CAmkES to seL4CP Transition Guide

Introduction

The goal of this document is to act as a guide for migrating applications built using CAmkES to the seL4 Core Platform (seL4CP). This is done by discussing the similarities and differences between the frameworks as well as providing various examples. A basic understanding of CAmkES and seL4CP might be needed before reading this documentation. Most of the examples in this guide are taken from the CAmkES Tutorial and the following example systems:

- seL4 Device Driver Framework (sDDF)
  - similar CAmkES version (no longer maintained)
- Linux virtual machines using VirtIO demo
  - similar CAmkES version

To avoid confusion between CAmkES and seL4CP, when referring to seL4CP terminologies, we always emphasize them with *italics*.

CAmkES and seL4CP are both designed for building systems with a static architecture (while seL4CP aims to have limited dynamicism). In general, CAmkES is a higher-level tool than seL4CP, it provides:

- a language to describe component interfaces, components, and the whole systems
- a collection of reusable CAmkES components and interfaces
- full integration in the seL4 build system (i.e. CMake)
- templates and a code generator to combine programmer-provided component code with generated scaffolding and glue code to build a system image
- runtime libraries (`libsel4camkes`) that provide various supports, including memory allocators, a subset of POSIX syscalls, inter-component transport mechanisms, interfaces for the CAmkES APIs etc...

seL4CP is a minimal seL4 operating systems framework, it intentionally doesn't prescribe a boiled system but provides an SDK that is designed to integrate with a build system of your choice. It provides:

- seL4CP tool that takes a programmer-provided system description as input and produces an appropriate system image
- runtime libraries that provide the C runtime for the *protection domain*, along with interfaces for the sel4cp APIs

And does not provide:

- a build system. seL4CP allows (and forces) users to choose their own build systems
- runtime libraries/tools for specific supports

Note that unlike CAmkES, seL4CP only supports the MCS kernel. This decision doesn't have significant impacts on migrating non-MCS-kernel-specific applications from CAmkES to seL4CP, as the scheduler of the MCS kernel can also handle round-robin threads.

This guide will demonstrate 1-to-1 mappings between CAmkES concepts and seL4CP concepts where they exist, as well as potential ways to implement similar systems on top of seL4CP where seL4CP does not provide a feature.

Overview

Given a simple 2-component CAmkES system:

```
component Ping {
    control;
    dataport Buf(0x1000) sharedbuf_1;
    dataport Buf(0x1000) sharedbuf_2;
    emits Done n1;
    consumes Ready n2;
}

component Pong {
    control;
    dataport Buf(0x1000) sharedbuf_1;
    dataport Buf(0x1000) sharedbuf_2;
    consumes Done n1;
    emits Ready n2;
}

assembly {
    composition {
        component Ping ping;
        component Pong pong;

        connection seL4SharedData channel1(from ping.sharedbuf_1, to pong.sharedbuf_1);
        connection seL4Notification ntfn1(from ping.n1, to pong.n1);

        connection seL4SharedData channel2(from ping.sharedbuf_2, to pong.sharedbuf_2);
        connection seL4Notification ntfn2(from ping.n2, to pong.n2);
    }
}
```

A mostly equivalent seL4CP system of this CAmkES system may look like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<system>
    <memory_region name="sharedbuf_1" size="0x1_000" />
```

```
    <memory_region name="sharedbuf_2" size="0x1_000" />

    <protection_domain name="ping" priority="42">
        <program_image path="ping.elf" />
        <map mr="sharedbuf_1" vaddr="0x3000000" perms="rw" cached="false"
            setvar_vaddr="send_buf" />
        <map mr="sharedbuf_2" vaddr="0x3001000" perms="r" cached="false"
            setvar_vaddr="recv_buf" />
    </protection_domain>

    <protection_domain name="pong" priority="42">
        <program_image path="pong.elf" />
        <map mr="sharedbuf_2" vaddr="0x3000000" perms="rw" cached="false"
            setvar_vaddr="send_buf" />
        <map mr="sharedbuf_1" vaddr="0x3001000" perms="r" cached="false"
            setvar_vaddr="recv_buf" />
    </protection_domain>

    <channel>
        <end pd="ping" id="0" />
        <end pd="pong" id="0" />
    </channel>

    <channel>
        <end pd="ping" id="1" />
        <end pd="pong" id="1" />
    </channel>
</system>
```

At the system level, there is a roughly 1-to-1 match between `assembly`/`component` in CAmkES and *`system`*/*`protection domain`* (*PD*) on seL4CP, but the way CAmkES and seL4CP implement inter-component/*PD* communication and shared memory are different.

Concepts

`assembly`, `composition` and *`system`*

The CAmkES way of defining a system:

```
assembly {
    composition {
        // your stuff
    }
}
```

and the seL4CP way of defining a system:

