Extending the Open Community Runtime with External Application Support

Jiri Dokulil
University of Vienna
Vienna, Austria
jiri.dokulil@univie.ac.at

Siegfried Benkner
University of Vienna
Vienna, Austria
siegfried.benkner@univie.ac.at

ABSTRACT

The Open Community Runtime specification prescribes the way a task-parallel application has to be written, in order to give the runtime system the ability to automatically migrate work and data, provide fault tolerance, improve portability, etc. These constraints prevent an application from efficiently starting a new process to run another external program. We have designed an extension of the specification which provides exactly this functionality in a way that fits the task-based model. The bulk of our work is devoted to exploring the way the task-parallel application can interact with an external application without having to resort to using files on a physical drive for data exchange. To eliminate the need to make changes to the external application, the data is exposed via a virtual file system using the filesystem-in-userspace architecture.

1 INTRODUCTION

The Open Community Runtime (OCR) [8] is a task-based runtime system for parallel applications on extreme scale/distributed memory systems. In OCR, all work is performed inside tasks, which are serial pieces of code that can be executed independently of other tasks. The tasks can only be synchronized using dependences, which define when a task may start. At runtime, an OCR program is represented as a dynamically generated directed acyclic task graph (DAG), which is processed by a scheduler in order to decide when and on which execution unit (worker) a task should be executed. All data in an OCR application that needs to persist outside a task has to be stored in a data block. To access a data block, it must either be created by a task or passed to the task as dependence before the start of the task.

Tasks decouple work from compute resources (threads, CPU cores) and data blocks decouple data from storage (memory), in order to better deal with performance variability and to improve portability. The complete control over tasks and data blocks gives the runtime interesting opportunities, like applying sophisticated scheduling mechanisms, or providing resilience. The runtime may restart failed tasks or maintain redundant copies of data, allowing it to recover from node failures. However, these extra capabilities come at a price. The whole application needs to be written using OCR. There is no conventional `main` function. In OCR, even the `main` is a task. It's not possible to write an application in the traditional way and only use the OCR for certain performance critical parts.

Another limitation is the requirement that the tasks are non-blocking. This means that once a task starts, it should run to completion without waiting for other tasks. Furthermore, blocking operations like long-running I/O should be avoided. In the existing OCR implementations [4, 9], if a task blocks, it effectively eliminates one thread from a fixed pool of worker threads, possibly resulting in under-utilization of CPU cores. One example of such operation is running an external program using the `system` call. As reimplementing all parts of the application in OCR may be difficult, it would be helpful if some code could be left outside, compiled independently into an executable, and then invoked from the OCR application.

Such extension to the OCR API is quite natural, since the external application can be viewed as a special kind of task – it has clearly defined inputs, outputs, and once started, it is expected to run till completion on its own. The inputs and outputs may be passed as files, without any involvement of the OCR runtime. However, it might be more efficient to directly use OCR data blocks for this purpose. As we expect the external application to be outside OCR and not modified for this purpose (the source codes may not even be available), the only way to make the data available is to expose it as files. Since the original motivation was to avoid writing the data out as files, we have used the File System in Userspace (FUSE) library available in Linux, which allows easy implementation of custom file systems, without having to write any code running in the kernel space.

The result of this design might not necessarily be increased performance. If a file is written by an application and immediately read by another one, there is a very high chance that the data will still be cached by the operating system, making the read operation very fast. What our approach does is keep the data in resources managed directly by the OCR runtime, rather than relying on automatic caching done by the operating system. This opens up new possibilities for improved scheduling strategies and novel optimizations.

Still, as performance is an important concern, we have executed several experiments, evaluating the performance of the proposed solution in comparison to using traditional files for input and output. As expected, the performance of reading a cached file is better than our solution, especially as it does not yet include most of the available low-level optimizations. However, the performance is more predictable and stable, as was demonstrated in one scenario, where the traditional file-based solution was much slower. Also, if we also consider the time needed to write the file, the file-based solution is slower.

The rest of the text is organized as follows. First, relevant concepts of the OCR specification are described in Section 2. Then, the OCR API extension that allows external applications to be executed as tasks is described in Section 3. Sections 4 and 5 describe the design of our solution in detail. An experimental evaluation is presented in Section 6, followed by related work in Section 7. Section 8 concludes the paper and discusses future work.

