Automatic Detection of Synchronization Errors in Codes that Target the Open Community Runtime

Jiri Dokulil[1(✉)] and Jana Katreniakova[2]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
jiri.dokulil@univie.ac.at
[2] Comenius University, Bratislava, Slovakia
katreniakova@dcs.fmph.uniba.sk

Abstract. The complexity of writing and debugging parallel programs makes tools that can support this effort very important. In the case of the Open Community Runtime, one major problem is ensuring that the program manages runtime objects correctly. For example, when one task uses an object and another task is responsible for deleting the object, the tasks need to be synchronized to ensure that the object is only destroyed once it is no longer being used. In this paper, we present a tool which observes program execution and analyzes it in order to find cases where the required synchronization is missing.

1 Introduction

Task-based runtime systems, including StarPU [1], HPX [7], UPC++ [15], or PaRSEC [2] have received a lot of interest given the increased complexity, performance variability, and heterogeneity of emerging architectures. The Open Community Runtime (OCR, [9]) is a recent specification [10] for an event-driven task-based runtime system developed within the US XStack targeting next generation extreme scale architectures. The basic idea of OCR is to use tasks to decouple computation from compute units and data blocks to decouple application data from specific memory. Synchronization is also abstracted by dependences among tasks. Events can be used to build more complex dependence patterns. The responsibility for work scheduling and data placement is moved to the runtime. The application issues tasks to the runtime, along with their dependences. The runtime examines this task graph (which should be a DAG) and decides when and where to execute the tasks.

Writing parallel programs is a difficult task [8]. This is especially true when writing programs directly at the level of a task-based runtime system like OCR. When the work is split into tasks, which are scheduled and executed by the runtime, the global execution-time context normally available as the stack trace is lost. Debuggers are not able to map a running task to the place where it was created, like they do with a function and the corresponding call site. This makes debugging task-based applications tricky. Tools that can support the developers' effort to write and debug such programs are therefore important.

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 3–15, 2018.
https://doi.org/10.1007/978-3-319-96983-1_1

To support our research on OCR [3–5], we created a single-threaded implementation of OCR, called OCR-V1, which can be used to aid the development of new OCR applications[1]. When an application is run with OCR-V1, the runtime checks that the OCR API is used correctly by the application, for example by testing that data blocks are not used after being released or destroyed. Many problems can be discovered this way, but because OCR-V1 uses a deterministic serial task schedule, its ability to detect synchronization errors is limited. Therefore, we have also extended OCR-V1 to collect execution traces, which can be analyzed to find synchronization errors. An unmodified OCR application is compiled and linked with the modified OCR-V1 runtime. When executed, the application generates an execution trace, which is then analyzed by a new tool that we developed to find errors. Due to the way synchronization is done in OCR, it is sufficient to use the instrumented runtime and the unmodified application.

Based on the OCR specification, we have defined a set of rules that a correct OCR application must follow. We detect errors by looking for violations of these rules. The rules (and errors) share one basic principle. Some operations on OCR objects (performed by the application via OCR API calls) need to happen in a certain order. For example, any object must not be used before it is created and it may not be used after it is destroyed. So, if one task accesses a data block and another task destroys the data block, the application must ensure that the access done by the first task happens before the delete operation in the second task. Dependences among the tasks have to be set up in such a way that there is causal relation (*happens-before*) among the operations. Our trace analyzer finds and reports instances where the synchronization is missing.

Existing tools like Valgrind/Helgrind [12] may not be able to detect these errors, as synchronization that is done internally by the runtime (for example, to ensure atomicity of concurrent operations) may appear as sufficient on the low level where Helgrind works. Naturally, we can only detect errors in the way the program interacts with the OCR runtime system, not application errors. If the desired algorithm is implemented incorrectly, but properly synchronized, no errors will be detected.

Our main contributions are: (1) the error-checking OCR-V1 runtime, which also generates execution traces of OCR programs; (2) definition of rules that a correct OCR application has to observe when dealing with OCR runtime objects; (3) trace analyzer, which finds violations of the rules in execution traces; (4) during our work, we have identified one problem where the OCR specification does not sufficiently specify how certain tasks should be synchronized.

The rest of the paper is organized as follows. First, related work is discussed in Sect. 2. In Sect. 3, we briefly describe key OCR concepts and explain how OCR programs are synchronized. Section 4 explains how we analyze OCR programs and find problems in them. Section 5 provides concrete examples of programs and the detected errors. The final section concludes the paper and discusses future work.