```
<system>
<!-- also your stuff -->
</system>
```

`component` and *protection domain* {#component}

The CAmkES way of adding a component:

```
component Foo {
    control;
    // cool features
}

assembly {
    composition {
        component Foo foo;
        // your other stuff
    }
}
```

A CAmkES component is typically multithreaded. The keyword `control` means it will contain a main function and have an active thread of control. Each `interface` (see CAmkES Interfaces) the component interacts with uses another thread within the component. A component that declares `control` will have an entry point `run`, which is provided by the component code.

seL4CP *PDs* are single-threaded and event-driven to keep the programming model and implementations simple. A *PD* provides one single thread of control, which is responsible for receiving messages (i.e. seL4 system calls) from other *PDs* and invoking `libsel4cp` entry points. A *PD* has three entry points: `init`, `notified` and, optionally, `protected`.

`libsel4cp` is a library that provides the C runtime for the protection domain, along with interfaces for the sel4cp APIs, we will discuss some of the `libsel4cp` functions in CAmkES Interfaces.

The seL4CP way of adding a *protection domain*:

```
<system>
    <protection_domain name="ping" priority="42">
        <!-- cool features -->
    </protection_domain>

<!-- also your other stuff -->
</system>
```

Unlike CAmkES components, it is mandatory to specify the priority of the *PD*. You may also want to specify other thread attributes, including *`passive`*. A *PD* of a passive server may look like this:

```
<protection_domain name="eth" priority="101" budget="160" period="300"
    passive="true" pp="true">
    <!-- cool features -->
</protection_domain>
```

budget and *period* are thread attributes for the scheduler of the MCS kernel. On CAmkES, you should be able to configure them as attributes (see thread attributes).

passive means this is an event-driven *protection domain*, that has no continuously running thread of its own. After it has initialised itself, its scheduling context is revoked, and it runs on the scheduling contexts of its invokers or a notification.

pp means this component has a *protected procedure call* entry point `protected`.

Note: The ability to configure the CPU affinity of a *PD* will be added in the future.

The configuration of hardware components however, will be covered in section Hardware.

`attributes, configuration` {#attributes}

CAmkES allows the programmer to configure custom attributes of components/connectors, such as the configurations for a driver component. Those attributes are accessible as global variables in the component code:

```
component EthdriverARM {

    // some cool stuff

    attribute int promiscuous_mode = 1;
    attribute int heap_size = 0x10000;
    attribute int dma_pool = 0x200000;
}
```

seL4CP doesn't provide such a mechanism, the alternatives and current workaround is discussed in Build.

Thread attributes (e.g., priority, budget, period CPU affinity) are also considered a type of attribute in CAmkES:

```
assembly {
    composition {
        component Mycomponent c;
    }
    configuration {
        c.period = 300;
        c.budge = 160;
```

```
        c.priority = 42;
        c.affinity = 1;
    }
}
```

On seL4CP, thread attributes are part of the *PD* attributes (see this example).

`connectors`, `connections` and *`memory regions`*, *`channels`*

CAmkES interfaces from different components are connected by CAmkES `connectors`. CAmkES provides various connectors for communication (e.g., `seL4VirtQueues`) and shared memory (e.g., `seL4RPCDataport`) for different purposes. There are also connectors for specific types of servers (e.g., `seL4Ethdriver`). Some connectors (e.g., `seL4VMDTBPassthrough`), are even used as ways to hack the build system. In contrast, seL4CP provides strictly minimum supports for memory mapping with *`memory regions`* and IPCs/notifications with *`channels`*. With seL4CP, it is up to you to implement the specific mechanisms you need.

With the help of *`memory regions`*, *`channels`* and potentially libraries that run on top of seL4CP (such as the `libsharedringbuffer` library), you might be able to implement similar features on seL4CP for most of the standard `connectors` and some of the global `connectors`.

An example of a global `connector` is `seL4RPCDataportSignal`, which combines a `dataport` and a seL4 notification object. The following example is a common use case of this connector:

```
procedure DriverInterface {
    // some cool APIs
};

component TheDriver {
    provides DriverInterface client;
}

component TheClient {
    control;
    uses DriverInterface driver;
}

assembly {
    composition {
        component TheDriver mydriver;
        component TheClient myclient;
    }
    configuration {
        connection seL4RPCDataportSignal(from myclient.driver, to mydriver.client);
```

```
    }
}
```

On seL4CP, a similar functionality can be implemented thus:

```
<?xml version="1.0" encoding="UTF-8"?>

<system>
    <memory_region name="sharedbuf" size="0x1_000" />

    <protection_domain name="mydriver" priority="43">
        <program_image path="mydriver.elf" />
        <map mr="sharedbuf" vaddr="0x3000000" perms="r" cached="false"
            setvar_vaddr="clientbuf" />
    </protection_domain>

