONCE events in the sense that they will disappear once they are satisfied and therefore, any dependence on them must be properly setup prior to their potential satisfaction. In other words, the `ocrAddDependence` calls starting at Line 145 must *happen-before* the satisfaction of `stage1aOut` and `stage1bOut`. This example shows three methods of satisfying an EDT's pre-slots:

- Through the use of an explicit dependence array known at EDT creation time as shown on Line 57;
- Through an output event as shown on Line 61. The GUID passed as a return value of the EDT function will be passed to the EDT's output event (in this case `stage1aOut`). If the GUID is a data-block's GUID, the output event will be satisfied with that data-block. If it is an event's GUID, the two events will become linked;
- Through an explicit satisfaction as shown on Line 88).

A.4. Using a Finish EDT

A.4.1. Code example

The following code demonstrates the use of Finish EDTs by performing a Fast Fourier Transform on a sparse array of length 256 bytes. For the sake of simplicity, the array contents and sizes are hardcoded, however, the code can be used as a starting point for adding more functionality.

```
/* Example usage of Finish EDT in FFT.
 *
 * Implements the following dependence graph:
 *
 * MainEdt
 *    |
 *
 * FinishEdt
 * {
 *       DFT
```

```
 *       /   \
 * FFT-odd  FFT-even
 *       \   /
 *      Twiddle
 * }
 *    |
 * Shutdown
 *
 */

#include ``ocr.h''
#include ``math.h''

#define N           256
#define BLOCK_SIZE  16

// The below function performs a twiddle operation on an array x_in
// and places the results in X_real & X_imag. The other arguments
// size and step refer to the size of the array x_in and the offset therein
void ditfft2(double *X_real, double *X_imag, double *x_in, u32 size, u32 step) {
    if(size == 1) {
        X_real[0] = x_in[0];
        X_imag[0] = 0;
    } else {
        ditfft2(X_real, X_imag, x_in, size/2, 2 * step);
        ditfft2(X_real+size/2, X_imag+size/2, x_in+step, size/2, 2 * step);
        u32 k;
        for(k=0;k<size/2;k++) {
            double t_real = X_real[k];
            double t_imag = X_imag[k];
            double twiddle_real = cos(-2 * M_PI * k / size);
            double twiddle_imag = sin(-2 * M_PI * k / size);
            double xr = X_real[k+size/2];
            double xi = X_imag[k+size/2];

            // (a+bi)(c+di) = (ac - bd) + (bc + ad)i
            X_real[k] = t_real +
                (twiddle_real*xr - twiddle_imag*xi);
            X_imag[k] = t_imag +

                (twiddle_imag*xr + twiddle_real*xi);
            X_real[k+size/2] = t_real -
                (twiddle_real*xr - twiddle_imag*xi);
            X_imag[k+size/2] = t_imag -
                (twiddle_imag*xr + twiddle_real*xi);
        }
    }
}

// The below function splits the given array into odd & even portions and
// calls itself recursively via child EDTs that operate on each of the portions,
// till the array operated upon is of size BLOCK_SIZE, a pre-defined
// parameter. It then trivially computes the FFT of this array, then spawns
// twiddle EDTs to combine the results of the children.
ocrGuid_t fftComputeEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t computeGuid = paramv[0];
    ocrGuid_t twiddleGuid = paramv[1];
    double *data = (double*)depv[0].ptr;
    ocrGuid_t dataGuid = depv[0].guid;
    u64 size = paramv[2];
    u64 step = paramv[3];
    u64 offset = paramv[4];
    u64 step_offset = paramv[5];
    u64 blockSize = paramv[6];
    double *x_in = (double*)data;
```

```
    double *X_real = (double*)(data+offset + size*step);
    double *X_imag = (double*)(data+offset + 2*size*step);

    if(size <= blockSize) {
        ditfft2(X_real, X_imag, x_in+step_offset, size, step);
    } else {
        // DFT even side
        u64 childParamv[7] = { computeGuid, twiddleGuid, size/2, 2 * step,
                               0 + offset, step_offset, blockSize };
        u64 childParamv2[7] = { computeGuid, twiddleGuid, size/2, 2 * step,
                                size/2 + offset, step_offset + step, blockSize };

        ocrGuid_t edtGuid, edtGuid2, twiddleEdtGuid, finishEventGuid, finishEventGuid2;

        ocrEdtCreate(&edtGuid, computeGuid, EDT_PARAM_DEF, childParamv,
                     EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_GUID,
                     &finishEventGuid);
        ocrEdtCreate(&edtGuid2, computeGuid, EDT_PARAM_DEF, childParamv2,
                     EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_GUID,
                     &finishEventGuid2);

        ocrGuid_t twiddleDependencies[3] = { dataGuid, finishEventGuid, finishEventGuid2 };
        ocrEdtCreate(&twiddleEdtGuid, twiddleGuid, EDT_PARAM_DEF, paramv, 3,
                     twiddleDependencies, EDT_PROP_FINISH, NULL_GUID, NULL);

        ocrAddDependence(dataGuid, edtGuid, 0, DB_MODE_RW);
        ocrAddDependence(dataGuid, edtGuid2, 0, DB_MODE_RW);
    }

    return NULL_GUID;
}

// The below function performs the twiddle operation
ocrGuid_t fftTwiddleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    double *data = (double*)depv[0].ptr;
    u64 size = paramv[2];
    u64 step = paramv[3];
    u64 offset = paramv[4];
    double *x_in = (double*)data+offset;
    double *X_real = (double*)(data+offset + size*step);
    double *X_imag = (double*)(data+offset + 2*size*step);

    ditfft2(X_real, X_imag, x_in, size, step);

    return NULL_GUID;
}

ocrGuid_t endEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t dataGuid = paramv[0];

    ocrDbDestroy(dataGuid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {

    ocrGuid_t computeTempGuid, twiddleTempGuid, endTempGuid;
    ocrEdtTemplateCreate(&computeTempGuid, &fftComputeEdt, 7, 1);
    ocrEdtTemplateCreate(&twiddleTempGuid, &fftTwiddleEdt, 7, 3);
    ocrEdtTemplateCreate(&endTempGuid, &endEdt, 1, 1);
    u32 i;
    double *x;

    ocrGuid_t dataGuid;
```

```
    ocrDbCreate(&dataGuid, (void **) &x, sizeof(double) * N * 3, DB_PROP_NONE, NULL_GUID,
        NO_ALLOC);

    // Cook up some arbitrary data
    for(i=0;i<N;i++) {
        x[i] = 0;
    }
    x[0] = 1;

    u64 edtParamv[7] = { computeTempGuid, twiddleTempGuid, N, 1, 0, 0, BLOCK_SIZE };
    ocrGuid_t edtGuid, eventGuid, endGuid;

    // Launch compute EDT
    ocrEdtCreate(&edtGuid, computeTempGuid, EDT_PARAM_DEF, edtParamv,
                 EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_GUID,
                 &eventGuid);

    // Launch finish EDT
    ocrEdtCreate(&endGuid, endTempGuid, EDT_PARAM_DEF, &dataGuid,
                 EDT_PARAM_DEF, NULL, EDT_PROP_FINISH, NULL_GUID,
                 NULL);

    ocrAddDependence(dataGuid, edtGuid, 0, DB_MODE_RW);
    ocrAddDependence(eventGuid, endGuid, 0, DB_MODE_RW);

    return NULL_GUID;
}
```

