The Journal of Supercomputing (2019) 75:2725–2760
https://doi.org/10.1007/s11227-018-2681-2

Consistency model for runtime objects in the Open Community Runtime

Jiri Dokulil[1]

Published online: 14 November 2018

Abstract

Task-based runtime systems are seen as a way to deal with increasing heterogeneity and performance variability in modern high performance computing systems. The Open Community Runtime is an example of such runtime system, with the benefit of having an open specification that defines its interface and behavior. This allows others to study the design and create alternative implementations of the runtime system. However, it turns out that the consistency model that defines how the OCR programs are synchronized when executed in parallel is not sufficiently defined. In the following text, we complete the consistency model by filling in the missing pieces and extending the memory model given in the specification.

Keywords Open Community Runtime · Consistency models · Causality

1 Introduction

The Open Community Runtime (OCR) is a task-based model for parallel and distributed computing [16]. The OCR working group maintains a specification [15], which describes the OCR application programming interface (OCR API) and the expected behavior of a runtime system that implements the OCR API. Among other things, it provides a memory model, which describes how data are handled and how are changes to the data are propagated—it provides a consistency model. The model is based on the *happens-before* relationship, which describes which operations (made by the user code within tasks) are guaranteed to happen in a particular order. Simply put, if one operation modifies data, the data are read by another operation, and the first operation *happens-before* the second operation, the second operation is guaranteed to see the changes made by the first operation. Having such model is essential for

✉ Jiri Dokulil
jiri.dokulil@univie.ac.at

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria

programmers' understanding of the code, but it can also be used in tools that detect errors such as data races [20,22].

However, the memory model only deals with user's data, which is stored in data blocks. It does not cover changes made to other OCR objects (like tasks or events) using OCR API. It would be possible to assume that full effect of each OCR API call is performed before the function call returns to the calling code (the user's code of a task), but this is too restrictive. The overall idea of the OCR is to allow as much asynchronicity as possible. It would be beneficial to be able to return from the API call immediately, while the effect of the call is still being evaluated by the runtime system. This may improve execution efficiency, but it is also beneficial for resiliency, which is another design goal of the OCR. If applying the effect a task has on the state of OCR objects can be delayed to a later point (even after the task has ended executing), it makes it easier to isolate that task from the rest of the runtime state. This may simplify checkpointing or allow redundant task execution, where a task may be executed twice at different locations, but only changes made by one of the two task clones are propagated to the global runtime state.

The goal of the following text is to fill the gaps in the OCR specification, to give a clean definition of the way the state of all OCR objects (not just data blocks) changes. The proposal is designed to allow the *deferred execution model* described in the previous paragraph, providing the API user (the application developer) with clear guarantees, while giving the runtime system the option to handle the API calls asynchronously. The specification already expects some of the changes not to be immediate. For example, once all dependences of a task have been satisfied, the task is ready to run. But the actual start of the execution does not have to be immediate. When the API call that fulfills the last dependence of the task is made, its effect does include the execution of the task, but it is not expected to start executing right away. The similar is true for events connected via a dependence. If an event is satisfied, the connected events should be satisfied as well, but these indirect satisfactions are not expected to happen immediately. The specification does not clearly define these situations, so our proposal also provides the necessary clarification.

The proposed *object model*, which defines the way the state of OCR objects is affected by the OCR API calls, is based on the existing OCR memory model. It turns out that the memory model provides a very good basis for the object model. The object model fully reuses the *happens-before* relation used to define the memory model, making it a very natural and well-fitting extension, which does not change the fundamental ideas laid by the memory model. As a result, the existing implementation does not have to be changed to implement the proposed object model. The reference implementation created by Intel/Rice University [16], a derived implementation from PNNL [14], and our OCR-Vx implementation [5–7] all already conform to the proposed object model. In some cases, they may provide stronger guarantees than required, so the model gives them room for optimizations.

2 Related work

Originally, C and C++ programming languages did not have a memory model that would define their behavior in a multi-threaded environment. Parallel execution was possible with a single thread per process, with the behavior of the synchronization primitives defined outside the language, for example in POSIX standards. The memory models were finally introduced by C++11 and C11 standards. The memory model defines the semantics of computer memory storage. A simplified view is that the memory model prescribes under which conditions under which a change made by a write operation to an object stored in memory needs to be seen by some read operation. For example, if both operations are in the same thread and there is a sequence point between them, the change has to be seen. A more complex example is when the thread that performed the write operation then releases a mutex. If this mutex is then acquired by another thread and that thread then reads the value, the change must be visible. In C/C++ the objects that control the execution (threads, mutexes, etc.) are managed the same way as "normal" objects used to store the data used by the program. For example, to use a mutex created by another thread, the user must be synchronized the same way as if it wanted to read a value written by the first thread. Also, the mutex can only be destroyed in a way that guarantees that no one else may be using it, just like when destroying a data object. The threads and mutexes don't live in separate worlds, one belonging to the application and the other to some runtime system. Similarly to C/C++, the OCR memory model is based on a relation that defines which changes are visible at different places. However, synchronization (a key to defining visibility) in C/C++ is controlled by locking and atomic operations. OCR uses a completely different approach—synchronization is defined using events and task dependences. Also, events are special objects, managed differently from application data.

Intel Threading Building Blocks (TBB, [11]) is a task-based parallelization library. The computation is performed by tasks that are synchronized with dependences, similarly to OCR. However, TBB is based on the C++ memory model. The management of application data is left completely to the application code and the TBB objects (e.g., tasks) are also normal C++ objects. For example, every task has a counter that tracks the number of unsatisfied incoming dependences. This counter is decremented using an atomic operation (covered by the C++ memory model) when a dependence is satisfied. No special treatment is necessary.

MPI is an example of a distributed programming model with such special objects. Application data are stored in the memory of the process and managed through the usual C/C++ (or Fortran) means, but MPI objects like communicators follow a different set of rules. The state of a communicator is distributed among the processes that are members of the communicator and the individual processes only get a handle which they can use to refer to the communicator in MPI calls. Some MPI objects are local, like `MPI_Status` or `MPI_Request`. Less obvious examples are `MPI_Datatype` and `MPI_Group`, which are used for communication, but they are managed locally. These behave as normal "local" objects. The `MPI_Comm` and `MPI_File` objects are "global". Most operations that modify these are collective operations and need to be done within a communicator by all members. As all members of a communicator need to make all of the calls and make them in the same order, this clearly defines when their effect takes place—there is a clear "before" and "after" for each operation. For example, `MPI_Comm_split` creates new communicators from an existing one. Since both members of the old communicator and the new communicators must participate in the call and no other processes can use the new communicators, the point in time where the new communicators can be used is clearly defined. Note that MPI does not guarantee that collective operations are synchronized with point-to-point communication and there is also no guarantee about ordering of collective operations on different communicators. It is the user's responsibility to avoid race conditions in such cases. Still, the requirement to modify "global" objects via collective operations on clearly scoped communicators is a fairly elegant solution to the problem of defining the ordering of such operations. It is still possible to run into synchronization issues, but the clear model helps in isolating these [3,8]. The MPI approach is not applicable in OCR, as there are no communicators in OCR and no collective calls. OCR objects have global visibility and any object can be modified by an OCR API call made by any task.

Some models avoid the problem by not using such global objects. For example, in OpenSHMEM (a PGAS library) scope of collective operations is defined by providing start, stride, and size—three numerical parameters passed to the function calls (like `shmem_barrier` and `shmem_collect`). Clearly, this is not possible in OCR, as tasks and events need to be created (and destroyed) in order for any computation to happen.

UPC++ [23] is a very interesting example of an APGAS model, which extends C++ for distributed computing. It is based on earlier work on UPC—Unified Parallel C [4]. UPC++ allows tasks to be executed on remote nodes, using futures for synchronization in a way that is similar to the way events are used in OCR. A typical invocation scenario for an asynchronous (remote) operation is to make a call which initializes the operation and returns a future. The future becomes ready once the operation has completed. This constrains the ordering of the operation: It cannot start before the call is made and it must finish before the future is satisfied. If one operation writes data to memory and another reads it, the way to ensure that the change is visible is to make sure that the second operation cannot start before the future that corresponds to the first operation becomes ready. Furthermore, UPC++ allows the application to define teams, which correspond to derived MPI communicators, and invoke collective operations on those teams. The ordering of these operations is similar to MPI. Unlike MPI, UPC++ does not guarantee ordering of point-to-point communication between two ranks (if order needs to be preserved, it must be enforced by dependences via futures). As a result, the fact that there is also no implicit ordering between collective and point-to-point operations does not really add to the overall complexity, as it might in MPI. Another type of entity in UPC++ is distributed objects. These are objects with globally valid name, similar to the way OCR objects are identified with GUIDs. Distributed creation of these objects is possibly the aspect of UPC++ most relevant to the work presented in this text. However, despite the creation being a collective call (it is local in OCR), no guarantees are given that after the constructor is called on one rank the name is valid on other ranks. The user must ensure that using a barrier or other custom synchronization, which guarantees that the constructor is called on a rank before the name is used. It would not be sufficient to construct the name and immediately use it to invoke a task on another rank—the invocation could be processed before the recipient calls the constructs the name. Such usage is generally valid in OCR. The objective of our work is to formalize the conditions under which it is valid.

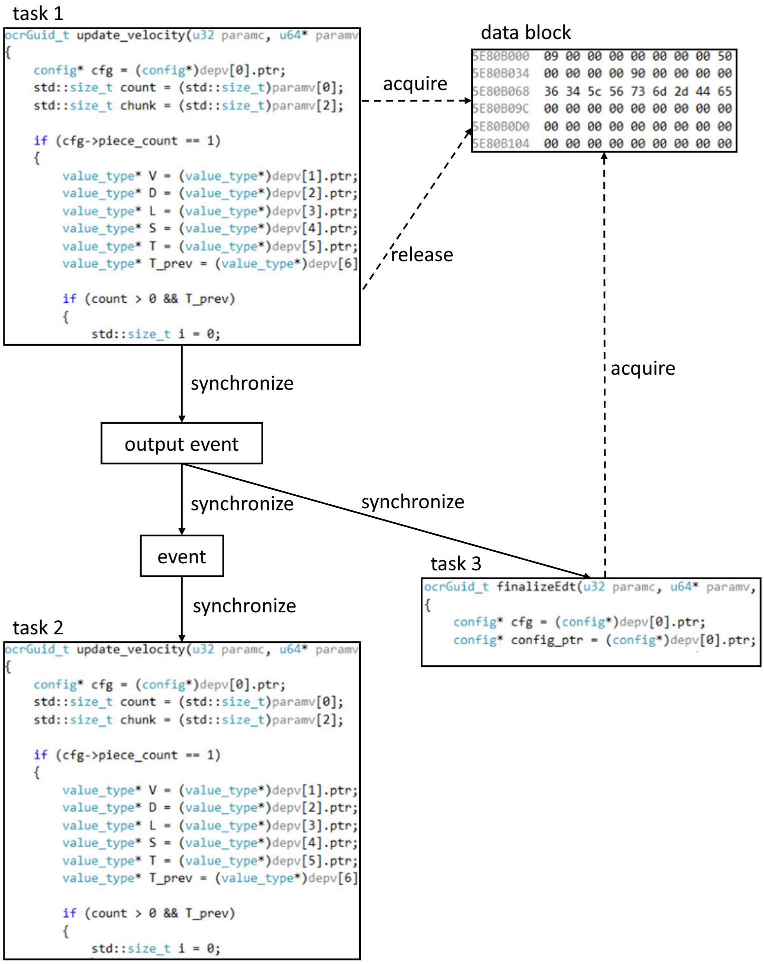
OpenCL standard includes a complex memory model, which prescribes the way different memory types are read and written [9,17]. One part of the model also describes how the host-side and device-side commands get synchronized. There are significant analogies between this and synchronization in OCR. The OpenCL queues contain commands (like kernel execution or memory transfers) and events, which are similar to OCR tasks and events. A *happens-before* relation is established among the commands, events, and API calls. It defines the order of operations and also controls visibility of memory operations, which is also similar to the way synchronization works in OCR.