2 OPEN COMMUNITY RUNTIME

The OCR specification defines the API and expected behavior of an OCR runtime. At the moment, there are three implementations: the reference OCR implementation created by Intel and Rice [9], an implementation (derived from the reference implementation) created at PNNL [7], and our implementation, which is called OCR-Vx [3]. Our work is done on OCR-Vx, although it could be also applied to the Intel/Rice implementation. This is probably also the case for the PNNL implementation, but unlike the other two, it is not publicly available.

To provide proper background for our work, we need to explain some components of OCR in greater detail. Beside tasks, an essential part of the OCR design are data blocks. A data block is an OCR object used to hold data. The contents of a data block (the actual data) are a contiguous array of bytes with a fixed size. The runtime does not interpret the contents of a data block in any way. Any structure is defined only inside the application. To access a data block inside a task, the data block needs to be passed to the task as an input dependence before the tasks starts, or it needs to be created by the task itself during its execution. When the task dependence is set, an access mode also needs to be specified. The access mode defines how the data block can be accessed concurrently from multiple tasks. There are four access modes:

- `CONST` – constant mode, where the contents of the data block is guaranteed not to change while the task is executing. Naturally, the task itself also cannot change the data.
- `EW` – exclusive write, where the task may change the data and it is guaranteed that no other task may be changing the data while the task is running.
- `RO` – read-only, non-coherent, where the task cannot change the data, but the data might be changed by other tasks.
- `RW` – read-write, non-coherent, where the task may change the data and the data may also be changed by other tasks. The other tasks need `RW` access (not `EW`).

The runtime is allowed to keep multiple copies of a data block and relocate the data, as long as the consistency model[1] is maintained and the running tasks are not affected. When a task starts, it is provided with pointers to all data blocks it has been given access to (via dependences) and these pointers need to remain valid as long as the task is running or until a data block is explicitly released by the task. So, for example, if there are two concurrent tasks (i.e., no particular order is enforced by their dependences), one with `EW` access and the other with `CONST` access to the same data block, the runtime may create a copy of the data block's data, give the copy to the second task and let the first task modify the original data block. The copies may even be distributed across several nodes in a cluster.

The design of data blocks makes programming OCR applications more complicated. The two main reasons are the necessity to provide all necessary data as dependences to a task and the fact that pointers are only valid within a single task. As soon as the task finishes, the data block may be relocated, invalidating the pointers. On the other hand, this gives the runtime greater flexibility, allowing it to migrate tasks and data automatically, but also making fully transparent fault tolerance possible. The runtime may make backup copies of data blocks and restart failed tasks from older copies in case of failure.

Another OCR concept that we should briefly mention are events. OCR tasks and dependences form a task graph, which is a DAG. To extend the type of synchronization patterns that can be expressed by the DAG, a new kind of object was introduced – an event. Events are nodes in the task graph that do no computation and only influence the synchronization of the application. There are several types of events, each with a straightforward set of rules that define their behavior. The simplest example is a `once` event, which waits for a satisfaction signal on its single input, forwards it to all outputs, and is automatically deleted. The other types are not used in our examples, so we would like to refer the kind reader to the OCR specification.

The interaction between the OCR application and the OCR runtime is defined by the OCR API – a set of C functions that the application may call to create and delete data blocks, tasks, and events, set dependences, query the number of workers etc. To allow OCR applications to also invoke external applications, we need to extend the API.

3 API

Conceptually, it's easy to integrate external applications into the OCR execution model. They are self-contained units of work with clearly defined inputs and outputs. On the actual API level, some care needs to be taken, due to the differences between normal OCR tasks and the tasks that represent external applications. Dealing with inputs is easy, as OCR tasks already assume they have a number of inputs, which are satisfied with data blocks, giving the tasks access to these data blocks. However, an OCR task only has one output – there is an output event associated with the task and if the task code returns an identifier of a data block, the data block is passed to the event and any task connected to the event receives the data block. In practice, these output events are mostly just used for synchronization, as tasks often need to output several data blocks. In this case, the task uses the OCR API directly (within the task code) to assign data blocks to the respective receiving tasks as dependences. The external application has no such option. Therefore, it is not possible to use exactly the same interface to create normal OCR tasks and tasks that correspond to invocation of external applications.