    <protection_domain name="myclient" priority="42">
        <program_image path="myclient.elf" />
        <!-- can't have a write only page on aarch64 seL4 -->
        <map mr="sharedbuf" vaddr="0x3000000" perms="rw" cached="false"
            setvar_vaddr="driverbuf" />
    </protection_domain>

    <channel>
        <end pd="mydriver" id="0" />
        <end pd="myclient" id="0" />
    </channel>
</system>
```

`sharedbuf` is a contiguous range of physical memory that's mapped into both *PDs* with the specified virtual address, caching attributes and permissions. The symbols `clientbuf` and `driverbuf` will be bound to the value `0x3000000` and are accessible in the application's code.

You can also set the physical address and the page size for a *memory region*:

```
<!-- physical memory for UART -->
<memory_region name="serial" size="0x1_000" phys_addr="0x9000000" />

<!-- RAM for a VM guest -->
<memory_region name="guest_ram" size="0x10_000_000" page_size="0x200_000" />
```

These are particularly useful for VMM and driver development.

Hardware connectors will be covered in Hardware.

CAmkES Interfaces and *channels* {#iface}

Apart from the entry point `run`, CAmkES also provides interfaces as abstract exposed interaction points of a component. Examples of interfaces are `dataports` (port interface), `emits`/`uses` (event interface) and `Ethdriver` (procedure interface). CAmkES sDDF makes use of the `Ethdriver` interface:

```
// the definition of `Ethdriver` in Ethdriver.idl4
procedure Ethdriver {
    int tx(in int len);
    int rx(out int len);
    void mac(out uint8_t b1, out uint8_t b2, out uint8_t b3, out uint8_t b4,
        out uint8_t b5, out uint8_t b6);
};
```

```
// code segments from CAmkES sDDF
assembly {
    composition {
        // some unrelated stuff

        // the connection of two interfaces
        connection seL4Ethdriver eth_driver_conn(from lwipserver.ethdriver,
            to ethdriver.client);
    }
}
```

This gives component `lwipserver` the ability to transmit data to component `ethdriver` by calling `ethdriver_tx`. On seL4CP, however, there are no built-in mechanisms for Ethernet drivers (or any other kind of drivers). What you might do instead, is to register *channels* for all methods you need:

```
<system>
    <!-- some cool configs for shared buffers -->

    <protection_domain name="eth" priority="101" budget="160" period="300" pp="true">
        <!-- some cool configs for shared buffer mappings -->
    </protection_domain>

    <protection_domain name="client" priority="100" budget="20000">
        <!-- some cool configs for shared buffer mappings -->
    </protection_domain>

    <!-- RX -->
    <channel>
        <end pd="eth" id="0" />
        <end pd="client" id="0" />
    </channel>
    <!-- TX -->
```

```
    <channel>
        <end pd="client" id="1" />
        <end pd="eth" id="1" />
    </channel>
    <!-- get MAC -->
    <channel>
        <end pd="client" id="2" />
        <end pd="eth" id="2" />
    </channel>
</system>
```

And implement the mechanisms using `libsel4cp` functions and entry points in the application's code, e.g., for the driver:

```
// in eth driver's PD

// your RX function
void eth_rx(void /* some cool parameters */)
{
    // some cool driver code

    // libsel4cp function
    sel4cp_notify(0);
}

// libsel4cp entry point
void notified(sel4cp_channel channel_id)
{
    switch(channel_id) {
        case 1:
            // handles TX
            return;
        case 2:
            // handles get mac
            return;
        default:
            // complains
            break;
    }
}
```

Port interfaces and event interfaces can also be implemented in similar ways.

Hardware {#hardware}

A hardware component represents an interface to hardware in the form of a component. A hardware component may look like this:

```
component Device {
    hardware;

    emits Interrupt irq;
    dataport Buf mem;
}
```

seL4CP doesn't have an abstraction for hardware. To allow a *PD* to access a particular device, the physical memory of the device needs to be mapped into the *PD*. You may also need to register the IRQ for it.

```
<system>
    <memory_region name="device_mem" size="0x10_000" phys_addr="0x30890000" />
    <!-- other cool stuff -->

    <protection_domain name="driver" priority="42">
        <map mr="device_mem" vaddr="0x30890000" perms="rw" cached="false"
            setvar_vaddr="device_base" />
        <map mr="device_ioport" vaddr="0x308a0000" perms="rw" cached="false"
            setvar_vaddr="device_ioport" />

        <irq irq="142" id="7" />
    </protection_domain>
    <!-- other cool stuff -->
</system>
```

To handle the IRQ, add an entry in the handler for *channels*.