Mixing different types of objects or the ways objects are accessed can be problematic. In MPI, care needs to be taken when combining collective communication with point-to-point. Similarly, combining atomic and non-atomic variables in C++ can lead to serious issues, even with the C++11 memory model [2,12]. In UPC++, the interaction of point-to-point communication, collectives, atomic operations, and global objects needs to be carefully managed. It is easy for an application developer to make incorrect assumptions about the way different types of operations interact, expecting some ordering to be enforced, while it is in fact not guaranteed. In OCR, there are also two types of objects, which are managed in two different ways. Data blocks, which contain the application data, need to be acquired by a task, before they are modified using C operations on pointers. The OCR objects are modified either by OCR API calls made inside tasks or implicitly by the runtime system. Our objective is to base both on the same foundations, reducing the cognitive burden placed on the programmer.

3 The Open Community Runtime and synchronization

To decouple work from execution units, all work in OCR is performed inside tasks. These tasks are scheduled by the OCR runtime system and they can be freely moved between execution units, including moving a task to a different node in a compute cluster. Giving such scheduling flexibility to the runtime system can improve execution efficiency, especially on heterogeneous systems [1,10,18].

To facilitate this task movement, all data need to be decoupled from storage. This is achieved by storing all application data in data blocks. A data block represents one contiguous piece of memory—a fixed-size sequence of bytes that can be used to store application data. A task can only access contents of a data block that it has acquired. There are only two ways to acquire a data block. The task may create a new data block or the data block is specified as a dependence of the task before it starts. This makes it more difficult for applications to manage their data but it makes the runtime system aware of all data a task may access, giving it much greater flexibility. The runtime system is allowed to move the data stored inside a data block around, including moving it to a different cluster node. The runtime system can make multiple copies of the data. It is even possible to provide different tasks with different (possibly old) versions of the data. Naturally, the runtime system needs to observe a set of rules which dictate when and how the different copies need to be synchronized. This is the OCR memory model.



```
ocrGuid_t update_velocity(u32 paramc, u64* paramv
{
    config* cfg = (config*)depv[0].ptr;
    std::size_t count = (std::size_t)paramv[0];
    std::size_t chunk = (std::size_t)paramv[2];

    if (cfg->piece_count == 1)
    {
        value_type* V = (value_type*)depv[1].ptr;
        value_type* D = (value_type*)depv[2].ptr;
        value_type* L = (value_type*)depv[3].ptr;
        value_type* S = (value_type*)depv[4].ptr;
        value_type* T = (value_type*)depv[5].ptr;
        value_type* T_prev = (value_type*)depv[6]

        if (count > 0 && T_prev)
        {
            std::size_t i = 0;
```

```
5E80B000  09 00 00 00 00 00 00 00 50
5E80B034  00 00 00 00 90 00 00 00 00
5E80B068  36 34 5c 56 73 6d 2d 44 65
5E80B09C  00 00 00 00 00 00 00 00 00
5E80B0D0  00 00 00 00 00 00 00 00 00
5E80B104  00 00 00 00 00 00 00 00 00
```

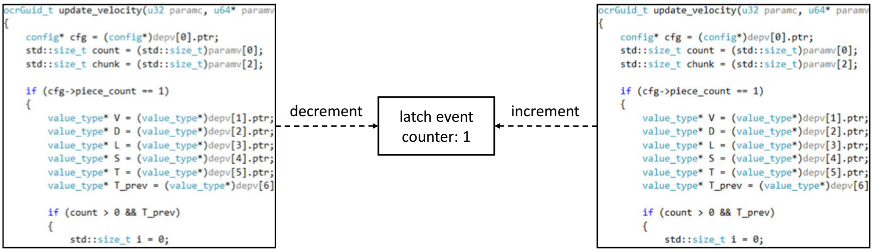
```
ocrGuid_t finalizeEdt(u32 paramc, u64* paramv,
{
    config* cfg = (config*)depv[0].ptr;
    config* config_ptr = (config*)depv[0].ptr;
```

```
ocrGuid_t update_velocity(u32 paramc, u64* paramv
{
    config* cfg = (config*)depv[0].ptr;
    std::size_t count = (std::size_t)paramv[0];
    std::size_t chunk = (std::size_t)paramv[2];

    if (cfg->piece_count == 1)
    {
        value_type* V = (value_type*)depv[1].ptr;
        value_type* D = (value_type*)depv[2].ptr;
        value_type* L = (value_type*)depv[3].ptr;
        value_type* S = (value_type*)depv[4].ptr;
        value_type* T = (value_type*)depv[5].ptr;
        value_type* T_prev = (value_type*)depv[6]

        if (count > 0 && T_prev)
        {
            std::size_t i = 0;
```

Fig. 1 An example of OCR tasks, data blocks, and synchronization

There is only one way to synchronize execution of OCR tasks. It is possible to specify task dependences to build a directed acyclic graph (a DAG) of tasks, which the runtime systems follow when making scheduling decisions. Events are lightweight objects that can be used in the DAG along with tasks, to specify more complex dependences. *Output events* are a special kind of events. Every task has an associated output event, which signals that the task has finished executing. Anything that depends on that output event cannot start before the task finishes. Figure 1 shows an example with three tasks, two events (one output, one normal), and a single data block. The dependences ensure that the two tasks at the bottom (tasks 2 and 3) of the figure cannot start before task 1 finishes.

In the example, two different tasks acquire the same data block, obtaining access to its data. As we can see, the second task (task 3) is synchronized to run after the first



```
ocrGuid_t update_velocity(u32 paramc, u64* paramv
{
    config* cfg = (config*)depv[0].ptr;
    std::size_t count = (std::size_t)paramv[0];
    std::size_t chunk = (std::size_t)paramv[2];

    if (cfg->piece_count == 1)
    {
        value_type* V = (value_type*)depv[1].ptr;
        value_type* D = (value_type*)depv[2].ptr;
        value_type* L = (value_type*)depv[3].ptr;
        value_type* S = (value_type*)depv[4].ptr;
        value_type* T = (value_type*)depv[5].ptr;
        value_type* T_prev = (value_type*)depv[6]

        if (count > 0 && T_prev)
        {
            std::size_t i = 0;
```

```
ocrGuid_t update_velocity(u32 paramc, u64* paramv
{
    config* cfg = (config*)depv[0].ptr;
    std::size_t count = (std::size_t)paramv[0];
    std::size_t chunk = (std::size_t)paramv[2];

    if (cfg->piece_count == 1)
    {
        value_type* V = (value_type*)depv[1].ptr;
        value_type* D = (value_type*)depv[2].ptr;
        value_type* L = (value_type*)depv[3].ptr;
        value_type* S = (value_type*)depv[4].ptr;
        value_type* T = (value_type*)depv[5].ptr;
        value_type* T_prev = (value_type*)depv[6]

        if (count > 0 && T_prev)
        {
            std::size_t i = 0;
```

Fig. 2 An example of a latch event modified from two tasks

task (task 1). Because of this, the OCR memory model guarantees that the second task sees all changes made by the first task. If there was no synchronization between the tasks, they could still both access the same data block, but there would be no guarantee that one task would see the changes made by the other task. This is the intention of the OCR specification. Task dependences not only ensure that tasks are executed in a certain order, but they also govern the visibility of updates to the data.

While the way the data stored in data blocks is updated is clearly defined by the memory model, the way the OCR objects are changed is not defined in detail. However, all OCR objects contain some state information or (meta)data. This may be read-only, like the size of a data block, but it may also be mutable. For example, the state of each OCR object can be changed by destroying the object.

Latch events are a good example of why these updates are important. A latch event is a special kind of event, used to define more advanced synchronization patterns. A latch event contains a counter. This counter may be incremented or decremented, either explicitly from a task or by a dependence. The latter case can be for example be used to automatically decrement the counter after a task finishes. The latch event is used to wait for the counter to reach zero.

In Fig. 2, two tasks explicitly update the counter. As the counter starts at 1 and one operation is increment and the other is decrement, their order is clearly important. If the counter is decremented first, it reaches zero, allowing all tasks that depend on it to start. If it is incremented and then decremented, it moves from 1 to 2 and then back, preventing the dependent tasks from starting. The existing OCR specification does not specify what should happen in such case.

First issue to deal with is the atomicity of the update. Even though not explicitly stated, it was clearly the intention of the authors of the specification to make the update an atomic operation. Even if the counter is updated from multiple tasks concurrently, the final value should be equivalent to some sequential ordering of the changes. It should not be possible for an update to interrupt another update, producing an inconsistent result.

Clearly, this does not solve the problem presented in Fig. 2. We also need to define a way that the application can use to specify the order in which multiple changes are applied to an object (e.g., the latch event). We call this the *object model*. It turns out that the foundations laid by the memory model can also be used for the object model. The basic idea is to use the synchronization established via events to also order changes made to OCR objects.

It may be tempting to define the object model by requiring all changes that tasks made to OCR objects to be immediately evaluated in an atomic way. Since tasks can only start (and make changes to OCR objects) after all their dependences have been satisfied, this would ensure that changes to OCR objects follow the dependences. However, not all changes to OCR objects are invoked directly by tasks. Some are performed implicitly by the runtime system. Still, this simple object model could be extended to also cover these cases.

There is a reason to use a more complex object model. Since the OCR targets distributed systems, it is possible that the data and metadata of the OCR objects are not available on the node that is executing the task that changes it. Requiring the change to be fully processed immediately would entail blocking the tasks until the change can be completed. For performance reasons, we want a model that allows the change to be applied asynchronously while the tasks continue to work, hiding the communication latency. Still, we would like to make this more complex model to be mostly compatible with the view that all changes are performed immediately. It turns out that this is possible, if we build our system around dependences in a way that is similar to the way the memory model handles data block changes.

Returning to the example shown in Fig. 2, to obtain a correct result, the application would need to ensure that dependences are set up to ensure that the two tasks run in a certain order. In this particular example, it only makes sense to make the task that performs the decrement depend on the task that increments the counter. The object model would then ensure that the counter is first incremented to 2 and then decremented back to 1. We allow the runtime system to make the actual updates in a deferred way, as long as the proper ordering is maintained. The value can actually be changed even after both tasks have finished. The OCR is designed in a way that prevents such delays from breaking the application. This is discussed in detail in Sect. 6, but the fundamental reason is that no task is allowed to wait for such a change to actually take place. It can be used in dependences but not as an explicit wait operation inside a task.

An important thing to note is that we also need to be careful about changes made by a single task. One task could increment and then decrement the counter of the latch. We need to make sure that even if the application of these changes is delayed, we still get the correct result. This is in fact a direct consequence of our requirement to apply changes in the order of their synchronization. Operations within a task are considered to be synchronized according their execution order (as per the rules of the C language).

4 The big picture

Before describing the way the OCR object model is built, we should first have a look at the big picture—see how the individual pieces fit together to define how OCR implementations and applications should behave.

First, there is the *happens-before* relation which defines how various actions performed by an OCR application are synchronized. Similarly to C/C++ memory model, if one action *happens-before* before another one, the second action should be able to see the changes made by the first one. In C/C++, synchronization is established through library calls (like working with mutexes) and atomic operations. In OCR, synchronization is established only by events and dependences. The relation is built on the actions that were actually executed by the application, not those in the source code. After the application terminates, we can look at all the actions and build the *happens-before* relation.