To create a task in OCR, the application first needs to create a task template using `ocrEdtTemplateCreate`. The template contains the pointer to the C function that implements the task, the number of input dependences and the number of parameters (simple numeric parameters passed to the task). Then, the template (via its identifier) is used to create a task with `ocrEdtCreate`. We've decided to follow this pattern, defining a way of creating an external task template and then the external tasks themselves.

A new function `ocrExternalTemplateCreate` has been introduced to create the template, expecting an identifier of the application, the number of inputs, and the number of outputs. To identify the applications, we have decided to use string identifiers, which

[1] OCR uses a relaxed consistency models. In a nutshell, to ensure that a task can see changes made by another task, they need to be properly synchronized. There has to be a path in the task graph from the writer to the reader.

are then used to look up the configuration details for the application in a stand-alone configuration file. The file defines how the application is started, the way data is passed to it, and how the results are obtained. For example, the following configuration file defines an external task `convert`, which has two inputs and one output. The first input is passed as a file, providing the file path as the `--in` argument. The second input is interpreted as an integer and the value is passed to the command line. A file name for the output is generated and passed via the command line. The data written to a file with that name will be returned as a data block.

```
[convert]
command=./convert --in=[in:0] --out=[out:0]
        --size=[arg:int:1]x[arg:int:1] --fast --level=3
```

Upon execution, this would be transformed to a command that looks like this:

```
./convert --in=/mount/ocr/db/523 --out=/mount/ocr/db/641
   --size=128x128 --fast --level=3
```

This assumes that the data block passed as the first input dependence is exposed as the file `/mount/ocr/db/523`, the data blocked passed as the second input contains the integer 128, and the external application is expected to write the result to `/mount/ocr/db/641`, which will then be available to the OCR application as the first output.

The configuration file may contain any number of entries with unique names. Once a template has been created, it can be used multiple times to create external tasks using `ocrExternalTaskCreate`. There are several differences compared to the `ocrEdtCreate` call, but the main one is an additional input argument that contains an array of event identifiers. There have to be as many events as there are outputs of the external task and each event will be satisfied with the data block from the corresponding output of the external application. Consider the following code example:

```
//create template for the external convert task
ocrExternalTemplateCreate(&convert_template, "convert",
     2, 1);
//array of output events, just one in this case
ocrGuid_t outs[] = { NULL_GUID };
//create the output event
ocrEventCreate(&outs[0], OCR_EVENT_ONCE_T,
     EVT_PROP_TAKES_ARG);
//connect the output event
ocrAddDependence(outs[0], sink, 0, DB_MODE_RO);
//create a task - an instance of the convert template
ocrExternalTaskCreate(&convert_task, 2, 1, outs);
//add inputs to the new task
ocrAddDependence(data,convert_task,0,DB_MODE_RO);
ocrAddDependence(size,convert_task,1,DB_MODE_RO);
```

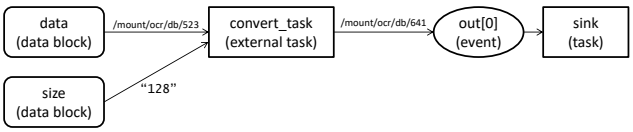

Figure 1: An example with a single external task with two inputs and a single output, which is forwarded by an event to a normal OCR task. A new data block will be created from the data written into the `/mount/ocr/db/641` file and passed to the `out[0]` event when the `convert_task` finishes.

In this example, a new external task `convert_task` is created, provided with input data (stored in data blocks `data` and `size`) and an event (stored in `outs[0]`) is used to pass the result to an existing task `sink`. As is the usual practice in OCR, the output event needs to be connected before the inputs are provided by satisfying the dependences of `convert_task`, because the external task may start as soon as the dependences are satisfied. If the output was connected after the inputs are satisfied, the task could finish before the output is connected, in which case the `out[0]` event would be satisfied and destroyed automatically, which would make it illegal to use the event as a source of a dependence. A graph that shows these objects and their dependences is depicted in Figure 1.

To make the API complete, `ocrExternalTemplateDestroy` and `ocrExternalTaskDestroy` are provided to delete templates and (unstarted) tasks.

