Orchestrating real-time containers: on the possibilities of schedule manipulation

Florian Hofer

September 10, 2019

In this document, an exploratory step by step investigation on process management and scheduling techniques are discussed.

1 Introduction

Emerging technologies such as the Internet of Things and Cloud Computing are reshaping the structure and control of industrial processes radically. The extent of these innovations, which allow the creation of highly flexible production systems, is so wide that today we talk about a fourth industrial revolution. Key enabling technologies such as distributed sensing, big-data analysis and cloud storage are taking the center stage in developing new industrial control systems.

The control of industrial processes, however, has not changed much over the last few decades, and there are reasons for it. For instance, typical control applications found in industrial processes have to respond to changes in the physical world within predefined time limits. Moving the execution of control tasks from devices physically co-located with the controlled process to cloud or fog computing platforms requires dealing with network delays that are difficult to predict. Moreover, while bare-metal solutions lend the control design full authority over the environment in which its software will run, it is not straightforward to determine under what conditions the software can be executed on cloud computing platforms due to resource virtualization properties. Yet, we believe that the principles of Industry 4.0 present a unique opportunity to explore complementing traditional automation components by a novel control architecture [1].

We believe that modern virtualization techniques such as application containerization [2, 1, 3] are key for adequate utilization of cloud computing resources in industrial control systems. The use of containerized control applications would yield the same advantages that traditional containerized microservices present: light and easily distributable control applications would be able to, for instance, run on any system and at the same time, be easily maintained and updated. Thus, enhanced scalability, portability, updatability and availability can be achieved [4]. Beyond the migration and resource savings, parallel operation of containers on devices such as PLCs and, in small amounts, on sensing and actuating field devices, is possible. This will increase reliability and robustness, while enabling further exploitation of Self-* properties including Self-awareness and Self-compare.

In this paper, we explore the feasibility of managing real-time control applications running on a shared resource environment. The contributions of this paper are:

- Evaluation of access and control techniques to vary a container's run parameters and constraints, including scheduling information;
- Exploration of implementation techniques of a task, which takes charge of automating those settings and constraints, further called orchestrator;
- Evaluation of monitoring and implementation of a monitoring interface;
- Static scheduling of application containers as real-time tasks;

The rest of this paper is structured as follows. Section 2 analyzes related work, while Section 3 introduces the problem and the environment. We next investigate how an orchestration and resource management might be performed analyzing a variety of options in Section 4, and perform a test implementation of the chosen method in Section 5. Finally, we document a first software implementation Section 6 and conclude in Section 7.

2 Related work

Containerizing control applications has been discussed in recent literature. Morga et al. [2], for instance, presented the concept of containerization of full control applications as a means to decouple the hardware and software life-cycles of an industrial automation system. Due to the performance overhead in hardware virtualization, the authors state that OS-level virtualization is a suitable technique to cope with automation system timing demands. The authors propose two approaches to migrate a control application into containers on top of a patched real-time Linux-based operating system:

- A given system is decomposed into subsystems, where a set of sub-units performs a localized computation, which then is actuated through a global decision maker.
- Devices are defined, where each component is an isolated standalone solution with a shared communication stack. Based on this, systems are further divided into modules, allowing a granular development and update strategy.

The authors demonstrate the feasibility of real-time applications in conjunction with containerization, even though they express concern on the maturity of the technical solution presented.

In related work, Goldschmidt and Hauk-Stattelmann [3] presented a similar solution. They perform benchmark tests on modularized industrial Programmable Logic Controller (PLC) applications to analyze the impact of container-based virtualization on real-time constraints. As there is no solution for legacy code migration of PLCs to this point, the migration to application containers could extend a system's lifetime beyond the physical device's limits. Even though tests showed worst-case latencies of the order of 15ms on Intel-based hosts (this may prevent direct application), the authors argue that the container engines may be stripped down and optimized for real-time execution. In a follow-up work of Goldschmidt et al. [5], a possible multi-purpose architecture has been detailed and the proposal tested in a real-world use case. The results show worst case latencies in the range of 1ms for a Raspberry PI single board computer, making the solution viable for cycle times in the range of 100ms to 1s. The authors state that topics like memory overhead, containers' restricted access and problems due to technology immaturity are still to be investigated.

Tasci et al. [1] address architectural details not discussed in [3] and [5], such as the definite run-time environment and how deterministic communication of containers and field devices may be achieved in a novel container-based architecture. They propose a Linux-based solution as host operating system, including both the single kernel preemption-focused PREEMPT-RT patch and the co-kernel oriented Xenomai. With the latter patch, this approach shows better predictability, although it suffers from security constraints. For this reason, they suggest limiting its application for safety-critical code execution. They analyze and discuss in detail inter-process messaging, focusing on the specific properties needed in real-time applications. Finally, they implement an orchestration run-time managing intra-container communication and show that task times of $500\mu s$ are possible.

The three solutions discussed above share one common property: they were based on bare-metal configurations. The solutions took into consideration real-time constraints, but limited to execution on physical hardware. Nonetheless, current trends in industry favor the flexibility of resource sharing through cloud computing. Thus, there is reason to investigate whether real-time control applications may also be ran on a shared infrastructure, their capabilities and limitations.

In 2014, Garcia-Vallas et al. [6] analyzed challenges for predictable and deterministic cloud computing. Even though the focus of their paper is on soft real-time applications, certain aspects and limits can be applied to any real-time systems. Merging cloud computing with real-time requirements is a challenging task; the authors state that the guest OS has only limited access to physical hardware and thus suffers from unpredictability of non-hierarchical scheduling, and thick stack communications.