[1] OCR-V1 is available at: http://www.univie.ac.at/ocr-vx/.

2 Related Work

There are existing tools that try to find errors in parallel programs. One type are tools that observe execution of the parallel program and check for various error conditions. Probably the best know example is Valgrind [12], which is mostly used to look for incorrect use of memory, but it also includes two modules that detect threading errors (Helgrind and DRD). Clang ThreadSanitizer performs similar function. Intel Inspector is an example of a similar commercial tool.

There are also tools which use static code analysis. For example, there are tools like FindBugs that analyze either source or bytecode of Java and try to find concurrency problems [13].

Another option is too look at the way operations are ordered in threads. The `rr` tool saves program execution and allows it to be deterministically replayed. This solves at least two problems: concurrency problems are often non-deterministic (two subsequent executions of the same program on the same data may not encounter the same failure) and running the parallel program in the debugger changes timing, potentially preventing the problem from occurring at all. An alternative approach is taken by CHESS [11] and Maple [14], which influence the execution of a multi-threaded program in order to systematically explore possible thread schedules.

The architecture of our system is similar to the first category (e.g., Valgrind and its modules), where the parallel execution of the program is analyzed and the analyzer looks for known error "patterns". One pattern that Helgind, DRD, and ThreadSanitizer check are data races where access to a shared variable from multiple threads is not properly synchronized – they check the presence of the happens-before relation among the operations. Our solution uses a similar approach, but as data in OCR is handled differently from plain C/C++ and data races are generally not an issue, we focus on the correct use of OCR objects. But the basic principle is the same: observe the behavior of the program and then check that concurrent operations have been properly synchronized.

Our approach could be also applied to similar programming models. The key requirement for such model is that synchronization is done on the task level using dependences. Examples of such models are OpenCL (kernels correspond to tasks), CUDA (with multiple streams and events), and StarPU. In TBB and UPC++, where fine-grained synchronization (locks, atomic operations, ...) can be used in tasks or if there can be malicious data races caused by individual read and write operations, Valgrind and ThreadSanitizer are better starting points.

It would also be possible to apply those fine-grained techniques to data accesses made by OCR tasks. Although data in OCR is stored in data blocks which are acquired and released as whole, there are two different pairs of access modes that can be used. The first pair are the constant and exclusive-write access modes, where the runtime is responsible for ensuring that all data access is consistent (there are no data races). The second pair are read-only and read-write access modes, which permit data races (both read-write and write-write). We do not consider these. There is however ongoing work done at Georgia Tech, attempting to also find such data races.

3 OCR and Synchronization

In OCR, all work (all application code) should run inside tasks, which are scheduled by the runtime. Similarly, all application data is stored in data blocks, which are relocatable blocks of data also managed by the runtime. The tasks are non-blocking, which means that once a task starts, it is expected to run to completion without waiting for any other work to be done. The only way tasks can synchronize is using dependences with the help of events. The application defines what the dependences are, but they are evaluated by the runtime, which figures out when a task is ready to start.

3.1 Event Driven Synchronization

Events are used to synchronize tasks in OCR, hence the name used for tasks in OCR – Event Driven Tasks (EDTs). Events and tasks in OCR can be connected using dependences. Tasks and events have slots that can be used as sources (post-slots) and destinations (pre-slots) of a dependence. A task has to wait for all of its pre-slots to be *satisfied* before it can start. Slots can either be satisfied directly using an OCR API call or they are satisfied automatically when they are connected to a post-slot of an event and the event itself is satisfied.

There are different types of events that have different rules that determine when the event is satisfied. The simplest one, the *once* event, gets satisfied as soon as its single pre-slot is satisfied. In other words, it directly propagates the satisfaction signal. We also say that the event has been *triggered* to distinguish satisfaction of the event from satisfaction of its pre-slot. Another interesting type of event is the *latch* event. It has two pre-slots and maintains an internal counter. The counter is incremented when the first of the two slots gets satisfied and decremented when the second slot gets satisfied. When the counter reaches zero, the event itself is satisfied and forwards the satisfaction signal – satisfies all pre-slots that are connected to its post-slot. Another important kind of events are *output* events. These are not a specific type of event (they are in fact *once* or *latch* events), but events used in a specific situation. For every task, there is a matching output event, which is satisfied after the task finishes.