4 RUNTIME ARCHITECTURE

Each OCR application runs as one process, with an instance of the OCR runtime present in each of the processes. Each external task is also started as a standalone process. Another necessary process is a FUSE module, which exposes data blocks as files to the external application. Only one process is used to serve all external applications. We will refer to the module as OCR-FUSE in the following text. When the OCR-FUSE module is started, a path has to be provided. This path defines where the newly started virtual filesystem is to be mounted. In the example from the previous section, this path would be `/mount/ocr`. When the OCR-FUSE module is running, the FUSE kernel module (`fuse.ko`) ensures that any I/O performed on the mounted directory is sent to the OCR-FUSE module through the `libfuse` library, which must to be linked to the OCR-FUSE module.

The architecture is shown in Figure 2. The OCR runtime spawns the application processes. Two different communication channels are used to communicate between the OCR runtime: a control channel and a data channel. The control channel is used to define which files should be used to expose the OCR data blocks to the application and to further synchronize the two processes. The ZeroMQ (ØMQ) communication library is used for this purpose, but there are many other viable alternatives, including raw sockets. The data channel is used to move data between the data blocks managed by the OCR runtime and the FUSE module.

When the external applications reads the file `/mount/ocr/db/523`, the `libfuse` tells OCR-FUSE to read `/db/523`. Earlier, the control channel was used as part of the invocation of the external application to notify OCR-FUSE that a certain data block should be made accessible as `/db/523`. As a result, OCR-FUSE knows to use the data channel connected to the OCR runtime that registered the `/db/523` data block to read the data of the data block. A similar process is used to write the results – the OCR runtime informs OCR-FUSE that `/db/641` will be used to write the output of the application, so that OCR-FUSE can redirect writes made to the file to the appropriate OCR runtime. Reading and writing of unregistered files can be considered an error or treated as a temporary file and redirected to `/tmp`.

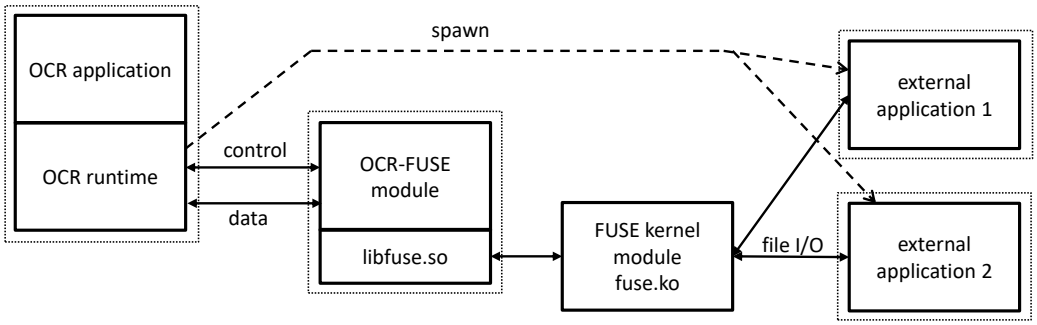

Figure 2: The architecture of the system, showing different components used to run two external applications from a single OCR application. The dotted boxes represent processes. The FUSE kernel module runs in the kernel and it is not a process.

This setup makes reading and writing to the files backed by OCR data blocks indistinguishable from reading and writing normal files on a physical device. Therefore, no special support is needed from the external application, as long as a path can be specified for the input and output files. Any reasonably designed application should support this. However, it would also be possible to use the FUSE file system to duplicate the contents of an existing filesystem and only redirect selected files to the OCR data blocks. We do not support this at the moment, although FUSE would make this modification quite easy. The remaining open question is how to exchange data between the OCR runtime (which holds the data blocks) and OCR-FUSE.

5 DATA BLOCKS AS FILES

The filesystem implementation in FUSE needs to provide several functions. Most of them deal with managing the filesystem – listing directories, creating and deleting files, opening files, etc. To read and write the contents of a file, two functions need to be provided: `read` and `write`. Aside from some identification of the file, these get a pointer to the buffer with data to be written or where the read data should be stored, the size of the requested read/write operation and an offset into the file. What we need for the `read` call is to read the part of the data block starting at the specified offset and store the required number of bytes into the buffer provided. The `write` call is more complicated, since it is also possible to write beyond the end of the "file", in which case the file is expected to grow to accommodate the newly written data.