The OCR memory model uses the *happens-before* relation to define when a change (write operation) made to a data block must be visible to an operation that reads the data block. This means that the OCR runtime system must propagate these changes and ensure that the read operation accesses the correct modified version of the data block. The proposed object model, which will be described later in the text, restricts the way the state of OCR objects (like events) are updated in a similar way, based on the *happens-before* relation. Note that we are using synchronization established by events (the *happens-before* relation) to define how events behave. This looks like the *happens-before* relation is being used while it is being constructed.

This is not actually the case. In a distributed environment, it is not possible to exactly define what the relation looks like at a given point in time. What can be done is to look at a finished OCR application, build the relation and determine if the program execution was correct—whether the constraints imposed by the memory and object models were observed. The common goal of runtime system and application developers is ensuring that the answer is always “yes”. From the runtime system’s point of view, the goal is to ensure that any “correct OCR application” is executed in a way that adheres to the models. From this point of view, a correct OCR application can be roughly defined as an application that follows the OCR API specification and uses synchronization (events and dependences) to ensure proper ordering of actions that need to happen in a certain order. For example, it has to ensure that any object is created before it is used.

Creating such a runtime system in a shared-memory environment is fairly straightforward. Later in the text, we will show a way to also do that in a distributed-memory environment where messages are used to communicate between distributed processes.

5 The *sequenced-before, synchronized-with,* and *happens-before* relations

The OCR 1.2.0 specification provides a memory model for OCR programs, to define how tasks can access data in data blocks concurrently. The memory model is based on three relations. First, the order among operations within a single task is defined by the *sequenced-before* relationship. This is provided by the C language used to implement the tasks and it is the natural ordering of operations performed by a C program, as one would expect. Second relation is *synchronized-with*, which is defined by dependences among OCR events and tasks. The simplest example is a task whose output event is used as a dependence for another task. In this case, it is natural to expect that the second task comes after the first task. There are more complex examples of *synchronized-with*, which will be covered later. The third relation is *happens-before*, which is a transitive closure of combined *sequenced-before* and *synchronized-with*.

The OCR specification does not provide a full definition of *synchronized-with*, it only shows a simple case of two connected tasks. To properly define it, we need two things. First, we need to define the set that the relation is applied to (domain and range). Second, we need to define which pairs of objects from the set are in the relation.

The domain and range of *synchronized-with* are the OCR API calls made by the application. We also include *implicit operations* performed by the runtime system in response to certain situations. For example, if two tasks are connected by dependences formed by a chain of events, the events in the middle of the chain are satisfied automatically by the runtime system in response to the previous links in the chain being satisfied. The actual list is given in “Appendix C”.

To describe the *synchronized-with* relation, we need to clarify the behavior of tasks and events. All tasks and events have a certain number of *pre-slots*. These are the actual targets of dependences. A task with five pre-slots (the number is defined when the task is created) needs to be set as a target of exactly five dependences. Events have one or two pre-slots, depending on the type of event. A pre-slot is said to be satisfied either if it is a target of a dependence and the dependence itself is satisfied or if an explicit OCR API call is used to satisfy the pre-slot. The origin of all dependences is an event. A dependence is satisfied when the source event is triggered. An event is triggered when its triggering condition (defined by the event type) is satisfied. The satisfaction of event’s triggering condition always only considers satisfaction of the event’s pre-slots.

Note that we have just described the way satisfaction propagates through events. A satisfied incoming dependence satisfies an event’s pre-slot. In the case of basic events, this is enough to satisfy the event’s triggering condition. The event gets triggered and satisfies all outgoing dependences. Tasks are sinks in the signal propagation. Satisfying task’s pre-slot may allow the task to eventually start but it requires no further propagation of the satisfaction signal. Besides sinks, we also need sources. There are two kinds: output events, triggered by completion of the corresponding task, and explicit pre-slot satisfactions performed by OCR API calls inside tasks.

The following rules build the *synchronized-with* relation: A task cannot start before its dependences are defined and satisfied; event’s pre-slots are satisfied before it satisfies pre-slots of connected events and tasks; a pre-slot of an event or task that is a target of a dependence can only be satisfied after the dependence is defined. Once again, a detailed definition is given in “Appendix C”.

Going back to example in Fig. 1, we can see if *synchronized-with* is defined as expected. The output event of task 1 is satisfied by an implicit OCR API call made at the end of task 1. As per our rules, this satisfaction call is *synchronized-with* all satisfactions that happen as a result of triggering of the output event. It satisfies pre-slots of the non-output event and task 3. From transitivity, we now know that the satisfaction of the output event at the end of task 1 is *synchronized-with* satisfaction of the pre-slot of task 3, which in turn is *synchronized-with* the actual start of task 3 (the first rule). Similarly, we obtain that the end of task 1 is *synchronized-with* the beginning of task 2, because satisfaction of the pre-slot of the non-output event is *synchronized-with* satisfaction of the connected pre-slot of task 2.

By combining *sequenced-before* and *synchronized-with* relations, we know that all operations inside task 1 *happens-before* all operations in task 3. Therefore, the release of the data block in task 1 *happens-before* acquisition of the data block in task 3 and changes made to the contents of the data block in task 1 must be visible in task 3.

6 Deferred execution

For performance reasons and to support resiliency features of the OCR, it is beneficial to allow the runtime system to defer evaluation of the OCR API calls, while allowing the user's code of the task to keep running. The performance benefits are clear. Forcing the operation to fully complete before returning means that in a distributed environment the task may be blocked waiting for communication to complete. Deferring operations is beneficial to resiliency, because it allows the runtime system to execute tasks in a speculative way. If all operations are not just running on the background (as they might be for performance reasons), but they are completely suspended, the task can be run without affecting the overall state of the computation. This can be very useful when the runtime system is making a snapshot of the computation. This snapshot may be later used to restart a failed computation. Allowing speculative task execution might decrease the cost (performance degradation) of making the snapshot.

An example comparing normal and deferred execution is shown in Fig. 3. On the left, we can see that the thread needs to be paused while communication takes place. On the right, the operations are placed in a local queue, which allows them to be evaluated later. The task runs to completion uninterrupted. Multiple queues (for multiple tasks) can be multiplexed to the same worker thread, saving resources.

To determine whether it is possible to actually defer the evaluation of the OCR operations, we need to examine the different effects of OCR API calls. There are three basic options, which are described in the following paragraphs.

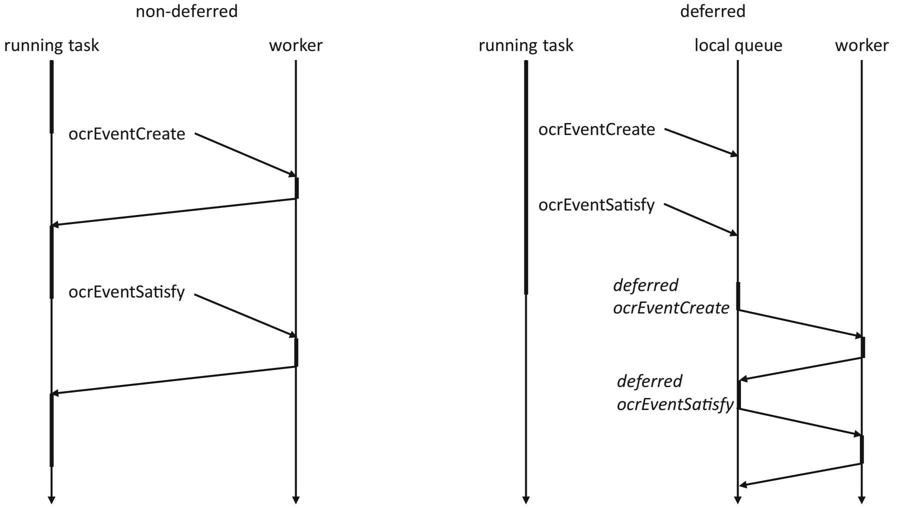


Fig. 3 Comparing normal and deferred execution of a task that makes two calls to the OCR API. The arrows correspond to threads and bold parts the intervals where the threads are active (not blocked)

The API calls change the internal state of OCR objects. All object types may be created or destroyed, events may be satisfied, tasks may have their dependences set, etc. The OCR API does not provide any way for the user's code to query the state of the OCR objects, nor can the code wait for an object's state to change. For example, the only way a task can find out that an event has been satisfied is to have the event connected to the task's pre-slot via a dependence. There is no API call that would allow a running task to find out the state of the event or to wait for the event to be satisfied. Therefore, it is possible to delay evaluating the API calls, even until after the task finishes. Naturally, the runtime system must take care to preserve the semantics of the API calls—the result produced in the presence of deferred execution should be equivalent to the result of non-deferred execution. Ensuring this is the objective of the proposed object model.

However, there is a second way in which a task can interact with objects outside of its own code and local data. A task may change the contents of data blocks and these data blocks may be read by other tasks. Clearly, it would not be a good idea to allow such changes to reach other tasks before the API calls are evaluated. If that were the case, a task could change a state of an OCR object via an API call and store the information that it has done so in a data block for other tasks to see. Another task would see that the change has been made, but if the API call gets deferred, its view would be inconsistent with the actual state of the system. Fortunately, to make changes to a data block visible to another task, the task that made the change has to release the data block. The data block is released either implicitly by the runtime system at the end of a task or via an API call. The runtime system can therefore also defer the release operation, preventing other tasks from seeing the updated information before the object state is changed. The OCR memory model is a good fit for the deferred API execution.

The final effect of an OCR API call is returning values to the calling code. There are three options. Error codes, handles of newly created objects, and pointer to a newly created data block. The error codes are not a major issue, since the specification already assumes that the runtime system may not be able to correctly identify all error conditions. If the evaluation is deferred, the call may immediately return an OK status. A special *nanny* mode should be provided by the OCR runtime systems, where the errors are checked strictly. As for the other two options, the runtime system can be designed in a way that allows it to immediately return a valid handle of an object as well as a valid pointer to a data block, without fully evaluating the OCR API calls. The actual creation of the object is deferred, but the runtime system must ensure that it is created with the same handle as the one returned to the user's code.

For a low-level example and further detailed discussion of deferred execution, see "Appendix D".

7 Object model

At this point, it is important to clarify how the concepts described in preceding sections (OCR memory model, the *happens-before* relation, and deferred execution) fit together. The extended *synchronized-with* relation defined in Sect. 5 and "Appendix C" serves to fill in a gap in the OCR specification, where the relation is not fully defined. With the extended *synchronized-with*, we can fully define the *happens-before* relation. This is then used by the OCR memory model to define when changes made to a data block by one task have to be seen by another task. The deferred execution model described in Sect. 6 is a way to delay evaluation of OCR API call, while maintaining correctness with respect to the OCR memory model.

The missing piece is defining a similar model for OCR objects. Clearly, it would be good to reuse the already defined *happens-before* relation. On the other hand, forcing the effects of API calls to be applied immediately would go against the deferred execution model. Our proposal is to use a solution similar to the memory model. Use the *happens-before* relation to define where a change made to OCR object's state needs to be seen by other actions that use that state. This way, if the change (write) is deferred, we can also defer the use (read), ensuring that they still happen in the right order, therefore maintaining correctness without having to block tasks while we wait for the changes to be processed.