3.2 State of OCR Objects

We have already introduced three types of OCR objects: tasks, events, and data blocks. All OCR objects carry some state. For example, a *latch* event needs to store the value of its counter. A data block needs to know the size of the corresponding buffer and it may also track which tasks have acquired it. However, the actual data stored in a data block is not considered to belong to the state of the data block object. The data plays a special role in the OCR specification and cannot be modified by OCR API calls, but only directly by reading and writing memory via a pointer. Note that dependences are generally not considered to be OCR objects. Therefore, adding a dependence is considered to be a change of state of the two connected objects (event and task/event).

3.3 The *happens-before* Relation

Events and dependences are used to define the *happens-before* relation among the operations performed inside tasks. If operation A *happens-before* operation B, it means that they are synchronized in a way that ensures that operation B sees the results of operation A. A simple example is when an output event of a task is used as a source of a dependence connected to a second task. In that case, the satisfaction of the event happens after the first task finishes and the second task can only start after the event is satisfied. Therefore, all operations done by the first task *happen-before* all operations done by the second task. The OCR memory model guarantees that all changes made by the first task are visible in the second task. This is true not only for changes to the application data (in data blocks), but also to state changes of runtime objects. For example, a newly created event is valid in the second task. Also, if the counter of a latch event is incremented from 0 to 2 by the first task and the second task decrements it, it is a valid operation which changes the value from 2 to 1. If the second task was not synchronized after the first task, it could run in parallel and try to decrement the counter while it is still zero, which is illegal.

There are only two types of operations that may change the state of OCR objects. First, the OCR API calls made inside the tasks (e.g., `ocrDbCreate` creates a new data block). Second, the runtime may modify the state automatically. For example, after a task finishes, the associated output event is satisfied. This also causes tasks and events connected (via dependences) to this event's post-slot to be also satisfied. Additionally, the finished task and its output event are automatically destroyed by the runtime. The only exception are data blocks, whose data is modified by memory reads and writes done inside tasks. But the state of the data block object itself (the size of the buffer, etc.) is still managed purely by OCR API calls. Because we only focus on the state of the OCR objects and not the application data and since all synchronization has to be done using OCR objects (tasks, events, and dependences), we only need to observe OCR API calls being made by the application and implicit operations done by the runtime. Since the runtime processes all the OCR API calls, we only need to instrument the runtime to collect the relevant data, not the application itself.

Consider the following example in OCR pseudo-code:

```
running tasks: t1
available tasks: t2
events: e1, e2
t1 {
  ocrAddDependence(NULL,t2); //allow t2 to start
  ocrAddDependence(e1,e2);//set up dependence e1->e2
}
t2 {
  ocrEventSatisfy(e1); //satisfy event e1
}
```

Here, `t2` has only one pre-slot, so when `t1` sets up a dependence from a NULL object to the pre-slot, it effectively satisfies, allowing `t2` to start. Then, `t1` goes

on to set up a dependence between events `e1` and `e2`. For correct execution, the dependence should be set up before `e1` is satisfied. Most of the time, the runtime will manage to set up the dependence before `t2` starts and satisfies `e1`, resulting in correct execution. However, it's also possible that after `t2` is allowed to start, `t1` gets suspended. This could for example be due to the OS scheduler suspending the thread. So, `t2` starts and satisfies `e1`. There is not yet a dependence connecting `e1` and `e2`, therefore `e2` is not satisfied and `e1` gets destroyed. Then, `t1` resumes and tries to add a dependence from the destroyed `e1`, resulting in an undefined behavior (e.g., a crash). There is a race condition among the two operations on the event. The error may be very hard to reproduce, especially if `t2` performs other work before satisfying `e1`. Although OCR-V1 attempts to detect application errors, this error would never be detected, because `t1` would always finish before `t2` can start due to the sequential task execution.

Using *happens-before*, we can clearly see the problem. To make sure that the dependence is set up in time, we need `ocrAddDependence(e1,e2)` *happens-before* `ocrEventSatisfy(e1)`. This is not the case here, only these hold: `ocrAddDependence(NULL,t2)` *happens-before* `ocrAddDependence(e1,e2)` and `ocrAddDependence(NULL,t2)` *happens-before* `ocrEventSatisfy(e1)`.