As the FUSE module is a separate process from the process that hosts the OCR runtime and the OCR application, it does not have a direct access to the data block that is to be exposed as a file. Several ways (protocols) of providing this access are available:

(1) One of the standard Linux two-sided IPC mechanisms can be used to send data between the two processes, for example sockets, pipes, or message queues. However, moving the data that is read and written by the external application this way may cause significant overhead. In these cases, the data is copied at least twice before it gets from the data block into the FUSE-provided buffer. A fixed size shared memory region could be used to exchange the data, ensuring that the data is copied exactly twice.

(2) The data block could be exposed as shared memory, which would then be mapped in the OCR-FUSE module. In this case, the data is copied just once – from the mapped memory into the buffer. The downside of this approach is the requirement that the data block needs to be in a shared memory area. We could make all data blocks this way or move exposed data blocks into a newly allocated shared memory when needed, but both options would increase overhead of the OCR runtime. Alternatively, we could require the OCR application to state that a data block needs to be in a shared memory when it is being created. This moves the responsibility to the application developer and it may be difficult to always figure out which data blocks may eventually end up being shared.

(3) As of Linux kernel 3.2, two new functions are available: `process_vm_readv` and `process_vm_writev`. These allow a process to read and write to the memory space of another process, requiring just the identifier (PID) of the target process, address, and size. This way, the OCR runtime only needs to let the FUSE module know the addresses of the data blocks and the OCR-FUSE module can then use the functions to directly read the data block contents, requiring only a single copy. This is exactly the kind of scenario for which the new functions were introduced into Linux.

Options 2 and 3 should both provide good performance of data exchange, with the option 3 being the better choice, as it avoids the requirement of placing data blocks in shared memory regions. On the other hand, it relies on functionality that may not yet be universally available. Even though Linux 3.2 was released in 2012, some Linux distributions being used today use an older version. For example, RedHat Enterprise Linux 6 uses kernel version 2.6. Even though RedHat Enterprise Linux 7 with kernel 3.10 has been available since 2014, some systems still run on the older release. Also, the new functions are specific to Linux, they are not part of the POSIX standard. It may be interesting to note that the Windows operating system provides similar functionality via `ReadProcessMemory` and `WriteProcessMemory`. With Dokan (Windows equivalent of FUSE), a similar solution can be deployed on Windows.

These options only cover the case where the external application reads data from an OCR data block. We also need to deal with two other cases. First, the application may also modify data in the data block, possibly attempting to enlarge it by writing beyond the end of the file. Second, the application may create new files to store the output. In both cases, the size of the file may be increased and the final size is not known in advance. For simplicity, consider only the case of a newly created file. Compared to the input data, even more options are available:

(1) Allocate memory for the new file inside the OCR-FUSE module and reallocate the buffer (using the exponential growth strategy) when the file needs to grow. As it is not possible to reallocate shared memory, the data would have to be copied to a data block inside the OCR runtime at the end. This means copying the data in memory once or twice, as discussed earlier regarding the input data. Even though the reallocation could be done by moving data from one shared memory block to a new (larger) memory block, this would most likely be less efficient, as the system `realloc` call has a chance to just grow the existing memory, with no copying required, or the copying could be eliminated by the operating system through the use of virtual memory, by mapping the physical memory pages with the old data to the virtual addresses in the new memory block.
(2) Allocate memory for the new file inside the OCR-FUSE module as shared memory and add new buffers (again, increasing the size exponentially) if it needs to grow. This way, the data is available to the OCR runtime directly. However, as data blocks in OCR need to be contiguous, the result would either have to be copied into a single data block or the result would have to be returned to the OCR application in multiple data blocks corresponding to the buffers.
(3) The previous option could be used, but with the data blocks allocated as shared memory inside the OCR runtime and mapped inside the OCR-FUSE module. This way, it would be easier to manage the lifetime of the data, as the OCR-FUSE module could unmap the block as soon as the file is closed. On the other hand, a protocol needs to be established to direct allocation of the data blocks in OCR from the FUSE module. This is not a problem, as those two need some communication channel anyway.
(4) It is also possible to allocate the file as a single buffer inside the OCR runtime and then reallocate it when the file grows. In that case, the block cannot be in shared memory, for the same reasons as in the first option. Also, the data needs to be copied to the buffer inside the OCR runtime. However, it can be directly copied from the source, which is a buffer provided by the FUSE library when it makes the `write` call. Using `process_vm_writev`, it's possible to copy the data from the FUSE buffer to the final destination (the data block) with just a single copy. If the function is not available, at least two copies are needed, for example using a shared, fixed-size shared memory area.