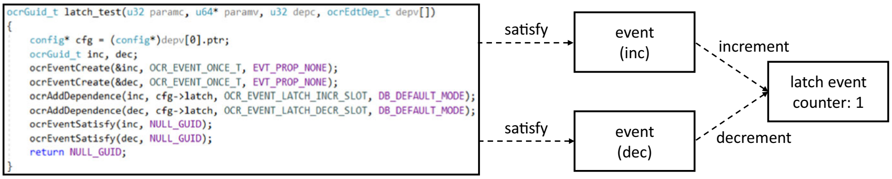
To define the object model, we need to split the effects of OCR API calls into two groups: *immediate* effects and *delayed* effects. The simplest explanation is that the delayed effects are effects that would not have to be evaluated immediately even if we used a simplified object model that tries to evaluate all OCR API calls right away. The most obvious example is starting a task as a result of its pre-slots getting satisfied. If the last pre-slot is satisfied by an OCR API call, no one would actually expect the task to be started as part of the satisfaction call. It is a delayed effect of the call. Maybe less obvious, but at least as important example is satisfaction of a pre-slot of the first event in a chain of events. The first event's pre-slot should be satisfied immediately, fulfilling the event's triggering condition. However, the actual triggering of the event is akin to starting of a task. It can be delayed, also delaying the triggering of the whole event chain.

With the groundwork already in place, the definition of object model itself is simple: The object model requires all effects to be evaluated atomically. Furthermore, for any two operations connected by *happens-before* relation, it requires that immediate effects of the first call are seen when immediate effects of the second call are evaluated and also when delayed effects of both calls are evaluated.

Notice that this allows evaluation to be deferred, as long as correct ordering is maintained. Also note that it is possible to evaluate delayed effects of the first operation after effects (both immediate and delayed) of the second operation. The object model only guarantees that a delayed effect is evaluated after (and therefore can see the results of) the immediate effects of the same operation but does not give any constraints about how much it can be delayed beyond that.

A formal definition is given in "Appendix E" along discussion of some low-level implications.

In most cases, the presented object model only provides a formal background for programmers' natural understanding of the way objects in OCR behave. It confirms the notion that two synchronized updates should happen in the specified order. However, there are cases where the exact behavior is not immediately obvious, where the formal model helps by providing clear definition of the expected behavior. Consider the example given in Fig. 4.



```
ocrGuid_t latch_test(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t depv[])
{
    config* cfg = (config*)depv[0].ptr;
    ocrGuid_t inc, dec;
    ocrEventCreate(&inc, OCR_EVENT_ONCE_T, EVT_PROP_NONE);
    ocrEventCreate(&dec, OCR_EVENT_ONCE_T, EVT_PROP_NONE);
    ocrAddDependence(inc, cfg->latch, OCR_EVENT_LATCH_INCR_SLOT, DB_DEFAULT_MODE);
    ocrAddDependence(dec, cfg->latch, OCR_EVENT_LATCH_DECR_SLOT, DB_DEFAULT_MODE);
    ocrEventSatisfy(inc, NULL_GUID);
    ocrEventSatisfy(dec, NULL_GUID);
    return NULL_GUID;
}
```

Fig. 4 An example of a latch event updated indirectly from a single task

If a task directly increments the counter of a latch event by satisfying the correct pre-slot and then decrements it the same way, it is natural to expect that the counter is first incremented and then decremented. However, in this example, two extra events are put between the task and the latch event. The task first satisfies the event connected to the increment pre-slot of the latch event and then the one connected to the decrement pre-slot. Satisfaction of the latch event's pre-slot is a delayed effect of satisfying the interposed event. Therefore, the object model places no restrictions on the relative order of the two delayed effects (increment pre-slot satisfaction and decrement pre-slot satisfaction) and the counter could be first decremented and then incremented. This may look inconvenient, but in a distributed environment it would be too costly to require chains of connected events to be fully evaluated as an immediate effect of the satisfaction of the first event in the chain.

8 Implementations of the object model

In shared-memory systems, implementation of the proposed model is straightforward. When an API call is made by a task, its immediate effects are evaluated as part of the API call, making the changes visible to all subsequent calls. This is possible, as we do not need to hide the latency of communication. Some care needs to be taken to either properly protect the changes with locks or use atomic operations, but this is also fairly easy. Therefore, well focus on distributed systems in the following text. We assume that the state of any object is maintained by one of the nodes in the system—the *owner* of the object. Interaction among nodes is facilitated by point-to-point messages. Changing a value of an object is performed by sending a message to its owner. Messages and delayed operations are managed by system workers, available on each node. These workers don't guarantee any ordering among the operations, except for cases specified later in this text. When an OCR object state is updated, the change is either atomic or properly protected by a lock. If a delayed operation causes further operations to be invoked, those are also delayed. An example of such situation is satisfaction a chain of events. Each "hop" in the chain may be delayed. Therefore, they are all processed by the system workers. Note that there may be OCR implementation that does not satisfy our constraints. For example, a state of an OCR object may not be maintained (owned) exclusively by a single node.

There are many ways a runtime system may implement the object model. We will provide several options in the following sections. A low-level discussion is provided in "Appendix F". In appendix, we also show a way to prove that these implementations actually ensure that the object model is maintained.

8.1 Blocking

As we have already mentioned, it is possible to block the calling task while an operation is being evaluated, avoiding the deferred execution model. With our object model, we can formulate a more precise definition for this option. It requires all immediate effects of an OCR API call to be fully evaluated before the call returns. The delayed effects can be evaluated by system workers at a later point in time.

The reference OCR implementation created by Intel and Rice University [16] uses this approach. All OCR API calls are translated to messages even if the operation can be processed locally. A message can be processed either in a blocking mode, where the sending task blocks until the message is processed, or in a non-blocking mode. The authors have decided which kind is appropriate by analyzing the individual cases and deciding whether a blocking call is required or not. This corresponds to our selection of immediate and delayed effects.

The downside of this solution is the fact that the tasks need to be blocked while waiting for the remote operation to finish. To compensate, the runtime system allows other tasks to execute on the same worker thread, while the original task is waiting. As a result, the original task may be suspended for longer than just the duration of the remote operation, but the overall utilization of the available compute units can be significantly improved.

This solution is in line with the OCR design philosophy of handing over the control of the execution to the runtime system. If there is always alternative work to do and the increased duration of the task does not have adverse effect on the task schedule, it can be very efficient. So far, the experience suggests that this works well for some codes, but it may also be problematic up to a point where switching of the feature altogether (forcing the whole thread to stall while waiting for the response) may significantly improve performance. Still, it is an interesting alternative to the deferred execution model.

8.2 Immediate confirmation protocol

To allow deferred execution, we can enqueue all operations to be executed later by a background worker thread. To satisfy the object model, the operations have to be handled by the worker in a certain way. First, they are executed in the order in which they are enqueued. Second, the worker must send out all messages that implement immediate effects of the operation and wait for all of them to be processed before moving on to delayed effects of the operation and then to the next operation. To check that a message has been processed, confirmation messages are used.

Initially, the distributed OCR-Vx implementation (OCR-Vdm) implementation did not use confirmation messages [7]. This was a source of race conditions. For example, if an object was created and the newly created handle was immediately used to satisfy an event that was at the beginning of a chain of events connected via dependences, it was possible for the event chain to be processed before the creation message, if the chain involved a third node (other than the node running the task and the node that owns the created object). A dependency mechanism was introduced to messages, preventing this kind of race condition, but it was not sufficient for all cases. For example, the indirect latch example in Fig. 4 could still be processed incorrectly.

To solve this, confirmation messages were introduced. If the OCR API call modified state of multiple objects (e.g., adding a dependence modifies the origin and destination of the dependence), the message could bounce around multiple nodes, making confirmation difficult. Later, this was changed to multiple smaller messages which only facilitated an update of a single object. At this point, it became apparent that a theoretical view of the problem is needed to justify the design, culminating in this work. The object model, where updates to each object are treated separately, validated the use of multiple simple messages with direct confirmation.

8.3 Further refinement

The immediate confirmation protocol can be further relaxed to improve efficiency while still providing all guarantees required by the object model. If multiple messages in a row are sent to the same remote node for processing, it is possible to only confirm the last one. This assumes that messages sent to one node cannot overtake each other and that they are processed in order in which they are received. Some designs might not provide such guarantees, but as they hold in OCR-Vdm, we already have a working experimental implementation of this optimization on a development branch.

As sending a long batch of messages to the same remote node is going to be rare, we might further improve the situation by reordering messages. In general, the object model does not allow that, so it is necessary to individually determine which pairs of messages can be reordered. For example, if the message implements event satisfaction and two messages target the same event, they obviously cannot be reordered. Even if they target different events, they still cannot be reordered. Delayed effect of satisfying the second event could then overtake immediate effect of satisfying the first event, producing incorrect behavior. However, adding two outgoing dependences to an event can be reordered, as the order of dependences does not matter. We could not, however, swap any of the messages with any message that can lead to satisfaction of the event.

A typical example where reordering may help significantly is the typical scenario for creating a new task. After the task is created, most of its incoming dependences are defined immediately. Defining such dependence entails two messages: one to the task and another to the source of the dependence. We can move all messages for the task forward, forming one larger group that can be sent in one batch and confirmed with a single confirmation message. The messages to the dependence sources can be grouped by owning nodes and send in groups as well.

9 Conclusion

The existing OCR specification does not sufficiently define the *synchronized-with* relation and the way synchronization is applied to the runtime objects. We have filled in these gaps and also provided some examples how the proposal can be implemented by a runtime system, to ensure correct synchronization. It turns out that the existing definition can be naturally extended from covering only data stored in data blocks to all runtime objects, without breaking established OCR practices (and programs). The implicit assumptions made in the specification and mostly adopted by application developers had to be made explicit and clarified.

The formal model can be used to reason about concurrency issues in OCR programs, possibly even allowing automatic checking tools to be deployed to find instances where the program violates the rules set by the OCR specification and the object model. For example, if an object is being destroyed, but its destruction is not properly synchronized to ensure that the destruction happens after all uses of the object.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF). The work was supported in part by the Austrian Science Fund (FWF) project P 29783 Dynamic Runtime System for Future Parallel Architectures.

Appendix A: Key OCR concepts

To explain our work, we need to use a large number of OCR-specific concepts. These are briefly introduced in the text and the full specification is available online [15], but for convenience we provide the following alphabetical list of the important concepts:

GUID A globally unique identifier. A GUID is an opaque identifier which serves as a handle that the OCR application uses to reference OCR objects in OCR API calls. The application gets a GUID when an object is created inside a task and it can be used in subsequent API calls made by the task or stored in a data block to allow other tasks to use it.

Pre-slot Once an OCR task starts, it should run to completion without waiting for other tasks. Following this philosophy, the OCR does not provide a way to synchronize a task with other tasks once it has started. The only way to synchronize a task is to provide a set of conditions that need to be fulfilled before the task can start. Each task has a fixed number of pre-slots (the number is defined when the task is created) and the task cannot start before all of them have been *satisfied*. The way to satisfy a pre-slot is to connect it to a *post-slot* of an *event* via a *dependence*.

Post-slot Output of an event, which may be connected to a pre-slot via a dependence.

Events Events are OCR objects used for synchronization. They have a number of pre-slots and one post-slot. An event monitors the satisfaction of its pre-slots and when a triggering rule is met, the event triggers. There are different types of events with different triggering condition, but the simplest *once* event has one pre-slot and triggers when that pre-slot is satisfied. After an event is triggered, it satisfies all pre-slots connected (using dependences) to its post-slot.

Once event An event triggered when its pre-slot is satisfied. It is destroyed automatically after it is triggered.

Sticky event An event triggered when its pre-slot is satisfied. It is not destroyed automatically. Subsequent satisfactions of the event's pre-slot are considered an error. An *idempotent* event is like a sticky event, but subsequent satisfactions are ignored and not considered an error.