4 Automatic Checking of OCR Programs

Our approach for checking of OCR programs is based on a OCR-V1, a single-threaded implementation of OCR. OCR-V1 was specifically designed to help debugging by exposing errors through explicit checks (using the standard C `asserts`). There are almost 100 checks like this in OCR-V1. Although they are very useful, these checks are only one of two parts of our system, which is complemented by the tracing functionality of OCR-V1 and the trace analyzer.

4.1 OCR Application Tracing and Trace Analyzer

As we have already shown with the example in the previous section, there are errors that cannot be detected by OCR-V1, since they only manifest when multiple tasks are executed concurrently. To cover these cases, we have extended OCR-V1 to export the list of operations (OCR API calls and implicit operations) performed by the OCR program. Only the operations relevant to synchronization are exported. Furthermore, OCR-V1 exports a subset of the *happens-before* graph that connects the operations. As the *happens-before* relation is transitive, we don't need to export the full graph, but only edges that are sufficient to construct it by transitive closure. The trace is loaded by the trace analyzer, which builds the full *happens-before* relation by performing a transitive closure. Then, it iterates through all OCR objects and checks that they are used correctly (the actual rules to check are described in Sect. 4.3). Rule violations are reported, along with the relevant context, like the file name and the line number of the location where the API call that violated the rule was made.

4.2 The *happens-before* Graph

To make checking the rules easier, the graph exported from OCR-V1 is not directly the graph of OCR API calls and *happens-before* relations among them. We modify the graph by introducing additional nodes and edges. For every operation performed by a task (*cause* node), there is also another node (*effect*) where the changed mandated by the operation is applied to the affected object. For example, when an OCR task invokes `ocrEventSatisfy(e1)`, the effect is the actual satisfaction of the event, which can be denoted as `e1.satisfied()`. The *happens-before* relation is also modified (extended) to ensure that the cause *happens-before* the effect, but also that if there is a *happens-before* relation among two causes, their effects also have this relation. This is achieved by *back edges*, which are edges connecting the effect of a cause to the operation that comes right after the cause. One cause can have multiple effects, for example connecting two events by dependence (`ocrAddDependence(e1,e2)`) changes both events (`e1.connectPostSlot(e2)` and `e2.connectPreSlot(e1)`). This format makes it easier to check if an event `e` is being used properly, as it is enough to check all actions applied to the event – `e.*`.

Furthermore, helper nodes (virtual operations) are added to objects. For example, we add `e.triggered()` to each event, signifying the point in time where the event is triggered. In the *happens-before* relation, this operation follows all satisfactions of the event and precedes satisfaction of all pre-slots connected to the event's post-slot. Also, a `x.destroyed()` node added to all objects that are automatically destroyed. This further simplifies checking of the rules.

Figure 1 shows an example of a graph of operations and their synchronization. The visualized graph corresponds to the example in Sect. 3.3.

4.3 Error Detection Rules

A set of rules are applied to the graph by the trace analyzer, in order to check for errors. We've already shown one example of such rule. For any *once* event, any `ocrAddDependence` call must *happens-before* satisfaction of the event. When viewed as by the effects of the operations, we require that `e.connectPostSlot(x)` *happens-before* `e.satisfy()`. The full list of rules is as follows:

1. Any use of an object must be (as per *happens-before*) between its creation and its destruction.
2. All dependences that start with a post-slot of a *once* or *latch* event have to be set up before the event is satisfied.
3. A *once* event can only be satisfied once.
4. ocrShutdown should be called from a task that comes after all other tasks.
5. Any valid (per *happens-before*) order of increments and decrements of a *latch* event must be correct – it must start with an increment, only reach zero once, and only reach zero at the end.