So far, we have seen that containerization techniques have been tested with early positive results in a variety of contexts. However, a combination of containers executed on cloud resources and control application containerization has not yet been examined in the literature, left alone the management and orchestration of the latter. In this paper, we assess the feasibility of a centralized resource orchestrator for a static set of real-time enabled containers.

3 Context and problem

With the rising needs of a more distributed system, the requirements for a running control tasks change as well. Different from before, the new software architectures foresee a distributed deployment of single functionalities, requiring the nodes to work together all in sync for a production process to work properly. The new setup also puts systems and architectures in front of new development requirements. Software modules are now to be exchanged and updates in a distributed manner as the functionality itself is not located in as single place anymore. Thus, requires for a change in controlling and management of these software modules, at the cost of additional complexity to increase maintainability and ease of use of such systems.

The trend to implement the single software modules into containers as isolated threads running in parallel on shared resource, reduces the hardware cost and increases software lifetime. This has the advantage of abstracting the controller implementation from its hardware host, making maintenance and redundancy quite easy to achieve. However, these containers are not made to run with hard real-time constraints by default. Therefore, some changes at the operating system level as well as some additional orchestration have to be performed in order to reduce latencies and augment the level of obtained determinism. The running containers may then be orchestrated statically, knowing all task, their actual execution time and periods, or dynamically where, as the tasks enter and leave, new schedules are computed on demand. However, if the maximum peak execution time of one task is not known, the schedule of containers is undecidable.

The containers are run using Docker this time and are started either by `runc` or docker's own `run`. If an `init` binary is specified, it will run as process number 1 allowing to capture and forward signals. Each container has a different PID space and userspace than the parent system. The mount points are separate and virtualized. If we need to share the host's PID space, e.g. for the orchestrator, we can specify the flag `--pid=host`. Until a real application image and data is available, the all-purpose software `rt-app` will be used to simulate the behavior of a set of real real-time containers. The configuration of RT-app is done via a JSON file. The file enables the specification of execution phases, including amount, type and duration of the computing simulation tasks. For the purpose of out project, each container will have only one task. Thus, the configuration will contain only one entry for thread and a global configuration. Times defined in the configuration are in μs. The configuration sample of the test container can be found in the git repository, inside the folder `prj/test_cont`.

The IEEE Std 1003.1 defines the interface every compliant operating system must have. The standard also foresees, in addition to the main process management features, an optional X/Open System Interfaces XSI conformance. The constants that define the presence and function of such interfaces might be found in the ¡unistd.h¿ header file. The values can be directly interrogated by verifying the constant as described in the standard. XSI defines in particular three option groups: encryption, realtime (threads) and advanced realtime (threads). As verified, only a few of the advanced realtime features are available in a standard Linux System. Some conditional compiler flags may be used in the source code for a more detailed output of versions and capabilities. Interestingly, the POSIX standard does not define other scheduling types than `other`, `fifo` and `RR`. There is no mentioning of `deadline`(EDF), `iso`, or `batch`. At the same time, the standard foresees a structure defining the scheduler's parameters, but it contains only one parameter, the absolute scheduling priority. This might need to be taken into consideration when trying to run the orchestrator in a non Linux environment.

4 Manipulating processes

In this part we are mainly interested in manipulating process priorities and scheduling behavior. The POSIX calls and standards are investigated to verify possibilities.

4.1 PID namespaces

Before considering option of sharing PID space with host, see if we can access from a privileged container nonetheless the namespace isolation. Posix defines functions ioctrl_ns to determine the parent namespace of the actual one. can help getting the inode of a parent namespace and read information. If PIDs can also be read is a matter of investigation. Maybe it can be mount as procext? What information can be read out from the pid is still to be investigated.

Manuals say that a parent can read child namespaces but not vice-versa. All inquiries of parent PID in fact once reached pid 1 (must not be 1 for a container) end up with a 0 response.

4.2 PID information

While in UNIX the PID table and information is part of the kernel memory, in linux the only way to access the process information is by parsing virtual files. If the orchestrator has to run on both systems, the PID retrieval must thus be implemented twice. Once via /proc pipe readout and once via sysctrl calls to read kernel memory. It is highly likely that the same is true for the other resources, such as memory or connectivity. An example code has been tested which gives the PID's of all matching process names. It might therefore be possible to poll once and verify cyclically for unbound containers. A hook-like structure might be more convenient. Such modification would require a kernel change and is therefore considered as a backup solution for to follow in a second moment.

4.3 Scheduling of processes

As default there is only limited access to the scheduling activities of a process. In the POSIX documentation we find the following functions:

- **sched_get_priority_max(int)** gets the max priority of the specified scheduling policy. On Linux, it is 0 for SCHED_OTHER and 1 for everything else.
- **sched_get_priority_min(int)** gets the min priority of the specified scheduling policy. On Linux, it is 0 for SCHED_OTHER and 99 for everything else.

- **sched_getparam(pid_t, struct sched_param *)** gets the scheduling parameters of a specified process ID
- **sched_getscheduler(pid_t)** gets the set scheduling policy for a PID
- **sched_rr_get_interval(pid_t, struct timespec *)** gets the length of the quantum used when using the round robin scheduling approach for realtime. With a Linux kernel, the round robin time slice is always 150 microseconds, and pid need not even be a real pid. At least that is what the manuals say. Interrestingly a test has shown that there are also 8000 or 16000 μs sometimes returned when querying the value.
- **sched_setparam(pid_t, const struct sched_param *)** sets the parameters for a specified process ID
- **sched_setscheduler(pid_t, int, const struct sched_param *)** sets the scheduling policy for a PID
- **sched_yield(void)** return the control to the scheduler (leave the execution state).