Latch event An event with two pre-slots (*increment* and *decrement*) and a counter. Initially, the counter is set to zero. When the increment slot is satisfied, the counter is incremented, and it is decremented upon satisfaction of the decrement slot. The event triggers and is automatically destroyed when the counter reaches zero from a nonzero value. At the very least, the counter needs to be incremented once and then decremented once.

Dependences A connection between a post-slot and pre-slot of a task or an event. A single post-slot can take part in any number of dependences. A single pre-slot of a task can be only used in one dependence. Event pre-slots can be used in multiple dependences, but depending on the type of the event, it may be an error to allow the pre-slot to be satisfied more than once.

Dependency satisfaction After a dependence is set up, it may eventually be satisfied. There are four ways a dependence may be satisfied: The origin of a dependence is an event, which gets triggered and satisfies pre-slots connected via dependences to its post-slot. A dependence is set up with an already satisfied sticky event as the origin. A dependence is set up with a data block as the origin. A dependence is set up with a `NULL_GUID` as the origin.

EDT A task in OCR is referred to as an Event Driven Task (EDT). However, as there is no other type of task in OCR, we will use the term *task*, rather than EDT.

Output event Every task has an associated output event, which is created automatically together with that task. The event is triggered automatically after the task finishes.

Finish tasks Finish tasks are the exception to the rule that the output event is satisfied after the associated task ends. For finish tasks (the finish flag is provided at task creation), the triggering of the output event waits not only for the task to finish, but also for all child tasks to finish. A child task is a task created by the original task or its children.

Data blocks Data blocks contain application data. To give a task access to the contents of a data block, the data block needs to be either created by the task or passed to the task via one of its pre-slots. When a pre-slot is satisfied, it is possible to also provide a GUID of a data block, which will give the task access to the data stored in the data block. If a pre-slot of an event (except for a latch event) is satisfied and a GUID is provided, the event will then pass the GUID to pre-slots connected to its post-slot. This way, a control dependence can also serve as a data dependence.

Access modes For a task to be able to run, exactly one dependence has to be set up for each of its pre-slots. When the dependence is set up, an access mode needs to be specified. There are four access modes: RO (read-only), RW (read-write), EW (exclusive write) and CONST (constant). These define whether the task can read and modify (RW, EW) or only read (RO, CONST) the data in the data block. The OCR memory model (in [15]) defines the exact rules of how the access modes apply to concurrently running tasks.

Data block acquisition and release To access a data block, a task needs to first *acquire* it. This is a processed performed automatically by the OCR runtime system. Data blocks passed to a pre-slot of the task are acquired automatically before the task starts. Newly created data blocks are acquired as part of their creation. Internally, acquiring a data block ensures that the task has access to the correct copy of the data (as per the OCR memory model). No data block can be accessed without being acquired. All data blocks acquired by a task need to be released. This can be done explicitly inside the task using the OCR API or automatically after the end of the task. Releasing a data block means the task has finished reading and/or modifying the data. A released data block cannot be read or written, it can also not be re-acquired. The fact that all read and write operations have to happen between acquisition and release of a data block is significant for the OCR memory model. Reliance on explicit acquire and release operations (release consistency) can improve performance by allowing more asynchrony [21].

OCR application OCR applications are C or C++ programs, which conform to the OCR specification. They provide the C/C++ functions that serve as bodies of tasks. The application is also fully responsible for defining the data structures used to store data in data blocks. An OCR application needs to be "pure"—all code of the application needs to be run inside OCR tasks, including the program entry point. The OCR needs to "own the main"—instead of the usual `main` function, a `mainEdt` function is used to build a single task, which is then executed automatically by the OCR runtime system. The task receives a data block with the command line parameters used to start the application. The main task should create other tasks to get the computation going.

OCR API The OCR API defines a C interface which the OCR application uses to issue commands to the OCR runtime system. The full list and documentation is available in the OCR specification, but a short description of relevant commands is given in "Appendix B".

Appendix B: OCR API (excerpt)

`ocrGuid_t` is a data type used to store GUIDs (handles) of objects. It is also passed to all OCR functions when an objects needs to be references.

`ocrEventSatisfy(event,data)` satisfies the pre-slot of event `event` with a data block `data`. The data block can also be a null data block (`NULL_GUID`).

`ocrEventSatisfySlot(event,data,slot)` satisfies a specified pre-slot of the event. Used with latch events that have two pre-slots.

`ocrAddDependence(source,destination,slot,mode)` sets up a dependence from the post-slot of `source` to the specified pre-slot of `destination`. The source is an event, the destination is an event or a task. For tasks, the `mode` parameter is used to set an access mode under which the data passed along the dependence will be available to the task. It is also possible to use a data block as the source, in which case the pre-slot is satisfied with the data block. For events, this is similar to `ocrEventSatisfySlot`. For tasks, it is the only way to pass a data block to the task directly.

`ocrEdtCreate(guid,tml,paramc,paramv,depc,depv,flags,hint, output)` creates a new task (EDT) from a task template `tml`. Numerical parameters may be passed to the task via `paramv` and it is possible to immediately satisfy some of the task's pre-slots via `depv`. An associated output event is returned via `output`.

`ocrDbCreate(guid,addr,size,flags,hint,allocator)` creates a new data block of the specified size, returning its GUID and a C address of the allocated memory.

`ocrEventCreate(guid,type,flags)` creates a new event of the specified type.

`ocrDbRelease(guid)` releases a data block that has been acquired by the current task. `ocrDbDestroy(guid)` destroys a data block.

`ocrShutdown()` instructs the runtime system to shut down. Used upon successful completion of the program.

`ocrAbort(code)` instructs the runtime system to shut down. Used upon failure of the program, returning an error code.

Appendix C: Definition of *synchronized-with*

First, we need to define what entities form the domain and the range of the relation. One obvious case is OCR API calls made by the application. For practical reasons, it is good to also include a *task-begin* operation—a NOP operation at the beginning of each task, which is *sequenced-before* all operations made inside the task. The last case is *implicit operations* made by the runtime system.

Implicit operations are OCR API calls which are not performed by the user's code inside tasks, but by the runtime system as a response to certain situations. For example, when an event is triggered, it should satisfy all pre-slots connected to its post-slot. The connected pre-slots are satisfied by implicit `ocrEventSatisfySlot` (for events) and ocrAddDependence (for tasks) operations. These play a key role in defining the *synchronized-with* relation.

To make the definition of *synchronized-with* more compact, we make several assumptions about the way the OCR applications are written. WLOG, we always assume that ocrEventSatisfySlot is used and that the simpler ocrEventSatisfy only serves as an alias for ocrEventSatisfySlot with slot being 0. We also do not consider task output events as a special case. Instead, we assume that there is a ocrEventSatisfy added at the end of any task that has a output event. The API call satisfies the output event explicitly. This can also be done with *finish* tasks, see "Handling of finish tasks" section of Appendix C for details. Note that to maintain correctness, the event satisfaction needs to be done after all data blocks have been released, including data blocks released implicitly at the end of the task. As we have already discussed, we expect all tasks to begin with an empty *task-begin* operation.

We also need to deal with the implicit operations. For clarity, we define a new API call ocrTaskSatisfySlot(T,D,I) (arguments are Task, Data block, Index of pre-slot), which works like ocrEventSatisfySlot(E,D,I) for events. Normally, ocrAddDependence would be used to satisfy a task's pre-slot with a data block (or NULL_GUID). There are two cases where we need to include implicit operations in *synchronized-with*. The first case occurs when the triggering condition of the event gets satisfied. When an event is triggered, it invokes ocrEventSatisfySlot (if destination is an event) or ocrAddDependence (if destination is a task) on all pre-slots connected to its post-slot. The second case is only valid for *sticky* and *idempotent* events. Even after such event is triggered, it may be used as a source of a dependence. In that case, the destination of the dependence is also satisfied via ocrEventSatisfySlot or ocrAddDependence.

We define that A *synchronized-with* B in the following cases (note that the underscore character is used as a wildcard in the API calls):

1. For a task T, A is ocrAddDependence(_,T,_,_) and B is *task-begin* of T.
2. For any event E, A is ocrEventSatisfySlot(E,_,_) and B is an implicit operation ocrEventSatisfySlot(_,_,_) invoked by the runtime system to satisfy a connected event's pre-slot after E is triggered.
3. For any event E, A is ocrEventSatisfySlot(E,_,_) and B is an implicit operation ocrTaskSatisfySlot(_,_,_) invoked by the runtime system to satisfy a connected task's pre-slot after E is triggered.
4. For any event E, A is ocrAddDependence(_,E,_,_) and B is the implicit operation ocrEventSatisfySlot(E,_,_) invoked by the runtime system when the dependence setup by A is satisfied.
5. For any task T, A is ocrAddDependence(_,T,_,_) and B is the implicit operation ocrTaskSatisfySlot(T,_,_) invoked by the runtime system when the dependence setup by A is satisfied.

For *once* events, the last two conditions are not necessary, since the requirement is that a dependence needs to be set up before it is satisfied. *Latch* events work in a similar way, but *sticky* and *idempotent* events don't have this requirement. Therefore, the ocrAddDependence might come after the event has been satisfied. Furthermore, this also covers the situation when ocrAddDependence is used with a data block or NULL_GUID as the origin—this also satisfies the pre-slot and provides synchronization.

Note that rules 2 and 4 often apply to the same operation B, meaning satisfaction of an event's pre-slot is synchronized after the dependence is set up and the origin event is satisfied. The same is true for rules 3 and 5 for pre-slots of tasks.

Having a proper definition of *synchronized-with* can provide us with new insights. For example, it turns out that idempotent events are a problem. An idempotent may be satisfied multiple times, but all satisfactions apart from the first one are ignored. However, given the way we have just defined *synchronized-with*, all satisfactions of the event form get *synchronized-with* actions that follow the triggering of the event. As we do not know how many times the event will be satisfied, it would be impossible to trigger the event. We could try to avoid this problem by having only the first satisfaction enter the *synchronized-with* relation. But it is very difficult to determine which one is first in a distributed system. The satisfactions may not be synchronized with each other. It could be left to the runtime implementation—the runtime system might choose to pick one satisfaction and use that for synchronization. Alternatively, the idempotent event could be defined not to establish *synchronized-with* at all. We have not encountered a use case that would benefit from such behavior. As far as we could determine, none of the existing OCR applications uses idempotent events. For these reasons, we have suggested removing idempotent events from the specification.

Using ocrAddDependence to satisfy an event

It is possible to use a NULL_GUID or a GUID of a data block as the source in a ocrAddDependence call, even if the destination is an event. This is very similar to using ocrEventSatisfySlot. However, as of OCR 1.2.0, the effect is not guaranteed to happen synchronously. The reason for this is the fact that the OCR runtime system may not know that the destination of the dependence is an event or that the source is a data block. When ocrEventSatisfySlot is used, it is the user's responsibility to make sure that the objects are indeed an event and a data block (or NULL_GUID). Forcing the runtime system to figure out the types of objects passed to ocrAddDependence would impose an unnecessary restriction and potentially slow applications down.

It is still possible to use ocrAddDependence to satisfy pre-slots of events but unlike ocrEventSatisfySlot, the actual satisfaction of the pre-slot is a delayed effect, which is executed by an implicit operation and it happens outside of the task. Therefore, the actions performed by the task after calling ocrAddDependence are not in the *happens-before* relation with the satisfaction of the pre-slot. The pre-slot may be satisfied even after the task has long been finished.

