Embedded System Design and Modeling

EE382N.23, Fall 2019

Lab #2

Refinement

Due: Monday, October 28th, 2019 (11:59pm)

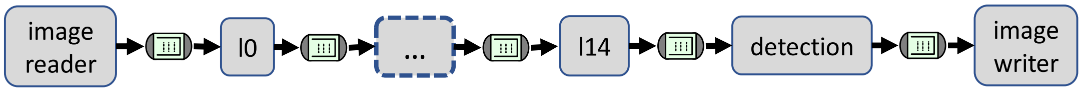
Instructions:

- Please submit your solutions via Canvas. Submissions should include a lab report (single PDF) and a single Zip or Tar archive with the source and supplementary files (code should include a README and has to be compilable and simulatable by running 'make' and ‘make test’, respectively).
- You are allowed to work in teams of up to three people and you are free to switch partners between labs and the project. Please submit one solution per team.

Tiny Yolo Object Detector Model Refinement

The purpose of this lab is to perform computation and communication refinement on the Tiny YOLO object detector SystemC specification model. As discussed in the lectures, the purpose of refinement process is to compile the abstract specification model into detailed computation and communication models to represent corresponding design decisions both for performance analysis as well as for further synthesis and implementation. You can start the refinement from the specification model developed in Lab 1.

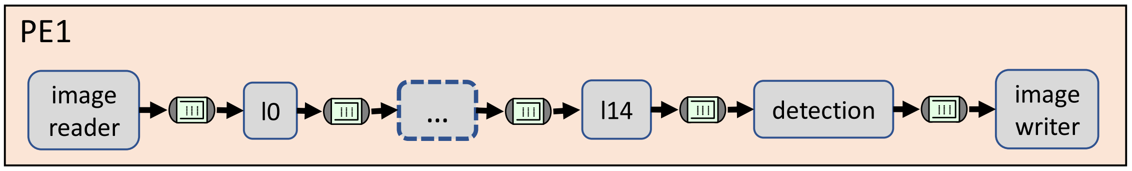
A diagram of the reference specification model is shown below:



(a) Computation Refinement

As a first step, we will refine the specification model into a computation model that reflects the partitioning and mapping of processes onto processing elements (PEs), where we will assume that we have allocated an Intel processor (*PE1*) to execute all the Kahn processes of Tiny YOLO.

1. Wrap all the processes of the specification and instantiate them under a new SystemC module that represents the processor PE:

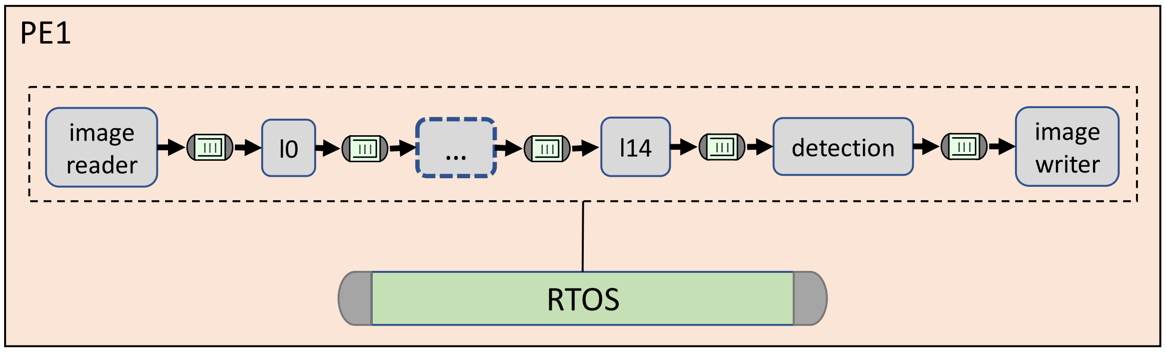


After deciding on a mapping, we can also make our model timed. Towards this end, you will need to annotate `process()` method of each Kahn process with a call to SystemC’s `wait()` with timing information representing the estimated execution time on the processor the process is mapped. For this lab, we will use the timing information you profiled in Lab1 (in ms, i.e. using `wait(<profiled_time_in_ms>, SC_MS)`.