In addition, if the scheduling is set to traditional, i.e. SCHED_OTHER, the niceness value for the complete fair scheduler can be adjusted with additional functions. POSIX as said, implements only three scheduling policies. Linux in addition has now:

```
    #define SCHED\_BATCH 3
    /* SCHED\_ISO: reserved but not implemented yet */
    #define SCHED\_IDLE 5
    #define SCHED\_DEADLINE 6
```

The newly introduced scheduling attributes are now extended as:

```
struct sched\_attr {
    __u32 size;

    __u32 sched\_policy;
    __u64 sched\_flags;

    /* SCHED\_NORMAL, SCHED\_BATCH */
    __s32 sched\_nice;

    /* SCHED\_FIFO, SCHED\_RR */
    __u32 sched_priority;

    /* SCHED\_DEADLINE */
    __u64 sched\_runtime;
    __u64 sched\_deadline;
    __u64 sched\_period;
};
```

And might be read and written through two Linux specific system calls which are specified as follows:

```
int sched_setattr(pid_t pid, struct sched_attr *attr,
    unsigned int flags);

int sched_getattr(pid_t pid, struct sched_attr *attr,
    unsigned int size, unsigned int flags);
```

This new data structure is an addition to the original sched_param which has been kept for standardization and backwards compatibility issues.

The commented scheduling mode in the list, i.e. SCHED_ISO, is an algorithm intended for isochronous (burst-like) tasks and has not been completely implemented yet. The brainfuck scheduler (BFS) developed by Con Kolivas is alternative version witch actually supports the ISO scheduling mode. The new processes will have a priority between RT and Normal/Other tasks. For further details see "A complete guide to Linux process scheduling" by Nikita Ishkov.

4.4 Other considerations

During the exploration of the namespace and PID function, some features to be implemented in future where discovered. The PID 1 inside the container is parent of all the namespace. If it dies, all the children are orphaned and will be killed. As parent, it is also target of sigs without implicit binding, thus it must manually handle incoming signals such as sig_int. The userspace in the container can be mapped. To avoid root as well as other accesses from a container, a proper map should be set up for all the containers at load.

5 Test implementation

The orchestrator will have to interact with the scheduler while monitoring the tasks. Ideally, the interaction is mutual. Unfortunately, no hint has been found suggesting that there is a way to "register" to the scheduler to get scheduling and process updates. Thus, apart from the system calls for getting and setting a process's scheduling attributes, there is no direct communication with the kernel. This means that the PIDs of all the processes must be queried manually, cyclically, considering time trends and thus the orchestrator must keep PID history.

5.1 PID search

In a first implementation test, we verify therefore the poll quality and speed of pid queries. As described in Section 4.2, there are two ways of polling the process table. In our testing environment we use Linux, and are therefore running for a virtual file-systems parsing. To ease the parsing, we can use the integrated tool "pidof" via a pipe opened through the call of popen. popen is a POSIX standard call that opens a one-way pipe to indicated purposely process called via shell. The processes can either be identified directly or via its parent process. Every container is in fact starting a daemon PID called 'docker-containerd-shim'. This eases grouping and process identification and is preferred to the previous solution. Another option is to scan the configured control groups for PIDs. Every container in Docker gets assigned its own control group. Therefore, each of these entries contains the list of PIDs running inside a running container. The latter technique is simpler and allows direct identification of containers. All three modes will be implemented in the final version, leaving the choice of operation either to the user or to automatic detection.

As soon as the orchestrator is up and running, it scans for target processes. The PID list is parsed and verified for matching process signatures or containers. Valid PIDs will be stored in memory for further processing. The whole process takes a fraction of a ms and does not put load on the CPU. With unlimited loop the CPU-load is below 3% (example taken from pidof).

An option to optimize process verification might be that the environment controlling the containers informs the orchestrator about containers entering or leaving the scheduler. This way, an orchestrator does not have to scan for new PID all the time. Kubernetes, for example, foresees the use of postStart and preStop handlers when installing a pod. The handlers can be specified in the yaml configuration file of the POD for each container. A Typical use for this might be the shutting down or preparation for a service such as a web server. In our case we could use it to signal a process start or stop. More details about this configuration can be found here. The Docker API on the other hand does not implement a pre- or post execution handler for container start or stop. Only the container configuration itself could contain a run command as part of the container to be executed upon startup. This might be also a standard configuration setting, signaling the orchestrator the presence of a new container process. Anyway, this will be a feature to be implemented in a second moment.

The test development is stored in git under src/testPosix/thread.c.

5.2 PID list

Once the PIDs have been acquired, the orchestrator has to maintain a list of active processes. For this purpose, and in order to match PIDs with scheduling information, a dynamic memory structure may be used. In order to store the information, the sorted resulting PID number array is sequentially used to fill a dynamic structure with scheduling data. Cyclically, the memory structure will be compared with the updated PIDs. New processes will be inserted in order in the structure, removed PIDs will be dropped. The actual type of dynamic data structure is irrelevant. Possible structures are linked lists, double linked lists, as well as sorted trees of any kind.