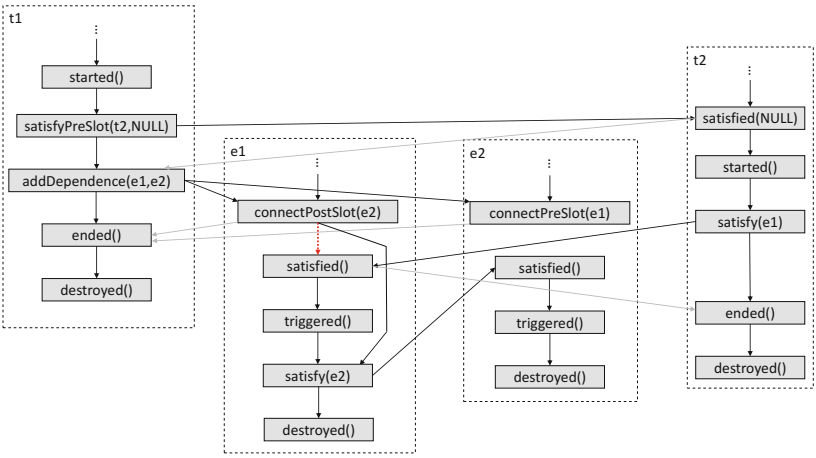

Fig. 1. The trace of the example in Sect. 3.3. Operations performed on two tasks (`t1` and `t2`) and two events (`e1` and `e2`). The black arrows are normal *happens-before* edges, the gray arrows are the back edges, which also contribute to *happens-before*. The red dotted arrow is the missing *happens-before* that would ensure that the event is used correctly. Note that *happens-before* is formed by transitive closure, so the shown arrows are only a subset. But even if transitivity is applied, it would not add the missing arrow. (Color figure online)

The first rule is probably the most important one, as it covers all types of objects and different possible error scenarios. The last rule, which checks *latch* events, is difficult to verify with a large number of increment and decrement operations, as we need to check all permutations of the operations.

5 Examples

To demonstrate the functionality of our tool, we have tried it on several OCR applications[2]. There are not many OCR applications and most of the existing ones have already been extensively debugged, so only very few errors were detected. Our tools are more useful when used by the application developer while the application is still being created, to identify problems as soon as possible.

5.1 Late Dependence Definition

The following code fragment is taken from an OCR tutorial. It is similar to the example given in Sect. 3.3. Two tasks `fill` and `print` are created and the output event of the `fill` task is used as a dependence for `print`, to make sure that `print` runs after `fill`. However, the dependence is added too late, after the `print` task is allowed to start. The task may run in parallel and destroy its output event before the dependence can be set up.

[2] https://xstack.exascale-tech.com/git/public?p=apps.git.

```
//create templates, fill has 1 pre-slot, print has 2
ocrEdtTemplateCreate(&fillTML, fill, 0, 1);
ocrEdtTemplateCreate(&printTML, print, 0, 2);
//create startEVT - an event which launches the computation
ocrEventCreate(&startEVT, OCR_EVENT_ONCE_T);
//create one instance of fill and print each
ocrEdtCreate(&fillEDT, fillTML, 0, 0, 1, NULL, &fillEVT);
ocrEdtCreate(&printEDT, printTML, 0, 0, 2, NULL, NULL);
//set up startEVT as predecessor of both tasks
ocrAddDependence(startEVT, fillEDT, 0, DB_MODE_EW);
ocrAddDependence(startEVT, printEDT, 1, DB_MODE_CONST);
//trigger the computation
ocrEventSatisfy(startEVT, NULL_GUID);
//set up a dependence from the output of fill to print
ocrAddDependence(fillEVT, printEDT, 0, DB_DEFAULT_MODE);
```

The trace analyzer reports the following error message:

```
ERROR: ONC.EVT may be satisfied before all post-slot are added
        Event 18:EVT.ONC-output-of(17:fill)
                satisfied by 73 in epilogue of 17:fill
        Missing happens-before from 52 in 10:mainEdt
                invoked from ocr\apps\app_lab.cpp:75
```

The error message tells us that there is a problem with the event with ID 18. The event is the output event of task 17, which is the `fill` task. The event is satisfied by operation 73, which is one of the operations executed automatically by the runtime after `fill` finished. In the main task (ID 10), the event is used to perform operation 52, which is at the specified line in the source code. This happens to be the last line of the example, where `ocrAddDependence` is called.

