Verification of the HASPOC Hypervisor: Proof of Information Flow Security

Christoph Baumann,
KTH Royal Institute of Technology, Stockholm, Sweden
cbaumann@kth.se

v1, November 4, 2018

About this document

This technical report describes details of the formal models and proofs in this repository. It serves as a companion piece to the the paper "Compositional Verification of Security Properties for Embedded Execution Platforms" by C. Baumann, O. Schwarz, and M. Dam submitted to the Journal of Cryptographic Engineering (JCEN).

Version information

This document is currently under revision because the descriptions of parts of the ideal model, the platform model, and the proofs still refer to earlier version of the development. The main difference is that there are additional message boxes between the components, i.e., the GIC and the memory on the ideal model and the in principle all the components in the refined model. Moreover, there is an additional buffer to receive IGC notification interrupts and inject them into the receiver's GIC. As these buffers were redundant we removed them in the current version.

In addition, the documentation is being adapted to take into account the presentation of the work in the paper. As of this date, the first 30 pages of this document have been revised. Subsequently updated versions will be provided shortly.

Revision History

v1-2018-11-04 first 30 pages revised

Contents

1 Overview

The formal development in this repository contains models and proofs regarding the information flow security of the HASPOC hypervisor design. The hypervisor runs on an ARMv8 SoC that is modeled as part of the verification effort, however the models abstract as much as possible from the intricacies of the actual hardware platform. Instead we introduce abstractions that specify in an axiomatic fashion the required properties of the hardware for establishing the desired isolation between guests. Similarly the implementation details of the hypervisor are hidden as much as possible and internal data structures are only exposed if they contribute to externally visible behaviour, i.e., if they influence the interactions of the hypervisor with other components in the SoC and the availability of transitions in the system schedule.

As explained in the companion paper, the rationale for this abstraction-based methodology is that the verification process is seen as a top-down approach, where individual components can be instantiated with more detailed models, as long as they preserve the axiomatic specifications under a suitable abstraction function (aka. refinement mapping).

The following sections serve as a documentation of the formal models and proofs and provide more technical descriptions as a supplement to what we were able to fit in the paper. The material is divided as follows.

- Section 2 introduces the *ideal model* as a specification of the system of separate interacting guests as implemented by the hypervisor. Thus it describes the desired information flows of the system by construction. The corresponding definitions can be found in `idealScript.sml`. System components and their idealized versions are defined in `axfuncsScript.sml`. File `idealInvProofScript.sml` contains the proof of the system invariant for the ideal model.

- Section 3 defines the *refined model*, i.e., the hypervisor design and the abstract model of the SoC. Again hardware components are specified in `axfuncsScript.sml`. The transition system of the refined model is contained in `refinedScript.sml` while the corresponding invariant is defined in `refinedInvScript.sml`. The hypervisor design is located in `hypervisorScript.sml`.

- Section 4 describes the *bisimulation* between the refined and ideal model. It presents the bisimulation relation as defined in `bisimScript.sml` and gives an outline of the bisimulation proof strategy. The bisimulation ensures that the same information flows exist on the ideal and refined model.

Both the ideal and refined model are parametrized, e.g., by the number of cores, peripherals, and guests, as well as the allocation of resources and the communication policy for each guest implemented by the hypervisor. These parameters are defined in `parametersScript.sml`.

In what follows we try to stick as much as possible to the names of the formal definitions. However, in order to improve readability we allow ourselves to introduce abbreviations where necessary. All abbreviations and their corresponding formal artifacts are listed in Appendix A.

2 Ideal Model

In what follows we define the idealized model of the HASPOC virtualization platform that is secure by construction. As a general approach in this document we try to keep the formalism high-level and use uninterpreted functions and types to hide hardware details. We main ideas for the model are:

- Guests are represented as separate, physically isolated virtual machines. Each guest contains a number of cores and peripherals, as well as an interrupt controller and a memory that is not shared directly with any other guests.

- In each guest, cores communicate with memory through message buffers that record the received and forwarded messages. Similarly peripherals use such message boxes to send DMA requests to memory and receive replies. However, no message boxes are used to handle memory-mapped I/O requests from the cores. They are forwarded directly from the memory component. The idea behind the message buffers is that they abstract away the second-stage memory management units (MMUs) and system MMUs that are present in the refined model. In the ideal model all addresses are intermediate physical addresses, i.e., the physical address space is hidden by the hypervisor. Address translation from virtual to intermediate physical addresses is assumed to take part within the core components and controlled completely by the guests.

- Peripherals can trigger physical interrupts, which are forwarded to cores within the same guest after being filtered by the interrupt controller. Moreover, cores can request the interrupt controller to send software-generated inter-processor interrupts (SGI) to other cores.

- Communication between guests is only possible through the IGC (Inter Guest Communication) channels; for each channel there is a corresponding memory page within the guest's memory. If a guest updates a page belonging to an outgoing channel through a write to memory, the memory page is copied into the corresponding memory region in the receiving guest. This simulates the shared memory semantics of the IGC mechanism as implemented by the hypervisor.

- Besides IGC messages guests can also send IGC notification interrupts for each of their outgoing channels. These interrupts can be triggered by the cores of that guest and are then stored in a dedicated send buffer for that channel that holds at most one message and the ID of the targetted core in the receiving guest's virtual machine. A separate transition moves the interrupts from there towards the targeted core by injecting a dedicated IGC interrupt into the receiving guest's (virtualized) interrupt controller. The targeted core is selected deterministically according to the hypervisor implementation and the power status of the cores in the receiving guest.

At any time only one such interrupt can be pending per channel. Any further interrupt requests are dropped while a former interrupt has not been acknowledged by a guest through the interrupt controller core interface.

- In order to facilitate interaction with the external world, peripherals of each guest may send and receive external event signals. These can encode all sorts of I/O data, including user input and network traffic.

An example ideal system configuration is shown in Figure 1.

2.1 Ideal Model Parameters

The ideal model is governed by a number of parameters as follows. These parameters and assumptions on them are reflecting the configuration file of the hypervisor and its associated security constraints. First there are global parameters:

- $ng \in \mathbb{N}^+$ – the non-zero number of guests in the system ($\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$)
- $ns \in \mathbb{N}$ – the number of inter-guest communication (IGC) shared memory channels (and corresponding interrupts)
- $cpol : \mathbb{N}_{ns} \to \mathbb{N}_{ng} \times \mathbb{N}_{ng}$ – a relation describing the communication policy for IGC. Each channel is represented by a pair of sender and receiver guest. We demand that for all s, s' $cpol(s) \neq cpol(s')$ if $s \neq s'$ and for all s, g that $cpol(s) \neq (g, g)$, i.e., the communication policy contains only unique entries and we do not allow channels with the same sender and receiver.
- $IGCa : \mathbb{N}_{ns} \to \mathbb{B}^{36} \times \mathbb{B}^{36}$ – the location of the shared memory pages in sender and receiver memory. Note that all channels are one page (4KiB) in size and that the maximum address size on ARM (intermediate) physical memory is 48 bits. Here $\mathbb{B} = \{0, 1\}$ is the type of bits. We require that the mapping is injective in both addresses if the channel involves the same guest, i.e., no sender or receiver address is used twice by one guest:

  $$\forall s \neq s' \in \mathbb{N}_{ns}, i \in \{1, 2\}.\ cpol(s)_i = cpol(s')_i \Rightarrow IGCa(s)_i \neq IGCa(s)_i\,.$$

  Here subscripts 1 and 2 denote the first and second component of a pair. Similarly to the requirement above, if one page is the source of one channel it must not be the target of another, i.e., all shared memory pages are dedicated to a single given channel.

  $$\forall s \neq s' \in \mathbb{N}_{ns}.\ cpol(s)_1 = cpol(s')_2 \Rightarrow IGCa(s)_1 \neq IGCa(s)_2$$

- $id_{\mathrm{IGC}} : \mathbb{N}_{ns} \to \mathbb{N}_{1024}$ – the number of the IGC notification interrupt associated with the given channel. IGC notifications appear in the receiving guest as (virtualized) peripheral interrupts that occupy range $[16 : 1019]$ on the ARM platform under verification, thus we require:

  $$\forall s \in \mathbb{N}_{ns}.\ id_{\mathrm{IGC}}(s) \in [16 : 1019]$$

Here $\mathbb{N}_n = \{i \in \mathbb{N} \mid i < n\}$ denotes the set of the first n natural numbers. Then there are local parameters characterizing each guest $g \in \mathbb{N}_{ng}$:

- $nc_g \in \mathbb{N}$ – the number of virtual cores for guest g
- $np_g \in \mathbb{N}$ – the number of peripherals associated with guest g, excluding the generic interrupt controller which is treated separatelly
- $\mathcal{P}_{g,p}$ – the states of peripheral p belonging to guest g
- $PIRQ_{g,p} \subseteq IRQ$ – the set of interrupt identifiers for interrupts sent by peripheral p of guest g. Note that we do not distinguish here between private (PPI) and shared peripheral interrupts (SPI) for simplicity. The set of interrupt IDs IRQ is defined by

  $$IRQ ::= \mathrm{pir}\,[16 : 1019] \mid \mathrm{sgi}^{\mathbb{N}_{16}}_{\mathbb{N},\mathbb{N}}\,.$$

  Here, "$\mathrm{pir}\,q$" signifies a peripheral interrupt with ID q, while "$\mathrm{sgi}^{q}_{c,d}$" stands for a software-generated interrupt q from (virtual) core c to core d. Note, that on our ARM platform there are only 16 interrupt IDs reserved for SGIs, but SGIs can be distinguished additionally by their sender and receiver core ID.

  We require that all interrupts in $PIRQ_{g,p}$ are peripheral interrupts. Moreover, IGC interrupt IDs for notification interrupts associated with incoming channels of g must be different from any peripheral interrupt number:

  $$\forall p \in \mathbb{N}_{np_g}, \mathrm{pir}\,q \in PIRQ_{g,p}, s \in \mathbb{N}_{ns}.cpol(s)_2 = g \Rightarrow id_{\mathrm{IGC}}(s) \neq q\,.$$

  Additionally, all IGC interrupt IDs are disjoint for a given guest.

  $$\forall g \in \mathbb{N}_{ng}, s, s' \in \mathbb{N}_{ns}.cpol(s)_2 = g = cpol(s')_2 \Rightarrow id_{\mathrm{IGC}}(s) \neq id_{\mathrm{IGC}}(s')$$

- $\mathcal{E}^{in}_{g,p}, \mathcal{E}^{out}_{g,p}$ – the sets of external input and output messages for guest g's peripheral p. We assume that the input and output sets are mutually disjoint, i.e., individual messages can be associated with their corresponding peripheral.
- $as_g \in \mathbb{N}_{37}$ – the virtual memory address space reserved for guest g in terms of page offset bits. For simplicity we assume here that guests receive amounts of memory pages that are a power of two. Then each guest owns a linear address space

  $$\mathbb{A}_g = \{a \in \mathbb{B}^{36} \mid a[35 : as_g] = 0^{36-as_g}\}$$

  starting at address 0^{36}. Here we use the interval notation $a[n : m]$ to denote the bit sequence $a_n, a_{n-1}, \ldots, a_{m+1}, a_m$ for $n \geq m$. If $n < m$, the empty sequence ε is returned. For $b \in \mathbb{B}$, b^n denotes the bitstring consisting of n consecutive b's. We demand for all channels s with $cpol(s) = (g, g')$. That $IGCa(s)_1 \in \mathbb{A}_g$ and $IGCa(s)_2 \in \mathbb{A}_{g'}$.

- $A_P^{g,p} \subset \mathbb{A}_g$ – the intermediate physical memory page addresses mapped to peripheral $p \in \mathbb{N}_{np_g}$ of guest g. We require that each peripheral of a guest occupies a disjoint region of memory.

$$\forall p, q \in \mathbb{N}_{np_g}.\ p \neq q \Rightarrow A_P^{g,p} \cap A_P^{g,q} = \emptyset$$

  In the formal proofs we add another constraint, demanding that peripherals have unique address regions across all guests that do not overlap with any other guest's virtual memory space. While this restriction of the hypervisor design is not strictly necessary for its correctness, it simplifies the bookkeeping of address ranges considerably.

  As mentioned above, the GIC is treated differently from other memory-mapped peripherals. In particular, the different address regions of our platform's controller (GICv2) are at fixed positions in each guest (the same as in the host). There are four regions that contain memory-mapped GIC interfaces:

  - A_{GICC} – the interrupt controller core interface, which is accessible to each guest, i.e., $A_{GICC} \subset \mathbb{A}_g$. Note that the GIC multiplexes the interface for each core at the same address, thus this is not a shared resource between guests. Nevertheless it is virtualized by the hypervisor for each core to facilitate interrupt isolation and IGC notifications.
  - A_{GICD} – the interrupt distributor, which is a centralized resource on our platform that delivers physical and software-generated interrupts to the core interfaces. Each guest has its own distributor at the same address, i.e., $A_{GICD} \subset \mathbb{A}_g$, which means that the hypervisor must virtualize and multiplex the distributor between guests.
  - A_{GICH} – the interrupt virtualization control interface is only accessible to the hypervisor, i.e., $A_{GICH} \cap \mathbb{A}_g = \emptyset$. This means that guests may not use hardware support for interrupt virtualization themselves.
  - A_{GICV} – similarly the virtual core interface is not accessible to the guest directly, i.e., $A_{GICV} \cap \mathbb{A}_g = \emptyset$. In fact, the hypervisor maps the guests' interrupt controller core interfaces to the virtual core interfaces for interupt virtualization and isolation purposes.

  The GIC interfaces will be explained in more detail in Section 3. The union of all four regions is denoted by the GIC address region A_{GIC} and we require that it is disjoint from any other memory-mapped peripheral interface for all guests g:

$$\forall p \in \mathbb{N}_{np_g}.\ A_P^{g,p} \cap A_{GIC} = \emptyset\,.$$

  Furthermore, no IGC channels are mapped onto peripheral memory:

$$\forall g, h \in \mathbb{N}_{ng}, p \in \mathbb{N}_{np_g}, q \in \mathbb{N}_{np_h}, s \in \mathbb{N}_{ns}.$$
$$cpol(s) = (g, h) \ \Rightarrow\ IGCa(s)_1 \notin A_P^{g,p} \wedge IGCa(s)_2 \notin A_P^{h,q}$$

All restrictions on the ideal model parameters are summarized by predicate `goodP` in `parametersScript.sml`. In addition, we define several abbreviations.

$$\begin{aligned} IGCin_g &= \{s \in \mathbb{N}_{ns} \mid \exists g' \in \mathbb{N}_{ng}.\ cpol(s) = (g', g)\} \\ IGCout_g &= \{s \in \mathbb{N}_{ns} \mid \exists g' \in \mathbb{N}_{ng}.\ cpol(s) = (g, g')\} \end{aligned}$$

These sets denote the in- and out-going IGC channels of guest g. Note that by construction we have for all $g \neq g'$ that $IGCin_g \cap IGCin_{g'} = \emptyset$, $IGCout_g \cap IGCout_{g'} = \emptyset$, and $IGCin_g \cap IGCout_g = \emptyset$, i.e., channels are used disjointly by guests and are either an input or output for any given guest. Similarly we can define the interrupts that a core of guest g can receive and send, using $PIRQ_g = \bigcup_{p \in np_g} PIRQ_{g,p}$ to denote all peripheral interrupts and $IPIRQ_g = \{\mathrm{sgi}^{id}_{c,c'} \mid c, c' \in \mathbb{N}_{nc_g}, id \in [8:15]\}$ to denote all inter-processor interrupts (IPI) within g. On our ARMv8 platform inter-processor interrupts with IDs 8-15 are reserved for Non-Secure execution and the hypervisor only supports these IDs for SGIs. The IGC interrupts for channels where guest g is the receiver are denoted by $IGCQ_g = \{\mathrm{pir}\,(id_{\mathrm{IGC}}(s)) \mid s \in IGCin_g\}$.

$$IRQ_g = IGCQ_g \cup PIRQ_g \cup IPIRQ_g$$

Cores can receive IGC interrupts, peripheral interrupts, and inter-processor interrupts from all the other cores in the same guest. On the other hand, cores do not send any interrupts directly. Inter-processor interrupts need to be required from the GIC explicitly (as emulated by the hypervisor). IGC interrupts are send through the hypervisor using a system call.

The state of the overall model is made up of a number of guest states including their core and peripheral states as well as the IGC channels containing the data sent from one guest to another one. We formulate the details below.

2.2 Memory Messages

Before we can define the ideal model semantics, we need to clarify how we model messages between the memory and the other components in the system, as they represent the main mode of communication, both in the refined and ideal model.

First, we introduce memory requests that can be sent to memory by cores or peripherals. Requests have type $\mathbb{R}$ as defined by the following abstract grammar.

$$\mathbb{R} ::= \mathrm{R}\ \mathbb{B}^{48}\ \mathbb{N}\ \mathcal{X} \mid \mathrm{W}\ \mathbb{B}^{48}\ \mathbb{N}\ \mathbb{B}^{*}\ \mathcal{X} \mid \mathrm{PTW}\ \mathbb{B}^{48}\ \mathcal{X}$$

Requests can be reads R a d x of d bytes from address a, writes W a d v x of a d-byte value v to address a, or page table lookups PTW a x, requesting the read of a 8-byte page table descriptor from address a. Here $x \in \mathcal{X}$ represents some additional message information, like a request ID, and access parameters, like memory types or cacheability attributes. We leave $\mathcal{X}$ uninterpreted here as it depends on the underlying ARMv8 model and all information needed for the proof is contained in the other fields of a request. However, an underlying

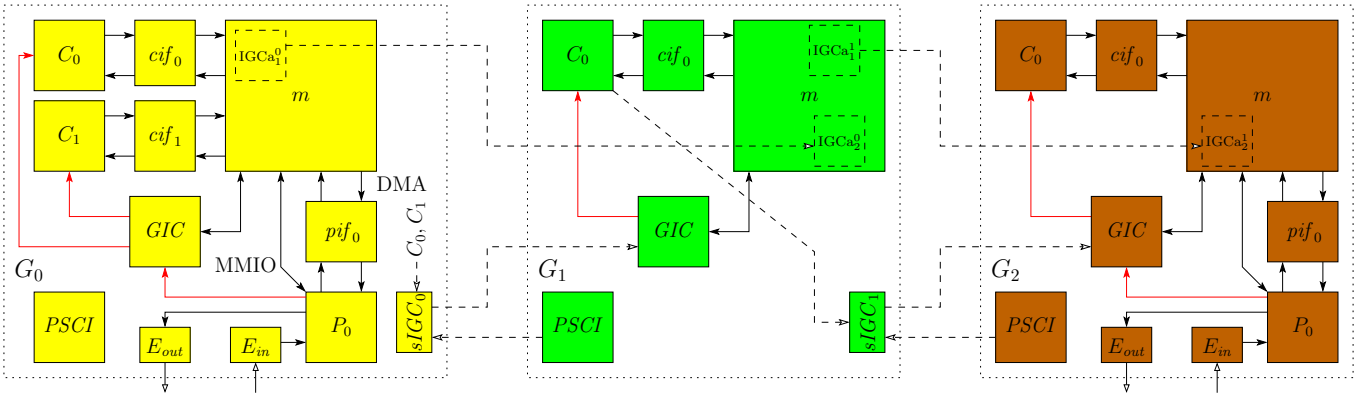

Figure 1: Example configuration of an ideal system with three guests, where G_0 owns two cores C_0 and C_1, one peripheral P_0, and some extra paraphernelia for buffering and signalling exlained below, G_1 owns only one core, and G_2 owns one core and one peripheral. Guest systems are separated modulo the IGC channels allowed from G_0 to G_1 and G_1 to G_2. Plain arrows denote inner guest data flow, red arrows show interrupt signalling and control, while dashed arrows represent IGC data and signalling flow.

assumption of our model is that each request has a unique message information field, thus no two requests are ever identical.

Note that we do not distinguish instruction and data accesses for simplicity. As the second-stage address translation set up by the hypervisor does not restrict the executability of guest memory, and also does not use permissions to restrict guest access to memory, instruction fetches and data accesses are handled in the same way, governed only by the validity bit of page table entries, marking pages as mapped or unmapped.

Moreover, the address in the request can be interpreted either as physical or intermediate physical and we use the same type for both cases depending on the context.[1] Note however, that requests with physical addresses only appear in the refined model (after translation by the (S)MMUs).

Once a request is processed by a receiver it may generate a reply for the sender. Replies have type $\mathbb{Q}$ that is defined as follows.

$$\mathbb{Q} ::= rplR\ \mathbb{R}\ \mathbb{B}^* \mid rplW\ \mathbb{R} \mid rplPTW\ \mathbb{R}\ \mathbb{B}^{64} \mid \mathrm{F}\ \mathbb{R}\ \mathcal{F}$$

Replies contain the request they are answering as well as a return value for read and page table walk requests. In case of writes, the reply does not contain further information and is meant as an acknowledgment message. A special case are fault replies $\mathrm{F}\ r\ f$ which indicate that request r caused a fault, e.g., for

[1] In retrospect, this decision was a case of premature optimization due to inexperience with HOL4 and it came to haunt us in the formal proofs when it was too costly to change the basic definitions. In general it is advisable to encode semantical differences using the type system, as we know now.

permission errors or when addresses are not mapped to peripherals or RAM. Additional fault information is provided via $f \in \mathcal{F}$. As above we leave the exact structure of $\mathcal{F}$ undefined. Note, that replies are only well-formed if the type of the reply matches the type of the request, e.g., for $rplW\ r$ request r must be a write request. Throughout our development we only consider such *good* replies.

We define a predicate $match : \mathbb{R} \times \mathbb{Q} \to \mathbb{B}$ to denote that a reply matches a request. It is defined by distinguishing the possible cases for requests below. Observe that matching replies are always well-formed, i.e., good replies. Note also that for each reply there exists only one unique matching request.

$$match(r, q) \equiv \exists f.\ q = \mathrm{F}\ r\ f \vee \exists v.\ q = \begin{cases} rplR\ r\ v & : \quad r = \mathrm{R}\ a\ d\ x \\ rplW\ r & : \quad r = \mathrm{W}\ a\ d\ v\ x \\ rplPTW\ r\ v & : \quad r = \mathrm{PTW}\ a\ x \end{cases}$$

When cores and peripherals send requests to memory, they are first received by the message boxes in the ideal model. As each of these components is dedicated to a specific core or peripheral, it is easy to associate received requests with their sender. Once requests are forwarded to the memory, however this implicit information is lost and we need to record it explicitly. The same holds for replies that need to be returned from memory to the requesting component. To this end we introduce sender identifiers $\mathbb{S}_g$ for messages within guest g:

$$\mathbb{S}_g ::= \mathrm{C}\ \mathbb{N}_{nc_g} \mid \mathrm{P}\ \mathbb{N}_{np_g}$$

Naturally, C c represents core c here, while P p stands for peripheral p. Then messages within the memory subsystem are represented as pairs of a sender identifier and a request ($\mathbb{R} \times \mathbb{S}_g$), or a reply ($\mathbb{Q} \times \mathbb{S}_g$).

Note that peripheral sender identifiers are only used for DMA messages. Core requests to memory-mapped I/O regions are forwarded by the memory to the targeted peripheral, including the sender core information. This is especially important for the interrupt controller that has I/O interfaces which are sensitive to the ID of the requesting core.

Finally we define the signatures for the message buffers at the interfaces between memory and cores, or peripherals respectively.

$$BUF = \{RR : \mathbb{R}^*;\ RS : \mathbb{R}^*;\ QR : \mathbb{R}^*;\ \}$$

Here RR records received requests that have not been answered yet, RS keeps track of requests that were forwarded to memory, and QR contains all replies that were received from memory for outstanding requests. We do not need to record forwarded replies as they are consumed by the cores or peripherals once they leave the message buffer.

Note that for historical reasons we use r for requests and q for replies in this document. This was changed in the JCEN paper as q for "query" may be more intuitive. We stick to the old notation here because it is also used like that in most cases for the formal definitions.

2.3 Guest Model

Guests are modeled as a record structure that is slightly different for each guest depending on the local parameters. We denote the guest state of guest g by $\mathbb{G}_g$ that contains the following components:

- $C : \mathbb{N}_{nc_g} \to \mathcal{C}_{id}$ – the states of all cores in guest g. The core states in the ideal model represent a reduced virtualized version of the actual ARMv8 cores. These virtual cores run only in exception level EL1 and EL0, and their MMU has only one stage of translation. As mentioned before this first stage of the MMU is considered part of the core.

  Since we do not want to introduce the full ARMv8 core model for the proof of the hypervisor design, the ideal core model, i.e., type $\mathcal{C}_{id}$ and its associated labeled transition system, is undefined here. In order to still be able to talk about important core components, we will introduce an abstract model of the below, that must be refined by the actual model.

- $cif : \mathbb{N}_{nc_g} \to BUF$ – the message buffers at the interface between cores and memory. Each core has one buffer that all of its memory messages have to pass.

- $m \in \mathcal{M}_g$ – the virtual memory of guest g. Again as we do not want to model the whole memory subsystem of our platform we represent it here via a placeholder type $\mathcal{M}_g$. Below we will introduce a sequentially consistent memory abstraction in order to be able to speak about the actual memory contents. The abstraction maps the guest's address space to 4KiB pages of memory.

- $P : \mathbb{N}_{np_g} \to \mathbb{P}_g$ – the peripheral states for guest g. Here $\mathbb{P}_g$ is a record $\{st : \mathcal{P}_g\,;\, ior : \mathbb{R} \to \mathbb{S}_g \cup \{\bot\}\}$ which contains the actual peripheral state component st and history information about the received and unanswered I/O requests and their senders.

  In particular, $\mathcal{P}_g = \bigcup_{p<np_g} \mathcal{P}_{g,p}$ and we demand that $P(p).st \in \mathcal{P}_{g,p}$ holds for all $p \in \mathbb{N}_{np_g}$. Moreover, if $P(p).ior(r) = s$ then a memory-mapped I/O request r from sender s was received by peripheral p and not answered yet. If ior is undefined for a given request then no such request is pending within the peripheral from any sender.

- $pif : \mathbb{N}_{np_g} \to BUF$ – the message buffers at the interface between peripherals and memory. Each peripheral has one buffer to be used for DMA accesses only. The buffers are bypassed by memory-mapped I/O accesses targetting the corresponding peripheral.

- $GIC \in \mathcal{G}_{id}$ – the state of the ideal interrupt controller of guest g. We treat the GIC as a special peripheral here since it both receives and sends interrupts from peripherals and to cores. Each guest has its own virtualized interrupt controller that manages the interrupts dedicated to that

particular guest. Again we do not give a detailed model for $\mathcal{G}_{id}$, but instead provide an abstract GIC semantics that only exposes the necessary architectural details and reflects the virtualization of interrupts by the hypervisor.