To test an implementation, a simple linked list data structure has been implemented as a separate source file src/testPosix/pidlist.c. The test program structure has been rewritten. Now two threads are launched to keep track of the running processes. Thread 1 will keep track of PIDs, add and remove list elements. The resulting PIDs array of a periodic scan is compared to the list's content. Thread 2, implemented in a second step, will take care of the distribution of resources among the listed PIDs.

The test implementation of Thread 1 works fluently. For the final version of the orchestrator, Kernel list based structures should be considered. The standard Linux linked lists use predefined macros for add and delete in the list. Differently from the managed list, this implementation does not cover the memory management of the structure. Memory allocation and freeing must still be done by the programmer.

5.3 PID scheduling

As a final step for the basic PID scheduling, we gather the information of a target process and set the new attributes. Resources, initially only CPU-affinity, are distributed according to configured settings, i.e. max values for resource occupation. This is for example a 80% boundary for each computation unit (CPU). The processes and its running parameters might be defined in a configuration file (*cmdfile*). If not, the real-time parameters of the process are read and stored. The daemon and other system tasks should be mapped to separate resources. Even though real-time tasks have higher priority, a separate CPU might be needed to keep enough boundary for new entries and maintain the system responsive. Affinity selection can be done via the functions `sched_setaffinity` and `sched_getaffinity`. In addition, a variety of auxiliary functions are also available to manage those. The macro commands starting with `CPU_*` may be used to create and modify CPU-Set masks, needed to change a process's affinity.

A CPU has a standard cyclic computation monitoring unit of 1 ms, separating it into 1000000 ticks. The different run-times, deadlines and periods of the periodic tasks will be used to find a common denominator on a 1ms basis to create a proper configuration file. The program will then inspect entering and leaving processes, and bind them to the specified resources. In the program the new entries found by thread two will be checked for affinity and resources. Once the task information has been gathered, the updated values will be set according to the actual configuration. During this phase, thread two will also read actual values and monitor the execution cap of tasks.

Thread 2 with be responsible for this task. Cyclically, and in a second moment by trigger, the new PIDs are configured to associate with specific resources. As this might introduce some resource peak, the reallocation must happen as soon as possible.

The new process entry might use any of the resouces and thus add some unwanted peak load. If possible, the daemon and consequently the new created container processes should be bound to a reserved CPU, where all the system tasks are also allocated.

5.4 Parameters and configuration

Parameter configuration can be done in multiple ways. The two major options taken into consideration are parameter file and command line parameters. The latter might work for a test phase, but might get to be a burden once the system and configurable parameters grow in size.

For some first tests, command line parameters will be used to set PIDs, their affinity, global thresholds and limits. Those will later be moved into a configuration file. The parameters include Ulimit, CPU masks of reserved computing units, memory maps and I/O resources that might be configured. For the implementation, the posix `getopt` parsing function will be used. To parse the configuration file, a predefined library will be used where possible.

When working with strings and buffers, such as configuration files, particular attention must be given to the possibility of buffer overflows and alike. This is a general weakness as documented in the CWE database. Finally, the combination of configuration file containing all process details and the command arguments, needed for example to select configuration file and set verbosity, might be the best option.

5.5 Logging and internal monitoring

RT-app includes a logging function to keep track of all tasks activity during execution. The logging itself writes to a file without rotation or buffers. This thus does not allow to use the default logging function for continuous monitoring purposes. If a ring-buffer is set in the configuration settings, the buffer is written to file only once the test is finished or interrupted. Thus, an adaptation of the logging function might be needed to allow readout of the task's performance. The source code is available and written in C. Changes should therefore be no major problem.

Some minor changes to the logging function have been successfully tested. Insertion of the ring-buffer dump every time it's capacity has been reached and overwriting of the log-file works as expected. Issues that might be encountered are the high load to disk I/O if a small disks buffer size forces to write to file, and consequently induces delays. An option to consider might be the writing to the stdout or a named FIFO buffer and piping the process to the Icinga plugin.

5.6 Icinga plugin

To monitor the app, the interaction of orchestrator and the monitoring software Icinga is foreseen. Icinga is a host-client system and relies on daemons and plugins on-site to exchange data. To reduce the overhead for the system, the orchestrator will share the obtained process details from the scheduler with the plugin running on our real-time host. In addition, Docker creates CGroups for the running containers, giving the monitoring process access to useful network and memory information. Also, possible might be to read disk I/O of a container, even though it could be that it is not necessary. The Icinga plugin made for Docker available online has a few limitations. In particular, it is limited to standard values such as CPU usage and network throughput. Additional statistics, such as deadline miss or monitoring of process execution itself might be acquired via a modified version of the plugin.

The instructions for a Nagions plugin, the open source parent of Icinga, are used to create the proper plugin output. The test development is stored in git under src/check_dockerRT/.

5.7 Thread and process communication

Finally, we have to specify how threads and tasks may communicate. The communication between the running software can be performed in different ways:

1. we might use the usual IPC, as it is done between processes via
   a) pipe - a connection between in and out, therefore not easily doable as the threads by default share the same stdin and stdout, can be used only for data streams;
   b) fifo (or named pipe) - similar to pipes, but rely on the file system to be able to connect two arbitrary processes in the system, may be used to talk bidirectionally;
   c) signals - such as *USR1* or *USR2* which might be sent between processes, but hardly doable on the same process but different threads. The process would need to signal itself and the amount of information exchanged is contained int the signal itself;
   d) sockets - might be used as a medium where both threads access externally to this resource and exchange information, can be used for streams and datagramms;
2. shared memory - probably the easiest and fastest type. The advantage in case of threads is that they share memory. Thus the only additional feature we need are locks/semaphores. Then, data can be exchanged in shared locations or FIFO buffers.