5.2 Conflicting Operations in Parallel Tasks

The following program was created specifically to demonstrate our tools. It shows a scenario where multiple tasks contribute to the error. The code shows the whole program, except for includes, function argument lists, and some unimportant arguments in function calls. Besides the `mainEdt` task, which is the entry point of any OCR program, there are three other tasks. Tasks `task1` and `task2` run in parallel. The `mainEdt` task creates a data block (called `data`) and passes it to both tasks. While `task1` only accesses the data block, `task2` destroys it. Task `task3` shuts down the runtime after `task1` and `task2` finish. A task graph for this example is shown in Fig. 2. This figure is generated as a side-effect by the trace analyzer tool (it generates a DOT file for GraphViz [6]).

```
void task1(/*arguments omitted for brevity*/){
 int i = *(int*)depv[0].ptr; //access the data block
}
void task2(/*arguments omitted for brevity*/) {
  ocrDbDestroy(depv[0].guid);//line 10 in the actual file
```

```
}
void task3(/*arguments omitted for brevity*/) {
  ocrShutdown();
}
void mainEdt(/*arguments omitted for brevity*/) {
  ocrGuid_t data,tml1,tml2,tml3,edt1,edt2,edt3,evt1,evt2;
  void* ptr;
  ocrDbCreate(&data, &ptr, 8);
  ocrEdtTemplateCreate(&tml1, task1, 0, 1);
  ocrEdtTemplateCreate(&tml2, task2, 0, 1);
  ocrEdtTemplateCreate(&tml3, task3, 0, 2);
  ocrEdtCreate(&edt1, tml1, 0, 0, 1, 0, &evt1);
  ocrEdtCreate(&edt2, tml2, 0, 0, 1, 0, &evt2);
  ocrEdtCreate(&edt3, tml3, 0, 0, 2, 0, 0);
  ocrAddDependence(evt1, edt3, 0, DB_MODE_NULL);
  ocrAddDependence(evt2, edt3, 1, DB_MODE_NULL);
  ocrAddDependence(data, edt1, 0, DB_MODE_RW);
  ocrAddDependence(data, edt2, 0, DB_MODE_RW);
}
```

When the program is executed and analyzed, the following error is reported:

```
ERROR: operation may be after destruction
        data block 13 destroyed by 78 in 19:task2
                invoked from ocr\src\src\apps\app_lab.cpp:10
        61: acquire in 17:task1 may be after destruction
```

The error message tells us that when the data block 13 (the `data`) is acquired by task 17 (type `task1`), it may already have been destroyed by `ocrDbDestroy` (line 10 of the actual source code), which is in task 19 (type `task2`).

Note that the identifiers of tasks and events are their actual IDs used by the runtime, so when the program was running, the `data` variable in the main task actually contained 13, `edt1` contained 17, etc. However, the identifiers of the operations, like 61 used for the acquire operation, are only internal identifiers of OCR-V1 and cannot be accessed from the application code. As is often the

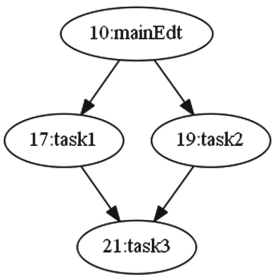

Fig. 2. Tasks and their dependences from the second example. The number is the ID of the task, the text label is the name of the C function which implements the task.

case when debugging programs based on logs, the developer therefore needs to carefully interpret the output to figure out what the operation is. In the case of 78, it is clear from the reference to the source code. To identify operation 61, one has to realize that the data block `data` is acquired by `task1` automatically before it starts, so there is no direct counterpart in the code.

5.3 SPMD Application – Synchronization Using Data Blocks

When we tested our tools on existing OCR applications, it reported a large number of errors in one of them. The application is an SPMD (single program multiple data) code which mimics the way MPI programs work[3]. There are virtual processes which are assigned numerical ranks and they can exchange data by send and receive calls using the rank numbers. Internally, the communication is handled by writing an identifier of the sent data into a so called *channel* data block, which is then read by the recipient. As part of the exchange, the sender creates an event which is satisfied by the recipient when the data is received. The tool reported that the event is being used but there is no guarantee that it's not used before it is created. There is no *happens-before* relation between the code that performs the send and the code that handles the received data.

This is not an error inside our tool. The relation really does not exist. The problem is that if two tasks acquire the same data block in exclusive write mode, no *happens-before* is established among them. However, looking from the outside, it seems it should not be detected as an error. The sender creates the event and then stores it in the channel data block. If the recipient initiates the receive operation (and acquires the channel data block) before the channel is updated by the sender, it does not see the event (it is not there yet), so it does not satisfy it. If the recipient acquires the channel data block after it has been modified, it means that the event has already been created and it can be satisfied. Because both sides acquire the data block in exclusive write mode, the recipient has to see one of the two consistent states.

On the other hand, it is conceivable to implement OCR so that the recipient sees the modified data block but the event is not yet valid. The specification [10] either needs to be updated to require the relation to be established in such cases or developers need to be very careful and avoid such scenarios.