A.4.1.1. Details

The above code contains a total of 5 functions - a `mainEdt()` required of all OCR programs, a `ditfft2()` that acts as the core of the recursive FFT computation, calling itself on smaller sizes of the array provided to it, and three other EDTs that are managed by OCR. They include - `fftComputeEdt()` in Line 67 that breaks down the FFT operation on an array into two FFT operations on the two halves of the array (by spawning two other EDTs of the same template), as well as an instance of `fftTwiddleEdt()` shown in Line 111 that combines the results from the two spawned EDTs by applying the FFT "twiddle" operation on the real and imaginary portions of the array. The `fftComputeEdt()` function stops spawning EDTs once the size of the array it operates on drops below a pre-defined `BLOCK_SIZE` value. This sets up a recursive cascade of EDTs operating on gradually smaller data sizes till the `BLOCK_SIZE` value is reached, at which point the FFT value is directly computed, followed by a series of twiddle operations on gradualy larger data sizes till the entire array has undergone the operation. When this is available, a final EDT termed `endEdt()` in Line 125 is called to optionally output the value of the computed FFT, and terminate the program by calling `ocrShutdown()`. All the FFT operations are performed on a single datablock created in Line 143. This shortcut is taken for the sake of didactic simplicity. While this is programmatically correct, a user who desires reducing contention on the single array may want to break down the datablock into smaller units for each of the EDTs to operate upon.