```
void notified(sel4cp_channel channel_id)
{
    switch(channel_id) {
        case 7:
            // handle interrupt
            return;

        // other channels

        default:
            // complains
            break;
    }
}
```

Build system {#build}

seL4CP is build-system-agnostic, so setting up build flags as it is done on CAmkES using CMake is up to the build system you choose for your project.

As we mentioned in `attributes`, `configuration`, CAmkES uses `attributes` that result in symbols being defined and available to the component code. For example, in CAmkES VMM development, `attributes` are used heavily for describing the layout of the whole system and the features of each VMM:

```
configuration {
    // telling the vswitch backend about the connections that VM0 likes to have
   vm0.vswitch_layout = [{"mac_addr": "02:00:00:00:AA:02", "recv_id": 0, "send_id":1},
                         {"mac_addr": "02:00:00:00:AA:03", "recv_id": 2, "send_id":3}];

   // telling the VMM what init functions should be call during the initialisation
   vm0.init_cons = [
                    {"init":"make_virtio_blk"},
                    {
                      "init":"make_virtio_con_driver_dummy",
                      "badge":"recv1_notification_badge()",
                      "irq":"handle_serial_console"
                    },
                    {"init":"make_virtio_con"}
                   ];
}
```

seL4CP does not have a standard build system. In existing seL4CP example systems, most of these variables and structures are hard-coded. In addition, for parameters in the `.system` file, such as *channel* IDs and IRQs, it is up to you to ensure that they match those specified in the application's code. It would be possible to have external build tools as a solution in the future, but more use cases are needed to determine what seL4CP actually needs to provide.

On CAmkES, you are also able to import components from other files or include header files for your `.camkes` file, which may not be necessary for seL4CP systems.

Libraries

Standard C library

Standard C library functionality is available on CAmkES, but seL4CP does not enforce any particular C library, or even the use of C as an implementation language.

Synchronization Primitives

CAmkES provides primitives for intra-component synchronization, e.g.:

```
/* Lock mutex m */
int m_lock(void);
```

```
/* Unlock mutex m */
int m_unlock(void);
```

which is not needed on seL4CP as *PDs* are single-threaded.

There are no built-in inter-component synchronisation primitives for either CAmkES or seL4CP as seL4 system calls are sufficient.

Allocator

CAmkES provides a DMA allocator to allocate and manage DMA buffers from a DMA pool, which is configured with the `dma_pool` attribute in CAmkES components. It also provides a seL4 capability object allocator that can be used to allocate seL4 capability objects from a managed pool. These tools are not available on seL4CP as one of its goals is to have capabilities known at *build* time.

Dynamic memory allocation in CAmkES is done by standard C library functions, which manipulate a static array that has been set up by CAmkES. The size of the array can be configured with the `heap_size` attribute in CAmkES components. On seL4CP, you can configure a *`memory region`* for this purpose, but you will also need to implement any allocators you need (at least for now).

Inter-component Transport Mechanisms {#sharedring}

The inter-component transport mechanisms on CAmkES are a virtqueue-like ring buffer library and the corresponding connectors `VirtQueueInit` and `seL4VirtQueues`. The mechanisms are commonly used in client-server systems and inter-VMM communications, e.g.:

```
component Comp {
    uses VirtQueueDev recv;
    uses VirtQueueDrv send;
}

assembly {
    composition {
        component Comp mydriver;
        component Comp myclient;

        component VirtQueueInit vqinit0;
        component VirtQueueInit vqinit1;

        connection seL4VirtQueues virtq_conn0(to vqinit0.init,
            from mydriver.send, from myclient.recv);
        connection seL4VirtQueues virtq_conn1(to vqinit1.init,
            from mydriver.recv, from myclient.send);
    }
```

```
    configuration {

        // driver virtqueue configurations
        // unique (per component) ID of the virtqueue
        mydriver.recv_id = 0;
        // size of the shared memory region of this virtqueue
        mydriver.recv_shmem_size = 0x1000;
        mydriver.send_id = 1;
        mydriver.send_shmem_size = 0x1000;

        // client virtqueue configurations
        myclient.recv_id = 0;
        myclient.recv_shmem_size = 0x1000;
        myclient.send_id = 1;
        myclient.send_shmem_size = 0x1000;
    }
}
```

The seL4 Device Driver Framework makes use of a ring buffer library that achieves the same goal. An example configuration for the ring buffers looks like this:

```
<system>
    <!-- shared memory for ring buffer mechanism -->
    <memory_region name="avail-1" size="0x200_000" page_size="0x200_000"/>
    <memory_region name="used-1" size="0x200_000" page_size="0x200_000"/>
    <memory_region name="avail-2" size="0x200_000" page_size="0x200_000"/>
    <memory_region name="used-2" size="0x200_000" page_size="0x200_000"/>