This may be a problem when combined with latch events, where proper ordering of increment and decrement operations must be ensured by the application. Incrementing the counter using ocrAddDependence and then decrementing it with ocrEventSatisfySlot may cause the increment to be evaluated after the counter is decremented. This is a simple example, but it shows that the counter of a latch event should always be directly incremented with `ocrEventSatisfySlot`. If it is satisfied by a dependence, the effect is always delayed and it is not possible to ensure it is evaluated before the matching decrement operation. Satisfaction of the pre-slot is only synchronized with the triggering of the latch event, so it is not possible to ensure that the decrement happens after the satisfaction, but before triggering of the event.

Handling of finish tasks

The requirement imposed on finish tasks is that the associated output event is satisfied only after all children have finished. A finish task may create more tasks (children), which in turn can recursively create further tasks. The usual implementation of a finish task is to make its output event a latch event. Initially, the event's counter is set to 1 and whenever a child task is created, the counter is incremented. There are two ways to decrement the counter. First, a child task could decrement it directly after it finishes and all data blocks are released. There already is a call to `ocrEventSatisfy` to satisfy the task's output event, so we add another one next to it (the actual order does not matter), which satisfies the decrement pre-slot of the latch event. The second option is to connect the child task's output event to the decrement pre-slot of the latch event using a dependence. In both cases, the overall result is that all operations inside all child tasks are *synchronized-with* with the triggering of the latch event. In other words, it ensures that anything that depends on the latch event is guaranteed to run after all of the child tasks and that any such operation sees the changes made by the child tasks.

Synchronization of task creation

When a task is created by the `ocrEdtCreate` call, the dependences are either provided directly as the `depv` argument or later using `ocrAddDependence`. The implicit or explicit addition of dependences establishes synchronization with the newly created task. However, there is one exception. A task may have zero pre-slots—no dependences. In this case, the task is ready to run as soon as the `ocrEdtCreate` call finishes. Let us denote the running task t_1 and the newly created task with no dependences t_2. Based on the rules established so far, t_1 is not *synchronized-with* t_2. This may not seem important, as a task with no dependences may not access any data blocks, therefore we don't need to establish which changes are visible inside the task. But, t_2 may create further tasks and provide those with data blocks to read. GUIDs of the data blocks are needed, but these can be provided via parameters (the `paramv` argument of `ocrEdtCreate`) even to a task with no pre-slots. Synchronization will be established between t_2 and the new tasks, but not with t_1.

One solution is to view the parameters passed to a task as an implicit data block, which is created (and released) by the creating task and destroyed at the end of the created task. The implicit `ocrAddDependence` used to assign the data block to the created task will establish a synchronization (rule 1). Alternatively, we can set up another rule:

6. For any task T, A is `ocrEdtCreate(T,_,_,_,_,_,_,_,_)` and B is *task-begin* of T.

Appendix D: Deferred execution low-level view

Probably the most interesting case where deferred execution is not straightforward is the `ocrDbCreate` call. `ocrDbCreate` returns GUID of the newly created data block as well as a pointer to the memory belonging to the data block. Obviously, those cannot be deferred. The task needs to get a valid pointer to memory that can be used to write data to the data block. The GUID also needs to be correct. First, it can be used in further OCR calls. That is not that much of a problem, since if the data block creation is deferred, the runtime system would also have to defer the following API calls, giving it enough time to figure out what the GUID should really be. It would be possible to use some kind of temporary identifier. The second way a GUID can be used is to store it in a data block. In that case, the runtime system loses track of the GUID and it can be read from the data block in a different task. This would make using a temporary identifier without global validity difficult. To sum up, the runtime system needs to be able to provide GUIDs for newly created OCR objects even if the actual creation of the object is deferred. In the case of data blocks, the runtime system also has to provide a valid block of memory as soon as the call returns. Since the memory needs to be available locally, it is also allocated locally, which is usually fast enough to be performed immediately before the `ocrDbCreate` call returns. It is also possible to devise a scheme that allows GUIDs to be created efficiently. For further discussion of the topic, see “Local creation of GUIDs and data blocks” section of Appendix D.

With some careful design, the OCR runtime system is able to defer evaluation of the OCR API calls, even if the task’s code finishes in the meantime. After the task’s code finishes, the runtime system needs to perform several implicit operations—all remaining data blocks need to be released and the task’s output event needs to be satisfied. These implicit API calls can be deferred the same way as the API calls made inside the task.

Consider this example of a task with one pre-slots, which was satisfied with a data block (available as `depv[0]`):

```
ocrGuid_t f(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t
    depv[])
{
  ocrGuid_t event, db;
  void* ptr;
  ocrEventCreate(&event, OCR_EVENT_ONCE_T,
     EVT_PROP_TAKES_ARG);
  *(ocrGuid_t*)depv[0].ptr = event;
  ocrDbRelease(depv[0].guid);
  ocrEventSatisfy(event,depv[0].guid);
  ocrDbCreate(&db,&ptr,1024,DB_PROP_NONE,NULL_HINT,
     NO_ALLOC);
  compute(ptr);//computation with no OCR API calls
  return db;
}
```

Deferring the operations where possible, it could be evaluated similar to the following pseudo-code:

```
ocrGuid_t f(u32 paramc, u64* paramv, u32 depc, ocrEdtDep_t
    depv[])
{
  ocrGuid_t event, db;
  void* ptr;
  ocrCreateGuidForEvent(&event);
  *(ocrGuid_t*)depv[0].ptr = event;
  //ocrDbRelease(depv[0].guid); - deferred
  //ocrEventSatisfy(event,depv[0].guid); - deferred
  ocrCreateGuidAndBufferForDb(&db,&ptr,1024);
  compute(ptr);//computation with no OCR API calls
  result = db;
  //end of the original code
  ocrEventCreate(&event, OCR_EVENT_ONCE_T,
     EVT_PROP_TAKES_ARG);
  ocrDbRelease(depv[0].guid);
  ocrEventSatisfy(event,depv[0].guid);
  ocrDbCreate(&db,&ptr,1024,DB_PROP_NONE,NULL_HINT,
     NO_ALLOC);
  //implicit API calls
  ocrDbRelease(db);//db was not released by the user
  ocrEventSatisfy(myOutputEvent,result);//result is db
}
```

As you can see, all operations were deferred and the rewritten code is still correct—the order of the real OCR operations is preserved and the data blocks are always released before they are used to satisfy an event. This is necessary to ensure that any task that depends on the event is guaranteed to see the changes made to the data block.

“Leaked” changes when using RO and RW access modes

The basic usage scenario is write-release-acquire-read, where one task modifies a data block, releases it and another synchronized task acquires the data block and reads the data. The synchronization means that the release of the data block *happens-before* acquisition of the data block. In that case, the OCR memory model guarantees that the data is visible. However, there is one alternative for data sharing. If one task acquires a data block in the RW (non-exclusive read-write) and another task acquires the same data block in RW or RO (non-constant read-only), the other task may see the changes made first task even if they are not synchronized.

This could allow them to share a GUID of an object (e.g., the event in the example above) before the object is actually created. This does not happen with EW and CONST access modes, event with deferred execution model, but it cannot be avoided with RW and RO. This is one case of the problem we have discussed at the end of Sect. 2 that affects programming models which combine different kinds of objects or different ways of accessing the objects. The memory model and the object model do not integrate seamlessly, allowing possibly unexpected behavior in a corner case.

We believe this limitation is a price worth paying for the benefits of deferrable execution and it is in fact quite natural. The behavior of both memory and objects is based on the *happens-before* relationship. If two operations are not in a *happens-before* relationship, they are considered to be concurrent, even if they run one after the other in reality. This is the case here—the creation of the object has no *happens-before* relation to the task where the object was used after being "leaked" using a RW data block. Writing and reading data blocks does not establish any synchronization.

Local creation of GUIDs and data blocks

As we have already discussed, even if the evaluation of OCR API calls is deferred, GUIDs and pointers to data blocks need to be returned immediately. The difficulty of returning a correct GUID even before the object is created depends on the way GUIDs are constructed. This is left to the implementation. The GUID often encodes type of the object it identifies. This is not a problem, since we already know which object is being created from the API call used, even if the evaluation of the call is deferred. Remember the `ocrCreateGuidForEvent` from the pseudo-code above.

In a distributed environment, the GUID also usually encodes some identification of the cluster node which "owns" the object. The way this owner is determined determines whether the GUID can be generated before the API call is evaluated. At the moment, the known behavior of OCR implementations is to either create the object locally, use a simple round-robin distribution scheme, or use the node which was specified as a hint to the API call. All of these can be evaluated locally and quickly, allowing the actual creation to be deferred.

The GUID also needs to somehow identify the object among all objects owned by the same node. This may be a local counter (on the owning node), a pointer to the owner's memory where the object's state is stored, or some other unique identifier. All of these are only known on the owning node, which may be different from the node where the create call was made. At the moment, three different solutions are being used. The first option is to request the identifier from the owning node when the GUID is created. This means that the calling task needs to be blocked until the remote node responds. The communication may be very efficient, but it is still likely to have impact on the task's performance. Second, it may be possible to request the identifiers in batches, caching the values. This way, most of the GUIDs can be created without communication. The last option is to use a richer GUID structure and allow nodes to locally create GUIDs owned by other nodes. For example, if the local identifier is a counter, a unique identifier created locally may be "first GUID created by node 4 for node 5". Node 4 can ensure that the counter values are unique (no other node needs to increment "node 4 for node 5" counter). The obvious disadvantage of this solution is using more bits of the GUID to identify both nodes that are involved. The advantage is the total absence of any required communication.

When creating a data block, it is also necessary to return a pointer to a valid memory pointer, even if the actual creation is deferred. Since the buffer always needs to be created locally to make the pointer valid locally, this should not be an issue. Some OCR implementations only allow data blocks not owned by the local node to be created with the `DB_PROP_NO_ACQUIRE` flag. If the flag is used, no pointer is returned, eliminating the problem. However, any distributed OCR runtime system needs to be able to move the contents of a data block between node, so it may be possible to also create the remote data block so that it looks as if its data was already moved to the node that made the `ocrDbCreate` call. For example, node 4 (where the `ocrDbCreate` call was made) creates a data block on node 5, but node 5 knows that the latest version of the data block is not on node 5, but on node 4. In this case, the "remote copy" on node 4 can be created immediately, but the creation of the data block on the owning node (node 5) can be deferred. At least one existing OCR implementation (OCR-Vx) provides this functionality.

Appendix E: Object model definition

In the following definition, we define when the modifications made by OCR API calls (including the implicit API calls) must be visible when another OCR API call is evaluated. As a reminder, *implicit OCR API call* is an action performed by the OCR runtime system in order to properly handle state transitions of OCR objects, namely triggering of events and execution of tasks. For example, if a *once* event is satisfied, it needs to be triggered at some in the future. When that happens, all pre-slots attached to the post-slot of the event need to be satisfied. These pre-slot satisfactions are implicit calls to `ocrEventSatisfySlot` (for pre-slots of events) and `ocrAddDependence` (for pre-slots of tasks). Another example is the implicit release of data blocks that were acquired by a task but not explicitly released.

Definition 1 For any OCR object X, all reads and writes to the state of the object have to be made atomically and there has to be a global order $\rightsquigarrow$ of the reads/writes that satisfies the following conditions for any two OCR API calls ocr_1 and ocr_2, such that ocr_1 *happens-before* ocr_2:

- $im_1 \rightsquigarrow im_2$ for any immediate effect im_1 of ocr_1 and any immediate effect im_2 of ocr_2;
- $im_1 \rightsquigarrow de_2$ for any immediate effect im_1 of ocr_1 and any delayed effect de_2 of ocr_2;
- $im_1 \rightsquigarrow de_1$ for any immediate effect im_1 of ocr_1 and any delayed effect de_1 of the same OCR API call;

where im_i and de_i modify the state of object X.