Once code is annotated, measure the throughput and latency of application via `sc_time_stamp()`. This function returns the current time as `sc_time` which you can perform arithmetic operations on and which can be printed by passing it into the `cout` stream or by casting it to seconds via `to_seconds()`.

2. Next, we will further refine the model to represent scheduling decisions. Since the processor can only execute a single task at a time, we need to serialize process executions on the processor. Assume that we decided to perform scheduling dynamically under the control of an operating system (OS). Introduce a preemptive operating system model that uses round-robin scheduling to serialize execution of the processes on the processor. Use the (minimal) API for such an operating system provided at [1]. Refine your model by defining an OS channel that implements the `os_api` interface. [1] also provides a new base class for Kahn process that has an `init()` method, which can be used to register tasks with operating system as well as any other initialization that cannot be executed in the constructor. Note that, in addition to Kahn processes and their timed waits, you will need to refine `write()/read()` calls and wrap any wait statements used for inter-process communication (IPC) and synchronization with `pre_wait()` and `post_wait()` as discussed in class.

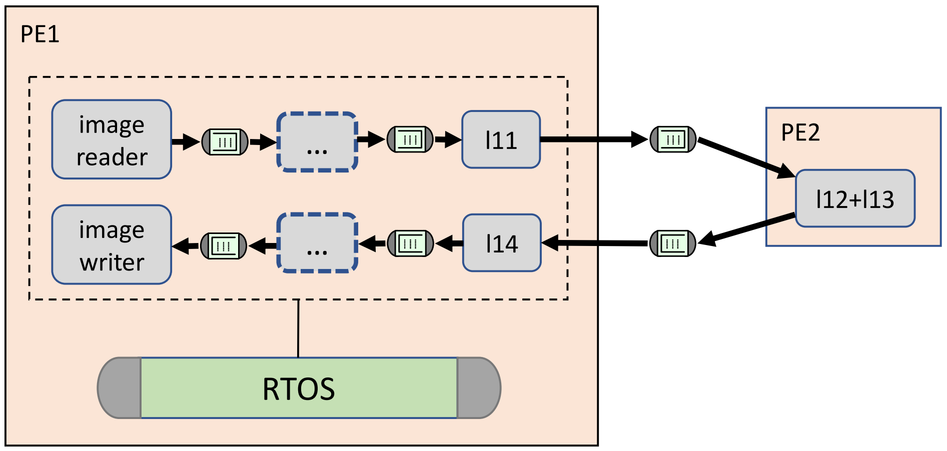
   A diagram of the refined model with OS is shown below:



   Measure the latency and throughput of the refined model and compare it with the previous results. How does serialization of tasks affect latency? How closely does your scheduled model match execution of the application on the real Intel processor?

3. To improve latency and throughput, we will further introduce a hardware accelerator into our system. This accelerator is used to speed up layers *l12* and *l13*, which are the most computationally heavy convolutional layers. In the general case, relative performance of accelerator compared to the processor depends on variety of factors but for this lab, we will simply assume that the accelerator can reduce forward propagation time of layers by factor of 5 (5x).

   Map layers *l12* and *l13* out into a new accelerator *PE2* and corresponding SystemC wrapper module, where the input queue of *l12* and the output queue for *l13* will be turned into inter-processor communication. In hardware, we need to perform static scheduling and serialization. To model the accelerator, fuse layers *l12* and *l13* by defining a new Kahn process that executes both layers sequentially in a loop. Scale the annotated execution times by the speedup factor. A diagram of the refined model with accelerator is shown below:



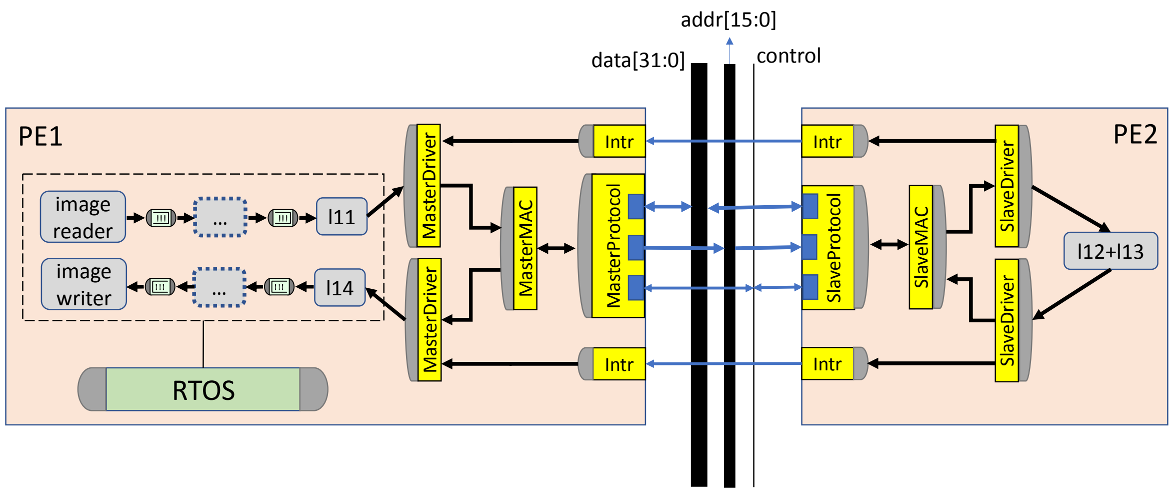
Measure the latency and throughput of the refined model and compare it with the previous results. By how much does the accelerator improve overall system latency and throughput? Do throughput and latency improve equally? Why or why not?

4. (Extra Credit) Implement an OS model that uses priority scheduling. How is latency and throughput of your design affected by different task priority settings?

(b) Communication refinement

As the final step, we will perform communication refinement and replace the zero-latency communication model with pin-accurate and transaction-level models (PAMs and TLMs). For this purpose, we will use a simple hardware bus protocol with address, data and control wires for all communication between PEs (CPU and Accelerator). A detailed, pin-accurate implementation of this bus protocol is provided in [1]. The bus model defines physical layer realizations for interrupt detection (*MasterHardwareSyncDetect*) and interrupt generation (*SlaveHardwareSyncGenerate*), bus wires and a protocol-level (physical) implementation each for master (*MasterHardwareBus*) and slave (*SlaveHardwareBus*) sides. In addition, media access (MAC) channels (named [*Master*|*Slave*]*HardwareBusLinkAccess*) show the methods of how to access the bus.

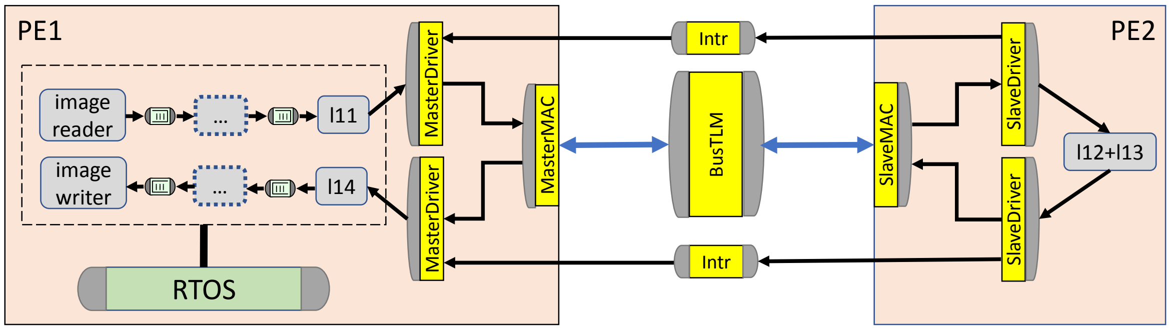
5. Refine the final computation model from part (a) to a PAM, where the CPU (*PE1*) is the bus master and the Accelerator (*PE2*) is a bus slave. In a PAM, the pin-accurate physical and MAC layer bus protocols are inlined into computing elements (*PE1* or *PE2*). Note that, to avoid deadlocks in case of communication with multiple bus slaves, each bus slave needs to send an interrupt signal to inform its corresponding master when it is ready. The PAM will need to include corresponding physical interrupt layer half channels inlined into the computing element. In the PAM, all communications is happening through connected signal wires. Within each computing element, there should be half-channels representing the drivers. The *l11*, *l14*, *l12+l13* processes do not call the bus/interrupt channels directly. Rather, they should remain unrefined, and a driver adapter should translate the `read()/write()` calls in the behaviors into bus communication, including all necessary synchronization. A diagram of the resulting PAM is shown below:



Note that interrupts generated by the accelerator might be missed by the *MasterDriver* if *PE1* is not performing a read or write. To address this problem, you will need introduce thread(s) into the top-level module of *PE1* that model the hardware of the processor continuously checking for interrupts and recording them in a flag (in reality, there should also be an interrupt service routine integrated with the OS model, but you are not required to model to this level of accuracy in this lab). Instead of directly calling the interrupt detection, the driver should check the flag and block when it is not set (either using an `sc_event` or a built-in channel like `sc_mutex` or `sc_semaphore`). Keep in mind that this blocking or synchronization channel needs to be properly integrated with the OS (`pre_wait()`/`post_wait()`) to trigger a context switch when blocking the calling process.

What is the throughput and latency of the PAM model? How does simulation execution time of the PAM compare with previous models?

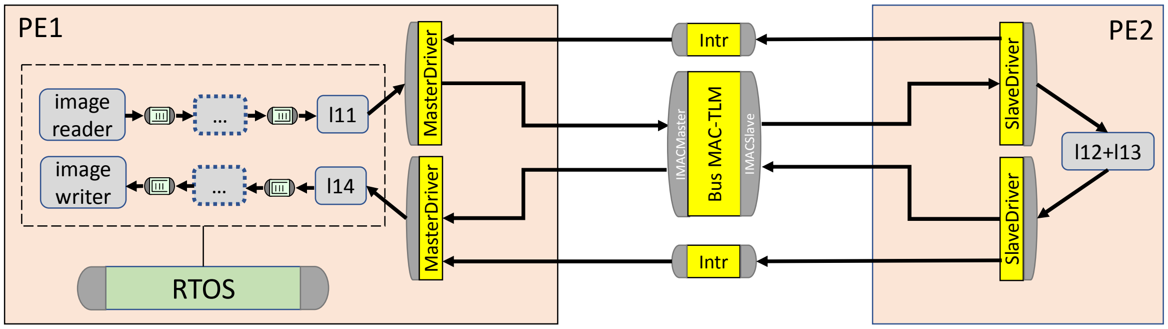
6. Next, we are going to replace the pin-accurate model of the system with a transaction-level model (TLM) at an abstracted level. For this, the protocol-level interface (both master and slave side) can be exchanged with a single *HardwareBusProtocolTLM* channel in which the communication is not performed via the wires previously instantiated, but through plain variables and events as a transaction-level model. Implement a corresponding transaction-level channel model of the bus and refine the existing PAM of the system into a TLM-based communication model. A TLM-based communication model has a similar structure as the PAM. However, physical layers and wires are replaced with a TLM bus channel that will transfer the data word by word. Furthermore, TLM versions of interrupts are realized by replacing physical interrupt layers and wires with single handshake channels (that act as interrupt TLMs). A diagram of the resulting TLM is shown below:



Does your TLM reach the same accuracy (in measured latencies and throughput) as the PAM? If not, explain why it is not possible for you to write a TLM that is 100% accurate. What is the expected and actual simulation speedup of the TLM compared to the PAM, i.e. does the achieved speedup match what you would theoretically expect?

7. (Extra Credit) Instead of a TLM at the protocol level, the MAC and bus protocol TLM channels together can be exchanged by a single MAC-TLM channel, where both the MAC and physical layers are abstracted away and data will be directly communicated without modeling it being sliced into bus words. Implement a MAC-TLM version of the bus and refine the communication model into a MAC-TLM based variant. Note that, to see the benefits of MAC-TLM, the feature maps need to be transferred all at once instead of transaction by transaction. You will need to send the entire output feature map of `l11` as a single struct-based token that encapsulates all data, instead of sending one float at a time. Likewise, the output feature map of *l12+l13* should also be sent as a single token.

   A diagram of MAC-TLM communication model shown below. In this case, a MAC-TLM channel encapsulates all the access and communication protocols. Note that interrupt models remain unchanged, and are still realized as single handshake channels:



   Can your MAC-TLM reach the same accuracy (in measured latencies and throughput) as the PAM and/or TLM? In general, under what conditions will a MAC-TLM be able to provide the same simulated timing behavior as a TLM or PAM? What is the expected and actual speedup between PAM, TLM and MAC-TLM?

References

[1] https://github.com/kammirzazad/KPN-Refinement