    <memory_region name="shared_dma-1" size="0x200_000" page_size="0x200_000" />
    <memory_region name="shared_dma-2" size="0x200_000" page_size="0x200_000" />

    <protection_domain name="mydriver" priority="43">
        <program_image path="mydriver.elf" />

        <map mr="avail-1" vaddr="0x5000000" perms="rw" setvar_vaddr="rx_avail" />
        <map mr="used-1" vaddr="0x5200000" perms="rw" setvar_vaddr="rx_used" />
        <map mr="avail-2" vaddr="0x5400000" perms="rw" setvar_vaddr="tx_avail" />
        <map mr="used-2" vaddr="0x5600000" perms="rw" setvar_vaddr="tx_used" />

        <map mr="shared_dma-1" vaddr="0x5800000" perms="rw"
            setvar_vaddr="rx_shared_dma_vaddr" />
        <map mr="shared_dma-2" vaddr="0x5a00000" perms="rw"
            setvar_vaddr="tx_shared_dma_vaddr" />
    </protection_domain>
```

```
    <protection_domain name="mycient" priority="42">
        <program_image path="mycient.elf" />

        <map mr="avail-2" vaddr="0x5000000" perms="rw" setvar_vaddr="rx_avail" />
        <map mr="used-2" vaddr="0x5200000" perms="rw" setvar_vaddr="rx_used" />
        <map mr="avail-1" vaddr="0x5400000" perms="rw" setvar_vaddr="tx_avail" />
        <map mr="used-1" vaddr="0x5600000" perms="rw" setvar_vaddr="tx_used" />

        <map mr="shared_dma-2" vaddr="0x5a00000" perms="rw"
            setvar_vaddr="rx_shared_dma_vaddr" />
        <map mr="shared_dma-1" vaddr="0x5800000" perms="rw"
            setvar_vaddr="tx_shared_dma_vaddr" />
    </protection_domain>

    <channel>
        <end pd="mydriver" id="0" />
        <end pd="mycient" id="0" />
    </channel>
</system>
```

You can see an example of how to set up and use the ring buffer in application code in the sDDF repository.

Virtual Machine development

The CAmkES VMM (distributed in various repositories, mainly camkes-vm, vmmplatsupport and libsel4vm) provides VM-related CAmkES components, templates, interfaces and libraries for creating VM guests and applications on seL4. The particularly important libraries are:

- `libcrossvm` that makes dataports and event interfaces available to the guest VM.
- VirtIO backends for networking, block devices, console and sockets.
- Virtual GICv2 support.

seL4CP VMM is an experimental VMM for 64-bit ARM platforms built on top of seL4CP. There are plans to improve the VMM, including x86 and RISC-V architecture support as well as VirtIO backends to make the seL4CP VMM more feature complete. These changes are currently in progress, we recommend you check the project's README to see the status as it progresses.

seL4CP VMM is subject to change as it is a work-in-progress. This document will be updated to adapt to the changes, but this might not be done in a timely manner. Please always check the seL4CP VMM manual for latest changes.

Both CAmkES VMM and seL4CP VMM support only one guest VM per instance of VMM. This is an intentional decision to maintain isolation between each VM/VMM.

While the seL4CP VMM is likley to change, the purpose of this example is to show the differences between similar non-trivial applications built on top of the both OS frameworks. This will also give you some idea of how to migrate existing CAmkES applications making use of virtual machines to seL4CP.

VM component

CAmkES provides a standard CAmkES VM component for ARM that describes a component with one VM guest, the interfaces that this component has (or may have), and the memory layouts and attributes that the VM guest has:

```
component VM {
    control;
```

Indicates the presence of a main control thread, previously discussed in `component` and *`protection domain`*.

```
    uses FileServerInterface fs;
    maybe consumes restart restart_event;
    has semaphore vm_sem;
    maybe uses Batch batch;
    maybe uses PutChar guest_putchar;
    maybe uses GetChar serial_getchar;
    maybe uses VirtQueueDev recv;
    maybe uses VirtQueueDrv send;
    consumes HaveNotification notification_ready;
    emits HaveNotification notification_ready_connector;
```

CAmkES interfaces, previously discussed in CAmkES Interfaces.

```
    maybe uses VMDTBPassthrough dtb_self;
    provides VMDTBPassthrough dtb;
```

A hack. Will be discussed in Device Passthrough.

```
    attribute int base_prio;
```

Used by the CAmkES VMM for creating threads, N/A on seL4CP as it is single-threaded.

```
    attribute int num_vcpus = 1;
```

The ability to config the number of vCPUs as an attribute and handles multiple vCPUs in CAmkES VMM. seL4CP VMM doesn't currently handle multiple vCPUs, but the feature will be added in the future.