For this program to execute correctly, it is apparent that each of the `fftTwiddleEdt` instances can not start until all its previous instances have completed execution. Further, for the sake of program simplicity, an instance of `fftComputeEdt-fftTwiddleEdt` pair cannot return until

the EDTs that they spawn have completed execution. The above dependences are enforced using the concept of *Finish EDTs*. As stated before, a Finish EDT does not return until all the EDTs spawned by it have completed execution. This simplifies programming, and does not consume computing resources since a Finish EDT that is not running, is removed from any computing resources it has used. In this program, no instance of `fftComputeEdt` or `fftTwiddleEdt` returns before the corresponding EDTs that operates on smaller data sizes have returned, as illustrated in Lines 94,97 and 101. Finally, the single `endEdt()` instance in Line 157 is called only after all the EDTs spawned by the parent `fftComputeEdt()` in Line 162, return.

A.5. Accessing a DataBlock with "Intent-To-Write" Mode

This example illustrates the usage model for datablocks accessed with the `Intent-To-Write (RW)` mode. The `RW` mode ensures that only one master copy of the datablock exists at any time inside a shared address space. Parallel EDTs can concurrently access a datablock under this mode if they execute inside the same address space. It is the programmer's responsibility to avoid data races. For example, two parallel EDTs can concurrently update separate memory regions of the same datablock with the `RW` mode.

A.5.1. Code example

```
/* Example usage of RW (Intent-To-Write)
 * datablock access mode in OCR
 *
 * Implements the following dependence graph:
 *
 *      mainEdt
 *      [ DB ]
 *       /  \
 * (RW)/     \(RW)
 *    /       \
 * EDT1       EDT2
 *    \       /
 *      [ DB ]
 *   shutdownEdt
 *
 */

#include "ocr.h"

#define N 1000

ocrGuid_t exampleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 i, lb, ub;
    lb = paramv[0];
    ub = paramv[1];
    u32 *dbPtr = (u32*)depv[0].ptr;

    for (i = lb; i < ub; i++)
        dbPtr[i] += i;

    return NULL_GUID;
```

```
}

ocrGuid_t awaitingEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u64 i;
    PRINTF("Done!\n");
    u32 *dbPtr = (u32*)depv[0].ptr;
    for (i = 0; i < N; i++) {
        if (dbPtr[i] != i * 2)
            break;
    }

    if (i == N) {
        PRINTF("Passed Verification\n");
    } else {
        PRINTF("!!! FAILED !!! Verification\n");
    }

    ocrDbDestroy(depv[0].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u32 i;

    // CHECKER DB
    u32* ptr;
    ocrGuid_t dbGuid;
    ocrDbCreate(&dbGuid, (void**)&ptr, N * sizeof(u32), DB_PROP_NONE, NULL_GUID, NO_ALLOC);
    for(i = 0; i < N; i++)
        ptr[i] = i;
    ocrDbRelease(dbGuid);

    // EDT Template
    ocrGuid_t exampleTemplGuid, exampleEdtGuid1, exampleEdtGuid2, exampleEventGuid1,
        exampleEventGuid2;
    ocrEdtTemplateCreate(&exampleTemplGuid, exampleEdt, 2 /*paramc*/, 1 /*depc*/);
    u64 args[2];

    // EDT1
    args[0] = 0;
    args[1] = N/2;
    ocrEdtCreate(&exampleEdtGuid1, exampleTemplGuid, EDT_PARAM_DEF, args, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &exampleEventGuid1);

    // EDT2
    args[0] = N/2;
    args[1] = N;
    ocrEdtCreate(&exampleEdtGuid2, exampleTemplGuid, EDT_PARAM_DEF, args, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &exampleEventGuid2);

    // AWAIT EDT
    ocrGuid_t awaitingTemplGuid, awaitingEdtGuid;
    ocrEdtTemplateCreate(&awaitingTemplGuid, awaitingEdt, 0 /*paramc*/, 3 /*depc*/);
    ocrEdtCreate(&awaitingEdtGuid, awaitingTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
        NULL,
        EDT_PROP_NONE, NULL_GUID, NULL);
    ocrAddDependence(dbGuid,              awaitingEdtGuid, 0, DB_MODE_CONST);
    ocrAddDependence(exampleEventGuid1, awaitingEdtGuid, 1, DB_DEFAULT_MODE);
    ocrAddDependence(exampleEventGuid2, awaitingEdtGuid, 2, DB_DEFAULT_MODE);

    // START
    PRINTF("Start!\n");
    ocrAddDependence(dbGuid, exampleEdtGuid1, 0, DB_MODE_RW);
    ocrAddDependence(dbGuid, exampleEdtGuid2, 0, DB_MODE_RW);
```

```
    return NULL_GUID;
}
```