Overall, the best option for both input and output data is to store all data inside the OCR runtime, in contiguous data blocks. Then, use `process_vm_readv` and `process_vm_writev` to copy the data into FUSE-provided read and write buffers, whenever a read or write operation is performed. However, if the fact that a data block will be used as input of an external application is known in advance, shared memory should be just as efficient. Furthermore, if the size of the output files can be determined in advance, shared memory should also be an efficient solution for the output data. Or, if the application can handle it, the outputs could be split into multiple data blocks. However, as the tools, libraries, and programming models for writing OCR applications are very limited at the moment, we would expect this option to be mostly avoided.

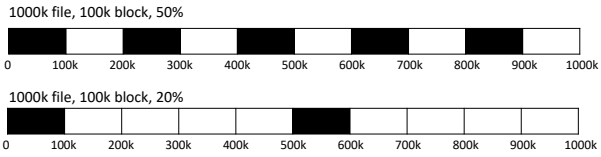

Figure 3: The read patterns used in the benchmark. The black boxes correspond to the blocks being read.

6 IMPLEMENTATION AND EXPERIMENTS

The proposed solution was implemented in OCR-Vx [4]. However, as the performance of the virtual filesystem is the interesting part, we have isolated it as a separate application, so that we can run experiments with minimal interference. The experimental setup is the same as Figure 2, except the OCR application and OCR runtime are replaced with a specialized application whose only job is to provide the data. The external application is also specifically designed to test the performance. It only works with the file system, using the standard file functions (`fopen`, `fread`, etc.). To test input performance, it reads a configurable fraction of the whole file, reading blocks of a specified size evenly distributed across the file. The blocks are read in sequence. For example, with a 1000k file, 100k blocks and 50% fraction read, it reads a 100k block at offset 0, then a 100k block at offset 200k, and so on, until it reads the final 100k block at offset 800k. Examples of the read pattern with 50% and 20% coverage are shown in Figure 3.

The experiments were performed on two very different systems. The first machine is a Linux server with two Intel Xeon X5680 CPUs (6 cores and 12 MB cache per CPU), with 24 GB RAM. The operating system is Red Hat Enterprise Linux Server 6.8, with kernel 2.6. Therefore, the `process_vm_readv` and `process_vm_writev` functions are not available on the machine. The second system has a single Intel Core i5-3210M CPU (2 cores, 3 MB cache), 8 GB RAM, and it runs Windows 10 Pro 64bit. We have used the machine directly, but also in a virtualized environment (Hyper-V) based on Docker. In this setup, the experiments were executed in a Docker container, with the Docker itself running in a Linux virtual machine (kernel 4.4.20) with 2 CPUs and 2GB RAM. Although it would be possible to run the experiments directly on the virtual machine (without Docker), we have used Docker to simplify the setup of the machine environment (compilers and libraries) and to obtain a more diverse set of experiments. The library versions used in the different setups were the following: FUSE 2.8.3, ZeroMQ 4.2.2 on the Linux server, FUSE 2.9.7-1 and ZeroMQ 4.2.2 in the virtual

	10%	20%	50%	100%	full
Linux, file	0.0035	0.0067	0.0154	0.0242	0.0413
Linux, shmem	0.0077	0.0174	0.0363	0.0619	0.0840
Linux, readv	N/A				
Windows, file	0.0058	0.0075	0.0217	0.0477	0.0664
Windows, shmem	0.0151	0.0290	0.0850	0.1444	0.1714
Windows, readv	0.0171	0.0333	0.0850	0.1659	0.1788
Docker, file	0.0023	0.0050	0.0104	0.0484	0.0493
Docker, shmem	0.0092	0.0203	0.0402	0.0696	0.1226
Docker, readv	0.0101	0.0196	0.0409	0.0706	0.0884
Docker, host file	0.0565	0.1030	0.3881	0.4780	0.4328