- $psci \in PSCI_g$ – the power controller of guest g, as virtualized by the hypervisor. Each guest may use the controller to power up and down its own cores. Within our model the state of the power controller is a record,

$$PSCI_g = \{pow : \mathbb{N}_{nc_g} \to \mathbb{B}\,;\; cmds : \mathbb{N}_{nc_g} \to CMD_g^*\}\,,$$

  where $pow(c)$ records whether core c of guest g is powered up and $cmds(c)$ is a queue of outstanding power control commands for c. We cover only powering up and down cores, hence $CMD_g ::= \text{start}\ \mathbb{N}_{nc_g} \mid \text{stop}\ \mathbb{N}_{nc_g}$. The power controller issues these commands to the targeted cores one by one, popping them from the queue. Since the guests do not have direct access to the power controller but rely on the hypervisor to issue commands, this rudimentary model of the power control interface suffices to describe the behaviour of the system at a high level of abstraction.

- $E_{out} \in \mathcal{E}_g^{in*}$ – a list of external inputs for the peripherals of guest g. Here $\mathcal{E}_g^{in} = \bigcup_{p<np_g} \mathcal{E}_{g,p}^{in}$ consists of the inputs for all peripherals. For simplicity we have one external interface per guest that asynchronously receives inputs from the external world and adds them to the list. From their they are distributed to the corresponding peripherals.

- $E_{out} \in \mathcal{E}_g^{out*}$ – a list of external outputs for the peripherals of guest g. Similar to the input interface we have $\mathcal{E}_g^{out} = \bigcup_{p<np_g} \mathcal{E}_{g,p}^{out}$ and the outputs from peripherals to the external world are recorded at a single point by adding them to the list.

- $sIGC \in \{pend : IGCout_g \to \mathbb{B}\,;\; tgt : IGCout_g \to \mathbb{N} \cup \{\bot\}\}$ – the state of the notification interrupt requests for the outgoing channels of guest g. The *pend* component records for which of these channels a notification interrupt is pending to be sent to the targeted guest. The second component records the targeted core for the interrupt within that guest. If $tgt(s) = c$ and $cpol(s) = (g, g')$ then we demand that $c \in \mathbb{N}_{nc_{g'}}$. Reflecting the hypervisor implementation, the target core in the ideal model will be selected according to the power state of cores in the receiving guest, i.e., a core is selected that is currently powered up.

Example instantiations of different guests and their connections are depicted in Figure 1. In the context of the JCEN paper, one guest is modeled as an instantiation of the generic system model, where actions that communicate between guests are considered external. Below we introduce the abstract models and transition systems for the different guest components.

2.3.1 Ideal Core Model

In order to describe the core semantics and prove the bisimulation with the refined model, we need to expose the crucial components that are architecturally visible to a guest in the ideal model. To this end we introduce an abstract core model with states $\mathcal{C}_{id}^{\text{abs}}$ that have the following components:

- $active \in \mathbb{B}$ – stores if the core is powered up and running
- $ps \in \mathbb{P}$ – stores the current processor state of the core as a record structure with type $\mathbb{P}$. We omit a detailed description of all its fields besides $mode \in \mathbb{B}$ which is the current mode of the core, either EL1 (1) or EL0 (0), and the IRQ mask bit $IRQm \in \mathbb{B}$ that disables asynchronous exceptions due to interrupts sent from the GIC.
- $pc \in \mathbb{B}^{64}$ – the program counter of the core
- $gpr : GPR_{\mathbb{G}} \to \mathbb{B}^{64}$ – this stores all the user (EL0) accessible registers of the core, e.g., general purpose registers, stack pointer registers, process state register, etc. We simplify here by assuming that all these registers are 64 bits wide.
- $spr : SPR_{\mathbb{G}} \to \mathbb{B}^{64}$ – this component collects all the special purpose registers that control system functionality and are accessible at EL1 or EL0, e.g., system control register, stage 1 MMU control, timer register, debug functionality, and so on.
- $RS \subseteq \mathbb{R}$ – a history variable recording currently unanswered memory requests sent by the core,
- $launch \in \mathbb{B}$ – a history variable saying that the core was just powered up.
- $int \in \mathcal{C}_{\text{int}}$ – other internal components of the core that we leave unspecified, e.g., the current instruction or the state of the pipeline. The internal state is only defined by a given instantiation of the core model. We use it mainly to couple the cores in the ideal and refined model, requiring that cores with the same internal state are able to perform the same operation. Moreover, some information from the internal state is needed when taking an exception.

We introduce an abstraction function $\lceil\cdot\rceil : \mathcal{C}_{id} \to \mathcal{C}_{id}^{\text{abs}}$ to map ideal cores C to their abstract state $\lceil C \rceil$. For simplicity we will use the same notation for all component abstraction functions.

The semantics of the ideal core is given by a labeled transition system $\xrightarrow{a}_{\mathcal{C}_{id}}$: $\mathcal{C}_{id} \times Act_{\mathcal{C}_{id}} \times \mathcal{C}_{id}$, where $Act_{\mathcal{C}_{id}}$ is a set containing the following action labels identifying the core transitions:

SndReq r – send memory request $r \in \mathbb{R}$ to the message buffer

RcvRpl q – receive memory reply $q \in \mathbb{Q}$ from the message buffer

RcvPhys – take an asynchronous exception due to an interrupt from the GIC

SndPow l – send a command list $l \in CMD_g^*$ to the power controller

RcvPow x – receive command $x \in CMD_g$ from the power controller

SndIGC s – request an IGC notification interrupt on channel $s \in \mathbb{N}$

Internal – internal step, e.g., an arithmetic or logical operation

StartUp g c – start executing as core c of guest g after power up

Note that the startup action is parametrized by guest index g and core index c. It initializes the core according to its index within the guest and the entry point for the guest.

The actions as listed abouve can be mapped to the send, receive, and internal actions in the generic system model of the JCEN paper in a straight-forward way, defining type $\mathbb{M}$ as the superset of all send and receive actions, and defining snd and rcv as a disjounction of the respective send and receive transitions. The internal action τ may multiplex Internal and StartUp g c by firing the latter whenever *launch* is active and hard-coding g and c according to the ideal core index that is being defined.

The precise semantics of the actions are depending on the core model instantiation, however we restrict any such instantiation via the core abstraction. The detailed proof obligations can be found in `axfuncsScript.sml`. Here we just provide an overview for each transition. As discussed in the paper, restrictions come in the form of safety properties, i.e., conditions on the abstract pre and post states, and liveness properties that determine when certain transitions must be enabled. That means that all of the following conditions are to be interpreted wrt. to some pre state $\lceil C \rceil$ and post state $\lceil C' \rceil$ of the corresponding transition.

Send Memory Request Whenever a request r is sent to the message buffer via SndReq r, the following preconditions hold:

- the core is active,
- no memory request is pending, i.e., $RS = \emptyset$, and
- the exception level is at most EL1, i.e., the ideal cores never exceed OS privilege level.

For the resulting ideal core state the following postconditions hold:

- the core is still active,
- the memory request r is recorded in RS,
- the history variable *launch* is unchanged,

- the internal state is updated to record any instruction accessing the GIC distributor via r, and
- the exception level is unchanged.

Note that the requirement to have no pending memory requests essentially forces any core instantiation to issue memory requests sequentially, i.e., it does not allow to have several memory requests pending at the same time.

We initially put the restriction here to reduce the complexity of the model, however we do not use it in the defintion of the invariants and bisimulation relation and it does not seem to simplify the proof significantly, hence it can be dropped if needed. However, this has implications on the definition of exception handling.

The restriction makes it harder to model side-effects of out-of-order execution in the cores, however it does not rule it out completely, as the core in our system not only contains the execution units but also the first stage MMU including corresponding TLB(s) and may contain virtually-indexed L1 caches that can satisfy out-of-order memory accesses.

In any case, the sequence of memory accesses issued by the core is in no way bound to a program order. The number of simultaneous memory accesses is in fact bound by the memory subsystem and thus depending on the specific hardware platform, hence it should be restricted using a model parameter (which in this development is implicitly assumed to be one).

Since it depends on the current instruction(s) and first stage MMU whether a memory request is sent by the core, we do not require a liveness property stating when the transition must be enabled. Instead, for the bisimulation proof we require an explicit bisimulation property that states when ideal and refined core must be able to perform the same guest step, including sending the same memory request. We will revisit this property when introducing the refined core model.

Receive Memory Reply If a memory reply q is received from the message buffer via RcvRpl q, the following preconditions hold:

- the core is active,
- a matching memory request r is pending in RS that matches q, and
- the exception level is at most EL1

For the resulting ideal core state the following postconditions hold:

- the core is still active,
- the memory request r is removed from RS,
- the history variable *launch* is unchanged,
- the internal state is updated to reflect the reception of replies for memory-mapped I/O accesses to the GIC distributor,

- the exception level is unchanged except when a fault is received,
- for supported read accesses to the GIC distributor the corresponding target register in the core is explicitly updated with the received read value according to the core semantics

The ideal core must be enabled to perform the transition whenever the pre-conditions hold, in particular when a matching request r is pending.

There are additional restrictions for the reception of MMU and memory faults, that cause the ideal core to take an exception. We have the following additional post-conditions:

- no memory request is pending, i.e., $RS = \emptyset$,
- the PC is set to the entry point of the corresponding exception handler, according to the interrupt vector table and the required offset, depending on the processor status register,
- the exception level is updated to EL1,
- the GPRs are unchanged,
- $spr(ELR\_EL1)$ contains the old PC,
- $spr(ESR\_EL1)$ contains extracted fault information from q,
- $spr(FAR\_EL1)$ contains the faulting address information,
- $spr(SPSR\_EL1)$ contains the old value of the processor status register,
- other special purpose registers of EL1 are unchanged, and
- the internal state indicates that the exception flushed the pipeline.

The enabling conditions for faults are the same as for regular replies.

Take Asynchronous Exception If action RcvPhys triggers an asynchronous exception due to an interrupt from the GIC, the following pre-conditions hold:

- the core is active,
- no memory request is pending, i.e., $RS = \emptyset$,
- the internal state indicates the pipeline as flushed, and
- IRQs are not masked in the processor status register.

After the exception we have the following postconditions:

- the core is still active,
- no memory requests were issued,

- the history variable *launch* is unchanged,
- the internal state still indicates the pipeline as flushed,
- the PC is set to the entry point of the corresponding interrupt handler, according to the interrupt vector table and the required offset,
- the exception level is updated to EL1,
- the GPRs are unchanged,
- $spr(ELR\_EL1)$ contains the old PC,
- $spr(ISR\_EL1)$ indicates an IRQ,
- $spr(SPSR\_EL1)$ contains the old value of the processor status register,
- other special purpose registers of EL1 are unchanged.

Ideal physical interrupts can always be received if the preconditions hold. Additionally the exception level must be at most EL1. Note that this requires any instantiation to be ready to take an asynchronous exception when the internal state is indicating the pipeline as being flushed (and only then). We will use this requirement in the bisimulation by coupling the internal states of the refined and ideal cores, such that (virtual) interrupts may always be received simultaneously in both models.

Request PSCI Functions Action SndPow l requests a list l of PSCI commands to be processed by the power controller (via the hypervisor as a proxy). The preconditions for the action are:

- the next instruction is a secure monitor call (SMC),
- its argument encodes list l of allowed commands (after filtering out bad commands in the hypervisor),
- no memory request is pending, i.e., $RS = \emptyset$
- the core is active and in EL1.

Then the following postconditions hold after the request is executed:

- the core is still active and in EL1,
- no memory requests were issued,
- the history variable *launch* is unchanged
- the internal state indicates that the pipeline was flushed (due to the call to the hypervisor and the corresponding exceptions to EL2 and EL3 in the refined model).

PSCI commands can always be requested if the preconditions hold.

Perform PSCI Commands If the ideal core performs action RcvPow x it receives a PSCI command x from the power controller and starts up or shuts down accordingly. The following preconditions hold for the transition:

- no memory request is pending, i.e., $RS = \emptyset$,
- the internal state indicates the pipeline as flushed (which is always the case for inactive cores),

Then the following postconditions apply:

- no memory requests were issued,
- the history variable *launch* and the power state depend on e:
  - for a start command, if the core is inactive, set *launch* to *true*, no other changes are made,
  - for a stop command, if the core is active, make it inactive and set *launch* to *false*,
  - if the core is already in the desired state, ignore the command,
- the internal state still indicates the pipeline as flushed.

Observe that the preconditions ensure that a core is never powered down in the middle of a memory request, i.e., it shuts down gracefully.

Request IGC notification For action SndIGC s, the core requests an IGC notification interrupt (from the hypervisor) for channel s. In this case the following precnditions hold:

- the next instruction is a hypercall (HVC),
- its argument encodes channel s, where $s = ns$ means that a non-existing or invalid channel was requested,
- no memory request is pending, i.e., $RS = \emptyset$,
- the core is active and in EL1.

After the call we have the postconditions:

- the core is still active and in EL1,
- no memory requests were issued,
- the history variable *launch* is unchanged, and
- the internal state indicates the pipeline as flushed (due to the call to the hypervisor).

IGC notifications can always be requested if the preconditions hold.

Internal Step Whenever the ideal core performs an internal step, we require that:

- core is active, and
- no requests to memory are pending.

Then we have the following postconditions

- the core is still active,
- no memory requests were issued,
- the history variable *launch* is unchanged
- the internal state may indicate that the pipeline was flushed, and
- the exception level may change but not higher than EL1.

Since internal steps can flush the pipeline, they often precede asynchronous exceptions, as these cannot be taken otherwise. This allows an instantiation some flexibility as to the definition of the internal state, since it can encapsulate the flushing of the pipeline before an exception into a dedicated internal step.

Start-up Step Ideal core c of guest g starts execution after a reset or power up via the StartUp g c action. It has only a single required pre-condition:

- the launch bit is set.

The transition has at least the following effects:

- all registers are set to their initial (potentially guest and core-specific) values,
- the core is active,
- *launch* is set to false,
- no memory requests were issued, and
- the internal state indicates the pipeline as flushed.

When a core is powered up in the refined model the EL3 boot code and EL2 hypervisor initialization is executed before the guest code can start executing. This boot phase is invisible in the ideal model. Instead there is one atomic startup transition that performs all the initialization steps that are implemented by the boot loader and the hypervisor.

Ideal Core Invariant The transitions of the ideal core establish a core invariant $Inv_{\mathcal{C}_{id}} : \mathcal{C}_{id}^{\text{abs}} \to \mathbb{B}$, that requires the following properties from the abstract ideal core configuration:

- at most one memory request is pending in RS,
- the exception level is at most EL1,
- inactive cores have no pending memory request and a flushed pipeline,
- if the *launch* bit is set, the core is inactive,
- if the pipeline is flushed, no memory request are pending,
- if the internal state indicates a pending supported read or write access to the GIC distributor, then (and only then) a corresponding memory mapped I/O read or write access to the GICD is pending,

This completes the definition of the ideal core model.

2.3.2 Power Controller Model

As power control is completely handled by the hypervisor and boot code, we only define a simplified model here. We do not use an abstraction function for the power controller but instead define the semantics for the ideal power controller directly. The same model will be used for the refined power controller with the difference that it talks to all cores of the system, using their refined model identifiers.

Recall that the state of the power controller $psci \in PSCI_g$ contains two components $psci.pow$ and $psci.cmds$ that model the power states and outstanding commands for the cores of guest g.

The semantics of the power controller of guest g is given by a transition relation $\xrightarrow{a}_{PSCI_g} : PSCI_g \times Act_{PSCI} \times PSCI_g$ for actions Act_{PSCI} that are defined as follows:

PowRcv l – receive a command list $l \in CMD_g^*$ from a core,

PowSnd x – send command $x \in CMD_g$ to the corresponding core,

PowIGC s c – send information for incoming IGC channel $s \in IGCin_g$ that notification interrupts should target the powered up core $c \in \mathbb{N}_g \cup \{\bot\}$. If no core is currently powered up then $c = \bot$ is reported.

Note that the first two actions carry the same information as the corresponding actions in the core. In fact PowRcv l and PowSnd x are just synonyms for SndPow l and RcvPow x that we introduce here as not to confuse the reader. In terms of the JCEN paper, the same message types are used for communication between the core and the power controller. The last action represents an external message that is sent to a different guest, namely the sender for channel s.

Receiving PSCI Requests This state transition is in fact a function, and $psci \xrightarrow{\text{PowRcv } l}_{PSCI_g} psci'$ holds if and only if for all $c \in \mathbb{N}_g$:

$$psci'.pow = psci.pow$$
$$psci'.cmds(c) = psci.cmds(c)@(l|_c)$$

Here "@" denotes list concatenation and $l|_c$ filters out all commands from l that are not of the form start c or stop c. Thus the power controller pushes all received to power commands to the queue for the core that is being targeted.

Sending PSCI Command When a command is sent, i.e., $psci \xrightarrow{\text{PowSnd } x}_{PSCI_g} psci'$, the following preconditions hold:

- if x is start c or stop c then x is a the head (hd) of the command queue for core c:
$$hd(psci.cmds(c) = x$$

We have the following postconditions:

- command x is popped from the corresponding queue:
$$psci'.cmds(c) = \begin{cases} tl(psci.cmds(c)) & : \quad x \in \{\text{start } c, \text{stop } c\} \\ psci.cmds(c) & \text{otherwise} \end{cases}$$

- the local power state record is updated accordingly:
$$psci'.pow(c) = \begin{cases} true & : \quad x = \text{start } c \\ false & : \quad x = \text{stop } c \\ psci.pow(c) & \text{otherwise} \end{cases}$$

Observe that the transition relation is again a function.

Sending Power Status for IGC Channel Transition $psci \xrightarrow{\text{PowIGC } s\ c}_{PSCI_g} psci'$ reflects a part of the hypervisor implementation of IGC notification interrupts. Before an IGC interrupt is sent to a guest, the hypervisor needs to determine a target core for the inter-processor interrupt used to implement the IGC notification. To this end it simply selects the core with the lowest index among the powered up cores of the receiving guest g.

As the hypervisor is only modeled explicitly in the ideal model we need to define a corresponding transition that transfers the corresponding information in to the IGC message buffer at the sender side. The power controller is the only entity in the receiver guest that "knows" about the power state of cores, therefore we add the PowIGC action to represent this part of the hypervisor implementation.

The action itself does not modify the power controller state as it is just a query, hence $psci' = psci$. However we require the following precondition.

$$c = \min\{c' \mid psci.pow(c')\}$$

Naturally, this transition only exists for the ideal power controller and is omitted from its refined counterpart.

2.3.3 Ideal Memory Bus Inferface Model

Similar to the power controller, also the semantics of the message buffers at the memory bus interface is defined explicitly. We give a common semantics for both core and peripheral message buffers that abstract from the second stage MMUs and the SMMUs, respectively. In the overall guest semantics additional constrains will define the differences in their behavior, mainly wrt. fault generation.

We introduce transition relation $\xrightarrow{a}_{BUF}$: $BUF \times Act_{BUF} \times BUF$ for the message buffers where Act_{BUF} defines the following actions:

BufRReq r – receive a memory request r from a core or peripheral (DMA),

BufSReq r – forward a received request r to memory,

BufRRpl q – receive a reply q from memory,

BufSRpl q – forward a received reply q from memory to the requesting core or peripheral,

BufFault q – generate a fault reply q for an illegal memory request and send it the requesting core or peripheral.

Note again, that BufRReq r and BufSRpl q are just synonmyms for the corresponding core actions SndReq r and RcvRpl q. Similarly BufFault q is a special case of RcvRpl q where q is a fault.

We now give the precise definitions for the different cases of the transition relation. To this end we use an inference-rule style of definition, however the upper and lower expressions are meant imply each other. All free variables that do not appear in the lower expression are assumed to be existentially quantified. The transition relation is exhaustively defined by the following five cases.

$$\frac{r \notin B.RR \qquad B' = B[RR \mapsto B.RR \cup \{r\}]}{B \xrightarrow{\text{BufRReq } r}_{BUF} B'} \text{ BufRcvReq}$$

A message may received if it is not yet pending in the buffer, then the message is added to the set of received and pending messages. The other message records, i.e., $B.RS$ and $B.QR$, are unchanged.

$$\frac{r \notin B.RS \qquad \forall q \in B.QR.\ \neg match(r,q) \qquad B' = B[RS \mapsto B.RS \cup \{r\}]}{B \xrightarrow{\text{BufSReq } r}_{BUF} B'} \text{ BufSndReq}$$

When forwarding a request r, it must have been received and not have been sent before, thus it is not pending in $B.RS$, nor was it answered already by a reply in $B.QR$. The sent messages record is updated, other records are unchanged.

$$\frac{r \in B.RS \qquad match(r,q) \qquad \begin{array}{c} r \in B.RR \\ B' = B[RS \mapsto B.RS \setminus \{r\}; QR \mapsto B.RS \cup \{q\}] \end{array}}{B \xrightarrow{\text{BufRRpl } q}_{BUF} B'} \text{BufRcvRpl}$$

In the BufRRpl q action, the buffer receives a reply q for a pending request r that was sent to memory. Note that only one such matching request r can exist according to the definition of *match*. The reply is registered in $B.QR$ and r is removed from $B.RS$, it is still pending in $B.RR$ though.

$$\frac{q \in B.QR \qquad match(r,q) \qquad \begin{array}{c} r \in B.RR \\ B' = B[RS \mapsto B.RR \setminus \{r\}; QR \mapsto B.QR \setminus \{q\}] \end{array}}{B \xrightarrow{\text{BufSRpl } q}_{BUF} B'} \text{BufSndRpl}$$

Finally, a received reply q may be forwarded to the requesting core or peripheral, if a corresponding matching request r is pending. Both r and q are removed from the message history variables.

$$\frac{q \notin B.QR \qquad \forall r' \in B.RS.\ \neg match(r',q) \qquad \begin{array}{c} q = \mathrm{F}\ r\ f \qquad r \in B.RR \\ B' = B[RS \mapsto B.RR \setminus \{r\}] \end{array}}{B \xrightarrow{\text{BufFault } q}_{BUF} B'} \text{BufFault}$$

Sending a generated fault reply is different from forwarding a reply that can be a fault generated by a peripheral for a memory-mapped I/O access or by the memory (even though we do not consider the latter in this work). We do not specify here in which cases a fault may be generated. However, if it is, there must be a request r pending that matches q and was not forwarded yet. Moreover q must not be a received fault reply. Similar to BufSRpl q, the faulting request is removed from $B.RR$, but all other variables are unchanged.

2.3.4 Memory Model

In order to define the memory semantics we define an abstract memory model $\mathcal{M}_g^{abs}$ for the memory of guest g. It is a record with the following components:

- $m : \mathbb{A}_g \to \mathbb{B}^{4096 \cdot 8}$ – the virtual memory of guest g modeled as a mapping of intermediate physical addresses to pages,
- $RR \subset \mathbb{R} \times \mathbb{S}_g$ – the set of received / pending memory requests from cores or peripherals with a given sender ID that have not yet been answered,
- $RS \subset \mathbb{R} \times \mathbb{S}_g$ – the set of forwarded memory-mapped I/O (MMIO) requests along with the corresponding sender ID, which are just cores in this work,

- $QR \subset \mathbb{Q} \times \mathbb{S}_g$ – the set of pending replies that were received from peripherals for MMIO requests from the cores, waiting to be forwarded to the recorded senders.

We overload notation and introduce functions $\lceil \cdot \rceil : \mathcal{M}_g \to \mathcal{M}_g^{\text{abs}}$ that abstract from the concrete guest memory states to the abstract ones. In practice we expect only one instantiation and abstraction function for the guest memories that are parameterized by the guest address space range and its owned peripheral I/O regions.

Relation $\xrightarrow{a}_{\mathcal{M}_\subseteq} \mathcal{M}_g \times Act_{\mathcal{M}} \times \mathcal{M}_g$ denotes transitions of the memory for actions $a \in Act_{\mathcal{M}}$. These are defined as follows:

MemRReq r id – receive a memory request r from a core or peripheral id,

MemSReq r id – forward a received MMIO request r to a peripheral or the GIC on behalf of core id,

MemRRpl q id – receive an MMIO reply q from a peripheral or the GIC for to be forwarded to core id,

MemSRpl q id – send or forward reply q to the requesting core/peripheral id wrt. a pending request,

MemIntrnl id – perform an internal step for a request by core/peripheral id.

Here we have $r \in \mathbb{R}$ for requests, $q \in \mathbb{Q}$ for replies, and $id \in \mathbb{S}_g$. Below we give a summary of the abstract specifications for the memory transitions, the detailed definition can be found in `axfuncsScript.sml`. Note that we use the same memory semantics for the ideal and refined model, the only difference being the memory range exposed to a given guest g in the ideal model.

Receive Memory Request If a memory request r is received from the message buffer of sender id via MemRReq r id, the behavioral specification does not impose any preconditions on the step. For the resulting abstract memory state the following postconditions hold:

- the pair (r, id) is added to RR,
- all other abstract state variable, in particular the memory contents, are unchanged.

As an enabling specification we require that the memory is always able to receive a memory request. We ignore the case where the exact same request from the same sender is already pending, because we simplify the cores and peripherals to send only one request at a time. Additionally, we assume that any practical instantiation will add serial numbers to the message information so that requests can be identified uniquely. Then no two identical requests from the same sender may exist in the system anyway.

Forward MMIO Request If a memory-mapped I/O request r is forwarded on behalf of core id, i.e., MemSReq r id is executed, we require the following preconditions:

- the address of r lies in the GIC area A_{GIC} or in the memory-mapped I/O region $A_P^{g,p}$ for some peripheral $p \in \mathbb{N}_{np_g}$,
- the pair (r, id) is pending in RR,
- the request was not sent yet, i.e., (r, id) is not pending in RS,
- no replies matching (r, id) are pending QR, e.g., because r was already sent earlier.

Then the transition has the following effects:

- the pair (r, id) is added to RS,
- all other abstract state variable, in particular the memory contents, are unchanged.

The enabling specification requires that any request (r, id) that fulfills the necessary preconditions listed above may be sent by the memory.

Receive MMIO Reply For action MemRRpl q id a reply q is received for an earlier request of sender id. We require the following preconditions:

- there exists a request (r, id) in RS,
- request r matches reply q, i.e., $match(r, q)$,
- the address of r lies in the GIC area A_{GIC} or in the memory-mapped I/O region $A_P^{g,p}$ for some peripheral $p \in \mathbb{N}_{np_g}$.

The transition has the following effects:

- the pair (q, id) is added to QR,
- the pair (r, id) is removed from RS,
- all other abstract state variable, in particular the memory contents, are unchanged.

The enabling specification requires that any reply (q, id) that fulfills the necessary preconditions listed above may received by the memory.

Send Reply for Received request When performing action MEMSRPL q id a reply q is returned to sender sender id for a currently pending request. We distinguish whether q is a reply to an MMIO request or a generated reply for a regular memory access. In the former case we have the following preconditions:

- there exists a request (r, id) in RS,
- request r matches reply q, i.e., $match(r, q)$,
- the address of r lies in the GIC area A_{GIC} or in the memory-mapped I/O region $A_P^{g,p}$ for some peripheral $p \in \mathbb{N}_{np_g}$.

The transition has the following effects:

- the pair (q, id) is removed from QR,
- the pair (r, id) is removed from RS,
- all other abstract state variable, in particular the memory contents, are unchanged.