A.5.1.1. Details

The mainEdt creates a datablock (`dbGuid`) that may be concurrently updated by two children EDTs (`exampleEdtGuid1` and `exampleEdtGuid2`) using the `RW` mode. `exampleEdtGuid1` and `exampleEdtGuid2` are each created with one dependence on each of them, while after execution, each of them will satisfy an output event (`exampleEventGuid1` and `exampleEventGuid2`). The satisfaction of these output events will trigger the execution of an awaiting EDT (`awaitingEdtGuid`) that will verify the correctness of the computation performed by the concurrent EDTs. `awaitingEdtGuid` has three input dependences. `dbGuid` is passed into the first input, while the other two would be satisfied by `exampleEventGuid1` and `exampleEventGuid2`. Once `awaitingEdtGuid`'s dependences have been setup, the `mainEdt` satisfies the dependences on `exampleEdtGuid1` and `exampleEdtGuid2` with the datablock `dbGuid`.

Both `exampleEdtGuid1` and `exampleEdtGuid2` execute the task function called *exampleEdt*. This function accesses the contents of the datablock passed in through the dependence slot `0`. Based on the parameters passed in, the function updates a range of values on that datablock. After the datablock has been updated, the EDT returns and in turn satisfies the output event. Once both EDTs have executed and satisfied their ouput events, the `awaitingEdtGuid` executes function *awaitingEdt*. This function verifies if the updates done on the datablock by the concurrent EDTs are correct. Finally, it prints the result of its verification and calls `ocrShutdown`.

A.6. Accessing a DataBlock with "Exclusive-Write" Mode

The `Exclusive-Write (EW)` mode allows for an easy implementation of mutual exclusion of EDTs execution. When an EDT depend on one or several DBs in `EW` mode, the runtime guarantees it is the only EDT throught the system to currently writing to those DBs. Hence, the `EW` mode is useful when one wants to guarantee there's no race condition writing to a Data-Block or when ordering among EDTs do not matter for as long as the execution is in mutual exclusion. The following examples shows how two EDTs may share access to a DB in `RW` mode, while one EDT requires `EW` access. In this situation the programmer cannot assume in which order the EDTs are executed. It might be that EDT1 and EDT2 are executed simultaneously or independently, while EDT3 happens either before, after or in between the others.