In other words, the order in which immediate changes are applied to an OCR objects need to observe the *happens-before* relation and delayed effects have to come after immediate effects of the same operation and all operations that happened before.

Consider the following code fragment:

```
ocrAddDependence(e1,e2,0,DB_DEFAULT_MODE);
ocrEventSatisfy(e1,NULL_GUID);
```

The immediate effect of the `ocrAddDependence` call is adding the pre-slot of e_2 to the list of pre-slots connected to the post-slot of e_1. When the `ocrEventSatisfy` call is made, this has to be visible, because *sequenced-before* implies *happens-before* relationship between the two calls. Triggering of e_1 is a delayed effect of the `ocrEventSatisfy`, so it needs to come after the immediate effect (linking of e_2) of the `ocrAddDependence`. Therefore, the runtime system has to make the implicit call `ocrEventSatisfy(e2,NULL_GUID)` that propagates e_1's satisfaction along the $e_1 \rightarrow e_2$ dependence.

Now, consider this code fragment:

```
ocrAddDependence(e1,e2,0,DB_DEFAULT_MODE);
ocrEventSatisfy(e1,NULL_GUID);
ocrAddDependence(e2,e3,0,DB_DEFAULT_MODE);
```

When the second `ocrAddDependence` is evaluated, we do not know whether e_2 has already been satisfied or not. Satisfaction of e_2 is a delayed effect of the `ocrEventSatisfy`. The object model does not require it to be applied before the immediate changes made in subsequent API calls. If e_2 is a *once* event, the code is not a valid OCR program, since OCR requires dependences where the *once* event is a source to be defined before the event is satisfied. In this case, the behavior is undefined. However, e_2 may be a *sticky* event. In that case, the code fragment is a correct OCR code. Either the $e_2 \rightarrow e_3$ dependence is added before e_2 is triggered, in which case the pre-slot of e_3 is satisfied as part of the triggering of e_2. If e_2 is triggered before the dependence is added, the pre-slot of e_3 will be satisfied as well. Adding a dependence where the source is a triggered *sticky* event has the (delayed) effect of satisfying the destination pre-slot.

The definition constrains the behavior of the OCR runtime system, but they should not impose a certain implementation. The implementations only have to guarantee providing equivalent results. For example, it is not necessary to use ensure atomicity and total ordering of all operations. If the implementation determines that two operations are independent, they may be performed concurrently. For example, setting up a dependence on two different pre-slots of a task can be done in parallel, the runtime system only needs to ensure that if all pre-slots of the task get satisfied, the task eventually starts. Maintaining atomicity only when updating the number of unsatisfied pre-slots is one of the possible solutions.

Another significant example is the deferred execution model. As is clear from their definition, delayed effects can be deferred. However, using the arguments presented in Sect. 6, it is possible to also defer immediate effects. The runtime system needs to maintain the appearance that they are resolved immediately, but as long as the relative order of the immediate effects is preserved and delayed effects cannot overtake any immediate effects it is possible to uphold the object model even in combination with deferred evaluation.

The effects of existing API calls

When determining whether API call effects are immediate or delayed, the existing API calls can be split into several categories:

- *Unaffected* These API calls are not affected by the proposed changes, since they return a value based purely on their parameters: argument handling (`getArgc`, `getArgv`), GUID management (like `ocrGuidIsNull`),
- *Object creation* All object creation calls create the object as an immediate effect. If a call to `ocrEdtCreate` results in a task becoming runnable, this effect is delayed.
- *Object destruction* All objects are considered to be destroyed immediately, with the exception of `ocrDbDestroy`, which requires the data block to only be destroyed once it has been released by all tasks.
- *Data block release* The effect of `ocrDbRelease` call is immediate.
- *Shutdown* The exact timing of the two shutdown API calls (`ocrShutdown` and `ocrAbort`) is implementation specific, as the calls mark the end of the program and no other tasks should depend on the task that made the call. Therefore, it only affects the execution of tasks not connected by a *happens-before* relationship and is not affected by the proposed object model.
- *Event satisfaction* The `ocrEventSatisfy` and `ocrEventSatisfySlot` calls have an immediate effect of satisfying the specified event's pre-slot and evaluating the event's trigger condition. However, if the trigger condition is met, the triggering of the event is a delayed effect of the call, as well as any further effects the event's triggering has on the objects that depend on the event.
- *Add dependence* The `ocrAddDependence` has the immediate effect of connecting the source's post-slot to the destination's pre-slot. Any other effects are delayed, most notably satisfaction of the destination's pre-slot, if the source is a null GUID, data block, or a satisfied sticky event.

Appendix F: Implementations of the object model and their correctness

The possible implementations have been described on a high level in Sect. 8. In the following text, we provide a low-level description. We also provide (partial) proofs that show that the implementations actually conform to the object model. We do not provide full proofs, as these would require going through many different cases, increasing the length of the text too much.

Causality

To explore how is the object model imposed by a runtime system, we will use the causality relationship. Causality is a relation that for two operations defines in which order they happened. It does not provide a total ordering of operations. Some pairs of operations are said to be *concurrent*, which means that causality between the two operations cannot be established in either direction. In the physical world, the operations may happen in any order, concurrently, or the order may be undefined[1]. To establish the behavior of a system, we will only use the causality relation where the ordering of operations is clearly established either because two operations happened in some order at the same place or because there is a clear cause and effect relation between the operations and we assume that causality is withheld in the physical world.

Definition 2 Two operations a and b are defined to be causally connected (denoted $a \rightarrow b$) if any of the following conditions holds:

A1. Both operations occur in the same process and operation a precedes operation b (co-located operations).
A2. Operation a is the sending of a message and operation b is the receipt of the message (cause and effect).
A3. There is an operation c, such that $a \rightarrow c$ and $c \rightarrow b$ (transitivity).

We only consider systems that make physical sense, i.e., operations cannot precede themselves (the relation is anti-reflexive) and messages observe the cause and effect rule. As a result, if a and b operations happen within the same process and $a \rightarrow b$, we know that operation a actually happens before operation b in the usual physical meaning of time.

We impose further limitations on the messaging system, which must hold for any two processes P_1 and P_2:

A4. If $send_{P_1 \rightarrow P_2}(m_1) \rightarrow send_{P_1 \rightarrow P_2}(m_2)$ (message 1 is sent from process 1 to process 2 before message 2 is sent from process 1 to process 2) then $recv_{P_1 \rightarrow P_2}(m_1) \rightarrow recv_{P_1 \rightarrow P_2}(m_2)$. In other words, messages sent between two processes cannot overtake each other.
A5. If $recv_{P_1 \rightarrow P_2}(m_1) \rightarrow recv_{P_1 \rightarrow P_2}(m_2)$ then $proc_{P_2}(m_1) \rightarrow proc_{P_2}(m_2)$. The messages are processed in the order they are received.
A6. If $proc_{P_2}(m_1) \rightarrow proc_{P_2}(m_2)$ then $send_{P_2 \rightarrow P_1}(m'_1) \rightarrow send_{P_2 \rightarrow P_1}(m'_2)$, where m'_1 and m'_2 are confirmation messages which tell the sender that the original messages (m_1 and m_2, respectively) have been processed. The confirmation messages are sent in the same order as the messages have been processed.

It is important to clearly distinguish the meaning of *happens-before* and causality in this text. Traditionally [13,19], they are two names for the same concept, but in our work, we need to work with two different relations, therefore the distinction. The *happens-before* relation denotes the relation of operations in the OCR model. The causality relation will be used to model what happens in the runtime system, which

[1] The special relativity tells us that it is not possible to establish that two spatially separated events occur at the same time (the relativity of simultaneity).

implements the OCR. Since we assume that each OCR object is stored exclusively on one node and all changes to its state must be evaluated on that node (as an atomic operation or guarded by a lock), if a change (write) to the object's state causally precedes access (read) to the state, the change will be visible. As a result, we can define the constraint on runtime system implementations like this:

Correct runtime system implementation

The definition of the object model does not assume that OCR is implemented in a certain way. We have already made some assumptions in Sect. 8 and we will make more in the following text. We will show how the object model can be applied to a runtime system that follows our assumptions. It is certainly possible to design a runtime system which does not meet the requirements but still adheres to the object model through some other means.

We expect the runtime system to ensure that reading and writing of data blocks follow the OCR memory model. All data blocks need to be acquired and released. The same messaging mechanism is used for data block acquisition and to implement other OCR API calls. We require the task scheduler to only run OCR tasks once all pre-slots have been satisfied. Satisfaction of task pre-slots is handled by explicit and implicit OCR API calls that follow the object model.

Lemma 1 *A runtime system that follows restrictions mentioned earlier follows the object model if all changes (of both immediate and delayed effects of API calls) are applied to objects atomically and for any two such changes a and b that realize effects of API calls that have to be order by $\rightsquigarrow$ as per Definition 1 the runtime system ensures $a \rightarrow b$.*

Proof As all changes to any object X are applied atomically by the owning process, there has to be some total ordering O of the changes of object X. When some of the changes are causally ordered, O must be a superset of the causal ordering of all operations on X, because we require causal ordering to observe the rules of the physical world. Since we require that the causality relation matches the $\rightsquigarrow$ relation, O is the required global order from Definition 1. □

Immediate confirmation protocol

The most basic protocol is to require each message that is used to process an OCR API call to be confirmed before processing the delayed effects and continuing to the next command in the task.

Lemma 2 *The immediate confirmation protocol is a correct implementation of the object model.*

Proof We assume that each effect is performed by the node that owns the affected object, when the owner gets to process a message requesting the effect to be processed. For any operation ocr_1, we will define $begin(ocr_1)$ and $end(ocr_1)$ as empty actions that can be causally ordered. They denote the beginning and the end of processing immediate effects of ocr_1. When we combine this, the premise of the immediate confirmation protocol, and the way we require message processing to be handled, then for any immediate effect im_1 of ocr_1 we get $begin(ocr_1) \rightarrow send(m_1) \rightarrow receive(m_1) \rightarrow proc(m_1) \rightarrow send(m'_1) \rightarrow receive(m'_1) \rightarrow end(ocr_1)$. The important part is that $begin(ocr_1) \rightarrow proc(m_1) \rightarrow end(ocr_1)$, since $proc(m_1)$ means that im_1 is applied to the object.

There are three cases when operations need to be ordered given in Definition 1. As discussed in the previous section, we need to show that in all cases, these cases are causally ordered. Let us start with the simplest case, $im_1 \rightsquigarrow de_1$ for an immediate and delayed effects of the same operation ocr_1. Let m_2 denote the message that instructs the owner of the object to apply de_1. The definition of the immediate confirmation protocol requires that $end(ocr_1) \rightarrow send(m_1)$. Combined with the observation from the previous paragraph and the way messages are handled, we get $proc(m_1) \rightarrow end(ocr_1) \rightarrow send(m_2) \rightarrow receive(m_2) \rightarrow proc(m_2)$, which can be shortened as $proc(m_1) \rightarrow proc(m_2)$. Since the messages correspond to im_1 and de_1, respectively, we know that application of these effects is causally ordered. Both are done atomically by the owner, which means this case is handled as it should be according to Definition 1.