In the other case, i.e., where the memory actually performs an operation on its contents, we need to distinguish reads, writes, and page table walks. The necessary preconditions for sending reply q are summarized as follows:

- there exists a request (r, id) in RS,
- request r matches reply q, i.e., $match(r, q)$,
- (q, id) is not contained in QR, i.e., it not the reply to some MMIO request,
- the address of r lies not in the GIC area A_{GIC} or some memory-mapped I/O region,
- the reply is not a fault, i.e., fault replies from memory are due to forwarded I/O replies only. We assume here that there are no intrinsic memory faults like parity or CRC check violations due to bad hardware, tampering, etc.

Executing the transition for different kinds of requests has different effects:

- in all cases the pair (r, id) is removed from RR and the other message record variables are unchanged,
- in case of reads and page table lookups:
  - the returned value is equal to the memory contents of the requested bytes (8 bytes for page table lookups),
  - all memory contents are unchanged,
- in case of writes:
  - the accessed bytes are updated according to the provided value,

– all other memory contents are unchanged.

The enabling specification requires that any reply (q, id) that fulfills the necessary preconditions listed above may received by the memory. For memory reads and page table walks we demand additionally that the read value contains the corresponding memory contents.

Internal Memory Step Via MemIntrnl id the memory may perform an internal step for some request that was received from sender id. We require the following necessary precondition on the step:

- there exists a request r such that (r, id) is contained in RR, i.e., there is a request of sender id pending for which the internal action is performed,
- request r is not addressing the GIC or some peripheral, i.e., it is not an MMIO request.

The resulting abstract memory state does not change for internal steps, i.e., while an internal step may represent some internal behavior like cache coherency operations, the memory contents as seen on the abstraction must not change.

As an enabling specification we require that the memory is always able to perform an internal step if the necessary preconditions hold.

2.3.5 Peripheral Model

The peripheral model is specified in `axfuncsScript.sml`. We use the same model for ideal and refined model peripherals. A detailed description will be added shortly.

2.3.6 Ideal GIC Model

The ideal GIC model is specified in `axfuncsScript.sml` and `idealScript.sml`. A detailed description will be added shortly.

2.3.7 IGC Notification Messaging Model

The semantics of IGC notification buffers is defined explicitly in `idealScript.sml`. A detailed description will be added shortly.

The following material describes the earlier version of the model. We will update the content as soon as possible.

2.3.8 Guest Transition System

The transitions of the guest are performed either by one of the cores or one of the peripherals of a guest. We model the core behaviour by a transition relation

δ_I^g. We denote a transition from core state $C \in \mathcal{C}_{id}$, memory input channel $m2c$, core output channel cif, and interrupt input $q_{in} \in IRQ_g$ by

$$\delta_I^g(C, m2c, cif, q_{in}) \ni (C', m2c', cif', e)$$

Here $e \in \mathcal{E}_g^*$ is a sequence of peripheral events, which only contains send and acknowledgment events for IGC interrupts as well as power control commands to stop and start other cores in this case. The transition also transforms the message boxes between core and memory, e.g., by consuming an input (clearing the resulting channel state of the message) and procuding new output messages. Note that the actual core model is a non-deterministic composition of sub-automata that describe core (δ_{Icore}^g) and MMU (δ_{Immu}^g) transitions. We omit a detailed description here for brevity. The actual transition relation also needs to take into account the automaton states for core, MMU, and memory.

Generally, the behaviour of the guest machines is similar to the original ARM semantics, restricted to EL1 and EL0 and excluding the second stage of address translation. There are two exceptions, namely for hypervisor and secure monitor calls. In the case of the IGC signalling hypercall, execution will not switch to EL2, but execute the effect of the call atomically, incrementing the *sIGC* counter for the requested channel using the event output. Secure monitor calls are used to power up or down different cores, which is modeled by a start(c) or stop(c) event and an update of the particular core's *active* flag.

To model the effect of power control commands on the core configurations $C : \mathbb{N}_{nc_g} \to \mathcal{C}_{id}$ of a guest we define the helper function $psci(C, e)$ that reads an event sequence e for such commands and produces a new configuration C' as accordingly.

$$C'(c) = \begin{cases} C_{off} & : \quad \text{stop}(c) \in e \wedge C(c).active \\ C_{reset}^{g,c} & : \quad \text{start}(c) \in e \wedge \neg C(c).active \\ C(c) & : \quad \text{otherwise} \end{cases}$$

Basically power control events may be power up or power down events for a core (resulting from an SMC call). In this case the corresponding cores enter into special states $C_{reset}^{g,c}$ (where the *active* flag is true and the registers are in a reset configuration for core c of guest g), or C_{off} respectively (where the *active* flag is false). We require that cores may only be stopped when they have no outstanding memory requests. To control this we add a history variable $curr \in$ None to each ideal core which records all outgoing core requests and is reset to None when a corresponding reply is received.

When the event sequence contains the request to send an IGC notification interrupt for channel s via event snd(igc_s), the *sIGC* component of the guest is set to one for the requested channel s. If it was already one before, then the send request has no effect. Received IGC interrupts on a channel s are acknowledged through the GIC which resets the *rIGC* flag for channel s to zero.

Note that all event signals are produced by the guest cores only through hypercalls, therefore they represent the effect of hypervisor actions that are only visible in the refined model.

For peripheral steps we have transition relations $\delta_P^{g,p}$ for each peripheral p. We denote a transition from peripheral state $P.st \in \mathcal{P}_{g,p}$, memory input channel $m2p$, and peripheral output channel pif under peripheral event sequence E by

$$\delta_P^{g,p}(P.st, m2p, pif, E) \ni (P.st', m2p', pif', e, Q_{out})$$

where $Q_{out} \subseteq PIRQ_{g,p}$ represents which of its interrupts are active for the peripheral after the step. Output $e \in \mathcal{E}^*$ is appended to the event sequence of g. Note that the peripheral event sequence input is not filtered for the peripheral in question to simplify the presentation.

Since we define a message passing protocol between the cores, peripherals, and memory, we need a separate transition rule for the memory, where it consumes the messages it gets and returns replies for such requests. We introduce a transition relation δ_M that takes a memory state m, a memory input channel $m_{in} \in \mathbb{R}$ from cores or peripherals as well as a sender identifier $s \in \mathbb{S}_g$, and produces a new memory state and an output $m_{out} \in \mathbb{Q}$. Note that this memory transition relation does not capture effects due to the cache hierarchy and weakly consistent memory ordering. In a future version of the proof we need to introduce them by exchanging the flat memory state with something more advanced. Also it does not capture the effects of memory-mapped I/O accesses; these are redirected to the concerned peripheral. We express memory transitions using again a funtional notation.

$$\delta_M(m, m_{in}, s) \ni (m', m_{out})$$

At last, the steps of the interrupt controller are executed by the guest-specific transition relation δ_Q^g. It transforms a GIC configuration GIC, memory input channel $m2g$, GIC output channel $g2m$, GIC state Q, as well as the IGC interrupt receive buffers for guest g, ignoring all accesses to configure interrupts that are not belonging to g. Additionally it depends on the raised peripheral interrupts so that they can be forwarded to the core interfaces.

$$\delta_Q^g(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC')$$

Note that usually the interrupt controller does not modify memory outside of its memory-mapped I/O ports but we leave this detail implicit here. Furthermore, GIC transitions may represent internal GIC steps, reconfiguration of the GIC, as well as signalling of peripheral, IPI, and IGC interruptsto the CPU interfaces of the cores. Note that we assume that a GIC step either responds to a memory-mapped I/O requests or forwards a peripheral interrupt, where the latter can only happen while there are no pending requests in the GIC's input channel.

Moreover, interrupts are never generated directly by the GIC, except for IPIs between cores of the same guest upon an explicit request by a core of that guest. That means in particular that peripheral interrupts are only raised by the corresponding peripheral itself and that IGC notification interrupts can only be raised through hypercalls. These and other assumptions on the GIC are formalized in Section 4.9.3. Note that, by construction, the ideal GIC of guest g can never make interrupts pending that do not belong to it.

The guest transitions are now defined as follows. At any time a guest can perform either a core, a memory, a GIC, or a peripheral step. We capture this notion of non-determinism by transition relation $\longrightarrow_g \subseteq \mathbb{G}_g \times \mathbb{G}_g$. It is defined by the following transition rules. For readability we define such rules using the convention to omit any components that do not change in the transition.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc_g} \\ G.C(c).active \qquad Q_{pend} = G.Q(c).pend \qquad msk = msk_g(G.GIC) \\ q_{in} = \max\{q \in Q_{pend} \mid (q,c) \notin msk \vee \exists c' \in \mathbb{N}_{nc_g}.\ q = \mathrm{sgi}^{id}_{c',c} \notin msk\} \\ \delta^g_I(G.C(c), G.m2c(c), G.cif(c), q_{in}) \ni (C', m2c', cif', e) \\ \forall c'.\ \mathrm{stop}(c') \in e \Rightarrow G.C(c').curr = \mathrm{None} \qquad G'.C = psci(G.C[c \mapsto C'], e) \\ G'.m2c = G.m2c[c \mapsto m2c'] \qquad G'.cif = G.cif[c \mapsto cif'] \\ G'.sIGC = \lambda s \in IGCout_g.\ \mathrm{snd}(\mathrm{igc}_s) \in e\ ?\ 1 : G.sIGC(s) \end{array}}{G \longrightarrow_g G'}\ \textsc{Core}$$

In case of a guest transition we only step active guests and we allow to choose the pending and enabled peripheral, IGC, or IPI interrupt with the highest priority to be delivered as input to the core. The core may however ignore these inputs, i.e., they are only removed from the core input and guest state if they are explicitly acknowledged, or deactivated respectively, through a memory-mapped I/O message to the GIC. Similarly, the IGC signal components for channel s are updated if there are new interrupts requested by the core through a $\mathrm{snd}(\mathrm{igc}_s)$ event that reflects the effect of the hypercall implementation.

Note that there may be at most one IGC interrupt pending per channel, thus if a second one is requested by the guest, this interrupt is merged with the one that is already pending and only one interrupt will be delivered eventually. Note moreover that we assume a global fixed priority order on interrupts here, where IGC interrupts have the highest priority. In fact, each core can set up its own priorities for interrupts, but we omit this feature here for simplicity.

The power control commands issued by the core transition relation are interpreted by the *psci* function, however only if all cores that are to be stopped do not have outstanding memory requests. Note that this is merely a scheduling restriction, i.e., before the SMC instruction to power down a core may be executed the ideal model must first be stepped in such a way that any pending memory request for the targeted core is finished.

Memory channels are updated in the obvious way, while the memory, peripheral states, the peripheral event sequence, and GIC are unaffected by core transitions.

Peripheral transitions on the other hand only update their memory channels, the event sequence, the peripheral interrupts, and the state of the peripheral that was stepped. The definition is straight forward, using $\circ$ for the concatenation

of sequences.

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np_g} \\ \delta_P^{g,p}(G.P.st(p), G.m2p(p), G.pif(p), G.E) \ni (P.st', m2p', pif', e, Q_{out}) \\ G'.P.st = G.P.st[p \mapsto P.st'] \\ G'.pif = G.pif[p \mapsto pif'] \qquad G'.m2p = G.m2p[p \mapsto m2p'] \\ G'.PI = G.PI[p \mapsto Q_{out}] \qquad G'.E = G.E \circ e \end{array}}{G \longrightarrow_g G'} \text{ PERIPHERAL}$$

Similarly GIC steps only modify the components related to GIC, its memory channels, and the active received IGC interrupts. In particular, the following rule allows the GIC to modify the sets of pending and active interrupts based on the raised peripheral interrupts and the messages it receives. We require that the output channel is empty of any previous GIC replies before the GIC can be stepped again. When there is no input present GIC transitions may signal periphral interrupts to the cores' CPU interfaces. In that case we require that any core whose pending physical interrupts change, does not have any out-standing memory replies. This "no-outstanding-request-interrupt-signalling" constraint is captured by the following predicate

$$\begin{array}{rcl} noris_g(G, c, Q, Q') & \equiv & G.C(c).curr \neq \text{None} \Rightarrow \\ & & (Q'(c).pend \cup Q'(c).ap) \cap (PIRQ_g \cup IPIRQ_g) \\ & & = \\ & & (Q(c).pend \cup Q(c).ap) \cap (PIRQ_g \cup IPIRQ_g) \end{array}$$

It is necessary, as in the refined model interrupt delivery requires the hypervisor to inject them as virtual interrupts, however the hypervisor may only be invoked while a core is not currently accessing memory.

$$\frac{\begin{array}{c} m2g = G.m2p(np_g) \qquad G.pif(np_g) = \text{None} \\ \delta_Q^g(G.GIC, G.Q, m2g, G.PI, G.rIGC) \ni (GIC', Q', g2m', rIGC') \\ \forall c \in \mathbb{N}_{nc_g}.\ noris_g(G, c, Q, Q') \\ G'.pif = G.pif[np_g \mapsto \text{None}] \qquad G'.m2p = G.m2p[np_g \mapsto m2g'] \\ G'.Q = Q' \qquad G'.rIGC = rIGC' \qquad G'.GIC = GIC' \end{array}}{G \longrightarrow_g G'} \text{ GIC}$$

Note that the constraint on targeted cores not having any outstanding memory requests does not restrict possible behaviours of the ideal model, as we will restrict cores later to only receive interrupts in between outstanding memory requests anyway. Thus, this condition is only necessary to restrict certain schedules of the ideal model that cannot be easily simulated by the refined model.

Memory transitions are defined by several rules, reflecting the handling of access requests to actual memory as well as the forwarding of memory-mapped I/O. We allow the memory to non-deterministically perform steps for an arbitrary core or peripheral as long as the sender has no outstanding reply from the memory in its message box. This simplifies the model as it sequentializes requests from a single core or peripherals to memory and rules out more advanced

implementations of such components. A more realistic model, where several requests per core or peripheral may be issued at the same time is subject to future research.

We first define the regular memory functionality for core and peripheral access, excluding memory-mapped I/O accesses. These cases are identified by the following predicate for core in guest configuration G of guest g, sending a request $r \in \mathbb{R}$ to peripheral $p \in \mathbb{N}_{np_g+1}$.

$$mmio(r, g, p) \equiv r \neq \text{None} \wedge adr(r)[47:12] \in A_P^{g,p}$$

Here function $adr : \mathbb{R} \cup \mathbb{Q} \to \mathbb{B}^{48}$ extracts the intermediate physical address from requess and replies.

Replies $q \in \mathbb{R} \cup (\mathbb{Q} \times \mathbb{S}_g)$ from peripherals to such requests are detected by a similar predicate.

$$mmio(q, g, c, p) \equiv \exists q' \neq \text{None}.\ q = (q', \text{C } c) \wedge adr(q')[47:12] \in A_P^{g,p}$$

The memory steps for regular accesses are then defined as follows.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc_g} \qquad \forall p \in \mathbb{N}_{np_g}.\ \neg mmio(G.cif(c), g, p) \qquad adr(G.cif(c)) \in \mathbb{A}_g \\ G.m2c(c) = \text{None} \qquad \delta_M(G.m, G.cif(c), \text{C } c) \ni (m', m2c') \\ G'.m = m' \qquad G'.cif = G.cif[c \mapsto \text{None}] \qquad G'.m2c = G.m2c[c \mapsto m2c'] \end{array}}{G \longrightarrow_g G'} \text{ MEM-CORE}$$

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np_g+1} \qquad \forall c \in \mathbb{N}_{nc_g}.\ \neg mmio(G.pif(p), g, c, p) \qquad adr(G.cif(c)) \in \mathbb{A}_g \\ G.m2p(p) = \text{None} \qquad \delta_M(G.m, G.pif(p), \text{P } p) \ni (m', m2p') \qquad G'.m = m' \\ G'.pif = G.pif[p \mapsto \text{None}] \qquad G'.m2p = G.m2p[p \mapsto m2p'] \end{array}}{G \longrightarrow_g G'} \text{ MEM-PERIPH}$$

Note that we require regular memory addresses to stay within the allocated address region for the current guest, e.g., for a guest with 4GiB allocated virtual memory ($as_g = 20$) the 12 most significant bits of the intermediate physical address in any request sent within this guest must be zero. Otherwise a memory fault is returned as defined below.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc_g} \qquad adr(G.cif(c)) \in \mathbb{A}_g \qquad G.m2c(c) = \text{None} \\ G'.cif = G.cif[c \mapsto \text{None}] \qquad G'.m2c = G.m2c[c \mapsto \text{F } adr(G.cif(c))] \end{array}}{G \longrightarrow_g G'} \text{ MEM-CFAULT}$$

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np_g+1} \qquad adr(G.pif(p)) \in \mathbb{A}_g \qquad G.m2p(p) = \text{None} \\ G'.pif = G.pif[p \mapsto \text{None}] \qquad G'.m2p = G.m2p[p \mapsto \text{F } adr(G.pif(p))] \end{array}}{G \longrightarrow_g G'} \text{ MEM-PFAULT}$$

As mentioned above, accesses to memory-mapped I/O regions are relegated from the memory input message box to the corresponding peripherals input message box and vice versa for replies from peripherals to such requests. Note that we exclude DMA accesses from one peripheral to another one here for simplicity and since this scenario seems unrealistic. Adding this possibility for

peripherals within the same guest is easy but would require additional transition rules. Instead we require the hypervisor to forbid these kind of DMA accesses through the system MMUs (SMMUs).

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc_g} \qquad p \in \mathbb{N}_{np_g+1} \qquad mmio(G.cif(c), g, p) \qquad G.m2p(p) = \text{None} \\ G'.cif = G.cif[c \mapsto \text{None}] \qquad G'.pif = G.pif[p \mapsto (G.cif(c), \text{C } c)] \end{array}}{G \longrightarrow_g G'} \textsc{Mem-I/Oreq}$$

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np_g+1} \qquad p \in \mathbb{N}_{nc} \qquad mmio(G.pif(p), g, c, p) \qquad G.m2c(c) = \text{None} \\ G'.m2c = G.m2c[c \mapsto G.pif(p)_1] \qquad G'.m2p = G.m2p[p \mapsto \text{None}] \end{array}}{G \longrightarrow_g G'} \textsc{Mem-I/Orpl}$$

Observe that DMA requests from the peripherals are only handled by memory if the peripheral has no outstanding memory-mapped I/O requests in its input channel. In order to avoid deadlocks in such situations we allow that peripherals may always be stepped and assume that they store replies to memory-mapped I/O requests internally until any outstanding DMA request has been consumed by memory from its input channel.

In addition to the rules above we allow communication with the external world at any time by adding arbitrary external events to the event sequence.

$$\frac{e \in \mathcal{E}_g \qquad G'.E = G.E \circ e}{G \longrightarrow_g G'} \textsc{External}$$

This completes the description of the guest model.

2.4 Platform Model

The overall ideal model of the HASPOC virtualization platform comprises a number of guest states and the data contained in the IGC channels. Formally, we introduce record $\mathcal{M}_I$ with the following two components:

- $G : \mathbb{N}_{ng} \to \mathbb{G}$ – the guest states of the system. Here $\mathbb{G} = \bigcup_{g \in \mathbb{N}_{ng}} \mathbb{G}_g$ and we demand that $G(g) \in \mathbb{G}_g$.
- $S : \mathbb{N}_{ns} \to \mathbb{B}^{4096 \cdot 8}$ – the data page stored for each IGC channel
- $tgt : \mathbb{N}_{ns} \to \mathbb{N} \cup \{\bot\}$ – the target core for any outstanding IGC interrupts per channel. Interrupts for channel s may target any core of the receiving guest (if it is powered up), i.e., if $cpol(s) = (g, g')$ then $tgt(s) \in \mathbb{N}_{nc_{g'}} \cup \{\bot\}$. The $\bot$ symbol means that no core of the receiving guest is ready (i.e., powered up) to receive the interrupt.

The transition relation of the overall system is denoted by $\xrightarrow{I} \subseteq \mathcal{M}_I \times \mathcal{M}_I$. Naturally it is always allowed to step any guest system. Before the actual step of a specific guest, the content of its ingoing IGC channels is copied into its local memory. After the step we update the content of its outgoing channels. This is done to correctly model the shared memory channels that are used for

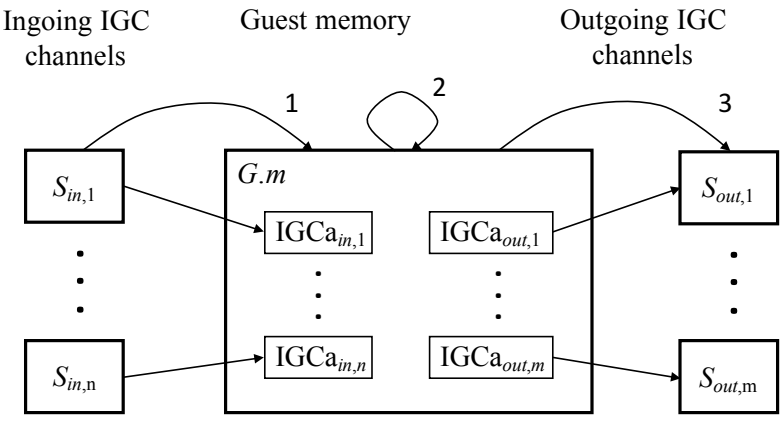

Figure 2: IGC memory sharing during a GUEST transition. In the first step ingoing channels are copied into guest memory to their dedicatd locations. In step 2 the guest performs a step, potentially updating an outgoing channel's memory area. After that step the content of these areas is copied into the corresponding IGC buffers.

IGC in the refined model. Figure 2 illustrates the mechanism. Since guests may use the IGC mechanism in arbitrary ways to establish a communication protocol, we must model the message buffers in the ideal model in a way that captures all possible interleavings of sender writes and receiver reads to those buffers. Therefore, for each guest step, the out-going buffers and receiving guests' memories are updated.

Moreover, whenever an IGC interrupt is raised for an outgoing channel, the targeted core is updated in accordance with the hypervisor implementation of the IGC mechanism. In particular we choose among the powered up cores of the receiving guest the one with the lowest index. The target computation for a given receiving guest G with index g is formalized as follows.

$$trgt_g(G) = \begin{cases} \bot & : \quad \forall c \in \mathbb{N}_{nc_g}.\ \neg G.C(c).active \\ \min\{c \in \mathbb{N}_{nc_g} \mid G.C(c).active\} & : \quad \text{otherwise} \end{cases}$$

The target is only updated for channels where an IGC interrupt to guest g' was raised on channel s in the last step from guest configurations G to G', as denoted by the following predicate.

$$IGCup_s^{g,g'}(G, G') \equiv cpol(s) = (g, g') \wedge G'.sIGC(s) > G.sIGC(s)$$

All such raised interrupts for which there is no powered up core to receive them are disable again immediately. Now the guest transition is formally defined as

follows.

$$\frac{\begin{array}{c} g \in \mathbb{N}_{ng} \qquad G = M.G(g)[m \mapsto M.G(g).m[IGCa(s)_2 \mapsto M.S(s) \mid s \in IGCin_g]] \\ G \longrightarrow_g G' \\ G'' = G'[sIGC \mapsto G'.sIGC[s \mapsto 0 \mid \exists g'.\ IGCup_s^{g,g'}(G, G') \wedge trgt_{g'}(M.G(g')) = \bot]] \\ M'.G = M.G[g \mapsto G''] \qquad M'.S = M.S[s \mapsto G''.m(IGCa(s)_1) \mid s \in IGCout_g] \\ M'.tgt = tgt[s \mapsto trgt_{g'}(M.G(g')) \mid IGCup_s^{g,g'}(G, G')] \end{array}}{M \xrightarrow{I} M'} \textsc{Guest}$$

In addition, we need another transition to deliver the IGC interrupts from one guest to another. This can happen non-deterministically for any channel where there is a pending interrupt and we deliver the interrupt to the targeted core of that channel, but only in case it has no currently outstanding memory requests. The latter constraint is similar to the "no-outstanding-request-interrupt-signalling" condition for the delivery of peripheral interrupts to the core interfaces. It is again necessary as in the refined model the corresponding inter-processor interrupt can only be received in between memory requests. Then we define the IGC notification delivery in the ideal model as follows.

$$\frac{\begin{array}{c} cpol(s) = (g, g') \qquad M.G(g).sIGC(s) = 1 \qquad M.tgt(s) = c \in \mathbb{N}_{nc_{g'}} \\ M.G(g').C(c).curr = \text{None} \qquad M.G(g').rIGC(s, c) = 0 \\ M'.G(g) = M.G(g)[sIGC \mapsto M.G(g).sIGC[s \mapsto 0]] \\ M'.G(g') = M.G(g')[rIGC \mapsto M.G(g').rIGC[(s, c) \mapsto 1]] \\ \forall g''.\ g \neq g'' \neq g' \Rightarrow M'.G(g'') = M.G(g'') \end{array}}{M \xrightarrow{I} M'} \text{IGC-IRQ}$$

Note that only the sending and receiving guests are affected. Observe moreover, that we do not allow to send the inter-guest interrupt, if a previous IGC interrupt for the channel is still pending at the same core. The receiver's GIC first needs to deliver that interrupt to the targeted core's CPU interface before another IGC interrupt may be triggered.

This completes the ideal model of the HASPOC platform. It is correct by construction as one can easily see for the following reasons:

- Guests are modeled as separate components of the model having their own cores, memories, peripherals, and interrupt components.
- For a given guest step (cf. rule Guest), only one guest state is affected along with all the IGC channels leading out of that guest. Moreover the step only depends on the state of the guest that is being stepped along with the IGC channels leading into that guest. In other words, a guest cannot directly read or modify other guests and their only means of communication is through the permitted IGC channels.
- Peripheral steps are bound to one guest and isolated from others, there is no direct information flow from one peripheral into other guests or vice versa. The same holds for peripheral interrupts which are tied to the guest owning the peripheral.

- IGC interrupt delivery is governed by the HASPOC communication policy (cf. rule IGC-IRQ), i.e., a guest cannot send IGC notification interrupts to other guests when it is not allowed to. Moreover all message channels are following that policy, i.e., there are no channels between guests that are not allowed to communicate.

In the next step we need to provide the refined model of the system where hypervisor steps, the second stage MMU, the SMMU, and the shared interrupt controller are visible.

3 Refined Model

The refined model H of the refined model is similar to an ideal model having only one guest state. However all of the cores of the underlying processor are present and they are extended with EL2 and EL3 functionality as well as a second stage of address translation for EL1 and EL0. Additionally, peripheral memory accesses are guarded by an SMMU and there is a global interrupt controller distributing the IGC, IPI, and peripheral interrupts among all cores. For each guest of each core the hypervisor needs to inject virtual interrupts in order to emulate the private interrupt controllers for each guest in the ideal model. In the refined model we see the execution of the boot and hypervisor code explicitly as well as their internal data structures.