```
    attribute int num_extra_frame_caps;
    attribute int extra_frame_map_address;
```

Attributes to initialise an allocator. N/A on seL4CP VMM because all memory is described in the system description at build time. The VMM does not manage the guest's virtual address space at runtime.

```
    // 7. attributes to describe the VM guest
    attribute {
        string linux_ram_base;
        string linux_ram_paddr_base;
        string linux_ram_size;
        string linux_ram_offset;
        string dtb_addr;
        string initrd_max_size;
        string initrd_addr;
    } linux_address_config;
    attribute {
        string linux_name = "linux";
        string dtb_name = "linux-dtb";
        string initrd_name = "linux-initrd";
        string linux_bootcmdline = "";
        string linux_stdout = "";
        string dtb_base_name = "";
    } linux_image_config;
```

Attributes that describe the memory layout of the VM guest. On seL4CP, `linux_address_config` attributes are handled as *memory regions* that are mapped to the VM's virtual memory. `linux_image_config` attributes are not handled by seL4CP, however, some of the attributes (e.g., `linux_stdout` and `linux_bootcmdline`) are configurable in the VM's DTS file.

```
    // 8. serial connections layout
    attribute {
        int send_id;
        int recv_id;
    } serial_layout[] = [];
}
```

Serial connections layout. An optional attribute that configures the connections the VM likes to have for the serial multiplexor. This is currently N/A on seL4CP VMM, but there are plans for adding a serial driver and a serial multiplexor as external libraries on top of seL4CP.

The standard CAmkES VM component also assumes the presence of other components, mainly server components:

```
assembly {
    composition {
        // other components

        component FileServer fs;
        component SerialServer ss;
        component TimeServer ts;
```

```
        // also define the connections and attributes
    }
    // other stuff
}
```

VMM requires access to the actual hardware. CAmkES provides a file server, a serial server and a time server etc. for such purposes. On seL4CP, it's up to you to implement these functionalities, or potentially make use of the existing external libraries on top of seL4CP.

The approach for VMM development on top of seL4CP is quite different. Most of the configuration is not done by seL4CP but by the seL4CP VMM and the external build system (that you provide). Take `vm_multi` as an example:

```
// vm_multi.camkes

component VM {
    // everything present in the standard CAmkES VM component
    VM_INIT_DEF()

    maybe uses VirtQueueDev recv1;
    maybe uses VirtQueueDrv send1;

    attribute vswitch_mapping vswitch_layout[] = [];
    attribute string vswitch_mac_address = "";
}

assembly {
    composition {
        // every needed server component
        VM_GENERAL_COMPOSITION_DEF()

        /* Other standard VM defintions (vm0, vm1) */
        VM_COMPOSITION_DEF(0)
        VM_COMPOSITION_DEF(1)

        /* vm0, vm1 serial connections */
        VM_VIRTUAL_SERIAL_COMPOSITION_DEF(0,1)

        // other necessary configurations, including server components

        /* vm0-vm1 connection */
        component VirtQueueInit vm1_vm0;
        connection seL4VirtQueues vm1_vm0_conn(to vm1_vm0.init, from vm0.send1,
            from vm0.recv1, from vm1.send1, from vm1.recv1);
```

```
        /* DTB Passthrough */
        connection seL4VMDTBPassthrough vm0_dtb(from vm0.dtb_self, to vm0.dtb);
        connection seL4VMDTBPassthrough vm1_dtb(from vm1.dtb_self, to vm1.dtb);
    }

    configuration {
        VM_GENERAL_CONFIGURATION_DEF()
        VM_CONFIGURATION_DEF(0)
        VM_CONFIGURATION_DEF(1)
        VM_VIRTUAL_SERIAL_CONFIGURATION_DEF(0,1)

        // virtqueue configurations are omitted

        vm0.vswitch_mac_address = "02:00:00:00:AA:01";
        vm0.vswitch_layout = [{"mac_addr": "02:00:00:00:AA:02",
            "recv_id": 0, "send_id":1}];

        vm1.vswitch_mac_address = "02:00:00:00:AA:02";
        vm1.vswitch_layout = [{"mac_addr": "02:00:00:00:AA:01",
            "recv_id": 0, "send_id":1}];
    }
}
```

Most of the work here is done by the C preprocessor macros, these are all defined in `vm.h`, and are concerned with specifying and configuring components that all VM(M)s need. We will not cover the details here, for more information on CAmkES VM components, see CAmkES VMM tutorial; for more information on virtqueues, see previous discussions in Inter-component Transport Mechanisms.

A similar VM system on top of seL4CP may look like this:

```
<!-- virtio_demo.system -->

<system>
    <!--
     Here we give the guest 256MiB to use as RAM. Note that we use 2MiB page
     sizes for efficiency, it does not have any functional effect.
    -->
    <memory_region name="guest_ram-1" size="0x10_000_000" page_size="0x200_000" />
    <memory_region name="guest_ram-2" size="0x10_000_000" page_size="0x200_000" />