A.6.1. Code example

```
/* Example usage of EW (Exclusive-Write)
 * datablock access mode in OCR
 *
 * Implements the following dependence graph:
 *
 *         mainEdt
 *         [ DB ]
 *        / |    \
 * (RW)/  |(RW)  \(EW)
 *    /   |       \
 * EDT1  EDT2     EDT3
 *    \   |       /
 *     \  |      /
 *      \ |     /
 *        [ DB ]
 *     shutdownEdt
 *
 */

#include "ocr.h"

#define NB_ELEM_DB 20

ocrGuid_t shutdownEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    // The fourth slot (3 as it is 0-indexed) was the DB.
    u64 * data = (u64 *) depv[3].ptr;
    u32 i = 0;
    while (i < NB_ELEM_DB) {
        PRINTF("%d ",data[i]);
        i++;
    }
    PRINTF("\n");
    // Destroying the DB implicitely releases it.
    ocrDbDestroy(depv[3].guid);
    // Instruct the runtime the application is done executing
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t writerEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    // An EDT has access to the parameters and dependences it has been created with.
    // ocrEdtDep_t allow to access both the '.guid' of the dependence and the '.ptr'
    // Note that when an EDT has an event as a dependence and this event is satisfied
    // with a DB GUID, the .guid field contains the DB GUID, not the event GUID.
    u64 * data = (u64 *) depv[0].ptr;
    u64 lb = paramv[0];
    u64 ub = paramv[1];
    u64 value = paramv[2];
    u32 i = lb;
    while (i < ub) {
        data[i] += value;
        i++;
    }
    // The GUID the output event of this EDT is satisfied with
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    void * dbPtr;
    ocrGuid_t dbGuid;
    u32 nbElem = NB_ELEM_DB;
    // Create a DataBlock (DB). Note that the currently executing EDT
    // acquires the DB in Intent-To-Write (RW) mode
    ocrDbCreate(&dbGuid, &dbPtr, sizeof(u64) * NB_ELEM_DB, 0, NULL_GUID, NO_ALLOC);
```

```
    u64 i = 0;
    int * data = (int *) dbPtr;
    while (i < nbElem) {
        data[i] = 0;
        i++;
    }
    // Indicate to the runtime the current EDT is not using the DB anymore
    ocrDbRelease(dbGuid);

    // Create the sink EDT template. The sink EDT is responsible for
    // shutting down the application.
    // It has 4 dependences: EDT1, EDT2, EDT3 and the DB
    ocrGuid_t shutdownEdtTemplateGuid;
    ocrEdtTemplateCreate(&shutdownEdtTemplateGuid, shutdownEdt, 0, 4);
    ocrGuid_t shutdownGuid;
    // Create the shutdown EDT indicating this instance of the shutdown EDT template
    // has the same number of parameters and dependences the template declares.
    ocrEdtCreate(&shutdownGuid, shutdownEdtTemplateGuid, 0, NULL, EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_GUID, NULL);
    // EDT is created, but it does not have its four dependences set up yet.
    // Set the third dependence of EDT shutdown to be the DB
    ocrAddDependence(dbGuid, shutdownGuid, 3, DB_MODE_CONST);

    // Writer EDTs have 3 parameters and 2 dependences
    ocrGuid_t writeEdtTemplateGuid;
    ocrEdtTemplateCreate(&writeEdtTemplateGuid, writerEdt, 3, 2);

    // Create the event that enable EDT1, EDT2, EDT3 to run
    // It is a once event automatically destroyed when satisfy
    // has been called on it. Because of that we need to make
    // sure that all its dependences are set up before satisfied
    // is called.
    ocrGuid_t eventStartGuid;
    ocrEventCreate(&eventStartGuid, OCR_EVENT_ONCE_T, false);

    // RW '1' from 0 to N/2 (potentially concurrent with writer 1, but different range)
    ocrGuid_t oeWriter1Guid;
    ocrGuid_t writer1Guid;