Again we need to introduce a number of parameters that constrain the refined model implementation:

- $nc \in \mathbb{N}$ – the number of cores on the actual hardware platform. For simplicity we assume here that $nc = \sum_{g=0}^{ng-1} nc_g$, i.e., all cores are used up by the guests.
- $np \in \mathbb{N}$ – the number of peripherals on the actual hardware platform, excluding the GIC, memory controller, and SMMUs. Again we assume $np = \sum_{g=0}^{ng-1} np_g$ for simplicity.
- $PIRQ_p$ – the identifiers of all the peripheral interrupts peripheral $p \in \mathbb{N}_{np}$ can trigger. We also define $PIRQ = \bigcup_{p \in \mathbb{N}_{np}} PIRQ_p$.
- $A_{ROM} \subset \mathbb{B}^{36}$ – the memory region for the ROM, represented as physical page addresses
- $a_{rkh} \in \mathbb{B}^{43}$ – aligned physical address of the 256 bit root key hash. The root key hash is allocated in read-only memory, i.e., $a_{rkh}[42:5] \in A_{ROM}$.
- $A_F \subset \mathbb{B}^{36}$ – the memory region of flash memory, represented as physical page addresses
- $A_{MC} \subset \mathbb{B}^{36}$ – the memory region for the memory controller setting up SDRAM
- $A_{SMMU} \subset \mathbb{B}^{36}$ – the memory region for the I/O ports of the system MMUs
- $A_{GIC} \subset \mathbb{B}^{36}$ – the memory region for the I/O ports of the GIC. We distinguish four disjoint regions of the GIC interface:
  - $A_{GICC} \subset A_{GIC}$ – the GIC CPU interface,
  - $A_{GICD} \subset A_{GIC}$ – the GIC distributor,
  - $A_{GICH} \subset A_{GIC}$ – the GIC virtual interface control interface, and
  - $A_{GICV} \subset A_{GIC}$ – the GIC virtual CPU interface.

Only the last region is directly accessible to guest accesses by mapping their GIC CPU interfaces to this region using the second stage MMU. The CPU interface and the virtual interface control registers are accessed by the hypervisor exclusively. Guest accesses to the distributer interface are intercepted by the hypervisor and emulated. Therefore the hypervisor, using GIC virtualization support, virtualizes interrupt control so that each guest seems to own a private GIC distributor and CPU interfaces. We denote the corresponding disjoint intermediate physical regions visible to guest g by $A^g_{GICC}, A^g_{GICD} \subset A_P^{g,np_g}$.

- $A_{boot} \subset \mathbb{B}^{36}$ – the memory region for the boot code and data copied from flash, where also the SMC handlers for power control reside
- $A_{HV} \subset \mathbb{B}^{36}$ – the memory region reserved for the hypervisor
- $A_{PT} : \mathbb{N}_{ng} \to 2^{\mathbb{B}^{36}}$ – the hypervisor memory reserved for the page tables of each guest. Here we assume that those page tables are allocated statically by the hypervisor. This is possible because the number of guests is bounded by the maximum number of the cores for a given architecture (32 in ARMv8). Moreover we require that page tables are not shared between different guests within the hypervisor, i.e., for all $g \neq g'$ we have $A_{PT}(g) \cap A_{PT}(g') = \emptyset$ and $A_{PT}(g) \subset A_{HV}$.
- $A_{PPT} : \mathbb{N}_{np} \to 2^{\mathbb{B}^{36}}$ – the hypervisor memory reserved for the SMMU page tables that govern peripheral DMA accesses. Again we require that these page tables are disjoint from those of the guests g, i.e., $A_{PT}(g) \cap A_{PPT}(p) = \emptyset$ and $A_{PPT}(p) \subset A_{HV}$ for all p.
- $A_{IGC} : \mathbb{N}_{ng} \times \mathbb{N}_{ng} \to 2^{\mathbb{B}^{36}}$ – the (potentially empty) set of physical memory pages shared by two guests for IGC purposes.
- $A_G : \mathbb{N}_{ng} \to 2^{\mathbb{B}^{36}}$ – the memory regions reserved for each guest. We demand for all guests g, g' with $g \neq g'$ that

$$A_G(g) \cap A_G(g') = A_{IGC}(g, g') \cup A_{GICV}$$

and $A_G(g) \cap (A_{HV} \cup (A_{GIC} \setminus A_{GICV}) \cup A_{SMMU} \cup A_F \cup A_{ROM} \cup A_{MC} \cup A_{boot}) = \emptyset$, i.e., different guests own disjoint portions of physical memory (modulo IGC channels and the virtual GIC CPU interface) that do not overlap with hypervisor memory or other system resources. We assume that the allocation of guest memory through the hypervisor is static and can be derived from the hypervisor configuration data. Note that the I/O ports for the GIC virtual CPU interfaces are the same for each core (similar to the GIC CPU interfaces), but the GIC keeps separate interface states internally. Figure 3 shows how different regions of guest memory are mapped into physical memory.

- $A_P : \mathbb{N}_{np+1} \to 2^{\mathbb{B}^{36}}$ – the memory regions reserved for each peripheral and their memory-mapped I/O ports which need to be mutually disjoint and also disjoint from all other memory regions. Note that $A_P(np) = A_{GIC}$.

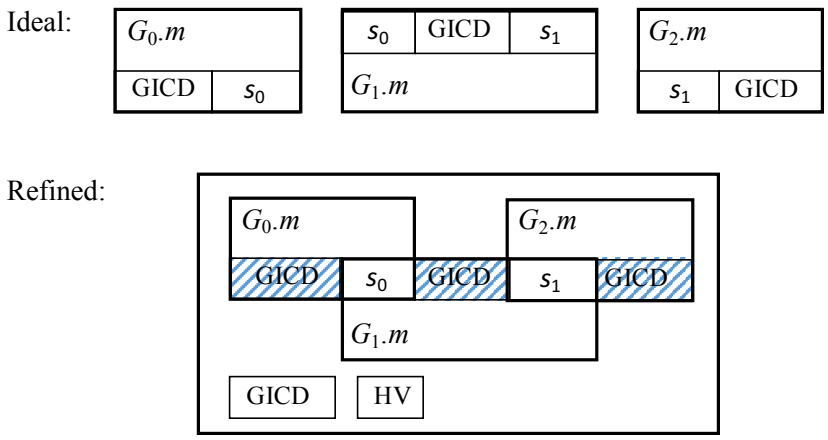

Figure 3: Mapping disjoint ideal guest memories into physical memory. Note that the guest memory areas for the GIC distributor are unmapped but that the IGC memory channels are shared between different guests. Moreover, hypervisor memory is not accessible to guests.

In addition to these parameters there are the images that are loaded during the boot process. Such images have abstract type *IMG* which is defined as a record with the following components:

- $adr \in \mathbb{B}^{36}$ – the page address where the image is stored
- $isz \in \mathbb{N}$ – the size of the image in number of pages
- $dadr \in \mathbb{B}^{36}$ – the page address in physical memory where the contained data should be copied
- $dsz \in \mathbb{N}$ – the size of the data to be copied in number of pages
- $data \in \mathbb{B}^{4096 \cdot 8 \cdot d}$ – the contained data, where we demand $d = dsz$
- $sig \in SIG(sk)$ – a cryptographic signature, i.e., a hash of the image data that was signed by some secret key $sk \in \mathbb{B}^{\alpha}$ with strength α. The integrity of image img can be checked using a public key k by predicate $check(k, img)$ which performs the necessary cryptographic operations. It holds true iff k is the public key matching secret key sk used for signing and the image data matches the signed hash.
- $entry \in \mathbb{B}^{48} \cup \{\bot\}$ – the entry address for an executable image. If the image is not executable this field is undefined ($\bot$).
- $EL \in \mathbb{N}_4 \cup \{\bot\}$ – the exception level at which the binary should be executed. We excluded an encoding for Secure EL1 and EL0 here since they are not used in the HASPOC platform. If the image is not executable this field is undefined ($\bot$).

For images $img \in IMG$ the address range occupied by it in memory is given by

$$Ra_i(img) = \{a \in \mathbb{B}^{36} \mid \langle img.adr \rangle \leq \langle a \rangle < \langle img.adr \rangle + img.isz\}\,.$$

Here we use $\langle \cdot \rangle \in \mathbb{N}$ to determine the binary value of a bit string. Similarly, the address range where the data of an image will be copied is denoted by

$$Ra_d(img) = \{a \in \mathbb{B}^{36} \mid \langle img.dadr \rangle \leq \langle a \rangle < \langle img.dadr \rangle + img.dsz\}\,.$$

The binary representation of an image img is obtained by $\langle img \rangle \in \mathbb{B}^{4096 \cdot 8 \cdot img.isz}$.

Now, we demand that all images (including the ones introduced below) are allocated in flash memory, i.e., $Ra_i(img) \subseteq A_F$, and that different images should not overlap, i.e., $Ra_i(img) \cap Ra_i(img') = \emptyset$ for $img \neq img'$. For executable images we require that their entry point is within the contained data, i.e., $img.entry \neq \perp \Rightarrow img.entry[47:12] \in Ra_d(img)$.

In the HASPOC platform we have to deal with the following images that we also treat as parameters to the system:

- $img_{rk} \in IMG$ – the image containing the public root key. We assume here that all images are signed for this key, i.e., $check(img_{rk}.data, img)$ should hold true for all of the above images. The root key itself is verified against the root key hash stored in read only memory. Additionally we require that $Ra_d(img_{rk}) \subseteq A_{boot}$ as well as $img_{rk}.entry = \perp$ and $img_{rk}.EL = \perp$, since the root key needs to be copied into the boot memory area and it is not an executable image.

- $img_{HV} \in IMG$ – an abstract representation of the image of the hypervisor code. We require that it should be copied into hypervisor memory, i.e., $Ra_d(img_{HV}) \subseteq A_{HV}$. The hypervisor image is executable and should be started in EL2 at the initial hypervisor entrypoint a_{hve0}, i.e., $img_{HV}.entry = a_{hve0}$ and $img_{HV}.EL = 2$.

- $img_{HVc} \in IMG$ – the image containing the configuration file for the hypervisor (reflecting many of the ideal model parameters). Similar to the hypervisor image we require $Ra_d(img_{HVc}) \subseteq A_{HV}$.

- $img^g_G \in IMG$ – the image containing the binary for guest $g \in \mathbb{N}_{ng}$. We assume that each guest has its own binary stored in flash which does not overlap, so we do not share binaries even if they are identical. Additionally each guest binary needs to be copied into the corresponding guest memory, i.e., $Ra_d(img^g_G) \subseteq A_G(g)$, and we mark these images as non-executable to the boot loader by $img^g_G.entry = \perp$ and $img^g_G.EL = \perp$. The guests are booted by the hypervisor which gets the entry point for the guests from its configuration file and always runs them in Non-Secure EL1 mode.

- $img^g_{Gc} \in IMG$ – the image with the configuration file for guest $g \in \mathbb{N}_{ng}$ (usually in form of a device tree). Again we demand $Ra_d(img^g_{Gc}) \subseteq A_G(g)$ as well as $img^g_{Gc}.entry = \perp$ and $img^g_{Gc}.EL = \perp$.

- $img_{Mc} \in IMG$ – the image with the configuration binary for the memory controller. We require $Ra_d(img^g_{Gc}) \subseteq A_{MC}$. Furthermore the entrypoint to the binary should run from EL3, i.e., $img^g_{Gc}.EL = 3$.

- $img_{boot} \in IMG$ – the image containing the ARM Trusted Firmware BL31 boot image that contains the SMC handlers for power management and the code to finally start the hypervisor. This code needs to be copied into boot memory, i.e., $Ra_d(img_{boot}) \subseteq A_{boot}$ and it should be executed in EL3 ($img_{boot}.EL = 3$).

Given these parameters we need to specify the refined model H. This is done by distinguishing three kinds of steps: guest steps, boot and hypervisor initialization, and hypervisor handlers. First, however, we need to define the overall state of the model.

3.1 System State

The refined model consists of three main components: the refined core states, the peripheral states, and the memory. In addition we add components to model the interrupt controller, the second stage MMU, and the SMMU. System States have type $\mathcal{M}_R$ that is a record with the following components:

- $C : nc \to \mathcal{C}$ – the actual cores of the system, containing all relevant registers and all exception levels. Their type is defined similarly to the ideal cores only that the mode has type $\mathbb{N}_4$ and *gpr* as well as *spr* are addressed by elements from the sets *GPR* and *SPR*. Those sets contain all the identifiers from the corresponding sets in the ideal model but are extended with all the registers and entries that are only accessible in EL3 and EL2. In addition we add a history variable $init \in \mathbb{B}$ to the core that turns true as soon as the core enters EL1 or EL0 the first time.

- $m : \mathbb{B}^{36} \to \mathbb{B}^{4096 \cdot 8}$ – the physical memory of the system denoted as a mapping of pages. Note that the maximum physical address size in ARM is 48 bits, i.e., it contains up 2^{36} pages of size 4KiB.

- $P.st : \mathbb{N}_{np} \to \mathcal{P}$ – the states of the peripherals in the system. We require in particular that $P.st(p) \in \mathcal{P}_p$, i.e., each peripheral has their own set of states.

- $GIC \in \mathcal{G} = \{gicc : GICC \times \mathbb{N}_{nc} \to \mathbb{B}^*; gicd : GICD \to \mathbb{B}^*; gich : GICH \times \mathbb{N}_{nc} \to \mathbb{B}^*; gicv : GICC \times \mathbb{N}_{nc} \to \mathbb{B}^*\}$ – the register state of the single generic interrupt controller (GIC) of the system, represented like in the ideal model but adding registers for the virtual interrupt control interface and each core's virtual interrupt interface. Let IPIs be denoted by the set $IPIRQ = \{\mathrm{sgi}^{id}_{i,j} \mid i, j \in \mathbb{N}_{nc}\}$ and $IGCQ = \{\mathrm{igc}_s \mid s \in \mathbb{N}_{ns}\}$. The set *VIRQ* denotes the virtual interrupts that can be received by each core separately. We define

$$VIRQ = \{virq_q \mid q \in PIRQ \cup IPIRQ \cup IGCQ\},$$

i.e., we allow all peripheral interrupts and IPIs as well as the IGC notifications to be injected into guests by the hypervisor as virtual interrupts. The, as in the ideal model, a mask function abstracts from the interface register state to yield the masked interrupts of each core.

$$msk, dmsk : \mathcal{G} \to IPIRQ \cup ((PIRQ \cup VIRQ) \times \mathbb{N}_{nc})$$

Additionally we have a *distributor mask* function $dmsk$, that reflects which interrupts are disabled for distribution to a specific CPU interface.

- $PI : \mathbb{N}_{np} \to 2^{PIRQ}$ – the set of peripheral interrupts being active within the system. Here we demand that $PI(p) \subseteq PIRQ_p$.
- $Q : \mathbb{N}_{nc} \to \{pend \subseteq IRQ; act \subseteq IRQ; ap \subseteq IRQ\}$ – currently pending, active, or active-and-pending physical interrupts for each core, where $IRQ = PIRQ \cup IPIRQ$, stored as a record of sets similar as in the ideal model, but for the complete system. Physical interrupts are the ones sent by peripherals or processors, as opposed to virtual ones, that are generated by the hypervisor implementation for the guest cores.
- $VI : \mathbb{N}_{nc} \to \{pend \subseteq VIRQ; act \subseteq VIRQ; ap \subseteq VIRQ\}$ – the set of virtual interrupts being pending, active, or active-and-pending for each core, where we restrict $VI(c)$ to contain IPIs only if they are addressed to core c. This component represents the virtual interrupts that are configured by the hypervisor to implement the GIC interrupts and inject them into the guests. Guest cores may acknowledge and configure virtual interrupts through a virtual GIC interface that is provided by the interrupt controller. Therefore the hypervisor only needs to intercept accesses to the GIC distributer registers which are not virtualized by hardware.
- $mmu : \mathbb{N}_{nc} \to MMU$ – the states of the second stage MMU per core. MMU states here are simplified as records with the following three components:
  - $active \in \mathbb{B}$ – a flag denoting that the second stage is inactive. Note that even if the second stage is turned off it still performs transitions, adding memory type information to requests, i.e., also the hypervisor and boot loader are communicating with memory through the (turned off) second stage MMU. During guest steps however, the second stage MMU should always be active.
  - $pto \in \mathbb{B}^{36}$ – the current page table base address (page table origin) for the second stage of translation. This should always point to a page table stored for a guest in the hypervisor.
  - $cfg \in \mathbb{C}$ – the other configuration information for the second stage of translation, e.g., translation granules, input and output address sizes. We abstract from these details here and let the configuration type $\mathbb{C}$ undefined.

– $walk \in \mathbb{W}$ – the internal state of the MMU keeping track of the progress of the translation table walk for the current request. We leave its type $\mathbb{W}$ uninterpreted here for simplicity.

Note that the MMUs in our model do not contain a TLB. We omit TLBs to simplify the proof since they add several complexities. See a detailed discussion below.

- $smmu : \mathbb{N}_{np} \to MMU$ – the states of the SMMU for each peripheral. Note that we assume that SMMUs work exactly as second stage MMUs for simplicity. Also we assume here that each peripheral has a dedicated SMMU, which may be an oversimplification for some hardware plaforms. Usually several devices share an SMMU and some are not even guarded by one so we assume an idealized scenario here.

- $c2u, u2m : \mathbb{N}_{nc} \to \mathbb{R}$ – the message channels from the cores to their second stage MMUs and from there to memory. Note that we use the same channel formats as in the ideal model. However, the channels from MMUs to memory contain requests with physical address now.

- $m2u, u2c : \mathbb{N}_{nc} \to \mathbb{Q}$ – the messages channel for replies from memory to the the second stage MMUs and from there to the cores. Again physical addresses are used in the channels from memory to the MMUs.

- $p2v : \mathbb{N}_{np+1} \to \mathbb{R} \cup (\mathbb{Q} \times \mathbb{S})$ – the message channels from the peripherals to their SMMU. As in the ideal model the peripheral output channels can contain both DMA requests to memory and replies to memory-mapped I/O accesses. The sender type $\mathbb{S}$ here is similar to $\mathbb{S}_g$ just with senders from domain $\mathbb{N}_{nc}$ instead of $\mathbb{N}_{nc_g}$ and $\mathbb{N}_{np}$ instead of $\mathbb{N}_{np_g}$. There is an additional channel here for replies from the GIC to memory-mapped I/O requests, however the GIC does not have an own SMMU.

- $v2p : \mathbb{N}_{np+1} \to (\mathbb{R} \times \mathbb{S}) \cup \mathbb{Q}$ – the message channels from the SMMUs to the corresponding peripheral. Note again that the input channel to peripherals may contain memory-mapped I/O requests just as in the ideal model. These messages are not sent from the SMMU but forwarded from the memory into the peripheral's input message box. Again we reserve an extra channel for requests to the GIC.

- $v2m : \mathbb{N}_{np} \to \mathbb{R}$ – the message channels from the second stage MMUs to memory. These channels contain only translated DMA access requests from the peripherals to memory and table walk requests of the SMMU. Replies to memory-mapped I/O requests are not handled by the SMMU but directly forwarded to the core which sent the request.

- $m2v : \mathbb{N}_{np} \to \mathbb{Q}$ – the message channels from memory to the second stage MMU of each peripheral. Again, note that we do not pass memory-mapped I/O requests to the SMMU, however table walk results and DMA replies are handled by it.

- $E : \mathcal{E}^*$ – a sequence of external events that implements communication between peripherals and the external world. Here $\mathcal{E}$ is the union of $\mathcal{E}_g$ for all guests g.

As mentioned above we do not model TLBs in this work. One reason is that the architecture does not provide a detailed specification of their structure and ARMv8 does not seem to enforce a strict separation for caching first and second stage translations. Moreover, it is infeasible to model the evictions from the TLBs deterministically in absence of a precise model. Therefore, we would have to add a lot of non-deterministic TLB behaviour to both the ideal and the refined model which would in turn complicate the bisimulation theorem and proof. Instead of a pure refinement (where steps of the abstract model are represented by one or more steps of the refined model) one would also need to introduce additional steps on the ideal model when proving the simulation of handler code. Therefore we leave the bisimulation of TLB behaviour as an open research question. This means in turn (similar to caches) that there may be some information flow in the model due to the TLBs.

In any case, the idea for the refined model is now to define a transition system on these components as follows:(1) Guest steps consisting of core transitions in EL1 and EL0, (2) System steps, i.e., MMU, peripheral, SMMU, memory, and external transitions which depend on and modify different parts of the system state related to them, (3) Hypervisor or boot code is executed whenever a core reaches EL2 or EL3. These steps are modeled as functions that transform a core state, hypervisor memory as well as the MMU and SMMU configurations. A detailed decription of these types of transitions is given below as rules for the overall refined transition relation $\xrightarrow{R} \subseteq \mathcal{M}_R \times \mathcal{M}_R$.

3.2 Guest Steps

Guest steps are modeled using the complete ARMv8 hardware model, where all instructions work as described in the instruction architecture manual and all registers and modes are present. The core and MMU part of this model are represented by the transition relation δ_R which takes a core state, second stage MMU input and output channels, as well as an interrupt input signal and produces new states for the core and the channels. Using functional notation we denote a transition by

$$\delta_R(C, u2c, c2u, q_{in}) \ni (C', u2c', c2u') \ .$$

Note, that in contrast to the idealized guest core transition relation, this one does not produce any power control or IGC notification events. This is due to the fact that these features need to be implemented with the help of the hypervisor that is invisible in the ideal model. Instead the involvement of the hypervisor is made explicit in the refined model. For instance, for sending an IGC, the hypervisor is invoked directly and it communicates with the GIC in order to activate the corresponding IPI to the appropriate core of the receiving guest.

Similarly, powering on and off other coresis implemented by the hypervisor using the run-time services that are implemented in EL3.

All this functionality is defined separately in the hypervisor transitions. We define the following transition rule to model guest steps using the core transition relation introduced above.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc} \qquad M.C(c).active \qquad M.C(c).ps.mode \leq 1 \\ Q_{pend} = M.Q(c).pend \cup M.VI(c).pend \qquad msk = msk(M.GIC) \\ q_{in} = \max\{q \in Q_{pend} \mid (q,c) \notin msk \vee \exists c' \in \mathbb{N}_{nc}.\ q = \mathrm{sgi}^{id}_{c',c} \notin msk\} \\ \delta_R(M.C(c), M.u2c(c), M.c2u(c), q_{in}) \ni (C', u2c', c2u') \\ M'.C(c) = M.C[c \mapsto C'] \\ M'.u2c = M.u2c[c \mapsto u2c'] \qquad M'.c2u = M.c2u[c \mapsto c2u'] \end{array}}{M \xrightarrow{R} M'} \textsc{Guest}$$

The update of the components is straight-forward. Observe that this rule is only activated for cores that are powered up and running in EL1 or EL0. In all other cases the functional hypervisor transitions must be used to progress the cores. As in the ideal model we filter interrupt pending input signals using the GIC configuration, however we also need to take the virtual interrupts into account. We configure the system so that physical interrupts take priority over virtual interrupts, such that the hypervisor receives any pending physical interrupts first before the guest can handle pending virtual interrupts. Moreover of all physical interrupts, the IGC notification interrupts are configured to have highest priority, so that we can always deliver them while they are pending and the receiving core is ready for it.

3.3 System Steps

System steps handle all the peripheral steps, SMMU, and second stage MMU transitions as well as internal memory steps, GIC steps, and external transitions. The latter MMU and memory transitions as well as answers to memory-mapped I/O requests from the peripherals are all in response to actions of the guests on the core. We use the same transition relations δ_M and δ^p_P as in the ideal model for memory and peripherals (with indices p adapted to the numbering in the refined model). For second stage MMU and SMMU transitions we use the same relation δ_{mmu} which takes an MMU state $mmu \in MMU$ as well as input channels to core/peripherals and memory and transforms them into new states and output channels. A transition is denoted as follows.

$$\delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m')$$

Since the MMUs communicate both with the cores/peripherals and the memory and we model these channels with a limited capacity, we need to define priorities for the handling of requests. First of all we never allow to step an MMU that still has outstanding requests to memory. Moreover we prioritize the handling of replies from memory over the handling of new requests from cores/peripherals. Finally a new such request may only be handled if the sender

has no outstanding replies from its MMU. Note that these constraints are a deliberate simplification as they rule out cores and peripherals that issue several memory requests at a time. Finding accurate semantics for such systems is a current research topic.

For the second stage MMUs their behavior and the constraints on the message protocol are captured by the following rule.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc} \qquad M.u2m(c) = \text{None} \\ M.u2c(c) = \text{None} \qquad c2u = (M.m2u(c) = \text{None})\ ?\ M.c2u(c) : \text{None} \\ \delta_{mmu}(M.mmu(c), c2u, M.m2u(c)) \ni (mmu', u2c', u2m') \\ M'.mmu = M.mmu[c \mapsto mmu'] \\ M'.c2u = M.c2u[c \mapsto \text{None}] \qquad M'.u2c = M.u2c[c \mapsto u2c'] \\ M'.u2m = M.u2m[c \mapsto u2m'] \qquad M'.m2u = M.u2m[c \mapsto \text{None}] \end{array}}{M \xrightarrow{R} M'}\ \text{MMU}$$

Observe that we set the core input channel for the MMU transition relation to None in order to prioritize the handling of memory replies as described above. This setting has to be reversed when updating the input channel again since otherwise an outstanding request from the core could be overwritten by the MMU and thus lost. The rest of the definition should be self-explaining and we define a similar rule for the SMMU transitions for DMA requests and replies.

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np} \qquad M.v2m(p) = \text{None} \qquad M.v2p(p) = \text{None} \\ p2v = (M.m2v(p) = \text{None})\ ?\ M.p2v(p) : \text{None} \qquad p2v \neq \text{None} \Rightarrow p2v \in \mathbb{R} \\ \delta_{mmu}(M.smmu(p), p2v, M.m2v(p)) \ni (smmu', v2p', v2m') \\ M'.smmu = M.smmu[p \mapsto smmu'] \\ M'.p2v = M.p2v[p \mapsto \text{None}] \qquad M'.v2p = M.v2p[p \mapsto v2p'] \\ M'.v2m = M.v2m[p \mapsto v2m'] \qquad M'.m2v = M.m2v[p \mapsto \text{None}] \end{array}}{M \xrightarrow{R} M'}\ \text{SMMU}$$

Peripheral steps are defined just as in the ideal model, only that now they communicate through the SMMU channels instead of directly with the memory.