Table 1: Performance results. The table shows execution time in seconds for different configurations, reading 128 MB file, using 1 MB chunks. The last column shows performance if the whole file is read using a single `fread` call. We use "file" to refer to the alternative where an actual file is used to store the data, while "shmem" and "readv" both refer to OCR-FUSE, with the two different options for moving data from the OCR runtime to the OCR-FUSE module. The last line shows performance if the file is stored on the host of the virtual machine. Note that on Windows, `ReadProcessMemory` is in fact used instead of `process_vm_readv`. We could not use `process_vm_readv` on the Linux server, as the kernel on the machine is too old and does not provide it.

Linux, and Dokan 1.0.4 and ZeroMQ 4.2.1 in the native Windows setup.

To test and compare read performance, three different tests were performed: reading a real file that has just been written to disk, reading a data block (via FUSE) available as shared memory, and reading a data block using `process_vm_readv` or `ReadProcessMemory`. All results are averages of 10 executions.

The results for a 128 MB file, 1 MB chunks, and 10%/20%/50%/100% file read are shown in Table 1. The table also shows the performance if the whole file is read using a single `fread` call. On the Linux server, writing the data to a file and reading it in the external application is about 2x faster than OCR-FUSE. This shows there is definitely room for further optimization. At the moment, we are using the high-level FUSE interface. The low-level interface might be able to perform faster. On Windows, the results are similar, with the plain file being 2.6x to 4x faster. The difference could be a result of using Dokan instead of FUSE. Shared memory and `ReadProcessMemory` provide very similar performance. In the virtual environment, the results are also similar, with plain files being 1.4x to 4x faster.

Interestingly, if we store the file not in the container's temporary filesystem, but in a directory on the physical host machine via the Docker virtual file system, the results are very different, with OCR-FUSE significantly outperforming the file-based solution by 5x to 9x (shown in the table as "Docker, host file"). Clearly, the reason for this is not extremely good performance of OCR-FUSE (although it is noticeably faster than the native Windows alternative), but very bad performance of reading the normal file. The conclusion to be made here is not that OCR-FUSE is superior, but that this demonstrates the potential benefits of tighter control over resources and data movement that it provides. The performance is more stable and predictable, compared to the alternative that depends on the caching done by the operating system. This allows better resource management and more accurate scheduling decisions.

An important consideration in all the experiments is that we do not count the time necessary to write the file and only consider read performance. In the FUSE-based solution, there is no file to write, but if a physical file is used, it has to be written at some point. Using a single `fwrite` call, the time necessary to create the 128 MB file on different systems looks like this: Linux 0.0818s, Docker 0.0848s, Windows 0.7915. These times almost always exceed the time needed to read data via OCR-FUSE and if we add together the time to write and read the file, it is always slower than OCR-FUSE. However, it may be possible to hide the time necessary to write the file in the OCR runtime by overlapping it with other computation, so we don't consider it in Table 1.

We have focused on the input data which is read by the external application. As for the other direction, the main difference is the need to potentially increase the size of the memory used to store the result in the OCR runtime. As reallocation can be handled by current operating systems very efficiently, it is not a major concern. For example, in the Docker environment, if we start with a 4 KB buffer and grow it to 128 MB using `realloc` to double the size in each step, the total time required is around 0.0002 s. Writing the data either through shared memory or `process_vm_writev` is the same as when the file is read, only the direction is reversed, but as the two processes are equivalent, it requires the same effort irrespective of the direction. The main difference would be the baseline we compare against, because as we have just discussed, writing a file to the physical filesystem is considerably slower.

7 RELATED WORK

Most programming environments provide a way to run an external application. In task-based runtime system, there is often the issue of wasting a thread from the thread pool to wait for the application to finish. It may be possible to work around this. For example, in the Intel Threading Building Blocks [6], it would be possible to start a new thread to wait for the external application to finish and once that happens, it's possible to manually start a task which depends on the result of the execution. This is not possible in OCR, as the application is not supposed to start new threads.