The easiest approach for the orchestrator's threads is to use shared memory and a semaphore/mutex. On the other hand, sockets and fifo pipes are more flexible. Thus, the latter will be used for the communication of orchestrator and monitoring plugin.

In our test implementation we add a pthread_mutex_t to be acquired before every access to the PID list. This way it is possible to instantiate a simple mechanism of intrinsic communication, without having to specify any additional data and protocol. The program changes are minimal and thus the code remains as readable as before. A fifo buffer, or a named pipe, will be used to pass run statistics to the planned Icinga plugin. The pipe is simply opened creating a virtual file descriptor and connect both communication partners. Management and information passing is done as if it were a file.

The two threads can manage the list independently as before. On every execution the actual thread has exclusive access to the list. This way, thread one can drop any removed item simply from the list and add new PIDs once discovered. Thread two has to scan the PID list anyway. If new items are found, the settings are populated and resources allocated and scheduled.

6 The orchestrator

In this section we detail function, implementation and tests. First, we discuss the assumptions taken for the implemented features. We then develop a static orchestrator and finally detail the test setup and configuration we use for performance tests.

6.1 Function principle

In [7] we assessed the feasibility of real-time tasks in the cloud. Now we address the system efficiency of this bounded execution instead. If we place these tasks into containers, we can increase their efficiency by managing and sharing the latter's resources. We will use an *orchestrator* and the needed algorithms to perform these tasks.

The orchestrator's primary goal is to allocate container resources to maintain determinism at best. To achieve this, it has to confine containers, and their running processes in their execution context. We have seen in Section 4 a variety valid techniques to obtain such isolation. Section 5 used these techniques to explore their appliance to check, maintain and monitor running containers. In the following we will describe the final implementation and its functions.

Each container will need three main resource allocations. It needs computing power to proceed with the algorithm, memory to store and manipulate data, and I/O to interact with the environment. If we start now by isolating computing resources, we can define the following relationships. A computation resource has a maximum utilization factor (limit) U_l of 1. The utilization factor for each processor has a value defined as the sum of run-times divided by their period, or

$$U = \sum_{i \in T} \frac{c_i}{p_i} \leq U_l = 1 \tag{1}$$

where T is the set of all tasks, p_i the period and c_i the computation time. When we schedule our tasks, we try to maximize U without exceeding it (over-subscription). However, to guarantee determinism we have to account for the worst execution for each period, thus $c_i = w_i$. This means that the worst case execution time (WCET) w_i of a task sets out the maximum amount that can be reclaimed or shared. If then this worst case is rare, this results in a high "waste" of resources. Our orchestration tool will try to optimize out this loss while moderating risks.

Ideal resource management is hard to guarantee. To ease a process' allocation and resource assignment, we start with some simplifying assumptions. First, we suppose that the real-time software running in the various containers is all independent. We will consider processes as black boxes where we do neither have specifications, nor source code. If we do not consider interdependence of the threads, we might simplify the scheduling algorithm to operate on multiple single core units. With dedicated memory and I/O, each unit has its own U_l of 1. Consequently, allocation might reach theoretical utilization rates converging to 69% for Rate Monotonic (RM) scheduling and up to 100% for Earliest Deadline First (EDF) [8]. Second, for our first implementations we assume knowledge of all the data of all possible real-time tasks. For simplicity, task deadlines track their period. We have thus the run-time values of WCET w_i, period and deadline ($p_i = d_i$). Given these two assumptions we perform the implementation of a first version, the static orchestrator.

The static orchestrator is the simplest of the three and works with a plain logic. It knows all real-time tasks that have to run, and on which resources they have to run. Using an offline computation, we write schedule parameters to a file. Once running, the orchestrator prepares the environment and reads the container configurations. It scans for running containers and if found, the algorithm assigns the configured resources to the new container. A successful implementation allows us to proceed to the next iteration and concludes this first part of the project.

The adaptive static orchestrator is a variant of the first program. It comes into play if we still know all existing applications, but not which of them actually runs or will run. This requires a more complex algorithm. To ease the implementation, we change the former solution in the following form. Instead of static assignments, the offline parameters set preferences. If the preferred resource is not available, the orchestrator binds the container to the best fitting resource pair. The preference list will contain more entries than resources might allow, but we do not expect all of them to execute. To reduce dynamism, we limit the maximum allowance of shifting containers to the set preference, and the determined best fit. If all containers would be allowed to shift to other resources, the schedule would get dynamic and undecidable again. A correct implementation of this version opens the path for the most complex of the algorithms. The advantage of this version is that we can configure all possible containers that can run on all nodes of a group of servers upfront. There is no need to specify in detail, and offline, which is running where. This allows a container management tool to dynamically decide the node that will run a new container.

An example of such a management tool is Kubernetes. Kubernetes can manage nodes, pods (groups of containers) and decide deployment and resource allocation (count). In a master's thesis, a student uses labels to configure and allocate real-time containers to capable nodes only.[1] Therefore, also migration and node mixing is possible. What Kubernetes cannot carry out, is monitoring resources in real-time and with no delay. As long as container allocation remains static, this makes no issue. If we have to reallocate resources dynamically, a real-time monitor gets mandatory.