    <!-- shared memory for ring buffer mechanism -->
    <memory_region name="avail-1" size="0x200_000" page_size="0x200_000"/>
    <memory_region name="used-1" size="0x200_000" page_size="0x200_000"/>
    <memory_region name="avail-2" size="0x200_000" page_size="0x200_000"/>
```

```
<memory_region name="used-2" size="0x200_000" page_size="0x200_000"/>

<memory_region name="shared_dma-1" size="0x200_000" page_size="0x200_000" />
<memory_region name="shared_dma-2" size="0x200_000" page_size="0x200_000" />

<protection_domain name="VMM-1" priority="254">
    <program_image path="vmm.elf" />
    <!--
        Currently the VMM is expecting the address set to the variable
        "guest_ram_vaddr" to be the same as the address of where the guest
        sees RAM from its perspective. In this case the guest physical
        starting address of RAM is 0x40000000, so we map in the guest RAM
        at the same address in the VMMs virutal address space.
    -->
    <map mr="guest_ram-1" vaddr="0x40000000" perms="rw"
        setvar_vaddr="guest_ram_vaddr" />

    <virtual_machine name="linux" vm_id="1">
        <!--
         The DTS given to Linux specifies that RAM will start
         at 0x40000000.
        -->
        <map mr="guest_ram-1" vaddr="0x40000000" perms="rwx" />
    </virtual_machine>

    <!-- shared memory for ring buffer mechanism -->
    <map mr="avail-1" vaddr="0x50000000" perms="rw" setvar_vaddr="rx_avail" />
    <map mr="used-1" vaddr="0x5200000" perms="rw" setvar_vaddr="rx_used" />
    <map mr="avail-2" vaddr="0x5400000" perms="rw" setvar_vaddr="tx_avail" />
    <map mr="used-2" vaddr="0x5600000" perms="rw" setvar_vaddr="tx_used" />

    <map mr="shared_dma-1" vaddr="0x5800000" perms="rw"
        setvar_vaddr="rx_shared_dma_vaddr" />
    <map mr="shared_dma-2" vaddr="0x5a00000" perms="rw"
        setvar_vaddr="tx_shared_dma_vaddr" />
</protection_domain>

<protection_domain name="VMM-2" priority="254">
    <program_image path="vmm.elf" />
    <map mr="guest_ram-2" vaddr="0x40000000" perms="rw"
        setvar_vaddr="guest_ram_vaddr" />

    <virtual_machine name="linux" vm_id="1">
        <map mr="guest_ram-2" vaddr="0x40000000" perms="rwx" />
    </virtual_machine>
```

```
        <!-- shared memory for ring buffer mechanism -->
        <map mr="avail-2" vaddr="0x50000000" perms="rw" setvar_vaddr="rx_avail" />
        <map mr="used-2" vaddr="0x5200000" perms="rw" setvar_vaddr="rx_used" />
        <map mr="avail-1" vaddr="0x5400000" perms="rw" setvar_vaddr="tx_avail" />
        <map mr="used-1" vaddr="0x5600000" perms="rw" setvar_vaddr="tx_used" />

        <map mr="shared_dma-2" vaddr="0x5a00000" perms="rw"
            setvar_vaddr="rx_shared_dma_vaddr" />
        <map mr="shared_dma-1" vaddr="0x5800000" perms="rw"
            setvar_vaddr="tx_shared_dma_vaddr" />
    </protection_domain>

    <channel>
        <end pd="VMM-1" id="2" />
        <end pd="VMM-2" id="2" />
    </channel>
</system>
```

This VM system defines the necessary *memory regions* `guest_ram-#` for the VMs and maps them into the virtual memory of the VMs by creating a ***`map`*** in the ***`virtual machine`*** element of the VMMs' *protection domains*. It also sets up the connections between two VMMs, as well as shared memory regions for Inter-component Transport Mechanisms (which is done by `seL4VirtQueues` in the CAmkES example). Unlike CAmkES, as a minimal seL4 operating systems framework, seL4CP doesn't have a way to describe the properties of the VM guest. It's up to you to do the necessary configuration, e.g., `vswitch_layout` and `vswitch_mac_address`, for your seL4CP VM system.

The examples are simplified for demonstration purposes, see the source code of the CAmkES version and the seL4CP version for details of the implementation.

For more details on the seL4CP VMM please see its manual.

Device Pass-through on 64-bit ARM platforms {#passthrough}

Given two DTS entries as example devices:

```
//linux.dts

// uart
pl011@9000000 {
    clock-names = "uartclk\0apb_pclk";
    clocks = <0x8000 0x8000>;
    interrupts = <0x00 0x01 0x04>;
    reg = <0x00 0x9000000 0x00 0x1000>;
    compatible = "arm,pl011\0arm,primecell";
};
```