The second case are two immediate effects im_1 and im_2 of two operations ocr_1 and ocr_2, where ocr_1 *happens-before* ocr_2. A runtime system that satisfies the requirements we put on runtime systems in this section (including the immediate confirmation protocol) ensures that in this case $end(ocr_1) \rightarrow begin(ocr_2)$. From this, we get $begin(ocr_1) \rightarrow send(m_1) \rightarrow receive(m_1) \rightarrow proc(m_1) \rightarrow send(m'_1) \rightarrow receive(m'_1) \rightarrow end(ocr_1) \rightarrow begin(ocr_2) \rightarrow send(m_2) \rightarrow receive(m_2) \rightarrow proc(m_2) \rightarrow send(m'_2) \rightarrow receive(m'_2) \rightarrow end(ocr_2)$, which can be simplified to $proc(m_1) \rightarrow proc(m_2)$, just as we needed.

The third case is immediate effects im_1 of ocr_1 and delayed effect de_2 of ocr_2, where ocr_1 *happens-before* ocr_2. Combining the arguments from the two previous cases, we get $begin(ocr_1) \rightarrow send(m_1) \rightarrow receive(m_1) \rightarrow proc(m_1) \rightarrow send(m'_1) \rightarrow receive(m'_1) \rightarrow end(ocr_1) \rightarrow begin(ocr_2) \rightarrow end(ocr_2) \rightarrow send(m_2) \rightarrow receive(m_2) \rightarrow proc(m_2) \rightarrow send(m'_2) \rightarrow receive(m'_2) \rightarrow end(ocr_2)$. This provides the required ordering of im_1 and de_2.

We still need to show that ocr_1 *happens-before* ocr_2 implies $end(ocr_1) \rightarrow begin(ocr_2)$. For ocr_1 *sequenced-before* ocr_2, this is trivial. The complicated part is showing it for ocr_1 *synchronized-with* ocr_2. The validity for the general *happens-before* case is then also trivial, since it can be broken down into individual *sequenced-before* and *synchronized-with* and causality is also transitive. Proving the property for *synchronized-with* requires analyzing all cases from the definition of the relation and showing that the way they are implemented with messages ensures the causality. We don't provide the analysis here for space reasons. □

The omitted part of the proof is not difficult, but there is one aspect that needs to be considered. The property that synchronization implies causality only holds for correct OCR programs. For example, consider an application where a once event is satisfied twice. Based on the definition of *synchronized-with*, both calls that satisfy the event should be *synchronized-with* any task that depends on the event, but this is not the case. When the event is satisfied the first time, the runtime system may propagate the satisfaction along dependences. When the second satisfaction is made, the dependences may have long been satisfied. Also, the event has likely already been destroyed.

Overall, we can see that the immediate confirmation protocol should guarantee that the object model is correctly observed. This is true even in the presence of deferred execution model, for reasons discussed in Sect. 6. Without deferred execution, $begin(ocr_1)$ and $end(ocr_1)$ events would occur inside the OCR API call. With deferred execution, they can be pushed back to a later point in time. As long as the causality between consecutive commands ($end(ocr_1) \rightarrow begin(ocr_2)$) is preserved, we get correct results.

Grouped confirmation protocol

Grouped confirmation protocol works like the immediate confirmation protocol, except if effects of consecutive operations $ocr_1, \ldots, ocr_n$ are resolved exclusively by messages sent to one node (same for all commands), it is necessary to only wait for the confirmation of the last message.

Due to the no-overtaking rules, we know that for two operations ocr_a and ocr_b that belong to the sequence (in this order), $proc(m_a) \rightarrow proc(m_b)$. Therefore, the required behavior is maintained among the commands within the sequence. For any command ocr_1 in the sequence, if there is a command ocr_2 outside the sequence, such that ocr_1 *happens-before* ocr_2, we know that either ocr_1 is the last command in the sequence or there is one such command. WLOG, assume it is the latter case and we denote the command ocr_3. Because ocr_3 is the last command, we need to wait for the confirmation of the command before proceeding to other commands, so $end(ocr_3) \rightarrow begin(ocr_2)$. Using the same arguments as in the case of the immediate confirmation protocol, we know that $proc(m_3) \rightarrow proc(m_2)$. Due to the non-overtaking rules, $proc(m_1) \rightarrow proc(m_3)$. Combined, we get the desired result $proc(m_1) \rightarrow proc(m_2)$.

Operation reordering

Both immediate and grouped confirmation protocols aim to ensure that the effects of OCR API calls are evaluated in the order specified by the application. Technically, this is not actually required by the object model in every case. For example, consider this pseudo-code:

```
running tasks: t0
available tasks: t1, t2
events: e1, e2
t0 {
  ocrAddDependence(e1, t1, 0, DB_MODE_RO);
  ocrAddDependence(e2, t2, 0, DB_MODE_RO);
}
```

The pseudo-code first lists the relevant OCR runtime objects and then the code of the tasks.

The immediate effects of the first `ocrAddDependence` call modify the state of e_1 and t_1, the second call changes e_2 and t_2. If e_1 or e_2 is a triggered sticky event, there is also a delayed effect that satisfies the dependence – updates t_1 or t_2, respectively, to note that the pre-slot 0 is now satisfied. As these affect completely different object, it seems that they can be freely reordered. However, there is one aspect that needs to be considered. Changing the ordering of the operations has another potential effect. In the original order, the delayed effects of the second command were guaranteed to be ordered after the immediate effects of the first command. This is no longer the case after the operations have been swapped. The interesting delayed effect of the second command is satisfaction of t_2's pre-slot, which may allow t_2 to start. Any operations performed by t_2 are no longer guaranteed to be ordered after the immediate effects of `ocrAddDependence(e1, t1, 0, DB_MODE_RO)`. By investigating what the command actually does, we can come to the conclusion that this is not a problem. There is no way a task can determine whether t_1 has already been added to the list of pre-slots connected to the post-slot of e_1. Similarly, the fact that the access mode of t_1's pre-slot has been set or not cannot be determined in t_2. Since each task's pre-slot should only be used once to set a dependence, it is also not possible for t_2 to change the value, in which case the order would be relevant.

Similarly, we can reorder operations in this case:

```
running tasks: t0
available tasks: t1
events: e1, e2
t0 {
  ocrAddDependence(e1, t1, 0, DB_MODE_RO);
  ocrAddDependence(e2, t1, 1, DB_MODE_RO);
}
```

Here immediate effects of both calls modify t_1. However, as one sets the mode for pre-slot 0 and the other for pre-slot 1, changing the order does not change the result. Also, both calls need to be evaluated before t_1 starts, since dependences have to be set exactly once before a task starts. So, t_1 starts only after both immediate and delayed effects of both calls have been evaluated.

The ability to change order of operations depends on the specific pair of commands. It is also possible to only swap parts of commands. In the last example, we could first set both pre-slots of t_1 and then connect them to e_1 and e_2. This may not seem like an improvement, but if all three objects happen to reside on three different nodes and we use the grouped confirmation protocol, we reduce the number of required confirmations from four to three, because the first two calls both only modify the state of t_1 and are therefore both sent to the node which owns t_1. A common pattern in OCR programming is to create a new task and then use a long sequence of `ocrAddDependence` calls which all specify the new task as the destination. In that case, we may be able to group a large number of messages, which we otherwise could not.

References

1. AlEbrahim S, Ahmad I (2017) Task scheduling for heterogeneous computing systems. J Supercomput 73(6):2313–2338. https://doi.org/10.1007/s11227-016-1917-2
2. Boehm H J, Demsky B (2014) Outlawing ghosts: avoiding out-of-thin-air results. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC ’14. ACM, New York, pp 7:1–7:6. https://doi.org/10.1145/2618128.2618134
3. Chen Z, Li X, Chen J Y, Zhong H, Qin F (2012) SyncChecker: detecting synchronization errors between MPI applications and libraries. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp 342–353. https://doi.org/10.1109/IPDPS.2012.40
4. Coarfa C, Dotsenko Y, Mellor-Crummey J, Cantonnet F, El-Ghazawi T, Mohanti A, Yao Y, Chavarría-Miranda D (2005) An evaluation of global address space languages: co-array fortran and unified parallel c. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’05. ACM, New York, pp 36–47. https://doi.org/10.1145/1065944.1065950
5. Dokulil J, Benkner S (2015) Retargeting of the Open Community Runtime to Intel Xeon Phi. In: International Conference On Computational Science, ICCS 2015. Procedia Computer Science, pp 1453–1462
6. Dokulil J, Benkner S (2017) The Open Community Runtime on the Intel Knights Landing architecture. In: 17th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2017)
7. Dokulil J, Sandrieser M, Benkner S (2016) Implementing the Open Community Runtime for shared-memory and distributed-memory systems. In: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), pp 364–368. https://doi.org/10.1109/PDP.2016.81
8. Fu X, Chen Z, Huang C, Dong W, Wang J (2014) Synchronization error detection of MPI programs by symbolic execution. In: 2014 21st Asia-Pacific Software Engineering Conference, vol 1, pp 127–134. https://doi.org/10.1109/APSEC.2014.28
9. Howes L, Munshi A (eds) (2015) The OpenCL Specification, Version 2.0. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf. Accessed 9 Nov 2017
10. Jin S, Schiavone G, Turgut D (2008) A performance study of multiprocessor task scheduling algorithms. J Supercomput 43(1):77–97. https://doi.org/10.1007/s11227-007-0139-z
11. Kukanov A, Voss MJ (2007) The foundations for scalable multi-core software in Intel Threading Building Blocks. Intel Technol J 11(04):309–322
12. Lahav O, Vafeiadis V, Kang J, Hur CK, Dreyer D (2017) Repairing sequential consistency in c/c++11. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017. ACM, New York, pp 618–632. https://doi.org/10.1145/3062341.3062352
13. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
14. Landwehr J, Suetterlein J, Márquez A, Manzano J, Gao GR (2016) Application characterization at scale: lessons learned from developing a distributed open community runtime system for high performance computing. In: Proceedings of the ACM International Conference on Computing Frontiers, CF ’16. ACM, New York, pp 164–171. https://doi.org/10.1145/2903150.2903166
15. Mattson T, Cledat R (eds) (2016) The Open Community Runtime Interface, version 1.2. https://www.univie.ac.at/ocr-vx/doc/ocr-v1.2.0.pdf. Accessed 24 Oct 2018
16. Mattson TG et al (2016) The Open Community Runtime: a runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–7. https://doi.org/10.1109/HPEC.2016.7761580
17. Munshi A (2009) The OpenCL specification. In: 2009 IEEE Hot Chips 21 Symposium (HCS), pp 1–314. https://doi.org/10.1109/HOTCHIPS.2009.7478342
18. Qureshi K, Majeed B, Kazmi JH, Madani SA (2012) Task partitioning, scheduling and load balancing strategy for mixed nature of tasks. J Supercomput 59(3):1348–1359. https://doi.org/10.1007/s11227-010-0539-3
19. Raynal M, Singhal M (1996) Logical time: capturing causality in distributed systems. Computer 29(2):49–56. https://doi.org/10.1109/2.485846
20. Wu Z, Lu K, Wang X, Zhou X, Chen C (2015) Detecting harmful data races through parallel verification. J Supercomput 71(8):2922–2943. https://doi.org/10.1007/s11227-015-1418-8

21. Yeo J, Yeom HY, Park T (2003) An asynchronous protocol for release consistent distributed shared memory systems. J Supercomput 24(1):25–41. https://doi.org/10.1023/A:1020937425960
22. Yu L, Sarkar V (2018) GT-Race: graph traversal based data race detection for asynchronous many-task parallelism. In: Aldinucci M, Padovani L, Torquati M (eds) Euro-Par 2018: Parallel Processing. Springer, Cham, pp 59–73
23. Zheng Y, Kamil A, Driscoll M B, Shan H, Yelick K (2014) UPC++: a PGAS extension for C++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp 1105–1114. https://doi.org/10.1109/IPDPS.2014.115