Without any additional assumptions, dynamic scheduling is undecidable. Therefore, in this project we will implement it at best using a variation and the following additional considerations. All tasks have a static, adaptive configuration. A parameter that changes is that we no longer set execution peak times as WCET. To reduce the amount of unused resources, we reduce the programmed WCET to a much lower value, w_{i_c}. Based on a maximum assumed risk, we retrieve the WCET from a task's probability distribution of execution times. We get these run-time values and probability distributions through experimental measurements. Those measurements give us an average ($\bar{c}_i$) and peak value ($c_{i_{max}} = \max c_i = w_{i_c}$). The closer we set our new w_{i_c} to the average $\bar{c}_i$, the

[1] "Real-time container clusters for seamless computing", Fariz Huseynli, Master Thesis TUM, September 17th, 2018

higher is the risk of exceeding it. A buffer time may hold for this, offloading the risk to a contemporaneity factor. In addition, monitoring the execution of tasks will allow the orchestrator to predict if the total computation time may exceed the maximum allowance. If so, some of the tasks closest to next in line will be rescheduled onto a different resource. The re-scheduling uses the adaptive static scheduler algorithm of the previous version.

This approach has one advantages. Before a task used the resource slot between average and peak value only with a low probability. If we reduce the assigned WCET, we free part of this slot to the use of other tasks. This increases resource utilization at a cost of a probability P to miss a deadline. Let us illustrate it with an example. If we have nine tasks with WCET $10ms$, period p_i of $50ms$ and $\bar{c}_i$ of $5ms$, a static scheduler cannot allocate them. The total utilization factor U reaches a value 1.8, way beyond the limit U_l. A probabilistic solution, yet, can handle this situation. We can put all tasks WCET w_i to $5ms$, resulting in a total utilization factor U of 0.9. The 10% margin ($5ms$) can be used by any of the tasks as buffer (U_B) to bridge the gap between $\bar{c}_i$ and WCET (w_{i_c}). The risk depends on the contemporaneity factor, i.e., the probability that two or more tasks exceed their average value at the same time, and the sum of their overtime is more than 10% of U.

Based on this example, we can now lie out the following. Setting w_{i_c} closer to $c_{i_{max}}$, the probability π_i of a task to exceed the set w_{i_c} decreases according to the PDF measured for the task. The relation in equation 2 gives the amount of buffer available for a task. It is the "left-over" computation time, decreasing the higher a task's w_{i_c} is set. Or put differently, if we set w_{i_c} closer to the average than to the peak, the resources freed will not be used with a probability $1 - P$.

$$U_B = 1 - U = 1 - \sum_{i \in T} \frac{w_{i_c}}{p_i} \tag{2}$$

Equation 3 describes the amount a tasks frees for resource sharing. c_{i_δ} gives also the maximum amount of buffer time the task i may reclaim.

$$U_{E_i} = c_{i_\delta} = \frac{w_i - w_{i_c}}{p_i} \tag{3}$$

If the buffer big enough to accommodate more than one exceeding tasks, the probability of failure reduces. At the same time, the amount of buffer reclaimed per task can also vary ($\bar{U}_E$). Even if there is only buffer enough for one task to fully run for a duration of WCET, there is space for multiple tasks exceeding w_{i_c} a little. A set of tasks that in sum do not exceed U_B are thus handled properly by the EDF scheduler. The overshoots will be accounted to the buffer until there is none left. Only then the GRUB algorithm preempts the task. The task set remains thus schedulable with a probability of failure P and resource savings as long as it complies with the mentioned parameters, where P is the sum of the random variables of the probability distributions. These savings expressed as utilization factor are defined as:

$$U_S = \sum_{i \in T} \frac{w_i - w_{i_c}}{p_i}. \tag{4}$$

If we apply the ceiling operator to the sum of all savings of all installed processing units, we get the number of CPUs that have been saved with the configured probability setting.

6.2 Test configuration

Theoretical aspects aside, the orchestrator created in this exploratory study needs also to be tested. A set of tests allows us to validate both the orchestrator and the various system configurations. For these tests we first prepare a set of systems with recent kernels and patches. We dispose then a set of test configurations to validate different running scenarios. The rest of this section details these steps further.

For the test results we get in this section, we reconfigured both AWS T3/C5 instances and the bare metal server used in [7] with the latest LTS operating system release, Ubuntu Server 18.04.2. In addition, we upgraded the kernel of all systems to their latest available version close or equal to 4.19.50, and patched it with the latest *PREEMPT-RT* release, $4.19.50 - rt22$. For the installation and update steps we used the same build scripts of [7] with only some minor changes. These changes should reduce interruption frequency and duration, significant details when trying to measure system introduced runtime noise. To increase preemption performance, we reused the configurations of the latest available official kernel releases (4.19.50 and 4.15.0-1035 for bare metal and AWS versions, respectively) as a template for the compilation process.

In these kernel configurations we changed the scheduler base tick settings in some test batches. Our system runs with dynamic ticks enabled (`CONFIG_NO_HZ_IDLE`) for the first round of test batches. In combination with high resolution timers (`CONFIG_HIGH_RES_TIMERS`), this enables higher scheduling granularity and fractional allocation of computation time. During this first round, the scheduler adapts the rescheduling of tasks to a minimum for each core with `CONFIG_HZ` as upper bound. The second batch runs with full ticks, a second recompiled kernel forcing scheduler evocation every millisecond (flag `CONFIG_HZ_FULL`). Other settings used during the performance tests such as scheduler isolation and RCU backoff are set as boot parameter and do thus not require kernel recompilation.