5.4 Performance

As OCR-V1 was designed with safety and not performance in mind, the extra overhead introduced by exporting the graph is noticeable but not a game changer. On a machine equipped with dual core (4 threads) Intel i7-7500U CPU, a highly tuned native OCR seismic simulation code, which executes 768517 tasks, takes 105 s to complete with OCR-V1. The graph data size is around 3.5 GB. Without the graph export, it takes 22 s, almost 5x faster. However, on the same

[3] https://xstack.exascale-tech.com/git/public?p=apps.git;a=tree;f=apps/libs/src/spmd.

machine, a shared-memory OCR implementation takes 2.4 s, improving the performance by another 9x, to a total speedup of around 44x. So, even though the graph export slows the execution down, it is still manageable for an application with hundreds of thousands of tasks.

The trace analyzer needs to explore the transitive closure of the exported *happens-before* subset. We store the closure as a dense adjacency matrix, which results in significant memory usage. The matrix is dense, because the existing OCR applications are often iterative algorithms and a task in iteration i is likely to be synchronized with all tasks from previous and subsequent iterations. The complexity of searching all permutations of n operations on a *latch* event is $n!$. For most other rules, the execution should be roughly n^2 for n operations.

The example from the first paragraph of this section cannot be analyzed in a reasonable amount of time. If we reduce it to just one thousand tasks (this version takes a quarter of a second to finish in OCR-V1), it produces around 48k operations and 77k edges. These can be analyzed in 10 s. However, if we double the number of tasks, the time goes up to over 40 s, following the predicted quadratic time complexity. This makes searching large graphs impractical.

In applications with some regular structure, it is possible to take a small workload and use that to check for errors. For example, the seismic simulation only has three different kinds of iterations (first, last, and all iterations in between) and in each iterations, there are 3 kinds of tasks (top, bottom, in between), so even a small run with 49 tasks in total is sufficient to test all these cases. As we are detecting all potential synchronization errors, increasing the number of tasks will not increase the chance of finding an error, as the process is not at all probabilistic. However, not all applications have a regular structure like this and it may not always be possible to test all cases with such a small sample.

6 Conclusion and Future Work

We have created a tool which can automatically detect synchronization errors in OCR applications, in cases where OCR objects are being used by the application without proper synchronization. While no automatic tool may detect all errors in such applications, any programmer aid is important for the difficult task of writing such programs.

In the future plan to use more sophisticated graph processing techniques to reduce overall memory footprint and processing time. We would also like to be able to efficiently handle common cases of very large *latch* events, without having to search all permutations.

Acknowledgment. The work was supported in part by the Austrian Science Fund (FWF) project P 29783 (Dynamic Runtime System for Future Parallel Architectures) and by VEGA 1/0684/16.

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03869-3_80
2. Bosilca, G., et al.: PaRSEC: exploiting heterogeneity to enhance scalability. IEEE Comput. Sci. Eng. **15**(6), 36–45 (2013)
3. Dokulil, J., Sandrieser, M., Benkner, S.: 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, February 2016
4. Dokulil, J., Benkner, S.: OCR extensions - local identifiers, labeled GUIDs, file IO, and data block partitioning. CoRR abs/1509.03161 (2015). http://arxiv.org/abs/1509.03161
5. Dokulil, J., Sandrieser, M., Benkner, S.: OCR-Vx - an alternative implementation of the open community runtime. In: International Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures, in conjunction with SC15, Austin, Texas (2015)
6. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) GD 2001. LNCS, vol. 2265, pp. 483–484. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45848-4_57
7. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D.: HPX - a task based programming model in a global address space. In: The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS) (2014)
8. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006). https://doi.org/10.1109/MC.2006.180
9. Mattson, T.G., et al.: The open community runtime: a runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2016)
10. Mattson, T., Cledat, R. (eds.): The Open Community Runtime Interface, April 2016. https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf
11. Musuvathi, M.: Systematic concurrency testing using CHESS. In: Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2008, p. 10:1. ACM, New York (2008)
12. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: SIGPLAN Notices, vol. 42, no. 6, pp. 89–100 (2007)
13. Rutar, N., Almazan, C.B., Foster, J.S.: A comparison of bug finding tools for Java. In: 15th International Symposium on Software Reliability Engineering, pp. 245–256, November 2004
14. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: a coverage-driven testing tool for multithreaded programs. In: SIGPLAN Notices, vol. 47, no. 10, pp. 485–502 (2012)
15. Zheng, Y., Kamil, A., Driscoll, M.B., Shan, H., Yelick, K.: UPC++: a PGAS extension for C++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 1105–1114, May 2014