    // parameters composed of lower bound, upper bound, value to write.
    // parameters are passed by copy to the EDT, so it's ok to use the stack here.
    u64 writerParamv1[3] = {0, NB_ELEM_DB/2, 1};
    // Create the EDT1. Note the output event parameter 'oeWriter1Guid'.
    // The output event is satisfied automatically by the runtime when EDT1
    // is done executing. This event is by default a ONCE event. The user must
    // make sure dependences on that event are set up before the EDT is scheduled.
    // This is the main reason why we have a start event. It allows to create all the
    // EDT before-hand and set up all the dependences before any scheduling occurs.
    ocrEdtCreate(&writer1Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv1,
         EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_GUID, &oeWriter1Guid);
    // Set up the sink EDT dependence slot 0 (2 dependences added so far)
    ocrAddDependence(oeWriter1Guid, shutdownGuid, 0, false);
    // EDT1 depends on the DB in RW mode on its slot '0'
    ocrAddDependence(dbGuid, writer1Guid, 0, DB_MODE_RW);
    // EDT1 depends on the start event on its slot '1'
    ocrAddDependence(eventStartGuid, writer1Guid, 1, DB_MODE_CONST);
    // At this poomt EDT

    // RW '2' from N/2 to N (potentially concurrent with writer 0, but different range)
    ocrGuid_t oeWriter2Guid;
    ocrGuid_t writer2Guid;
    u64 writerParamv2[3] = {NB_ELEM_DB/2, NB_ELEM_DB, 2};
    ocrEdtCreate(&writer2Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv2,
         EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_GUID, &oeWriter2Guid);
```

```
    // Set up the sink EDT dependence slot 1 (3 dependences added so far)
    ocrAddDependence(oeWriter2Guid, shutdownGuid, 1, false);
    ocrAddDependence(dbGuid, writer2Guid, 0, DB_MODE_RW);
    ocrAddDependence(eventStartGuid, writer2Guid, 1, DB_MODE_CONST);

    // EW '3' from N/4 to 3N/4
    ocrGuid_t oeWriter3Guid;
    ocrGuid_t writer3Guid;
    u64 writerParamv3[3] = {NB_ELEM_DB/4, (NB_ELEM_DB/4)*3, 3};
    ocrEdtCreate(&writer3Guid, writeEdtTemplateGuid, EDT_PARAM_DEF, writerParamv3,
        EDT_PARAM_DEF, NULL,
                 EDT_PROP_NONE, NULL_GUID, &oeWriter3Guid);
    // Set up the sink EDT dependence slot 2.
    // At this point the shutdown EDT has all its dependences
    // and will be eligible for scheduling when they are satisfied
    ocrAddDependence(oeWriter3Guid, shutdownGuid, 2, false);
    // EDT3 request the DB in Exclusive-Write (EW) mode. This is essentially
    // introducing an implicit ordering dependence between all other EDTs
    // that are also acquiring this DB. The actual EDTs execution ordering
    // is schedule dependent.
    ocrAddDependence(dbGuid, writer3Guid, 0, DB_MODE_EW);
    ocrAddDependence(eventStartGuid, writer3Guid, 1, DB_MODE_CONST);

    // At this point all writers EDTs have their DB dependence satisfied and
    // are only missing the start event to be satisfied.
    // Doing so enable EDT1, EDT2, EDT3 to be eligible for scheduling.
    // Because there's no control dependence among the writer EDT, the
    // runtime is free to schedule them in any order, potentially in parallel.
    // In this particular example, EDT1 and EDT2 can execute in parallel because
    // they both access the DB in RW mode which allow for concurrent writers.
    // EDT3 will be executed in mutual exclusion with EDT1 and EDT2 at any point in time
    // since it requires EW access.
    ocrEventSatisfy(eventStartGuid, NULL_GUID);

    return NULL_GUID;
}
```