All test use the EDF scheduler including the Greedy Reclamation of Unused Bandwidth (GRUB) server shipped with the Linux kernel since version 4.13. The latter is an improvement to Constant Bandwidth Servers (CBS) used to manage asynchronous tasks in a real-time setting. In fact, the CBS algorithm is unable to manage WCET overshoots and task misbehaviour; for instance, an EDF real-time task going to sleep would risk starving a system. The GRUB algorithm, yet, is able to manage such situations and preempt the starvation causing task. In addition, it can reclaim the unused bandwidth of EDF tasks and reallocate it to other EDF processes, permitting them to continue over their WCET without forcing them to throttle.

Another advantage of the EDF scheduler is the ease to determine feasibility of a schedule. Each task i in an EDF schedule requires three parameters: programmed worst case runtime w_{i_c}, deadline d_i and period p_i, where the relation is as follows:

$$w_{i_c} \leq d_i \leq p_i \tag{5}$$

For a generic periodic task, the run-time w_{i_c} is set equal to the WCET w_c, p is the repetition period. The deadline d is the relative point in time since period start where the computation must be finished. During run-time, the scheduler will use these settings to compute the absolute deadline D_i and the absolute period end P_i based on the absolute start time S_i of the new period. A partitioned or affinity managed system can furthermore be considered equal to a set of isolated subsystems. If now the deadline of a task d_i is equal to the period p_i of a task, a schedule consisting of n tasks running on one of the partitioned subsystems is feasible if

$$\sum_{i \in n} \frac{w_{i_c}}{p_i} \leq 1 = U_l, \tag{6}$$

which also equals the theoretical utilization limit U_l [8].

The ideated tests consist of a set of preliminary boundary tests, and a simulation test. In particular, we have created the following preliminary test settings:

- *Test lower bound:* homogeneous period and run-time among all containers executing on the same resources with a WCET w_i smaller than the best case scheduler's wake-up granularity ($1000 \mu s$);
- *Test upper bound:* homogeneous period and run-time among containers executing on the same resources, with a runtime to period ratio ($\frac{t_i}{p_i}$) close to 0.5.
- *Test diversity:* mixed periods and run-times each container executing on the same resources.

As a conclusive test, we emulate the scenario of our initial problem statement. The parallel operation of multiple instances of the flow control software is verified creating an additional test set. The test configuration foresees a homogeneous period and run-time among all containers on the same resources, with timing based on the values of Section "Problem Statement" in [7](first edition) ($10ms$ run-time, $100ms$ deadline and period).

For the first test batch execution, we set up one resource configuration for each of the tests discussed in the previous section. The first group consists of ten containers with a WCET w_i of $900 \mu s$. With a period and deadline of $10ms$, this results in a utilization factor U_l of 0.9. This configuration represents a scheduling challenge as the best case kernel interrupt setting (HZ_1000) limits scheduler's refresh rate to $1000 \mu s$. The latter exceeds the container's WCET, forcing the kernel to more than 1000 scheduler invocations per second, continuously using the high resolution timer slicing feature. The second configured group includes two containers with $2.5ms$ WCET and $5ms$ period. Although this configuration has only two containers, scheduling consists a challenge in a relatively small period. Group number three consists of a mixed set of containers: one container set to $2.5/5ms$, one $900 \mu s/10ms$, and one configured as $3/9ms$ for worst case computation time and deadline/period, respectively. The orchestrator manages configuration and resource affinity automatically based on a static scheduling file. The last test in the batch is the simulation of the parallel operation of multiple containers running the flow-control program.

The execution of tests will be supported by test tools and scripts. We describe in Section 6.2.1 the tool we used for the application simulation and test. Furthermore, we created two bash scripts to support sequence and container management. Section 6.2.3 describes script details while Section 6.2.2 explains the used execution parameters of the orchestrator. We will detail tests results and conclusions thereof after each implementation description.

6.2.1 rt-app

We perform the resource efficiency tests by placing a higher number of real-time applications on the same set of resources. For this purpose, we used the real-time test software *rt-app* [**?**] to create configurable dummy applications. *rt-app* allows a variety of simulated actions, from pure computation loops, to memory and I/O operations, as well as synchronization simulations. We place those applications individually into containers and run them on shared resources. The configuration of each of the containers specifies their running periods and computation times. For simplicity, we set the relative computation deadline d_i to the same as the period p_i in all tests. In addition to the worst case runtime w_i, the configuration requires a simulated runtime parameter t_i which is the supposed effective runtime of the simulated task. As the number of loops to perform is a constant value set at startup, the actual execution time depends solely on interaction with the system. *rt-app* tracks the absolute elapsed time between computation start and computation end. Monitoring the resulting value will therefore allow detection of execution jitter and capture of eventual deadline miss caused by the tested system configuration.

To emulate real-time computation, *rt-app* uses a loop performing exponential dummy computations. Usually, to relate loops to time, a calibration phase measures the needed time to execute one loop of this computation. This result determines the number of loops required to reach the desired run-time, t_i. Due to granularity and precision issues, the resulting application run-time might differ slightly. The measured performance ranged between 17 and 40 nanoseconds with a accuracy of 1. This makes it hard to predict the resulting execution time and either under-perform or overshoot at every cycle. As example, a rounding error in pretest 1 could cause the test program to perform about 3000 loops less than required. To reduce this issue, we applied a patch to the calibration procedure of *rt-app*. Instead of an average loop time in ns, for loop times lower than $1\mu s$ rt-app now stores the average execution time of 1000 loops in ns. In addition, to align loop execution period among all containers and avoid anomalies, we perform the calibration in advance and pass it as a parameter to all instances of *rt-app*. The code changes have been commited to GitHub and will be listed for a pull request.