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np} \\ \delta_P^p(M.P.st(p), M.v2p(p), M.p2v(p), M.E) \ni (P.st', v2p', p2v', e, Q_{out}) \\ M'.P.st = M.P.st[p \mapsto P.st'] \\ M'.p2v = M.p2v[p \mapsto p2v'] \qquad M'.v2p = M.v2p[p \mapsto v2p'] \\ M'.PI = M.PI[p \mapsto Q_{out}] \qquad M'.E = M.E \circ e \end{array}}{M \xrightarrow{R} M'}\ \textsc{Peripheral}$$

Again we need four rules for the memory, handling requests from cores and peripherals, as well as memory-mapped I/O accesses. They are similar to the rules from the ideal model as well using a modified *mmio* predicate that checks

whether addresses of requests lie in $A_P(p)$ instead of $A_P^{g,p}$.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc} \qquad \forall p \in \mathbb{N}_{np}.\ \neg mmio(M.u2m(c), p) \\ M.m2u(c) = \text{None} \qquad \delta_M(M.m, M.u2m(c), \text{C } c) \ni (m', m2u') \\ M'.u2m = M.u2m[c \mapsto \text{None}] \qquad M'.m2u = M.m2u[c \mapsto m2u'] \end{array}}{M \xrightarrow{R} M'} \textsc{Mem-Core}$$

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np} \qquad M.m2v(p) = \text{None} \qquad \forall c \in \mathbb{N}_{nc}.\ \neg mmio(M.v2m(p), c, p) \\ \delta_M(M.m, M.v2m(p), \text{P } p) \ni (m', m2v') \qquad M'.m = m' \\ M'.v2m = M.v2m[p \mapsto \text{None}] \qquad M'.m2v = M.m2v[p \mapsto m2v'] \end{array}}{M \xrightarrow{R} M'} \textsc{Mem-Periph}$$

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc} \qquad p \in \mathbb{N}_{np+1} \\ mmio(M.u2m(c), p) \qquad M.v2p(p) = \text{None} \qquad M'.u2m = M.u2m[c \mapsto \text{None}] \\ M'.v2p = M.v2p[p \mapsto (M.u2m(c), \text{C } c)] \end{array}}{M \xrightarrow{R} M'} \textsc{Mem-I/Oreq}$$

$$\frac{\begin{array}{c} p \in \mathbb{N}_{np+1} \qquad c \in \mathbb{N}_{nc} \qquad mmio(M.v2m(p), c, p) \qquad M.m2u(c) = \text{None} \\ M'.m2u = M.m2u[c \mapsto M.p2v(p)_1] \qquad M'.p2v = M.p2v[p \mapsto \text{None}] \end{array}}{M \xrightarrow{R} M'} \textsc{Mem-I/Orpl}$$

GIC steps are defined just like for the ideal model. The GIC transition relation δ_Q is similar to δ_Q^g only that it ranges over all interrupts of the system and that it also manages the virtual interrupts. Note that we do not need the "no-outstanding-request-interrupt-signalling" constraint here, as it was only necessary to couple the signalling of peripheral interrupts in the ideal model with the injection of virtual interrupts in the refined model. Here, peripheral interrupts can be signalled to the core interfaces at any time but these GIC steps are invisible in the simulation by the ideal model.

$$\frac{\begin{array}{c} m2g = M.v2p(np) \qquad M.p2v(np) = \text{None} \\ \delta_Q(M.GIC, M.Q, M.VI, m2g, M.PI) \ni (GIC', Q', VI', g2m') \\ M'.p2v = M.p2v[np \mapsto g2m'] \qquad M'.v2p = M.v2p[np \mapsto \text{None}] \\ M'.Q = Q' \qquad M'.VI = VI' \qquad M'.GIC = GIC' \end{array}}{M \longrightarrow_g M'} \text{GIC}$$

Finally, we also allow external steps in the refined model, which are defined just like for the ideal one.

$$\frac{e \in \mathcal{E} \qquad M'.E = M.E \circ e}{M \longrightarrow_g M'} \textsc{External}$$

This finishes the definition of the system steps that can be triggered by guests or peripherals. Note that guest cores and peripherals cannot send requests to memory directly but need to pass through their second stage MMUs or SMMUs respectively. Moreover, observe that there are no system or guest steps modifying the second stage MMU or SMMU components. These are configured by

hypervisor code exclusively. We are simplifying the model here since we know from the architecture that guests cannot change second stage MMU registers in EL1 or EL0. The SMMU is configured through memory-mapped I/O accesses which are not visible here. We need to add a verification condition that guests cannot access these memory regions in order to keep the model sound. Proving this would require a further refinement step where the configuration of the MMUs and SMMUs through the hypervisor becomes visible in detail. Here we completely hide the reconfiguration of second stage MMUs and SMMUs in the steps of the secure boot loader and the hypervisor.

3.4 Boot and Hypervisor Steps

Whenever a core is running in exception levels EL2 or EL3, hypervisor or boot code is executing. Note that this is an assumption that is baked in the refined model. It would require a refinement to the machine code level to actually prove it. Instead of modelling the execution of the virtualization platform code on such a low level we instead provide a functional abstraction that models all the relevant effects of such code execution on the refined model. We name this function δ^c_{HV} which captures all hypervisor and boot functionality running on core c. It transforms a core, a second stage MMU, memory and all SMMUs into a new configuration. Additionally it reads from the input memory channel of the core and may produce a new memory request (used here to communicate with the GIC) as well as peripheral event outputs $e \in \mathcal{E}^*$ to power up or down any core in the system. We denote such a transition as follows.

$$\delta^c_{HV}(C, mmu, m, smmu, u2c) = (C', mmu', m', smmu', c2u', e)$$

Note that the hypervisor transitions are modeled as a function, i.e., they are deterministic. Also they only operate on one core and second stage MMU since ARMv8 processors do not allow one core to modify the registers of another one. However, since the hypervisor has unlimited access to memory and as SMMUs are configured through memory-mapped I/O accesses, each core can in principle modify the whole memory and reconfigure any SMMU. Nevertheless in the bisimulation proof with the ideal model we will show that the effects of hypervisor code execution are limited to a single guest, modulo effects due to inter-guest communication.

Observe also that in principle several cores may execute hypervisor code at the same time which has the potential for race conditions on shared resources such as the SMMUs and the GIC. We need to be careful in defining the granularity of the hypervisor transitions so as not to rule out certain interactions between hypervisor threads on different cores. For the boot code this is not an issue since it is executed only on a single core. We also ignore the interaction of the hypervisor core with peripherals and their interrupts, assuming that the hypervisor does not interfere with them. In deed, this is possible since the hypervisor does not configure guest peripherals during boot, ignoring their potential interrupts, and during run-time the configuration of the SMMUs is fixed, avoiding any collision between DMA accesses and SMMU reconfiguration.

We introduce the hypervisor (and boot) rule for the refined model which requires that all channels of the core to be stepped are empty, thus cores must first handle all their outstanding memory requests before switching execution level. We make an exception for memory replies from the GIC so that the hypervisor can communicate with it. To that end, predicate $gicio(x)$ checks if the addresses contained in requests or replies $x \in \mathbb{R} \cup \mathbb{Q}$ are pointing to A_{GIC}.

$$\frac{\begin{array}{c} c \in \mathbb{N}_{nc} \qquad M.C.ps.mode > 1 \qquad M.u2c(c) \neq \text{None} \Rightarrow gicio(M.u2c(c)) \\ M.c2u(c) = \text{None} \qquad M.u2m(c) = \text{None} \qquad M.m2u(c) = \text{None} \\ \delta^c_{HV}(M.C(c), M.mmu(c), M.m, M.smmu, M.u2c(c)) \ni (C', mmu', m', smmu', c2u', e) \\ \forall c'.\ \text{stop}(c') \in e \Rightarrow M.C(c').curr = \text{None} \qquad M'.C = psci(M.C[c \mapsto C'], e) \\ M'.m = m' \qquad M'.mmu = M.mmu[c \mapsto mmu'] \qquad M'.smmu = smmu' \\ M'.c2u = M.c2u[c \mapsto c2u'] \qquad M'.u2c = M.u2c[c \mapsto \bot] \end{array}}{M \xrightarrow{R} M'} \text{HYPERVISOR}$$

Note that the *psci* function used here is similar to the one introduced for the ideal model, but it only resets the EL1 and EL0 registers as well as the processor state and active flag. Thus all registers set by the hypervisor and boot code stay configured. We omit a formal definition here. Again we have a condition that only cores which have no outstanding memory requests may be stopped. Again the history variable *curr* is introduced to the core, with the same semantics as in the ideal model.

Observe also, that all hypervisor and boot steps are invoked by this rule. The specific functionality that needs to be executed depends mainly on the core state upon entry into EL2 or EL3. Then δ_{HV} can be thought of as a finite state machine that models the effects of initialization and handler code. Its details are defined below.

3.4.1 Hypervisor Transition System

In order to define the hypervisor transition system conveniently we introduce the hypervisor state $hv \in \mathbb{H}$ for the hypervisor running core of the refined model. It is a record with components C, *mmu*, *smmu*, *c2u*, and *u2c*, denoting the corresponding components of the respective core, as well as m, denoting the complete physical memory, and an external event sequence e used for power control. Instead of defining transition function δ^c_{HV} directly, the hypervisor and boot transition system is defined by functional transformation rules of this hypervisor state. Note that for memory the hypervisor is only supposed to modify boot and hypervisor, memory, but for simplicity we still define the transitions on the complete memory mapping.

When defining transitions we will have to identify different states of the hypervisor computation where a given transition is enabled. Since *hv* contains the complete memory as well as registers of the guest, such states are not unique. Therefore we introduce state sets $\sigma_{\mathrm{x}} \subseteq \mathbb{H}$ in order to identify them. For example, σ^c_{reset} denotes the possible hypervisor states of a core after reset.

Below we introduce the hypervisor transition system for core $c \in \mathbb{N}_{nc}$ by defining the state predicates σ_{x} and a transition relation $\rightsquigarrow_c \subseteq \mathbb{H} \times \mathbb{H}$. However,

this relation is in fact a function (i.e., every state is mapped to a unique next state), we use the relational notation for ease of representation. We define unique transitions from all the identified hypervisor states in the sets σ_{x}.

Let $\Sigma = \bigcup_{\mathrm{x}} \sigma_{\mathrm{x}}$ be the union of hypervisor states for which we define transitions. Then the hypervisor transition function δ^c_{HV} can be constructed as follows. We first take its arguments C, mmu, m, $smmu$, and $u2c$ to create a hypervisor state hv where $hv.c2u = \text{None}$ and $hv.e = \varepsilon$ are defined to be empty. If $hv \in \Sigma$ there exists a unique transition $hv \rightsquigarrow_c hv'$ and we return $hv'.C$, $hv'.mmu$, $hv'.m$, $hv'.smmu$, $hv'.c2u$, and $hv'.e$ as outputs of δ^c_{HV}. Otherwise the transition function stutters, i.e., we return the respective components of hv. We omit a formal definition here.

Note that this setting allows to perform hypervisor steps at any time even if the hypervisor program itself cannot make progress because it is missing inputs (here these may only be responses from the interrupt controller) or it is in an unidentified state. However we will define the transitions in such a way that the latter case cannot occur and the hypervisor does not get stuck indefinitely (assuming a fair scheduler). In particular we define seven groups of transitions that reflect the boot and hypervisor functionality:

- Initialization,
- Power control SMC handlers,
- GIC distributor virtualization,
- Physical and software interrupt virtualization,
- IGC notification requests,
- IGC notification reception, and
- MMU fault handling.

Below we identify the hypervisor states and transitions for each group. In the transition rule definitions we again follow the convention that we do not mention unchanged components. Specifically, core registers and memory addresses that are not explicitly mentioned are not modified. This abstracts from the internal workings of the hypervisor implementation, e.g., we do not model how the hypervisor modifies general purpose registers or its stack in hypervisor memory during intermediate computations. Nevertheless, these effects are not visible in the ideal model and thus not part of the bisimulation between ideal and refined model, so it is fine to ignore them here.

Moreover in order to define the transitions we sometimes need to know which guest owns a given core $c \in \mathbb{N}_{nc}$ belongs. This information is given by the guest projection function $\gamma : \mathbb{N}_{nc} \to \mathbb{N}_{ng}$, i.e., refined core c is owned by guest $\gamma(c)$. In addition $\kappa : \mathbb{N}_{nc} \to \mathbb{N}_{nc}$ maps a given core index to its ideal index within the guest it belongs to. We will define these and other projections more thoroughly in Sect. 4.1.

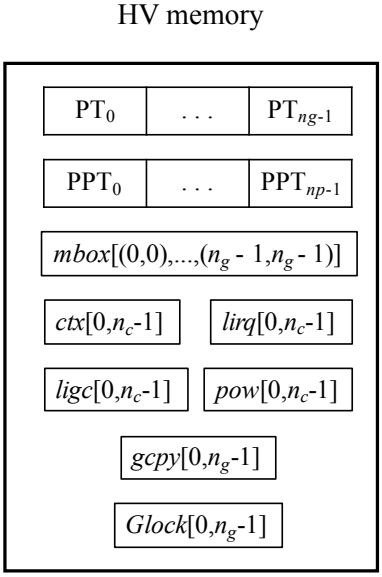

Figure 4: Data structures in hypervisor memory.

3.4.2 Hypervisor Data Structures

As explained above, the hypervisor transitions also affect memory content. However for all transitions except those related to the boot process these effects are limited to the hypervisor memory region, where its internal data structures are stored. Below we introduce those structures that are relevant for the hypervisor transition system and the bisimulation with the refined model. They are also summarized in Fig. 4.

During run-time, the hypervisor is entered from a previous guest computation on several occasions, e.g., to serve hypercalls or for handling peripheral interrupts. On each entry to hypervisor, i.e., after an exception from the guest or a physical interrupt, the guest context is stored in general and special purpose registers. In particular the general purpose register content of the guest are still present in the registers, the program counter is saved to the *ELR_EL2* special purpose register (increased by 4 in case of system calls), and the guest's processor state is saved in *SPSR_EL2*. After that the hypervisor needs to save this guest context information in its own memory. Specifically, the guest context of core c is saved in a page at $a^c_{ctx} \in A_{HV}$. We introduce a context abstraction from that page to a record, that distinguishes the saved PC value for system calls, i.e., it restores from memory the context of the guest at hypervisor entry depending on a bit that says whether the entry was caused by a system call.

$$ctx : \mathbb{B}^{4096 \cdot 8} \times \mathbb{B} \times \mathbb{N}_{nc} \to \{ps \in \mathbb{P};\ pc \in \mathbb{B}^{64};\ gpr : GPR_{\mathbb{G}} \to \mathbb{B}^{64}\}$$

We omit a formal definition here. Note, that we do not save the special purpose register of a guest, since at ARMv8 all exception levels have a separate set of

special purpose registers and the hypervisor does not need to use the ones of the guest for its own computations. Note also that the saved guest context is only relevant in hypervisor handlers that are not executing atomically, then it is exposed in the intermediate states of the handler transitions.

In addition to a guest's execution context intermediate states of the hypervisor also need to store information about the cause of the handler execution. This is particularly necessary when handling peripheral and IGC interrupts for the guest. The last received peripheral or IGC interrupts (along with the sender) on any given core are stored in pages a_{lirq}, and a_{ligc} respectively. Again we introduce abstraction functions, hiding implementation details.

$$\mathit{lirq} : \mathbb{B}^{4096\cdot 8} \times \mathbb{N}_{nc} \to \mathit{PIRQ} \cup \mathit{IPIRQ} \quad \mathit{ligc} : \mathbb{B}^{4096\cdot 8} \times \mathbb{N}_{nc} \to (\mathbb{N}_{ns} \times \mathbb{N}_{nc}) \cup \{\bot\}$$

For the implementation of the IGC mechanism another data structure is used. The hypervisor uses a message box to control when notification interrupts may be sent to a potential receiving core. The notifications themself are implemented by software-generated interrupts (SGI). For each channel the message box contains a bit that is one when a notification interrupt was sent for this channel but not yet received. Upon reception of the interrupt, the hypervisor clears the bit and injects a virtual interrupt into the guest. After that, new notification interrupts may be sent.

The message box is implemented as a 32x32 table of bytes (since there may be at most 32 cores and thus guests in ARMv8 processors) and therefore is 1KiB big. It is allocated at a physical base address that is 1KiB-aligned with the 38 most significant bits denoted by $a_{mbox} \in \mathbb{B}^{38}$. We require that $a_{mbox}[37:2] \in A_{HV}$ holds, i.e., the box is allocated within a page of hypervisor memory. The message box entry for the channel from guest g to guest g' in the 1KB message box $b \in \mathbb{B}^{1024\cdot 8}$ is retrieved by

$$\mathit{mbe}(b, g, g') = \langle b[32 \cdot 8g + 8g' + 7 : 32 \cdot 8g + 8g'] \rangle \,.$$

In order to access entries in the IGC message box for the channel from guest g to guest g' of a given hypervisor state hv we define the following shorthand.

$$\mathit{mbox}(hv, g, g') = \mathit{mbe}(hv.m(a_{mbox}[37:2])[b + 1024 \cdot 8 - 1 : b], g, g')$$

Here $b = 1024 \cdot 8 \cdot \langle a_{mbox}[1:0] \rangle$ encodes the message box base offset (in number of bits) based on the lower two bits of a_{mbox} which are used to determine the location of the 1KiB message box within its containing 4KiB page.

Note that the IGC message box basically functions as a lock for the notification interrupts. Thus there is a locking policy to set and clear the mesage box entry. Since any entry is only relevant for two guests (the sender and the receiver for the corresponding IGC channel), and only one core of the receiving guest is actually interrupted by the SGI (the guest's powered up core with the lowest index) this policy is somewhat simple compared to other locking schemes.

In particular the receiver can reset the message box entry to zero at any point without risking race conditions with other writers, as noone else is allowed to

reset the entry but the receiver. However, several cores of a sender guest may compete for sending a notification, thus setting a message box entry to one needs to be implemented like acquiring a lock using the appropriate atomic memory operations. Whichever core of the sender succeeds in setting the bit is allowed to send the SGI to the receiver. The other cores simply drop their core requests as one notification is being sent anyway on behalf of the guest they belong to.

Nevertheless, note that these implementation details are not exposed in the refined model. Here the updates of the message box are modeled as atomic transactions and if several hypervisor handlers of the same guest are ready to set the message box entry for the same channel to one, the non-deterministic choice of the scheduling for the refined model computation decides which handler succeeds. Whenever one of the other competitors is scheduled again, it will see that the message has already been sent by another core and perform the transition that drops the request.

For selecting the right receiver core for the IGC notification, the hypervisor also keeps track of the cores that are powered on in the system. Function $pow : \mathbb{B}^{4096 \cdot 8} \times \mathbb{N}_{nc} \to \mathbb{B}$ abstracts from the data structure holding this information that is allocated in page $a_{pow} \in A_{HV}$.

Besides the hypervisor data structures described above, hypervisor memory also contains the page tables for the second-stage MMUs and the SMMUs. We do not introduce them here formally since they are set up once and for all during initialization and thus are invariant during any further hypervisor transitions.

3.4.3 Boot and Initialization

Before any guest can run, the HASPOC platform has to be initialized by the hypervisor. After reset the boot loader copies validates all relevent images to main memory and transfers control to the hypervisor which takes several steps to configure the system. This configuration process is partly concurrent and illustrated in Fig. 5

We distinguish the following steps.

- After a cold reset, i.e., power-up of the system, several transitions are possible:
  - PCold – we need to distinguish the primary core (index 0) from the secondary ones. In this transition, the primary core executes the complete boot code and enters the hypervisor.
  - SCold – after a cold reset the secondary cores just configure a few EL3 system registers and then go to sleep, i.e., themselves of again, waiting to be woken up by the primary core with a *start* power control command after it finishes the initial hypervisor set-up.
  - Abort – if the primary core executing the boot code is not able to verify the signatures of its input images it will fail and get stuck.
- PInit – the primary core initializes internal hypervisor data structures. No configuration is yet performed for the guests.

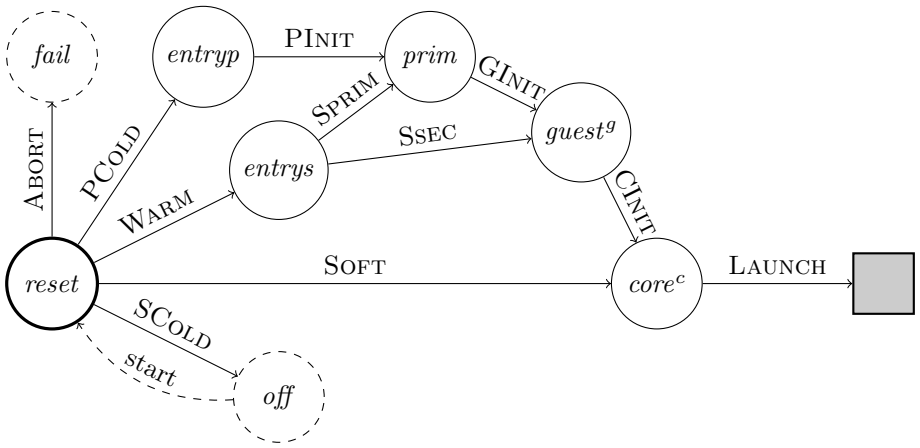

Figure 5: Transition diagram for the boot and initialization phase. Thick circles denote entry states and dashed circles stand for sink states. Boxes denote states in which the guest is executing.

- GInit – the executing core initializes the guest-specific data structures for the guest it belongs to. These include the second-stage page tables and the SMMU page tables for the peripherals belonging to that guest. Moreover the SMMUs are configured and activated in this step. In case the primary core is executing it also wakes up the secondary cores which are mapped to the first core of each guest. Note that we have $\kappa(0) = 0$, i.e., the primary core in the refined model is also the primary core of some guest in the ideal model.

- CInit – when the guest's data structures in the hypervisor are set up, the primary core of that guest will be launched. Before this can be done, some virtualization-related registers on the core have to be set up, e.g., the page table origin, or certain hypervisor registers that control the second-stage MMU and traps for guest execution. The second-stage MMUs for the current core are configured and activated in this step.

- Launch – finally the guest is starting its computation. On the HASPOC platform we start guests with only their primary cores being powered-on. Afterwards the guest can request power-up for its other cores. Before that these cores are still in the powered-off state reached by the secondary cold boot.

- Warm – we call it a warm boot, when the secondary cores are woken up again by the primary core. Note that the state in which the secondary cores are powered up are different from the cold reset state, e.g., certain system registers may still be set up, so the boot loader can distinguish between the warm and cold reset. However we omit going into details

here for simplicity. With this transition the secondary cores enter the hypervisor as well.

- SPRIM – after the warm boot of a secondary core we need to distinguish whether this core is mapped to a primary or secondary core of some guest in the ideal model. If it is the primary core we jump to the code which is responsible to implement the guest-specific data structures for the corresponding guest.
- SSEC – if the secondary core of a guest is (warm) booted, all the guests data structures have already been set up. We can jump directly to the configuration of the virtualization-related registers. Afterwards guest execution can also be launched on this core.
- SOFT – when a guest requires to power down one of its cores, just like during secondary cold boot, these cores are not reset completely. We assume that all relevant system registers are restored as in the state when the core was powered down earlier. Then the guest can be re-launched directly without further configuration. However the EL1 and EL0 register state is wiped.

We start definining the transitions and states for the initialization phase by introducing the reset states $hv \in \sigma^c_{\text{reset}}$. They are identified by the following conditions.

- $hv.C = C^c_{reset}$ – all components of the core that are not written during the initialization phase and all EL1 and EL0 registers are in their reset configuration. This requires in particular:
  - $hv.C.active = 1$ – the core is powered up.
  - $hv.C.spr(MPIDR\_EL1)$ contains the encoding for the core's index c. Other identification registers must contain valid data as well. We omit the technical details here.
  - $hv.ps.mode = 3$ – the core is running in EL3.
- All input images are in place. In particular for all $img \in \{img_{rk}, img_{boot}, img_{HV}, img_{HVc}, img_{Mc}\}$ and $a \in Ra_i(img)$ we have:

$$hv.m(a) = \langle img \rangle [\langle a \rangle - \langle img.adr \rangle + 4096 \cdot 8 - 1 : \langle a \rangle - \langle img.adr \rangle]$$

  Moreover, for all $g \in \mathbb{N}_{ng}$, $img \in \{img^g_G, img^g_{Gc}\}$, and $a \in Ra_i(img)$ we require the same property.
- $hv.u2c = \text{None}$ – there are no outstanding memory replies.

Observe that we do not restrict the value of the program counter. It will help us to distinguish the different forms of reset.

We first define the fail case. Predicate $fail : hv \to \mathbb{B}$ is true if there exists an image img from the set of HASPOC images IMG_H listed above such that

$\neg check(img_{rk}.data, img)$, i.e., if the signature check on them fails, or the hash of the root key $img_{rk}.data$ is not equal to the root key hash stored at ROM address a_{rkh}. Then the fail transition is formalized as follows.

$$\frac{\begin{array}{c} hv \in \sigma^c_{\text{reset}} \qquad c = 0 \\ fail(hv) \qquad hv.C.pc = hv.C.spr(RVBAR\_EL3) \qquad hv'.C.pc = a_{fail} \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-ABORT}$$

The transition is only allowed for the primary core after a cold reset. This type of reset is identified by the PC being equal to the hardware-defined reset address which is stored in the read-only register *RVBAR_EL3*. In the fail case we jump to address a_{fail} with $a_{fail}[47:12] \in A_{ROM}$ which is different from all other addresses that we introduce in the remainder of this document. Once the system reaches this state it is stuck there as we do not define a transition out of it. This reflects the boot loader implementation entering an endless loop for any integrity check failure.

If the primary core succeeds in booting the hypervisor, all images have been copied at the right place. This property represents the functional correctness of the boot code. Strictly speaking we would not need to specify it here, since the hypervisor model abstracts from the machine code execution. However we still include it in the following transition as a sanity check that every refinement to the machine code level has to pass.

Before the hypervisor is entered we also need to configure the *SCR_EL3* since this is only possible in the highest exception level EL3. We introduce the desired value $scr_H \in \mathbb{B}^{32}$ to be set up as follows.

- $scr_H[10] = 1$ – EL2 executes at AArch64 mode.
- $scr_H[8] = 1$ – the HVC instruction is enabled.
- $scr_H[0] = 1$ – EL1 and EL0 are running in non-secure mode.
- all other bits are set to zero, disabling among other features the trapping of interrupts to EL3 and enabling the SMC instruction.

Moreover, we set the exception vector base address to page $vbar_{psci} \in \mathbb{B}^{36}$ which is where the power control handlers are allocated. Then the primary cold boot configuration step is modeled by the definition below.

$$\frac{\begin{array}{c} hv \in \sigma^c_{\text{reset}} \qquad c = 0 \qquad \neg fail(hv) \\ hv.C.pc = hv.C.spr(RVBAR\_EL3) \qquad \forall img \in IMG_H, a \in Ra_d(img). \\ hv'.m(a) = img.data[\langle a \rangle - \langle img.dadr \rangle + 4096 \cdot 8 - 1 : \langle a \rangle - \langle img.dadr \rangle] \\ hv'.C.spr(SCR\_EL3) = scr_H \\ hv'.C.spr(VBAR\_EL3) = 0^{16} \circ vbar_{psci} \circ 0^{12} \\ hv'.C.pc = a_{iep} \qquad hv'.C.ps.mode = 2 \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-PCOLD}$$

The exception level is lowered to EL2 and the program counter is set to the address a_{iep} with $a_{iep}[47:12] \in A_{HV}$ to denote the state where we enter the

hypervisor. More specifically, such states are captured by the set $hv \in \sigma^{entryp}_{\text{INIT}}$ where $hv.C.pc = a_{iep}$ and $hv.C.ps.mode = 2$. From these states the primary core sets up the internal hypervisor data structures, i.e., the IGC message box, GIC lock, and power state for all cores. However, internally the hypervisor implementation has a lot more to set up, e.g., it's own address translation.