A.6.1.1. Details

A.7. Acquiring contents of a DataBlock as a dependence input

This example illustrates the usage model for accessing the contents of a datablock. The data contents of a datablock are made available to the EDT through the input slots in depv. The input slots contain two fields: the guid of the datablock and pointer to the contents of the datablock. The runtime process grabs a pointer to the contents through a process called “acquire”. The acquires of all datablocks accessed inside the EDT has to happen before the EDT starts execution. This implies that runtime requires knowledge of which datablocks it needs to acquire. That information is given to the runtime through the process of satisfaction of dependences. As a result, a datablock’s contents are available to the EDT only if that datablock has been passed in as the input on a dependence slot or if the datablock is created inside the EDT.

A.7.1. Code example

```
/* Example to show how DB guids can be passed through another DB.
 * Note: DB contents can be accessed by an EDT only when they arrive
 * in a dependence slot.
 *
 * Implements the following dependence graph:
 *
 *      mainEdt
 *      [ DB1 ]
 *         |
 *       EDT1
 *         |
 *      [ DB0 ]
 *    shutdownEdt
 *
 */

#include "ocr.h"

#define VAL 42

ocrGuid_t exampleEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    ocrGuid_t *dbPtr = (ocrGuid_t*)depv[0].ptr;
    ocrGuid_t passedDb = dbPtr[0];
    PRINTF("Passing DB: 0x%lx\n", passedDb);
    ocrDbDestroy(depv[0].guid);
    return passedDb;
}

ocrGuid_t awaitingEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    u32 *dbPtr = (u32*)depv[0].ptr;
    PRINTF("Received: %u\n", dbPtr[0]);
    ocrDbDestroy(depv[0].guid);
    ocrShutdown();
    return NULL_GUID;
}

ocrGuid_t mainEdt(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[]) {
    // Create DBs
    u32* ptr0;
    ocrGuid_t* ptr1;
    ocrGuid_t db0Guid, db1Guid;
    ocrDbCreate(&db0Guid, (void**)&ptr0, sizeof(u32), DB_PROP_NONE, NULL_GUID, NO_ALLOC);
    ocrDbCreate(&db1Guid, (void**)&ptr1, sizeof(ocrGuid_t), DB_PROP_NONE, NULL_GUID,
        NO_ALLOC);
    ptr0[0] = VAL;
    ptr1[0] = db0Guid;
    PRINTF("Sending: %u in DB: 0x%lx\n", ptr0[0], db0Guid);
    ocrDbRelease(db0Guid);
    ocrDbRelease(db1Guid);

    // Create Middle EDT
    ocrGuid_t exampleTemplGuid, exampleEdtGuid, exampleEventGuid;
    ocrEdtTemplateCreate(&exampleTemplGuid, exampleEdt, 0 /*paramc*/, 1 /*depc*/);
    ocrEdtCreate(&exampleEdtGuid, exampleTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF, NULL,
        EDT_PROP_NONE, NULL_GUID, &exampleEventGuid);

    // Create AWAIT EDT
    ocrGuid_t awaitingTemplGuid, awaitingEdtGuid;
    ocrEdtTemplateCreate(&awaitingTemplGuid, awaitingEdt, 0 /*paramc*/, 1 /*depc*/);
    ocrEdtCreate(&awaitingEdtGuid, awaitingTemplGuid, EDT_PARAM_DEF, NULL, EDT_PARAM_DEF,
        NULL,
        EDT_PROP_NONE, NULL_GUID, NULL);
```

```
    ocrAddDependence(exampleEventGuid, awaitingEdtGuid, 0, DB_DEFAULT_MODE);

63  // START Middle EDT
    ocrAddDependence(db1Guid, exampleEdtGuid, 0, DB_DEFAULT_MODE);

    return NULL_GUID;
}
```

A.7.1.1. Details

The mainEdt creates two datablocks (`db0Guid` and `db1Guid`). Then it sets the content of `db0Guid` to be an user-define value, while the content of `db1Guid` is set to be the guid value of `db0Guid`. Then the runtime creates an EDT (`exampleEdtGuid`) that takes one input dependence. It creates another EDT (`awaitingEdtGuid`) and makes it dependent on the satisfaction of the `exampleEdtGuid`'s output event (`exampleEventGuid`). Finally, mainEdt satisfies the dependence of `exampleEdtGuid` with the datablock `db1Guid`.

Once `exampleEdtGuid` starts executing function "exampleEdt", the contents of `db1Guid` are read. The function then retrieves the guid of datablock `db0Guid` from the contents of `db1Guid`. Now in order to read the contents of `db0Guid`, the function satisfies the output event with `db0Guid`.

Inside the final EDT function "awaitingEdt", the contents of `db0Guid` can be read. The function prints the content read from the datablock and finally calls "ocrShutdown".

B. OCR API Extensions

This chapter will contain documentation for the extensions to OCR's core APIs. For the moment, refer to the header files located in `inc/extensions` for more information.

C. Implementation Notes

This appendix contains details on the ways in which the various OCR reference implementations differ from the specification. This section will keep evolving as the implementations become more compliant.

C.1. General notes

This section describes the limitations common to all OCR implementations.

Data block access modes The data block access modes are only supported using the `l`ockableDB implementation of data blocks. The `r`egularDB implementation ignores the modes. Furthermore, the read-only mode has not been fully tested.

D. OCR Change History

September 2014 Release of OCR 0.9 including the first version of this specification.

April 2015 Release of OCR 0.95. Fixed some typos in the spec and cleaned up some subtle flaws in the memory model.

June 2015 Release of OCR 1.0.0. Restructured the specification for clarity and updated the API to make the names of the memory modes more intuitive. Also moved the API documentation from doxygen to human-readable TEX with proper specification language.