During test execution, the orchestrator and *rt-app* monitor run-time behavior of the system. *rt-app* creates a log file detailing each execution cycle to be inspected for possible deadline failures. The orchestrator reports scan times and run-time budgets. It can thus see anomalies in container behavior and report it. We detail the execution parameters for the orchestrator in the next section. The test scripts, the source code and a detailed description of the orchestrator can be found in the project repository, and in [9].

6.2.2 Orchestrator settings

For our tests we used the following parameters for the orchestrator:

- the affinity parameter is set either to 1 or to 1-2. The first set of tests uses only one real-time CPU, thus the former. All other sets go for multicore testsl;

-f the force flag is needed to allow the orchestrator to reconfigure the system. This is to ensure that all options such as Hyper-threading are disabled;

- the run-time of our tests is limited to $900s$;

- we set the update thread policy of the container scan. With a default rate of $5000\mu s$ we use this setting to have a more reliable execution and continuous. The chosen policy is `FIFO`;

- the container deadline based threads can reclaim any unused bandwidth. This feature thus permits a task to grab all the free CPU-time there to not miss a deadline;

All other parameters are set to default. In case of exceptions, we detail the additional parameters in the sections following.

6.2.3 Execution and automation scripts

During the tests we used three automation scripts. The first one automates the container management. It systematically applies all commands to the configured containers for our test suite. The usage is very similar to docker itself and thus we do not detail it further. The second script contains all the tests to be executed, the preparatory and the final steps. It is the main script that is run when starting a test batch. The last script manages kernel settings, updates the boot manager and reboots the system when needed. It is the launching script and performs multiple batch runs with varying parameters. The varying parameters thus are: scheduler isolation, full dynamic ticks and RCU fallback CPU numbers in combination with standard PREEMPT_RT of full dynamic tick enabled kernels. Again, the test scripts can be found in the project repository, and in [9].

6.3 Static scheduler - structure

We implement the static scheduler as the first of our solutions for this project. The solution divides into three primary code "*.c" files, each of them with a specific function.

The main file, named "schedstat.c" prepares for execution. It configures the system and verifies if the running process has all required capabilities. If successful, the main thread starts the linked threads and waits for interruption or termination.

Among its duties is also to test what detection mode is available. The latest version of the static orchestrator includes four detection modes:

- via CGroup settings;

- using the parent PID of the container daemon;
- comparing the signature of the command line;
- reading the Docker event server output and reacting to it.

The default selection is the first in list, while the event server detection is always active. If CGroups are not available, the orchestrator switches down the list to the next detection mode.

In preparation for execution, the orchestrator identifies environment parameters and configures the kernel. In particular, the RT-containers are isolated via *CGroups* from system tasks. A separate control group called `system` is created and before allocating containers, all possible system or user tasks are moved from the root into the new group. The CPUs are then divided in two sets and allocated exclusively to either `system` or the running containers. The division and size of the CPU-sets can be configured via command line parameters. In addition, at start-up, the orchestrator sets the maximum real-time run-time to -1. This is to avoid throttling of RT tasks and to enable CPU affinity for EDF scheduled processes.

The first of the *main* started threads manages updates. It scans for containers and keeps their run-time parameters current. A linked list serves as a shared data structure among all threads. The scan executes periodically with a preset scan time of $5ms$. By default, every 100^th scan the thread verifies if there are new containers present. The main thread parses these parameters from the CLI input at start-up. If the thread interrupts or stops, it prints a list of statistics for each real-time capable process. We placed the source code for this thread in “update.c” and the used data structures are part of a developed shared static libary. The second started thread keeps track of resources. Its goal is to monitor resource and set optional run-time parameters. A shared library has been created for tasks such as interaction with the kernel, reading JSON files or managing data structures. Further detail can be found in the manual inside the project repository.

This first implementation of the tool is simple and reduced in functionality. Although this simplicity, a set of performance tests will show if it allows some resource savings and to which extent. Test execution and performance test results are part of a near future paper submission. Please refer to it for more detail.

7 Conclusions

In this report we have seen the concept of resource orchestration to increase hardware utilization and reduce operational costs. Via a implementation example we have seen how static resource management is viable and opens the door to more complex orchestration scenarios. We discussed algorithm variants, their advantages and possible difficulties. As a follow-up we plan to implement and test the adaptive and dynamic variants of the orchestration software.

References

[1] T. Tasci, J. Melcher, and A. Verl, "A container-based architecture for real-time control applications," in *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*. IEEE, jun 2018.

[2] A. Moga, T. Sivanthi, and C. Franke, "OS-level virtualization for industrial automation systems: Are we there yet?" in *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC 16*. ACM Press, 2016.

[3] T. Goldschmidt and S. Hauck-Stattelmann, "Software containers for industrial control," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2016.

[4] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, sep 2016.

[5] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *Journal of Systems Architecture*, vol. 84, pp. 28–36, 2018.

[6] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, oct 2014.

[7] F. Hofer, M. Sehr, A. Iannopollo, I. Ugalde, A. Sangiovanni-Vincentelli, and R. Barbara, "Industrial control via application containers: Migrating from bare-metal to iaas," 2019. [Online]. Available: https://www.florianhofer.it/papers/?id=5238198429fe9f5ffb540b91b221a7d71136034c

[8] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[9] Test archive. [Online]. Available: http://bit.ly/2XdoYPn