In this step the primary core also wakes up all secondary cores that are primary within their corresponding guest using an iterative version of the concatenation operator $\circ$ to create the appropriate power control event sequence.

$$\frac{\begin{array}{c} hv \in \sigma^{entryp}_{\text{INIT}} \qquad c = 0 \qquad \forall g, g' \in \mathbb{N}_{ng}.\ mbox(hv', g, g') = 0 \\ Glock(hv.m(a_{glk})) = \bot \qquad hv'.C.pc = a_{iprm} \qquad S = \{c' \mid c' > 0 \wedge \kappa(c') = 0\} \\ hv'.e = \circ_{c' \in S} \text{start}(c') \qquad \forall c' \in S \cup \{0\}.\ pow(hv'.m(a_{pow}), c') = 1 \\ \forall c' \notin S \cup \{0\}.\ pow(hv'.m(a_{pow}), c') = 0 \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-PINIT}$$

The program counter is set to a_{iprm} with $a_{iprm}[47:12] \in A_{HV}$. The resulting hypervisor states $hv \in \sigma^{prim}_{\text{INIT}}$ are identified by the conditions $hv.C.pc = a_{iprm}$ and $hv.C.ps.mode = 2$.

In the next step the hypervisor sets up the guest data structures. For a given guest g it has to fill the page table area according to a *golden image* $GI_g : A_{PT}(g) \to \mathbb{B}^{4096 \cdot 8}$ that represents a page table set up such that all guests have disjoint memories (modulo IGC channels). We will constrain the golden image later in more detail by putting requirements on the translations the MMU produces using that page table configuration. Similarly for each guest there is a golden image $GIP_g : A_{PPT}(g) \to \mathbb{B}^{4096 \cdot 8}$ for the SMMU page tables that constrain peripheral DMA accesses. Note that each peripheral is in principle allowed to write the whole memory of the guest it belongs to, therefore one golden SMMU page table image is sufficient for all peripherals of a guest.

For the current guest the hypervisor also configures and enables all the SMMU dedicated to its peripherals. The projection function $\gamma : \mathbb{N}_{np} \to \mathbb{N}_{ng}$ maps peripherals to their owner. A configuration for the SMMU that its the golden image for that guest is provided by $GIPconf_g \in \mathbb{C}$. We assume here that the top-level page table is stored at the minimal address in the ares of page tables reserved for each guest, i.e., the page table origin component of the SMMUs must point to these addresses. Formally, the desired SMMU configuration for guest g is given by a set $SMMU^g_{GI} \subset MMU$ as follows.

- $SMMU^g_{GI}.active = 1$ – the SMMU is active and running.
- $SMMU^g_{GI}.pto = \min\{A_{PPT}(g)\} \circ 0^{12}$ – page table origin is configured to point to the top level page table.
- $SMMU^g_{GI}.cfg = GIPconf_g$ – The remaining control registers of the SMMU are set up according to the golden image configuration.

Moreover, we set $smmu(p).walk = \bot$ initially, i.e., no translation is currently active, as we assume that all peripherals are still idle. With these principles in

mind, we define the guest initialization step as follows.

$$\frac{\begin{array}{c} hv \in \sigma^{prim}_{\text{INIT}} \qquad \gamma(c) = g \\ \forall a \in A_{PT}(g).\ hv'.m(a) = GI_g(a) \qquad \forall a \in A_{PPT}(g).\ hv'.m(a) = GIP_g(a) \\ \forall p \in \mathbb{N}_{np}.\ \gamma(p) = g \Rightarrow hv'.smmu(p) \in SMMU^g_{GI} \\ \forall p \in \mathbb{N}_{np}.\ \gamma(p) = g \Rightarrow hv'.smmu(p).walk = \bot \qquad hv'.C.pc = a_{igst} \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-GINIT}$$

The program counter points to address a_{igst} with $a_{igst}[47:12] \in A_{HV}$. The corresponding hypervisor states $hv \in \sigma^{guest,g}_{\text{INIT}}$ for guest g are identified by the conditions $hv.C.pc = a_{igst}$ and $hv.C.ps.mode = 2$. The last step before launching the guest requires to complete up all core-specific preparations. In particular the second-stage MMUs have to be set up according to set of configurations $MMU^g_{GI} \in MMU$ as follows.

- $MMU^g_{GI}.active = 1$ – the SMMU is active and running.
- $MMU^g_{GI}.pto = \min\{A_{PT}(g)\} \circ 0^{12}$ – page table origin is configured to point to the top level page table.
- $MMU^g_{GI}.cfg = GIconf_g$ – The remaining control registers of the MMU are set up according to the golden image configuration.

Moreover the hypervisor control registers *HCR_EL2* needs to be updated to the value $hcr^{vi}_H \in \mathbb{B}^{64}$ for $vi \in \mathbb{B}$ with the following bits set.

- $hcr^{vi}_H[31] = 1$ – EL1 is running in AArch64 mode.
- $hcr^{vi}_H[19] = 1$ – trap SMC instructions from EL1 to EL2.
- $hcr^{vi}_H[18:15] = 1111$ – trap accesses to identification registers to EL2. In order to maintain the ideal view of the system for the guest, the hypervisor needs to virtualize all reads to identification registers. However we omit the discussions of these handlers here, as they require to introduce a large amount of tedious details.
- $hcr^{vi}_H[7] = vi$ – this bit signals virtual interrupts to the guest system, however we don't constrain it and allow both values throught the execution of the system.
- $hcr^{vi}_H[5:3] = 111$ – trap all physical interrupts during guest execution to EL2.
- $hcr^{vi}_H[2] = 1$ – treat translation walks to device memory as Permission Faults.
- $hcr^{vi}_H[1] = 1$ – force all data cache invalidations by set/way indexing to clean the cache as well. This is needed to maintain memory coherency and isolation in the presence of caches.

- $hcr_H^{vi}[0] = 1$ – enable the second stage MMU. This bit is already mirrored in the configuration of the MMU but we list it here anyway.
- all other bits are set to zero, disabling a number of traps and cache configuration options. In a cache-aware model some of these options need to be revisited. We leave this task to future research.

Furthermore, we need to set the EL2 exception vector base address to the page address $vbar_H \in \mathbb{B}^{36}$ where the hypervisor handler code is allocated. Finally, the hypervisor data structures recording the last IRQ or IGC interrupt are initialized to default values. Now the core configuration step is defined as follows.

$$\frac{\begin{array}{c} hv \in \sigma_{\text{INIT}}^{guest,g} \qquad \gamma(c) = g \qquad \forall c \in \mathbb{N}_{nc}.\ \gamma(c) = g \Rightarrow hv'.mmu(c) \in MMU_{GI}^{g} \\ \forall c \in \mathbb{N}_{nc}.\ \gamma(c) = g \Rightarrow hv'.mmu(c).walk = \bot \\ hv'.C.spr(HCR\_EL2) = hcr_H^{vi} \\ hv'.C.spr(VBAR\_EL2) = 0^{16} \circ vbar_H \circ 0^{12} \qquad lirq(hv'.m(a_{lirq}), c) = 1023 \\ ligc(hv'.m(a_{ligc}), c) = \bot \qquad hv'.C.pc = a_{icor} \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-CINIT}$$

After the step the PC is pointing to a_{icor} with $a_{icor}[47:12] \in A_{HV}$, denoting the states before the launch of the guest. Formally such states $hv \in \sigma_{\text{INIT}}^{core,c}$ are identified by $hv.C.pc = a_{icor}$ and $hv.C.ps.mode = 2$. The launch of the guest is described by the following transition.

$$\frac{\begin{array}{c} hv \in \sigma_{\text{INIT}}^{core,c} \qquad \gamma(c) = g \qquad \forall r \in GPR_{\mathbb{G}}.\ hv'.C.gpr(r) = C_{reset}^{g,\gamma(c)}.gpr(r) \\ \forall r \in SPR_{\mathbb{G}}.\ hv'.C.spr(r) = C_{reset}^{g,\gamma(c)}.spr(r) \\ hv'.C.ps = C_{reset}^{g,\gamma(c)}.ps \qquad hv'.C.pc = C_{reset}^{g,\gamma(c)}.pc \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-LAUNCH}$$

We simulate the reset state as defined in the ideal model. In particular we have $C_{reset}^{g,\gamma(c)}.ps = 1$ and $C_{reset}^{g,\gamma(c)}.pc = 0^{16} \circ img_G^g.entry$, i.e., guest execution starts in EL1 and at the entry point as defined in the corresponding guest image. Note that initially the first-stage MMU of the guest is disabled, thus the PC is set to a zero-extended intermediate physical address.

While this step finishes the transitions for the primary core, there are still steps of the secondary core left to define. First we cover the cold boot transition.

$$\frac{\begin{array}{c} hv \in \sigma_{\text{reset}}^{c} \qquad c \neq 0 \qquad hv.C.pc = hv.C.spr(RVBAR\_EL3) \\ hv'.C.spr(VBAR\_EL3) = 0^{16} \circ vbar_{psci} \circ 0^{12} \\ hv'.C.spr(SCR\_EL3) = scr_H \qquad hv'.e = \text{stop}(c) \end{array}}{hv \rightsquigarrow_c hv'} \text{ INIT-SCOLD}$$

We simply set up the *SCR_EL3* register and the exception vector interrupt base address as in the primary cold boot step. After that the running core is powered down by a power control command. If it is restored by another core, its PC will be reset pointing to the corresponding power control handler. Such states are contained in the set $\sigma_{\text{reset}}^{c}$.

The subsequent transition takes execution into the hypervisor entry point for secondary guests. This warm boot state is identified by predicate $warm : \mathbb{H} \to \mathbb{B}$, but we do not give a formal definition here how the implementation distinguishes it from the cold boot and the soft boot state.

$$\frac{hv \in \sigma^c_{\text{reset}} \qquad c \neq 0 \qquad warm(hv) \qquad hv'.C.pc = a_{ies} \qquad hv'.C.ps = 2}{hv \rightsquigarrow_c hv'} \textsc{Init-Warm}$$

The following states $hv \in \sigma^{entrys}_{\text{INIT}}$ are identified by the conditions $hv.C.pc = a_{ies}$ and $hv.C.ps = 2$. From here the core either jumps to the guest or core initialization step, depending on whether it is the primary or secondary core of its corresponding guest.

$$\frac{hv \in \sigma^{entrys}_{\text{INIT}} \qquad \gamma(c) = 0 \qquad hv'.C.pc = a_{iprm}}{hv \rightsquigarrow_c hv'} \textsc{Init-Sprim}$$

$$\frac{hv \in \sigma^{entrys}_{\text{INIT}} \qquad \gamma(c) \neq 0 \qquad hv'.C.pc = a_{igst}}{hv \rightsquigarrow_c hv'} \textsc{Init-Ssec}$$

Lastly, we define the soft boot transition starting a subset of σ^c_{reset}. As for warm boot we assume a predicate $soft : \mathbb{H} \to \mathbb{B}$ that identifies a soft reset state based on the hypervisor register and memory content. Then the transition takes execution directly to the launch of the guest core.

$$\frac{hv \in \sigma^c_{\text{reset}} \qquad c \neq 0 \qquad soft(hv) \qquad hv'.C.pc = a_{icor} \qquad hv'.C.ps = 2}{hv \rightsquigarrow_c hv'} \textsc{Init-Soft}$$

This finishes the definition of the boot and initialization phase. There are some noteworthy observations to be made.

- Guests are launched concurrently and the order in which they are launched is depending on the non-deterministic scheduling of the overall refined model. However, for each guest, first its primary core becomes active and all other cores of that guest are still powered down.
- The SMMUs are configured once and for all after all guests are launched. Their configuration and page tables are never touched again.
- There is no interaction with the GIC during the boot phase. We require that the GIC is initially turned off completely and only activated upon request of a guest. Likewise, no physical or virtual interrupts are pending, but to avoid any unexpected interruptions, the boot and hypervisor code masks all exceptions during its execution.
- Peripherals of the guests are not touched during the boot and initialization phase. We assume that they remain inactive until they are initialized by the owning guest.

- We defined a number of hypervisor code addresses above. All of these (and the ones to be defined below) are mutually different, so that the different hypervisor states can be distinguished. Similarly we assume that $warm(hv)$ and $soft(hv)$ are never true simultanously and do not hold in the cold boot state, i.e., $hv.C.pc \neq hv.C.spr(RVBAR\_EL3)$.

3.4.4 Power Control SMC Handlers

For the power control handlers we first define the possible entry states $\sigma^{entry}_{\text{SMC}}$. For every such state hv we have:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = hv.C.spr(VBAR\_EL2) +_{64} 0^{53}10^{10}$ – The program counter is pointing to the exception vector for synchronous exceptions from a lower exception level executing in AArch64 mode (exception vector base address + offset 0x400). At this address the hypervisor exception handler dispatcher code is located.
- $hv.C.spr(ESR\_EL2)[31:26] = 010111$ – The exception syndrome register holds exception code 0x17, denoting an SMC exception.
- $hv.C.u2c = \text{None}$ – There are no outstanding replies from the GIC.

Since we trap all guest SMC calls to EL2, the entry states as defined above cover all states produced by a refined guest step executing an SMC instruction. The particular service requested by the guest is encoded in an intermediate constant of the SMC instruction that is stored in the lower 16 bits of *ESR_EL2* when the exception is taken. We do not go into the technical details here but instead assume a decoding function

$$decsmc : \mathbb{B}^{16} \to \bigcup_{c \in \mathbb{N}_{nc}} \{\text{start}(c), \text{stop}(c)\} \cup \{\bot\}$$

which returns the desired power control operation, or $\bot$ in case the encoding is not recognized. Then there are two possible steps that can be taken; either the request is permitted by the mapping of cores to guests or it is denied. Only in the former case the power control command is issued via the event signal. We depict the transition system in Fig. 6 where thick circles denote hypervisor entry points and gray boxes denote exit points to guest execution. The corresponding rules are defined as follows.

$$\frac{\begin{array}{c} hv \in \sigma^{entry}_{\text{SMC}} \qquad cmd = decsmc(hv.C.spr(ESR\_EL2)[15:0]) \\ \exists c'.\ cmd \in \{\text{start}(c'), \text{stop}(c')\} \wedge \gamma(c') = \gamma(c) \\ hv'.e = cmd \qquad pow(hv'.m(a_{pow}), c') = (cmd = \text{start}(c'))\ ?\ 1 : 0 \\ hv'.C.pc = hv.C.spr(ELR\_EL2) \qquad hv'.C.ps = PS(hv.C.spr(SPSR\_EL2)) \end{array}}{hv \rightsquigarrow_c hv'} \text{SMC-Issue}$$

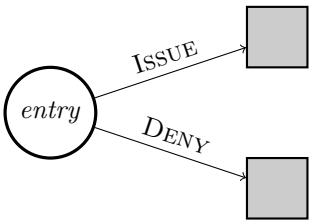

Figure 6: Transition diagram for the power control SMC handlers.

The above rule describes the successful case, i.e., when the guest requests to power on or off a core that belongs to it. The event signal is updated and the power control action is performed according to the HYPERVISOR rule of the refined system. Also the internal hypervisor structure keeping track of core's power status is updated. Moreover the guest is restored with only the program counter changed, pointing to the address of the next instruction which was saved in the *ELR_EL2* system register. In particular, the general purpose registers, the special purpose registers and the memory of the guest are not affected by this hypervisor transition. Function $PS : \mathbb{B}^{64} \to \mathbb{P}$ casts the stored processor state from binary representation back to a processor state record.

$$\frac{\begin{array}{c} hv \in \sigma^{entry}_{\text{SMC}} \qquad cmd = decsmc(hv.C.spr(ESR\_EL2)[15:0]) \\ cmd = \bot \vee \exists c'.\ cmd \in \{\text{start}(c'), \text{stop}(c')\} \wedge \gamma(c') \neq \gamma(c) \qquad hv'.e = \varepsilon \\ hv'.C.pc = hv.C.spr(ELR\_EL2) \qquad hv'.C.ps = PS(hv.C.spr(SPSR\_EL2)) \end{array}}{hv \rightsquigarrow_c hv'} \text{SMC-DENY}$$

If the guest request to power on or off a core that does not belongs to it, or sends an invalid request, the hypervisor simply ignores the SMC interrupt and restores the guest. No power control events are generated.

3.4.5 GIC Distributor Virtualization

In the ideal model every guest has their own interrupt controller, however in the real system (and the refined model) there is only one GIC. On the HASPOC platform and its demonstrators this controller implements the GICv2 standard for ARM interrupt controllers and consists of four memory-mapped components: 1. an interrupt distributor that is shared by all cores, 2. physical CPU interrupt interfaces for each core, 3. virtual CPU interrupt interfaces for each core, and 4. the virtual interrupt control interface for each core.

Within the context of this document virtual interrupt and their control interfaces are only visible in the refined model, while the distributor and physical interfaces are represented in the *GIC* component(s) in both models. In the bisimulation between both models the ideal physical CPU interfaces for the guests are mapped to the virtual interfaces in the refined model for each core. This virtualization is supported natively by GICv2 interrupt controlers. For the

distributor however, the hypervisor needs to create the illusion that each guest has their own (guest-specific) interrupt distributor by emulating accesses to it.

This virtualization is non-trivial and requires the hypervisor to trap guest memory accesses to the (intermediate physical) GIC distributor address region, decode these accesses to decide whether the request to the distributor by the guest is allowed, and communicate modified requests to the GIC on behalf of the guest. Finally for reads of distributor registers the hypervisor also needs to filter out all information about interrupts not visible to the guest in question to uphold the isolation of information flow between guests.

For some of these reads the hypervisor can avoid interaction with the GIC by keeping local copies of the distributor configuration that is kept up to date with every update of the distributor. This is useful, since different hypervisor threads (running on separate cores) need to synchronize their accesses to the GIC in order to avoid race conditions. Internally the hypervisor needs employs a locking discipline for accessing the GIC distributor which increases the performance overhead of the virtualization solution, thus hypervisor GIC accesses are best avoided.

Below we list all the registers of a GICv2 interrupt distributor and how accesses to them are virtualized.

- *GICD_CTLR* – contains two bits to enable and disable the forwarding of secure and non-secure interrupts to the cores. We always keep the distributor enabled for non-secure interrupts.

  Reads Return a local copy of the state of the enabled bit for the requesting guest.

  Writes If the guest wants to disable its distributor, disable forwarding for all its interrupts individually in the distributor and clear the local copy of the enabled bit. Similarly, activate forwarding of all of the guest's interrupts when it requests to enable its distributor and set the local enabled bit. We assume here for simplicity that the enabling and disabling of all of the guests interrupts can be performed by a single GIC access, which is most likely not the case, depending on the interrupt IDs associated with a guest. Extending the transition system to cover all scenarios is straight-forward but tedious.

- *GICD_TYPER* – contains information about the interrupt controller including the number of interrupt lines and whether security extensions are supported. This is a read-only register.

  Reads Return a fixed virtualized copy for each guest.

  Writes Ignored

- *GICD_IIDR* – contains version information about the implementation of the GIC. This is a read-only register.

  Reads Return a fixed virtualized copy for each guest.

Writes Ignored

- *GICD_IGROUPRn* – maps interrupt IDs to either Group 0 (used for Secure interrupts) or Group 1 (used for Non-Secure interrupts). Guest access to these registers is restricted and Guest interrupts are always in Group 1. Upon guest access to these registers, inject a Data Abort into the guest.

- *GICD_ISENABLERn*/*GICD_ICENABLERn* – registers containing bits to enable and disable the forwarding of particular interrupts, excluding SGIs which are always enabled. These bits are used to implement the *GICD_CTLR* virtualization.

  Reads Keep a local copy of the last write attempted by the guest. Return it with all other interrupts shown as disabled.

  Writes Analyze the written data to determine which of the guests interrupts should be enabled or disabled. Update the local copies of the bits for the requesting guest accordingly and send the corresponding modified write request to the GIC. Updates of interrupts not belonging to that guest are ignored in the GIC request (similar to requests to its ideal GIC). Note that SGIs are always enabled, thus the GIC will ignore any writes to the corresponding bits.

- *GICD_ISPENDRn*/*GICD_ICPENDRn* – registers containing bits to view and control (set and clear) the pending status for each interrupt. Modifying these bits is useful, e.g., when saving and restoring the interrupt state of the currently running process in an operating system, however we only allow modification to interrupts belonging to the requesting guest.

  Reads Since interrupts may become pending asynchronously we cannot keep (fresh) local copies of these registers. Perform the read to the GIC and return a filtered version to the guest, marking all interrupts not belonging to the guest as not pending.

  Writes Forward a modified write to the GIC like for the enable and disable bits.

- *GICD_ISACTIVERn*/*GICD_ICACTIVERn* – registers containing bits for controlling (set and clear) the active status of each interrupt. Handle reads and writes in the same way as accesses to the pending bits.

- *GICD_IPRIORITYRn* – registers containing the eight-bit priority set for each interrupt. Since we do not cover dynamic interrupt priorities in our models we need to assign priorities to interrupts of guests according to our fixed priority order to match the semantics of the ideal model. Both in the refined and the ideal model reads and writes to the priority registers have the following effect:

  Reads Return the standard priority values as stored in the local copy.

  Writes Ignored

To support dynamic priorities we would handle reads and writes to the priority registers in the same way as access to the enable and disable bits, however also the GIC model would have to be extended.

- *GICD_ITARGETSn* – registers containing bit masks to determine the target core(s) for each interrupt.

  Reads Return a local copy to the guest where information for un-owned interrupts is suppressed. Moreover the core IDs from the refined model need to be converted to the corresponding core IDs in the ideal model.

  Writes Update the local copy according to the eligible parts of the guest request and send a modified write to the GIC translating the ideal core IDs to refined ones.

- *GICD_ICFGRn* – containing bits for each interrupt to control whether it is edge-triggered or level-triggered. Handle reads and writes in the same way as accesses to the enabled and disabled bits.

- *GICD_NSACRn* – optional Secure registers. These are not accessible to the guest and for every read or write to them the hypervisor injects a Data Abort into the guest.

- *GICD_SGIR* – controls the generation of SGIs. We allow guests to send inter-processor interrupts to their own cores but not to those of other guests. Note that SGI IDs used by the guest do not interfere with the ones used for the IGC implementation since the GIC exposes the sender for each SGI and IGC interrupts are never sent within a guest. Therefore we can share the SGI IDs for different purposes in the refined model.

  Reads Return a copy of the last attempted write by the guest to this register.

  Writes Update the local copy and analyze the written value to determine which cores are targeted by the SGI. Translate the request so that only the refined IDs of cores belonging to the guest are targeted. The SGI ID is unmodified.

- *GICD_CPENDSGIRn*/*GICD_SPENDSGIRn* – contain bits to set and clear the pending status of SGIs for each core.

  Reads Perform the read to the GIC and return a filtered and ID-translated version to the guest, marking all SGIs coming from cores of other guests as not pending. Such SGIs are used to implement the IGC notification mechanism and should not be visible to the guests.

  Writes Forward a modified write to the GIC, where updates to SGIs from cores not belonging to the guests are ignored and core identifiers are translated from the refined to the ideal model.

- *ICPIDR2* – an identification register containing the implemented GIC architecture version. This register is read-only.

  Reads Always return a value that states that the GIC implements GICv2.

  Writes Ignored

Let *GICD*, as introduced earlier, be the set of these register names. The distributor registers are allocated in the pages with addresses A_{GICD}. For simplicity we assume that all distributor registers fit in one page, i.e., $A_{GICD} = \{a_{GICD}\}$, then they can be identified by the twelve-bit offset into that page of a given address addressing the GIC distributor. In the ideal model the associated intermediate physical page address $a^g_{GICD} \in A_P^{g,np_g}$ is mapped to a_{GICD} by the second-stage address translation.

We introduce a function $decgicd : \mathbb{B}^{12} \to GICD \cup \{\bot\}$ to decode the addresses GIC distributor registers from an address offset, returning $\bot$ if the access cannot be mapped to a register. Note that GIC distributor registers are either byte- or word-addressable.

The local copies of the distributor registers for guest g mentioned above are stored in a page with physical page address $a^g_{gcpy} \in A_{HV}$. We define an abstraction function $gcpy_g : \mathbb{B}^{4096 \cdot 8} \times GICD \to \mathbb{B}^8 \cup \mathbb{B}^{32}$ that extracts the requested register from such a memory page. Initially this copy reflects the turned-off state of the GIC for all guests. After that it contains the value that the ideal GIC register has according to the last (modified) write to the GIC by a core of the corresponding guest. Observe that not all registers of the ideal GIC are represented in this local copy, e.g., the registers viewing the interrupt state cannot be mirrored in this way.

Accesses to this data structure and the GIC distributor need to be synchronized. To this end we introduce lock $Glock : \mathbb{B}^{4096 \cdot 8} \to \mathbb{N}_{nc} \cup \{\bot\}$, containing the index of the core that acquired it, or $\bot$ in case it is free. The corresponding lock variable is stored in a hypervisor page with address $a_{glk} \in A_{HV}$. Note that an optimized implementation could provide separate locks for each copy data structure and the GIC, however we use a global lock here for simplicity.

As explained above, write accesses to the GIC from a guest g may not be forwarded without being processed by the hypervisor in order to maintain isolation between guests also in the GIC configuration. Intuitively, any write request for register r that aims to update it with value v must be converted to a request such that the resulting value in the GICD register r is $\eta^r_g(v)$, where η^r_g is a function that filters out all settings and information for interrupts that do not belong to guest g and replaces them with default values.

As an exception, if $r = GICD\_CTLR$, any desired value v is transformed into $\eta^r_g(v) = v$ but that value is only stored in the local copy, reflecting the state of ideal GIC. The actual write to the GIC needs to be modified so that it always stays enabled. Note, that we cannot simply replace values v in the write request with $\eta^r_g(v)$, because GIC registers are not updated like ordinary memory, instead the written value is interpreted by an update function ν to determine the resulting value of the register.

To capture the necessary modifications to requests sent to the GIC as described above, we introduce a conversion function ρ_g for each guest, that takes an ideal model GIC request or reply $ir \in \mathbb{R} \cup \mathbb{Q}$ and manipulates it according to the register it is meant for and the guest's access rights. We omit a formal definition here; informally we demand that for write requests or reply ir_w, the conversion $\rho_g(ir_w)$ changes register address and the written value in such a way that the GIC registers contain $\eta_g^r(v)$ for the value v that is stored in the local copy of the register.

In addition, for requests and replies ir_e that are emulated by the hypervisor entirely, $\rho_g(ir_e)$ returns None. Ideal read requests ir_r that are actually sent to the GIC are not changed at all, i.e., $\rho_g(ir_r) = ir_r$.

Lastly, read replies $q \in \mathbb{Q}$ to the hypervisor from the GIC distributor in the refined model are mapped to the ideal model read replies by $\rho_g(q)$, where all information about interrupts not belonging to guest g is filtered out and replaced with default values. Intuitively, the filtering of read replies replaces any returned read value v with $\eta_g^r(v)$.