```
// interrupt controller
intc@8000000 {
    phandle = <0x8001>;
    interrupts = <0x01 0x09 0x04>;
    reg = <0x00 0x8000000 0x00 0x10000 0x00 0x8010000 0x00 0x10000
           0x00 0x8030000 0x00 0x10000 0x00 0x8040000 0x00 0x10000>;
    compatible = "arm,cortex-a15-gic";
    ranges;
    size-cells = <0x02>;
    address-cells = <0x02>;
    interrupt-controller;
    interrupt-cells = <0x03>;
};
```

Device pass-through for CAmkES VMM is done by a dummy interface `VMDTBPassthrough` and the corresponding connector `seL4VMDTBPassthrough`. They are used as a way to hack the CAmkES build system. The connector parses the DTB entries (including the corresponding IRQ(s)) from the programmer-provided `devices.camkes` file, e.g.,

```
// devices.camkes

vm.dtb = dtb([{"path": "/pl011@9000000"},]);

// Interrupt Controller Virtual CPU interface (Virtual Machine view)
vm.mmios = ["0x8040000:0x1000:12",];
```

and gives CAmkES VMM the access to the results through the connection:

```
component VM vm;
connection seL4VMDTBPassthrough vm_dtb(from vm.dtb_self, to vm.dtb);
```

On seL4CP, device pass-through is done by mapping *memory regions* to the *virtual machine*:

```
<!--
github.com/au-ts/sel4cp_vmm/blob/main/board/qemu_arm_virt_hyp/systems/simple.system
-->

<system>
    <memory_region name="guest_ram" size="0x10_000_000" page_size="0x200_000"/>

    <!--
     We intend to map in this UART into the guest's virtual address space so
     we define the memory region here.
    -->
    <memory_region name="serial" size="0x1_000" phys_addr="0x9000000" />
```

```
    <!--
     We need to map in the interrupt controller's (GIC) virtual CPU interface.
     This is then mapped into the guest's virtual address space as if it was
     the actual interrupt controller. On ARM GICv2, not all of the interrupt
     controller is hardware virtualised, so we also have a virtual driver in
     the VMM code.
    -->
    <memory_region name="gic_vcpu" size="0x1_000" phys_addr="0x8040000" />

    <protection_domain name="VMM" priority="254">
        <program_image path="vmm.elf" />
        <map mr="guest_ram" vaddr="0x40000000" perms="rw"
            setvar_vaddr="guest_ram_vaddr" />
        <virtual_machine name="linux" vm_id="1">
            <map mr="guest_ram" vaddr="0x40000000" perms="rwx" />
            <!--
             For simplicity we give the guest direct access to the platform's UART.
             This is the same UART used by seL4 and the VMM for debug printing. The
             consequence of this is that the guest can just use the serial without
             trapping into the VMM and hence we do not have to emulate access.
            -->
            <map mr="serial" vaddr="0x9000000" perms="rw" cached="false" />
            <!--
             As stated above, we need to map in the virtual CPU interface into
             the guest's virtual address space. Any access to the GIC from
             0x8010000 - 0x8011000 will access the VCPU interface. All other
             accesses will result in virtual memory faults, routed to the VMM.
            -->
            <map mr="gic_vcpu" vaddr="0x8010000" perms="rw" cached="false" />
        </virtual_machine>
        <!--
            When the serial that is mapped into the guest receives input, we
            want to receive an interrupt from the device. This interrupt is
            delivered to the VMM, which will then deliver the IRQ to the guest,
            so that it can handle it appropriately. The IRQ is for the
            platform's PL011 UART the VMM is expecting the ID of the IRQ to be 1.
         -->
        <irq irq="33" id="1" />
    </protection_domain>
</system>
```

Future work on the seL4 Core Platform

The following is a (non-exhaustive) list of indented future additions to seL4CP:

- Limited dynamic memory management.
  - There will still be a static amount of memory available, however there plans to allow for having a protection domain temporarily assign a memory region to its virtual address space. Note that there is still discussion to be had on this, the details are not finalised.
- Multi-threaded protection domains.
- Multi-core configurations.
  - This will allow the use of the SMP kernel. For example you may want to pin certain PDs to certain CPU cores.
- Other architecture support such as x86 and RISC-V.
- Virtualisation support.
  - As you can see from the examples in this guide, we have been experimenting with virtual machines on top of seL4CP. The changes necessary to do this are not yet mainlined, but should be in the near future.

Note that some of these features have already been implemented and are currently in the process of being mainlined into seL4CP.

Reference

1. seL4 Core Platform manual
2. CAmkES manual
3. libsel4camkes manual (the best version I can find)