In the .NET environment, process execution is not directly provided as an asynchronous task (the native task-parallel model of .NET), however it is possible to implement it with the existing library functions. For example, `RunProcessAsTask` is provided here: https://github.com/jamesmanning/RunProcessAsTask. However, this only deals with the execution of the task, not providing a way to pass data to and from the task without using files. Technically, it would also be possible to expose memory as files in .NET, in a similar way that we do, although it would not be possible to expose the managed memory directly using shared memory or `ReadProcessMemory`.

StarPU [1], another task-based runtime system, could be extended to provide external tasks, in a very similar way that we did in OCR. In StarPU, the data (the *buffers*) is also managed by the runtime, so it could be exposed the same way that we do with OCR. Similar modifications could be made to other task-based runtimes,

like OmpSs [2] or HPX [5]. However, unlike in OCR, with less strict constraints placed on the application, it would also be also possible to implement the functionality directly in the application and not in the runtime.

In general, it's possible to expose the memory of any process as files with FUSE using the same techniques that we use. Furthermore, on Linux, memory of a process `$pid` is exposed by the pseudo-file `/proc/$pid/mem`. However, to be able to read memory of another process, the reader (in our case the FUSE module) would have to attach to the process like a debugger would (using `ptrace`), stop the process, and detach after reading the data to restart the stopped process. This might be acceptable in some cases, especially as the process to be stopped is probably waiting for the external application to finish. However, it would not be possible to allocate and reallocate memory for the output this way.

8 CONCLUSION AND FUTURE WORK

Our proposal shows that OCR can be extended with the ability to invoke external applications in a way that fits well with the existing OCR design philosophy. The FUSE library can be used to expose data of the OCR application to the external application in an efficient way that requires no changes to the external application. This gives the OCR runtime grater control of the resources used by the applications. Our experiments have shown that the performance of such design is roughly 2x to 4x slower compared to using a normal file. This assumes ideal circumstances for the file storage, where it is fully cached by the operating system and the time necessary to write the files is not considered. In some scenarios, OCR-FUSE can already provide better performance.

In the future, we plan to further optimize the system, both on the OCR level and in the FUSE module. Once the optimizations are done, we will run a much larger set of experiments, assessing the performance under realistic application scenarios, using a combination of file reads and writes, also including memory mapped files. Also, it would be interesting to see how the file-based solution works if a ramdisk is used as storage, rather than relying on caches.

We plan to integrate the external application execution into the task scheduler of the OCR runtime. At the moment, an eager execution strategy is used, running the external applications as soon as the data blocks used by them can be acquired. We would also like to further investigate the option of running the external applications inside Docker containers. External applications that use accelerators (like GPUs) also pose an interesting challenge.

Another interesting possibility created by OCR-FUSE is the fact that even while a data block is being accessed by the external application via OCR-FUSE, the data block can still be available to tasks running inside OCR. This access can fully observe the access modes that the external and normal tasks use for the data blocks. For example, a data block may be concurrently read by multiple external applications and OCR tasks in RO mode, while it is being modified by external applications or tasks that acquire the data block in RW mode. The OCR memory model and the OCR-FUSE design make this possible.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience; Euro-Par 2009* 23 (2011), 187–198. Issue 2.

[2] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. 2012. Productive Programming of GPU Clusters with OmpSs. In *IPDPS 2012 Parallel Distributed Processing Symposium*. https://doi.org/10.1109/IPDPS.2012.58

[3] Jiri Dokulil, Martin Sandrieser, and Siegfried Benkner. 2015. 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.*

[4] J. Dokulil, M. Sandrieser, and S. Benkner. 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)*. 364–368. https://doi.org/10.1109/PDP.2016.81

[5] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX - A Task Based Programming Model in a Global Address Space. In *The 8th International Conference on Partitioned Global Address Space Programming Models (PGAS)*.

[6] Alexey Kukanov and Michael J. Voss. 2007. The Foundations for Scalable Multi-Core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 04 (2007), 309–322.

[7] Joshua Landwehr, Joshua Suetterlein, Andrés Márquez, Joseph Manzano, and Guang R. Gao. 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, NY, USA, 164–171. https://doi.org/10.1145/2903150.2903166

[8] Tim Mattson and Romain Cledat (Eds.). 2016. *The Open Community Runtime Interface*. https://xstack.exascale-tech.com/git/public?p=ocr.git;a=blob;f=ocr/spec/ocr-1.1.0.pdf.

[9] T. G. Mattson et al. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761580