Note that ρ_g affects read or written values as well as parts of the address (potentially addressing a different register of the GIC distributor, e.g., to simulate disabling a guests distributor), but they never change the page addresses of requests and replies. Such a translation from the intermediate to the physical address has to be performed explicitly by the hypervisor. Additionally, the conversion function only has an effect on requests or replies targetting page a^g_{GICD}, i.e., messages that target other page addresses are not modified.

Then, intuitively, when a guest sends a write requests $w \in \mathbb{R}$ to its GIC distributor, the hypervisor will instead send the address-translated $\rho_g(w)$) (or no request at all in cases of emulation), a write reply $w \in \mathbb{Q}$ will appear in the refined model in the same way. Read requests $r \in \mathbb{R}$ sent by the guest to the GIC distributor will appear in the refined model as $\rho_g(r)$, i.e., either identically or not at all (in case of emulation). On the other hand, refined read replies $r \in \mathbb{Q}$ from the GIC distributor to the hypervisor will be forwarded to the guest in the ideal model as $\rho_g(r)$ (with physical addresses reverted to intermediate physical ones).

Now we introduce the set of entry states for the GIC distributor virtualization handler as follows. A hypervisor state hv from a computation on a core mapped to guest g is in set $\sigma_{\text{GICD}}^{entry,g,r}$ with $r \in \mathbb{R}$ if and only if:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = hv.C.spr(VBAR\_EL2) +_{64} 0^{53}10^{10}$ – The program counter is pointing to the exception vector for synchronous exceptions from a lower exception level executing in AArch64 mode (exception vector base address + offset 0x400).
- $hv.C.spr(ESR\_EL2)[31:26] = 100100$ – The exception syndrome register holds exception code 0x24, i.e., a Data Abort from a lower exception level.
- $hv.C.spr(ESR\_EL2)[24] = 1$ – The instruction specific syndrome field

holds valid information. This field $ISS = hv.C.spr(ESR\_EL2)[24:0]$ needs to be set up as follows:

  - $ISS[22] = 0$ – The aborted access was either byte-sized or word-sized.
  - $ISS[21] = 0$ – An aborted load does not require sign extension. We require this to simplify the exposition, always returning zero-extended load results to the guest.
  - $ISS[8] = 0$ – The abort was not due to a cache management or address translation instruction.
  - $ISS[7] = 0$ – The abort was not caused by a first-stage table lookup.
  - $ISS[5:2] = 0001$ – The attempted access caused a second-stage Translation Fault at any level of translation.

- $hv.C.spr(HPFAR\_EL2)[39:4] = a^g_{GICD}$ – The Hypervisor IPA Fault Address Register contains the intermediate physical page address of guest g's GIC distributor. The offset into that page must be obtained from the faulting virtual address stored in register *FAR_EL2*.

- $hv.C.u2c = \text{None}$ – There are no outstanding replies from the GIC.

- r denotes the request of guest g that was intercepted by the hypervisor. There are two cases distinguished by bit 6 of the syndrome field.

  1. $ISS[6] = 0$ – the intercepted request was a read of the GIC distributor. The requested access size is either a byte, determined by $B = \neg hv.C.spr(ESR\_EL2)[23] = 1$, or a word, determined by $B = 0$. The request is accessing the ideal GIC address region of guest g with offset o taken from the *FAR_EL2* register, i.e., $o = hv.C.spr(FAR\_EL2)[11:0]$.

$$r = \mathrm{R}\ (a^g_{GICD} \circ o)\ (4 - 3B)$$

  2. $ISS[6] = 1$ == the intercepted request was a write to the GIC distributor. The written value is stored in the GPR identified by bits 20-16 of the syndrome field.

$$r = \mathrm{W}\ (a^g_{GICD} \circ o)\ (4 - 3B)\ hv.C.gpr(ISS[20:16])[31 - 24B : 0]$$

Note that reads and writes that perform an address write-back are not covered by this definition. For such memory accesses $ISS[24]$ is 0 and virtualization is hard as we are lacking sufficient information on the attempted access. For simplicity we do not allow the guest to access the GIC distributor register with such accesses and inject a data abort in case it tries anyway.

The possible transitions of the GICD virtualization handler are shown in Fig. 7. Either the guest access can be emulated by the hypervisor or the modified request has to be sent to the GIC. Read requests forwarded back to the guest are subject to filtering by the ρ_g function. Below we define predicates to distinguish

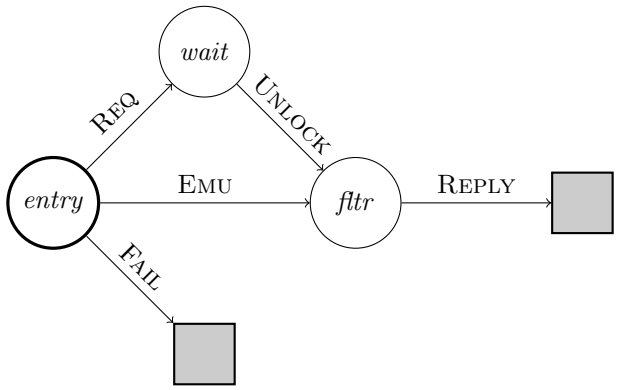

Figure 7: Transition diagram for the GIC distributor virtualization handler.

the different cases which are based on register name $r \in GICD$, register offsets $o \in \mathbb{B}^{12}$, a write flag $w \in \mathbb{B}$ and a size encoding $B \in \mathbb{B}$ denoting a byte access.

$$\begin{aligned} failgicd(r, B) \quad &\equiv \quad r \in \{GICD\_IGROUPn, GICD\_NSACRn\} \vee \\ &\qquad B \wedge r \notin \{\ GICD\_IPRIORITYRn, GICD\_ITARGETSRn, \\ &\qquad\qquad GICD\_CPENDSGIRn, GICD\_SPENDSGIRn \quad \} \\ failgicd(o, B) \quad &\equiv \quad decgicd(o) = \bot \vee failgicd(decgicd(o), B) \vee \neg B \wedge o[1:0] \neq 00 \end{aligned}$$

A request fails if it targets secure registers or the target cannot be determined, if it is a byte access to a word-adressable GIC distributor register or if a word access is not aligned. In these cases in our models the GIC sends back a Data Abort and we simulate this behaviour in the hypervisor as well. We simplify here a bit, as in the real hardware possibly different exceptions may be generated, or the GIC might just ignore certain invalid requests.

$$\begin{aligned} reqgicd(r, w) \quad &\equiv \quad r \in \{\ GICD\_ISPENDRn, GICD\_ICPENDRn, \\ &\qquad\qquad GICD\_ISACTIVERn, GICD\_ICACTIVERn, \\ &\qquad\qquad GICD\_CPENDSGIRn, GICD\_SPENDSGIRn \quad \} \vee \\ &\qquad w \wedge r \in \{\ GICD\_CTLR, GICD\_ISENABLERn, \\ &\qquad\qquad GICD\_ICENABLERn, GICD\_ITARGETSRn, \\ &\qquad\qquad GICD\_ICFGRn, GICD\_SGIR \quad \} \\ reqgicd(o, w) \quad &\equiv \quad reqgicd(decgicd(o), w) \end{aligned}$$

For accesses as described above the hypervisor needs to send a translated request to the hypervisor. Otherwise the guest access is emulated. In both cases the hypervisor needs to update its local copies of the accessed registers, except for those that are read-only or depending on interrupt signals, as denoted by predicate $updgicd : \mathbb{B}^{12} \to \mathbb{B}$.

$$emugicd(o, w) \quad \equiv \quad \neg reqgicd(o, w)$$

$$
\begin{aligned}
\mathit{updgicd}(r) \quad &\equiv \quad r \notin \{\ \mathit{GICD\_TYPERn}, \mathit{GICD\_IIDRn}, \mathit{IPCIDR2}, \\
&\qquad\quad \mathit{GICD\_ISPENDRn}, \mathit{GICD\_ICPENDRn}, \\
&\qquad\quad \mathit{GICD\_ISACTIVERn}, \mathit{GICD\_ICACTIVERn}, \\
&\qquad\quad \mathit{GICD\_CPENDSGIRn}, \mathit{GICD\_SPENDSGIRn}\ \} \\
\mathit{updgicd}(o) \quad &\equiv \quad \mathit{updgicd}(\mathit{decgicd}(o))
\end{aligned}
$$

Now we can define the three outgoing transitions from Fig.7 formally. For Emu and Req we distinguish the read and write case. Note that the register offsets within the GIC distributor page are not stored in the *HPFAR_EL2* register but taken from the virtual address used by the guest which is stored in *FAR_EL2* at exception entry. Moreover bit 23 of the syndrome field determines whether the aborted memory access was byte- or word-sized.

$$
\frac{\begin{array}{c}
g = \gamma(c) \qquad \mathit{hv} \in \sigma_{\mathrm{GICD}}^{\mathit{entry},g,r} \\
o = \mathit{hv}.C.\mathit{spr}(\mathit{FAR\_EL2})[11:0] \qquad B = \neg \mathit{hv}.C.\mathit{spr}(\mathit{ESR\_EL2})[23] \\
\mathit{failgicd}(o,B) \qquad \mathit{hv}'.C.\mathit{spr}(\mathit{SPSR\_EL1}) = \mathit{spsr2} = \mathit{hv}.C.\mathit{spr}(\mathit{SPSR\_EL2}) \\
\mathit{hv}'.C.\mathit{spr}(\mathit{ELR\_EL1}) = \mathit{hv}.C.\mathit{spr}(\mathit{ELR\_EL2}) \\
\mathit{hv}'.C.\mathit{spr}(\mathit{FAR\_EL1}) = \mathit{hv}.C.\mathit{spr}(\mathit{FAR\_EL2}) \\
\mathit{hv}'.C.\mathit{spr}(\mathit{ESR\_EL1}) = 10010 \circ \mathit{spsr2}[2] \circ 1 \circ 0^{25} \\
\mathit{hv}'.C.\mathit{pc} = \mathit{hv}.C.\mathit{spr}(\mathit{VBAR\_EL1}) +_{64} 0^{52} \circ \mathit{ghoff}(\mathit{spsr2}) \\
\mathit{hv}'.C.\mathit{ps} = \mathit{hv}'.C.\mathit{ps}[\mathit{mode} \mapsto 1]
\end{array}}{\mathit{hv} \rightsquigarrow_c \mathit{hv}'} \text{ GICD-Fail}
$$

The Data Abort is injected into the guest by simulating the effect of an exception entry into EL1. For setting the PC and the exception cause in the syndrome register we need to distinguish whether the GIC access was attempted at user or kernel level, using function *ghoff* defined below. Bit 2 in the saved processor state determines in our setting if the guest was executing in EL1 or EL0. In the first case exception code 100101 is injected into the guest, signalling a Data Abort from the same level of exception. Otherwise, exception code 100100 signals a Data Abort from a lower exception level to the guest handler running in EL1. The lower bits of the exception syndrome registers are always initialized identically upon an exception into EL1, independent of its cause.

To determine the position of the exception handler, i.e., the offset of its exception vector relative to the guests Vector Base Address, we introduce function $\mathit{ghoff} : \mathbb{B}^{64} \to \mathbb{B}^{12}$ which decodes the required offset from the saved processor state, namely 0x000 for exceptions from the same exception level where *SP_EL0* is used as stack pointer (determined by processor state bit 0), 0x200 for exceptions from the same exception level where *SP_EL1* is used as stack pointer, and 0x400 for exceptions from a lower exception level.

$$
\mathit{ghoff}(d) = \begin{cases}
0010 \circ 0^8 & : \quad d[2] \wedge d[0] \\
0000 \circ 0^8 & : \quad d[2] \wedge \neg d[0] \\
0100 \circ 0^8 & : \quad \neg d[2]
\end{cases}
$$

For guest accesses that are in a valid form their emulation is defined as follows. Writes and reads are distinguished by bit 6 in the syndrome fieldand

the target register is encoded in bits 20-16. The action is only enabled if the GICD lock is not taken. Otherwise any hypervisor step is stuttering until the lock is free again.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma^{entry,g,r}_{\text{GICD}} \\ o = hv.C.spr(FAR\_EL2)[11:0] \qquad B = \neg hv.C.spr(ESR\_EL2)[23] \\ esr = hv.C.spr(ESR\_EL2) \qquad emugicd(o,0) \\ \neg failgicd(o, \neg esr[23]) \qquad \neg esr[6] \qquad Glock(hv.m(a_{glk})) = \bot \\ reg = decgicd(o) \qquad q = rplR\ (a^g_{GICD} \circ o)\ gcpy_g(hv.m(a^g_{gcpy}), reg) \\ ctx(hv'.m(a^c_{ctx}), 0, c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}), 0, c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}), 0, c)).ps = PS(hv.C.spr(SPSR\_EL2)) \qquad hv' \in \sigma^{fltr,q}_{\text{GICD}} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-EMU-R}$$

For emulated reads this step has no effect other than saving the guest context and moving the hypervisor into a state in $\sigma^{fltr,q}_{\text{GICD}}$ where the read reply q will be forwarded to the guest. Note that the step may only be executed if the GIC lock is free.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma^{entry,g,r}_{\text{GICD}} \\ o = hv.C.spr(FAR\_EL2)[11:0] \qquad esr = hv.C.spr(ESR\_EL2) \\ B = \neg esr[23] \qquad emugicd(o,1) \qquad \neg failgicd(o, \neg esr[23]) \\ esr[6] \qquad Glock(hv.m(a_{glk})) = \bot \qquad R = decgicd(o) \\ v = gcpy_g(hv.m(a^g_{gcpy}), R) \qquad u = hv.C.gpr(esr[20:16])[31-24B:0] \\ v' = \nu(R, v, u) \qquad updgicd(o) \Rightarrow gcpy_g(hv'.m(a^g_{gcpy}), R) = \eta^R_g(v') \\ q = \text{W}\ (a^g_{GICD} \circ o)\ (4-3B)\ u \qquad ctx(hv'.m(a^c_{ctx}), 0, c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}), 0, c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}), 0, c)).ps = PS(hv.C.spr(SPSR\_EL2)) \qquad hv' \in \sigma^{fltr,q}_{\text{GICD}} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-EMU-W}$$

The emulated write case is similar to the read case, just that we have a conditional update of the local copy of the targeted register R in case it is writable. The update of the local copy of that register is computed according to the GIC update function ν which takes a GIC register name, its old value, and the written value to produce the new value for the register which is filtered by function η^R_g for the specific register. These values are either bytes or words in size, depending on bit B extracted from the exception status register. The hypervisor state is again in $\sigma^{fltr,q}_{\text{GICD}}$ where the guest will be restored.

If a request needs to be sent to the GIC distributor, we take the REQ transition as shown in Fig. 7. Again we distinguish the read and the write cases. To avoid concurrent accesses to the local copy and GICD registers of guest g, the

GICD lock must be acquired.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma^{entry,g,r}_{\text{GICD}} \qquad o = hv.C.spr(FAR\_EL2)[11:0] \\ esr = hv.C.spr(ESR\_EL2) \qquad B = \neg esr[23] \\ reqgicd(o,0) \qquad \neg failgicd(o,B) \qquad \neg esr[6] \qquad Glock(hv.m(a_{glk})) = \bot \\ Glock(hv'.m(a_{glk})) = c \qquad ctx(hv'.m(a^c_{ctx}),0,c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}),0,c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}),0,c)).ps = PS(hv.C.spr(SPSR\_EL2)) \\ hv'.mmu.active = 0 \\ hv'.C.pc = a_{gdw} \qquad hv'.c2u = \text{R } (a_{GICD} \circ o)\ (4-3B) \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-Req-R}$$

For a read we simply issue a read request for the targeted GICD register, adjusting the address to the physical address of the distributor. The size of the access is again determined by bit B. We also need to disable the second-stage MMU as the hypervisor only uses one stage of translation. Moreover we save the guest context in hypervisor memory while waiting for the reply from the GIC. This intermediate hypervisor state is signified by the program counter pointing to address a_{gdw}.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma^{entry,g,r}_{\text{GICD}} \\ o = hv.C.spr(FAR\_EL2)[11:0] \qquad esr = hv.C.spr(ESR\_EL2) \\ B = \neg esr[23] \qquad reqgicd(o,1) \qquad \neg failgicd(o,B) \qquad esr[6] \\ Glock(hv.m(a_{glk})) = \bot \qquad R = decgicd(o) \qquad v = gcpy_g(hv.m(a^g_{gcpy}),R) \\ v' = \nu(R,v,hv.C.gpr(esr[20:16])[31-24B:0]) \\ updgicd(o) \Rightarrow gcpy_g(hv'.m(a^g_{gcpy}),R) = \eta^R_g(v') \\ Glock(hv'.m(a_{glk})) = c \qquad ctx(hv'.m(a^c_{ctx}),0,c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}),0,c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}),0,c)).ps = PS(hv.C.spr(SPSR\_EL2)) \\ hv'.mmu.active = 0 \qquad hv'.C.pc = a_{gdw} \\ hv'.c2u = \rho_g(\text{W } (a_{GICD} \circ o)\ (4-3B)\ hv.C.gpr(esr[20:16])[31-24B:0]) \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-Req-W}$$

The rule for sending write requests to the GIC distributor is similar to the read rule. The guest context is saved and we issue a write taking the byte or word part of the data stored in the guest's target register. The write is manipulated to maintain the GICD virtualization using the ρ_g emulation function for guest g. We update the local copy of the GIC distributor registers as necessary. Again the program counter is set to address a_{gdw}, marking the state of the hypervisor waiting for a reply from the GIC.

This state, or rather set of states, is identified by $\sigma^{wait}_{\text{GICD}}$ and for every contained hypervisor state hv we have:

- $hv.C.pc = a_{gdw}$ – the PC is still at the same position after sending the GIC distributor access request.
- $\neg hv.C.spr(ESR\_EL2)[6] \Rightarrow hv.u2c \in \{rplR\ a\ v \mid a[47:12] = a_{GICD}\}$ – there is an outstanding read reply from the GIC distributor when we have

requested a read before. We do not explicitly check that the address and size matches the previous request, assuming that the GIC does not send spurious read replies, i.e. it only sends replies for addresses that have been requested.

- $hv.C.spr(ESR\_EL2)[6] \Rightarrow hv.u2c \in \{rplW\ a \mid a[47:12] = a_{GICD}\}$ – there is an outstanding read reply from the GIC distributor when we have requested a write before. Again we do not need to match the address, size, and written value explicitly.

Then the reception of the GIC reply triggers step Unlock which is defined separately for reads and writes as follows.

$$\frac{\begin{array}{ccc} g = \gamma(c) & hv \in \sigma^{ret}_{\text{GICD}} & esr = hv.C.spr(ESR\_EL2) \\ B = \neg esr[23] & \neg esr[6] & q = \rho_g(hv.u2c) = rplR\ a\ v \\ R = decgicd(a[11:0]) & & gcpy_g(hv'.m(a^g_{gcpy}), R) = v \\ Glock(hv'.m(a_{glk})) = \bot & hv' \in \sigma^{fltr,q}_{\text{GICD}} & hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-Unlock-R}$$

A read reply is filtered by the ρ_g emulation function and the new value is stored in the local copy of the GIC register to be returned to the guest in the next step. We release the lock on guest g's local copy and GIC registers. The hypervisor state is in the set $\sigma^{fltr,q}_{\text{GICD}}$ where the PC is pointing to a_{gdfltr}. From there the handler returns to the guest.

$$\frac{\begin{array}{ccccc} & & hv \in \sigma^{ret}_{\text{GICD}} & & \\ Glock(hv'.m(a_{glk})) = \bot & q = hv.u2c & hv' \in \sigma^{fltr,q}_{\text{GICD}} & hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{GICD-Unlock-W}$$

For the write case we simply release the lock on GIC and its local copies and set the PC to a_{gdfltr} as well. The corresponding hypervisor states $hv \in \sigma^{fltr,q}_{\text{GICD}}$ that are reached both through Req and Emu are identified as follows.

- $hv.C.pc = a_{gdfltr}$ – the PC is at the position after executing the emulation step or receiving the reply from the GIC.
- $hv.u2c = \text{None}$ – no further reply from the GIC is pending.
- q is a history variable that records the converted GIC reply for reference in the bisimulation relation.

From these states transition Reply returns to the guest and injects the converted GIC reply into the guest in case of reads. We again distinguish the read

and write case.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma^{fltr,q}_{\text{GICD}} \\ o = hv.C.spr(FAR\_EL2)[11:0] \qquad esr = hv.C.spr(ESR\_EL2) \\ \neg esr[6] \qquad q = rplR\ a\ v \qquad hv'.C.gpr(esr[20:16]) = zxt_{64}(v) \\ \forall R \neq esr[20:16].\ hv'.C.gpr(R) = ctx(hv.m(a^c_{ctx}), 0, c)).gpr \\ hv'.C.pc = hv.C.spr(ELR\_EL2) +_{64} 4_{64} \\ hv'.C.ps = PS(hv.C.spr(SPSR\_EL2)) hv'.mmu.active = 1 \qquad hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'} \text{ GICD-Reply-R}$$

For read replies the reply value is taken from (possibly filtered) reply and stored in the lower 8 or 32 bits of the guest's target register, depending on whether the read was byte- or word-sized. The upper bits are set to zero by function zxt_n which performs a zero extension of shorter bit strings to n bits. The remaining guest state is restored from the saved guest context, setting the PC to the instruction following the GICD memory access. The mode is again EL1 or EL0 and the second-stage MMU is re-enabled, finishing the handler execution.

$$\frac{\begin{array}{c} hv \in \sigma^{fltr,q}_{\text{GICD}} \\ hv.C.spr(ESR\_EL2)[6] \qquad hv'.C.gpr = ctx(hv.m(a^c_{ctx}), 0, c)).gpr \\ hv'.C.pc = ctx(hv.m(a^c_{ctx}), 0, c)).pc +_{64} 4_{64} \\ hv'.C.ps = ctx(hv.m(a^c_{ctx}), 0, c)).ps \\ hv'.mmu.active = 1 \qquad hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'} \text{ GICD-Reply-W}$$

For writes, nothing has to be done save restoring the guest and turning the MMU back on. This finishes the definition of the hypervisor transition system for the virtualization of the GIC distributor. Note, that in both the ideal and refined model accesses to the GIC from different cores are serialized, i.e., there are no race conditions when accessing the GIC. This is sound as the hypervisor implementation acquires the GICD lock before accessing the GIC distributor registers for a given guest. For GIC accesses in different guest no synchronization is required in the ideal model. However in the refined model accesses to the GIC distributor need to acquire the lock first. Accesses to any other register of the GIC need no locking as these registers are banked for each core.

In fact, even for the GIC distributor we could relax the locking policy, since most registers allow to configure each interrupt individually. As different guests own different interrupts there would be no race condition. The only GICD registers where this is not the case are the interrupt configuration registers which determine the trigger type of interrupts. Therefore it would suffice to lock the GIC for accesses to these registers, however as we cannot get rid of locking completely, we omit this optimization here for simplicity.

3.4.6 Peripheral IRQ and SGI Virtualization

During guest execution, all asynchronous exceptions, i.e., peripheral interrupts and inter-processor interrupts, are routed to the hypervisor running in EL2 due

to the setting of the interrupt trap bit (IMO, bit 4) in the hypervisor control register (*HCR_EL2*). While the hypervisor is running, however, all interrupts are masked. Thus only guests are interrupted by asynchronous exceptions. Upon such an exception the hypervisor is in a state identified by the set $\sigma^{entry}_{\text{IRQ}}$. For each contained hypervisor state hv the following conditions hold.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = hv.C.spr(VBAR\_EL2) +_{64} 0^{53}10010^7$ – The program counter is pointing to the exception vector for asynchronous exceptions from a lower exception level executing in AArch64 mode (exception vector base address + offset 0x480).
- $hv.C.spr(ISR\_EL1)[7] = 1$ – The interupts status register shows a pending IRQ interrupt. On the HASPOC platform we only expose IRQs to the guest, i.e., no FIQs or SErrors are forwarded to it.

Note that at the entry point we cannot distinguish inter-processor interrupts sent by another core of the guest from inter-processor interrupts used to implement IGC notifications. Both handlers share the initial state and the first transition. For the virtualization, a number of registers of the GIC CPU interface and the virtual interrupt control interface need to be accessed. We give a short description of the relevant functionality below.

- *GICC_AIAR* – the (aliased) interrupt acknowledge register contains the ID of the last non-secure interrupt delivered to the core at bits 9-0. For SGIs bits 12-10 also contain the ID of the sender core. This sender ID allows us to distinguish intra-guest SGIs from IGC notifications and to share SGI IDs for guest and hypervisor use. When this register is read the interrupt state change from pending to active or active-and-pending. The latter case occurs when a peripheral interrupt is still asserted at its source, which is usually true for level-triggered interrupts. For SGIs a read of the register always changes the interrupt state to active, it is changed to active-and-pending only when a second SGI with the same ID and from the same sender is asserted. Note that we only need to treat non-secure interrupts here, as guests have no access to secure interrupts.
- *GICH_LRn* – the 64 list registers of the virtual interrupt control interface allow the hypervisor to set up virtual interrupts for the guest, allowing to configure a virtual interrupt ID (bits 9-0) and priority for the interrupt to the guest. For SGIs also the sender core ID can be virtualized. Bit 31 of each list register distinguishes between physical and software generated interrupts. In the first case the virtual interrupt is tied to a physical one (bits 19-10) and all actions the guest performs on its virtual interface to handle the interrupt (in particular, deactivating it) are mapped accordingly to the physical interface by the GIC. Unfortunately this behaviour is not supported for SGIs, thus bit 31 needs to be set to zero when forwarding intra-guest SGIs to the guest. The GIC constantly updates the

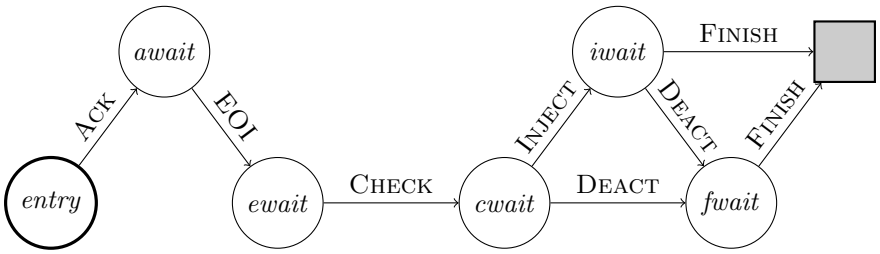

Figure 8: Transition diagram for the physical and software interrupt virtualization handlers.

state of the virtual interrupts (bits 29-28) according to the actions of the guest handling it.

- *GICC_CTLR* – the CPU interface control register contains several flags to control the behaviour of the physical CPU interface. One flag, *EOIModeNS* controls the way interrupts are deactivated. Usually a write of an interrupt ID to the *GICC_EOIR* register signals the end of handling for that interrupt and deactivates it. However if the flag is set a write to the end-of-interrupt register only lowers the running interrupt priority, avoiding the core to be interrupted again while the interrupt signal stays active. This is useful for interrupt virtualization, as after the hypervisor execution the physical interrupt will stay active until the guest deactivates the corresponding virtual interrupt. Note that this behaviour is not supported for SGIs, here the hypervisor needs to explicitly deactivate the physical SGI by writing the *GICC_DIR* register. For each CPU interface we require that $GICC\_CTLR.EOIModeNS = 1$ always holds after initialization.

- *GICC_EOIR* – the end-of-interrupt register is written in order to signal the end of interrupt handling for a certain interrupt (bits 9-0 for the interrupt ID and 12-10 for the sender ID in case of SGIs). Note that due to the setting of the *GICC_CTLR.EOIModeNS* bit, interrupts stay active even after the hypervisor writes this register and only a priority drop is performed for the corresponding interrupt.

- *GICC_DIR* – the deactivate interrupt register is used to deactivate interrupts in case $GICC\_CTLR.EOIModeNS = 1$. It has the same layout as *GICC_EOIR*.

To virtualize any physical interrupt that has been trapped to it, the hypervisor takes a number of steps that are also depicted in Fig. 8.

- Ack – the hypervisor sends an acknowledgement of the interrupt to the GIC by sending a read request for the interrupt acknowledgment register of the core's interrupt interface (*GICC_AIAR*).

- EOI – in the answer from the GIC the hypervisor learns the ID of the interrupt that triggered the asynchronous exception and saves its ID locally. Before the list register is updated, a priority drop is requested from the GIC by a write to the end-of-interrupt register. However the physical interrupt is still active, it just will not interrupt the hypervisor again before it is being handled by the guest. In case of SGIs we also must take a DEACT step later to deactivate the physical SGI. As explained above this deactivation cannot be done automatically when the guest requests it, thus we deactivate the SGI prematurely, allowing for potential new SGIs from the same sender to invoke the hypervisor.

- CHECK – the hypervisor checks if the last received interrupt is already pending for the guest in the list registers ($GICH\_LRn$) of the virtual interrupt control interface, sending another read request to the GIC. We assume here for simplicity that each guest is configured to receive at most 64 interrupts. Then we can associate each interrupt with a specific list register and a single 32-bit read to the dedicated register can retrieve the required information. Note that when reserving the list registers for SGIs, we need to reserve one register for each desired sender and SGI ID combination, these are at most 64 interrupts in each core interface.

- DEACT – If the interrupt cannot be injected, or in case of SGIs, the interrupt is deactivated by a write to the $GICC\_DIR$ register.

- INJECT – if the interrupt is was not already pending (it may be active though) the interrupt is added to the list of virtual interrupts for the guest by a write to the corresponding list register in the virtual interrupt control block of the GIC. If the interrupt was active it becomes active-and-pending, i.e., we must update the interrupt's status bits in the list register. For peripheral interrupts we mark the virtual interrupt as an hardware interrupt (bit 31), so that the physical interrupt can be handled by the guest through its virtual interface without involvement of the hypervisor. For inter-processor interrupts within the guest we also adjust the sender ID of the corresponding SGI to map the refined core indexes to the ones in the ideal model.

- FINISH – after the list register (or potentially $GICC\_DIR$) has been written, we resume guest execution. The virtual interrupt interface will eventually deliver the interrupt into the guest.

In what follows we define these transitions formally and introduce the different sets of identified intermediate hypervisor states. We start with the interrupt

acknowledgement step.

$$\frac{\begin{array}{c} hv \in \sigma^{entry}_{\text{IRQ}} \qquad ctx(hv'.m(a^c_{ctx}), 0, c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}), 0, c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}), 0, c)).ps = PS(hv.C.spr(SPSR\_EL2)) \\ hv'.C.pc = a_{irqaw} \qquad hv'.mmu.active = 0 \qquad hv'.c2u = \text{R } a_{IAR}\ 4 \end{array}}{hv \rightsquigarrow_c hv'} \text{ IRQ-Ack}$$

The hypervisor saves the guest context, disables the second-stage MMU, and issues a 32-bit read to the $GICC\_AIAR$ register at a_{IAR} with $a_{IAR}[47:12] \in A_{GICC}$ and $a_{IAR}[1:0] = 00$. The next hypervisor transition is enabled in states $hv \in \sigma^{await,v,g}_{\text{IRQ}}$ for $v \in \mathbb{B}^{32}$ and $g \in \mathbb{N}_{ng}$, identified as follows.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqaw}$ – The program counter is pointing to the address were we stopped in the hypervisor in order to request the read to the acknowledgment register.
- $hv.C.u2c = rplR\ a_{IAR}\ v$ – The read value v is returned to the core from the GIC. For this v we have in particular:
  - $\langle v[9:0] \rangle \in [8:1023]$ – the first eight interrupts are used for secure SGIs and are not injected into EL2. Moreover the last four interrupt IDs have special meaning, e.g., interrupt 1023 denotes a spurious interrupt and thus should be dropped. We do not add an additional branch to the hypervisor transition system to handle these cases for simplicity. Instead they are handled by the Drop transition.
  - $\langle v[9:0] \rangle \in [8:15] \Rightarrow \gamma(\langle v[12:10] \rangle) = g$ – any Non-Secure SGI received (IDs 8-15) was sent by a core belonging to guest g. Otherwise the interrupt needs to be treated as an IGC notification (defined later).

Note that, in case of peripheral interrupts, the information we are receiving from the GIC may actually be outdated, as other cores of the same guest have access to the peripherals and are free to deactivate peripheral interrupts at will without even handling them through the GIC. While this should be considered a software error on behalf of the guest, the hypervisor cannot easily prohibit such behaviour and will inject peripheral interrupts it receives from the GIC as virtual interrupts into a guest's core even if the corresponding physical interrupt might no longer be signalled by the peripheral. Note that the GIC in such a scenario will signal a spurious interrupt to the guest as soon as it tries to handle such a virtual interrupt, both in the refined and the ideal model. The bisimulation proof however is complicated by this issue as we will see later.

In the next step, a write request for the end-of-interrupt register at address a_{EOIR} with $a_{EOIR} \in A_{GICC}$ is sent in order to require a priority drop for the

interrupt that was acknowledged in the previous transition. We also save this interrupt in the core's hypervisor variable *lirq*.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma_{\text{IRQ}}^{await,v,g} \qquad id = \langle v[9:0] \rangle \qquad s = \langle v[12:10] \rangle \\ q = id \in [8:15]\ ?\ \text{sgi}_{s,c}^{id} : id \qquad lirq(hv'.m(a_{lirq}), c) = q \qquad hv'.C.pc = a_{irqew} \\ hv'.u2c = \text{None} \qquad v = (q = \text{sgi}_{s,c}^{id})\ ?\ 0^{19} \circ \text{bin}_3(s) \circ \text{bin}_{10}(id) : 0^{22} \circ \text{bin}_{10}(q) \\ hv'.c2u = \text{W}\ a_{EOIR}\ 4\ v \end{array}}{hv \rightsquigarrow_c hv'} \text{IRQ-EOI/DROP}$$

We decode the received interrupt from the value returned from the *GICC_AIAR* register, distinguishing the special case of SGIs where also the sender ID needs to be saved. The value to be written to the end-of-interrupt register is the 10-bit interrupt ID together with the 3-bit sender ID in case of an inter-processor interrupt within guest g. After the transition the program counter points to a_{irqew} and the hypervisor is waiting for a reply from the GIC, confirming the end-of-interrupt request. Such states $hv \in \sigma_{\text{IRQ}}^{ewait}$ are defined by the following conditions:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqew}$ – The program counter is pointing to the address were we halted in the hypervisor in order to request the write of the end-of-interrupt register.
- $\exists v \in \mathbb{B}^{32}.\ hv.C.u2c = rplW\ a_{EOIR}$ – the input message box to the core contains the confirmation of the write to *GICC_EOIR*.

In the CHECK transition out of the states defined above the hypervisor needs to read the List register corresponding to the interrupt ID (and potentially an SGI sender). We use function $a_{LR}^{g} : PIRQ \cup IPIRQ \rightarrow \mathbb{B}^{48}$ to denote this address for guest g, where $a_{LR}^{g}(q)[47:12] \in A_{GICH}$ and $a_{LR}^{g}(q)[2:0] = 000$.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma_{\text{IRQ}}^{ewait} \qquad q = lirq(hv.m(a_{lirq}), c) \\ hv'.C.pc = a_{irqcw} \qquad hv'.u2c = \text{None} \qquad hv'.c2u = \text{R}\ a_{LR}^{g}(q)\ 4 \end{array}}{hv \rightsquigarrow_c hv'} \text{IRQ-CHECK}$$

The GIC reply is consumed from the input message box and a new read request is sent. The program counter is pointing to address a_{irqcw} within the hypervisor. Then the reply from the GIC may be processed in states $hv \in \sigma_{\text{IRQ}}^{cwait,c,l}$ on core c, for $l \in \mathbb{B}^{32}$, identified as follows.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqcw}$ – The program counter is pointing to the address were we stopped in the hypervisor in order to request the read to the list register.
- $hv.C.u2c = rplR\ a_{LR}^{\gamma(c)}(lirq(hv.m(a_{lirq}), c))\ l$ – The value l of the requested list register is returned to the core from the GIC.

Now there are two possible transitions starting in these state. If the interrupt has not ID 1020-1023 and it can be injected, i.e., it is not currently pending or active-and-pending (bit 28 equals 1), then the list register(s) are updated. The following predicate denotes this case for guest g.

$$inj_g(l,q) \equiv q \in PIRQ_g \setminus [1020:1023] \cup IPIRQ \wedge \neg l[28]$$

In particular, the request to be sent is determined by bit 29 of the list register, denoting an active interrupt. Function $injlr : \mathbb{B}^{48} \times \mathbb{B}^{32} \times PIRQ \cup IPIRQ \to \mathbb{R}$ computes this write request to a given address depending on the read list register value and the interrupt ID. It also uses projection function κ to map refined core indices to ideal ones and $prio_q \in \mathbb{B}^5$ for a 5-bit encoding of the fixed priority assigned to interrupt q on the HASPOC platform.

$$injlr(a,l,q) = \begin{cases} \text{W } a \text{ 4 } 0101 \circ prio_q \circ 0^{10} \circ \text{bin}_3(\kappa(s)) \circ \text{bin}_{10}(id) & : \neg l[29] \wedge q = \text{sgi}_{s,c}^{id} \\ \text{W } a \text{ 4 } 1101 \circ prio_q \circ 0^3 \circ \text{bin}_{10}(id) \circ \text{bin}_{10}(id) & : \neg l[29] \wedge q \in PIRQ \\ \text{W } a \text{ 4 } l[31:30] \circ 11 \circ l[27:0] & : l[29] \end{cases}$$

Basically, if bit 29 is zero, it means the interrupt is currently inactive and we need to generate it newly. Bit 31 denotes a hardware interrupt and is thus zero for SGIs. The next bit determines the group of the interrupt and is always one to denote Group 1. The next two bits denote the interrupt state (00 for inactive, 01 for pending, 10 for active, and 11 for active-and-pending), followed by the fixed interrupt priority. For SGIs we zero all the following configuration bits except the ones indicating the sender core ID (bits 12:10). For hardware interrupts, bits 19-10 contain the physical interrupt ID that we set equal to the virtual ID stored in the ten least significant bits for both cases.

If bit 29 is one, the same interrupt that we want to inject is currently active, i.e., we can simply copy the entry of this active interrupt and set its status to active-and-pending.

With the auxiliary functions introduced above the definition of the INJECT transition is straight-forward.

$$\frac{\begin{array}{c} g = \gamma(c) \qquad hv \in \sigma_{\text{IRQ}}^{cwait,c,l} \qquad q = lirq(hv.m(a_{lirq}), c) \qquad inj_g(l,q) \\ hv'.C.pc = a_{irqiw} \qquad hv'.u2c = \text{None} \qquad hv'.c2u = injlr(a_{LR}^g(q), l, q) \end{array}}{hv \rightsquigarrow_c hv'} \text{ IRQ-INJECT}$$

The new value for the PC in this case is a_{irqiw} and the hypervisor waits for the confirmation of the write access from the GIC. We define the corresponding set of hypervisor states after the successful interrupt injection, i.e., $hv \in \sigma_{\text{IRQ}}^{iwait}$:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqiw}$ – The program counter is pointing to the address where the hypervisor stopped, waiting for the list register write confirmation.

- $\exists v \in \mathbb{B}^{32}.\ hv.C.u2c = rplW\ a_{LR}^{\gamma(c)}(lirq(hv.m(a_{lirq})))$ – the input message box to the core contains the confirmation of the write to the list register.

In these states, the interrupt needs to be disabled by a write to the *GICC_DIR* register in case it is an inter-processor interrupt. The same has to be done for all interrupts that cannot be injected in states $\sigma_{\text{IRQ}}^{cwait,c,l}$. With a_{DIR} being the location of that register, we define the deactivation transition for both sets of states.

$$\frac{\begin{array}{c} q = lirq(hv.m(a_{lirq}), c) \\ hv \in \sigma_{\text{IRQ}}^{iwait} \wedge q \in IPIRQ \vee hv \in \sigma_{\text{IRQ}}^{cwait,c,l} \wedge \neg inj_g(l,q) \qquad hv'.C.pc = a_{irqfw} \\ hv'.u2c = \text{None} \qquad v = (q = \text{sgi}_{s,c}^{id})\ ?\ 0^{19} \circ \text{bin}_3(s) \circ \text{bin}_{10}(id) : 0^{22} \circ \text{bin}_{10}(q) \\ hv'.c2u = \text{W}\ a_{DIR}\ 4\ v \end{array}}{hv \rightsquigarrow_c hv'}\ \text{IRQ-DEACT}$$

In both cases the PC is set to address a_{irqfw}, where the hypervisor waits for the final confirmation of the GIC in order to finish the virtualization handler execution. These states $hv \in \sigma_{\text{IRQ}}^{fwait}$ are identified by:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqfw}$ – The program counter is pointing to the address were we stopped in the hypervisor in order to request the write of the end-of-interrupt register.
- $\exists v \in \mathbb{B}^{32}.\ hv.C.u2c = rplW\ a_{DIR}$ – the input message box to the core contains the confirmation of the write to *GICC_DIR*.

The finishing transition is triggered in both sets of states $\sigma_{\text{IRQ}}^{fwait}$ and $\sigma_{\text{IRQ}}^{iwait}$. In the latter ones this is the case for hardware interrupts as these do not need to be deactivated explicitly, i.e., all interrupts except SGIs.

$$\frac{\begin{array}{c} hv \in \sigma_{\text{IRQ}}^{fwait} \vee hv \in \sigma_{\text{IRQ}}^{iwait} \wedge q \in PIRQ \\ hv'.C.gpr = ctx(hv.m(a_{ctx}^{c}), 0, c)).gpr \\ hv'.C.pc = ctx(hv.m(a_{ctx}^{c}), 0, c)).pc \qquad hv'.C.ps = ctx(hv.m(a_{ctx}^{c}), 0, c)).ps \\ hv'.C.spr(HCR\_EL2)[7] = 1 \qquad hv'.mmu.active = 1 \qquad hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{IRQ-FINISH}$$

We simply restore the guest state and the second-stage MMU in the mode it was executing before the interruption. In the both cases of starting states, we also signal a virtual IRQ as pending to the guest by setting the *VI* bit (bit 7) in the hypervisor control register. Note that due to the requests sent to the GIC (in the successful case) the virtual interrupt controller state holds now the injected interrupts from where they can be delivered to the guest. However, these effects are modeled outside the hypervisor transition system.

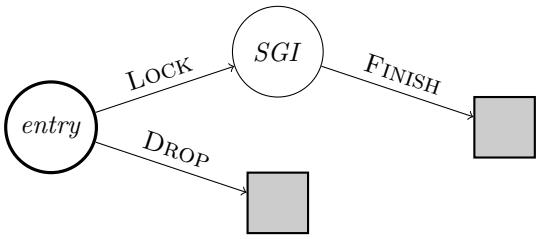

Figure 9: Transition diagram for the IGC notification request handler.

3.4.7 IGC Notification Request

Guests may also invoke the hypervisor in order to request IGC notification interrupts for a communication channel on which they have the sender role. In the current hypervisor implementation this mechanism is requested by writing to an IGC control page which is intercepted by the second-stage MMU and control is transfered into EL2 via a Data Abort. Thus the hypervisor provides a virtual device interface to the guest, quite similar to the GIC distributor virtualization. To simplify the presentation, in the models presented here, guests instead use a hypercall for IGC notification requests.

The entry states for the IGC notification request handler $hv \in \sigma_{\text{SIGC}}^{entry,v}$ for $v \in \mathbb{B}^{16}$ are then defined as follows.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = hv.C.spr(VBAR\_EL2) +_{64} 0^{53}10^{10}$ – The program counter is pointing to the exception vector for synchronous exceptions from a lower exception level executing in AArch64 mode (exception vector base address + offset 0x400). At this address the hypervisor exception handler dispatcher code is located.
- $hv.C.spr(ESR\_EL2)[31:26] = 010110$ – The exception syndrome register holds exception code 0x16, denoting an HVC exception.
- $hv.C.spr(ESR\_EL2)[15:0] = v$ – The requested hypervisor service is encoded in immediate constant v, taken from the HCV instruction that caused the exception and stored in the Exception syndrome register. We introduce function $decsigc : \mathbb{B}^{16} \to \mathbb{N}_{ns} \cup \{\perp\}$ that returns the requested IGC channel from the constant (or $\perp$ for an invalid encoding).
- $hv.C.u2c = \text{None}$ – There are no outstanding replies from the GIC.

Figure 9 shows the transition diagram for the IGC notification request handler. First there is a descision if the request can be granted. This depends on the platforms communication policy and the requested channel as well as on the power state of the receiver's cores and the state of the IGC message box. The

following predicate captures all conditions that must be met for a successful grant to guest g communicating with guest g'.

$$\begin{array}{rcll} grant_{g,g'}(hv, v) & \equiv & decsigc(v) \neq \bot \wedge cpol(decsigc(v)) = (g, g') \\ & \wedge & \exists c.\ \gamma(c) = g' \wedge pow(hv.m(a_{pow}), c) \\ & \wedge & mbox(hv, g, g') = 0 \end{array}$$

If the request is granted, the message box is updated to lock the inter-processor interrupt that is used to notify the receiving guest. We use SGI ID 15 for IGC notifications, but as explained earlier we do not reserve this ID for IGC purposes only. Then that interrupt is requested from the GIC to be sent to the core with the lowest ideal index that is also powered on. After a confirmation from the GIC is received, or if the request was not granted, the guest is restored at the instruction after the HVC. We first define the fail case as follows.

$$\frac{\begin{array}{cc} hv \in \sigma^{entry,v}_{\text{SIGC}} & \forall g'.\ \neg grant_{\gamma(c),g'}(hv, v) \\ hv'.C.pc = hv.C.spr(ELR\_EL2) & hv'.C.ps = PS(hv.C.spr(SPSR\_EL2)) \end{array}}{hv \rightsquigarrow_c hv'}\ \text{SIGC-Drop}$$

Note that for hypercalls (as for SMCs) the value stored in the exception link register (*ELR_EL2*) already points to the instruction following the HVC instruction that caused the exception. In case the request is granted, the corresponding transition is defined by:

$$\frac{\begin{array}{c} hv \in \sigma^{entry,v}_{\text{SIGC}} \qquad grant_{\gamma(c),g'}(hv, v) \qquad ctx(hv'.m(a^c_{ctx}), 0, c)).gpr = hv.C.gpr \\ ctx(hv'.m(a^c_{ctx}), 0, c)).pc = hv.C.spr(ELR\_EL2) \\ ctx(hv'.m(a^c_{ctx}), 0, c)).ps = PS(hv.C.spr(SPSR\_EL2)) \\ mbox(hv', g, g') = 1 \qquad \kappa(t) = \min\{\kappa(c') \mid \gamma(c') = g' \wedge pow(hv.m(a_{pow}), c')\} \\ hv'.c2u = \text{W}\ a_{SGIR}\ 4\ 0^8 \circ 0^{7-t}10^t \circ 0^{12}1^4 \\ hv'.mmu.active = 0 \qquad hv'.C.pc = a_{snsgi} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{SIGC-Lock}$$

The hypervisor first saves the guest context and then sets the lock bit in the message box for the channel to one. The target core t is determined using the projection function κ to retrieve the ideal core indices associated with the refined cores belonging to the receiving guest g'. The a request is sent to update GIC register *GICD_SGIR* located at address a_{SGIR} with $a_{SGIR}[47:12] \in A_{GICD}$. The 32-bit value written encodes a request to send SGI with ID 15 (bits 3-0 = 11) to a single core (bits 25-24 = 00) with index t (bits 23-16 in unary encoding). Again we need to turn of the second-stage MMU to send the GIC request. The program counter is set to a_{snsgi} where the hypervisor waits until a confirmation of the write is returned from the GIC.

The states $hv \in \sigma^{SGI}_{\text{SIGC}}$ enable the finishing step of the handler:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{snsgi}$ – The program counter is pointing to the address were the hypervisor stopped after writing to the Send SGI register.

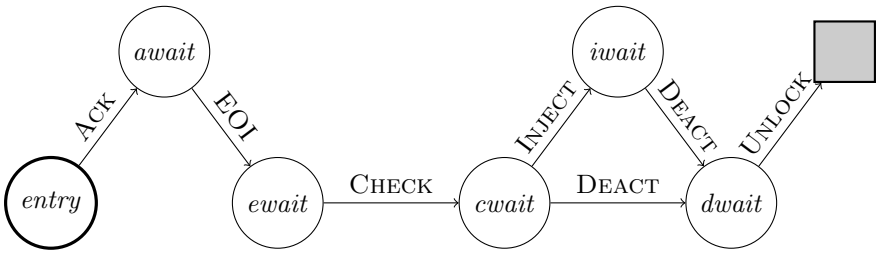

Figure 10: Transition diagram for the IGC reception handler.

- $\exists v \in \mathbb{B}^{32}$. $hv.C.u2c = rplW\ a_{SGIR}$ – the input message box to the core contains the confirmation of the write to *GICD_SGIR*.

Then we only need to restore the guest context from memory and turn back on the second-stage MMU. Note that the restored program counter already points to the instruction following the HVC that triggered this handler.

$$\frac{\begin{array}{c} hv \in \sigma^{SGI}_{\text{SIGC}} \qquad hv'.C.gpr = ctx(hv.m(a^c_{ctx}), 0, c)).gpr \\ hv'.C.pc = ctx(hv.m(a^c_{ctx}), 0, c)).pc \qquad hv'.C.ps = ctx(hv.m(a^c_{ctx}), 0, c)).ps \\ hv'.mmu.active = 1 \qquad hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'}\ \text{SIGC-Finish}$$

3.4.8 IGC Notification Reception Handler

As with all other physical interrupts, the SGI between cores of two different guests, as generated by the IGC notification request, may only interrupt guest computations and takes execution to EL2 for handling. The initial hypervisor state after the interruption is identical to the entry state for peripheral IRQs and intra-guest SGIs, i.e., $\sigma^{entry}_{\text{RIGC}} = \sigma^{entry}_{\text{IRQ}}$. Also the following transition, where the hypervisor needs to read the interrupt acknowledgment register in the GIC CPU interface is identical, i.e., RIGC-Ack = IRQ-Ack, notably it saves the context of the guest into memory.

Afterwards the transition systems for both handlers differ slightly, as shown in Fig. 10. In particular the hypervisor performs the following steps.

- RIGC-EOI – from the *GICC_AIAR* value returned by the GIC, the hypervisor learns that the interrupt was an SGI from the core of another guest, therefore it invokes the IGC reception handler. First, similar as in IRQ-EOI, the end-of-interrupt register needs to be written, in order to drop the priority of the physical SGI interrupt.

- RIGC-Check – then, like in as in IRQ-Check, the hypervisor checks the list register dedicated to the virtual IGC interrupt to determine if the interrupt can be injected into the guest.

- RIGC-INJECT – if the virtual IGC interrupt is not currently pending (or active-and-pending) the hypervisor updates the corresponding list register(s) in order to inject the virtual interrupt into the guest. This step is similar to IRQ-INJECT.
- RIGC-DEACT – whether the inerrupt can be injected or not, in both cases the SGI is dectivated through a write to *GICC_DIR*. This step is similar to IRQ-DEACT.
- RIGC-UNLOCK – finally the hypervisor returns to the guest to which the pending virtual interrupt will eventually be delivered. The hypervisor also clears the lock bit in the IGC message box entry for the corresponding channel. As with intra-guest SGIs, this implementation allows new IGC notifications to be sent and discarded by the receiving hypervisor while the guest has not finished handling any pending virtual IGC interrupt.

In the states $hv \in \sigma_{\text{RIGC}}^{await,g',c',g}$ for guest g, the hypervisor received the reply from the GIC to the read of the *GICC_AIAR* register. They are identified as follows.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{irqaw}$ – The program counter is pointing to the address were we halted in the hypervisor in order to request the read to the acknowledgment register.
- $hv.C.u2c = rplR\ a_{IAR}\ v$ – The read value v is returned to the core from the GIC. For this v we have in particular:
  - $\langle v[9:0] \rangle = 15$ – the interrupt was caused by an SGI with ID 15, that is reserved for IGC notification requests.
  - $\langle v[12:10] \rangle = c' \wedge \gamma(c') = g' \neq g$ – the SGI was sent by a core belonging to guest g' which is different from g. Then it has to be injected as an IGC notification virtual interrupt into the guest.

  Note that no other SGI IDs are ever used to send inter-processor interrupts between different guests on the HASPOC platform.

In these states the physical interrupt is dropped in the next hypervisor step by sending a write to the end-of-interrupt register.

$$\frac{\begin{array}{cc} hv \in \sigma_{\text{RIGC}}^{await,g',c',g} & ligc(hv'.m(a_{ligc}), c) = (s, c') \qquad hv'.C.pc = a_{rnew} \\ hv'.u2c = \text{None} & hv'.c2u = \text{W}\ a_{EOIR}\ 4\ (0^{19} \circ \text{bin}_3(c') \circ 0^6 1^4) \end{array}}{hv \rightsquigarrow_c hv'} \text{RIGC-EOI}$$

This is accomplished, as in IRQ-EOI, by a write to the *GICC_EIOR* register, signalling the GIC to drop the running interrupt priority for SGI 15 (bits 9-0) from sender c' (bits 12-10). We also save the received interrupt identifier in a hypervisor variable. The PC is set to a_{rnew} where the hypervisor is waiting

for the write confirmation from the GIC. It is received in states $hv \in \sigma^{ewait}_{\text{RIGC}}$ identified by:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{rnew}$ – The program counter is pointing to the address were we stopped in the hypervisor in order to request the write of the end-of-interrupt register.
- $hv.C.u2c = rplW\ a_{EOIR}$ – the input message box to the core contains the confirmation of the write to $GICC\_EOIR$.

In the following step the hypervisor probes the list register at address $a^{IGC}_{LR} : \mathbb{N}_{ns} \to \mathbb{B}^{48}$ with $a^{IGC}_{LR}(s)[47:12] \in A_{GICH}$ associated with the virtual IGC interrupt for the particular channel s.

$$\frac{\begin{array}{c} g = \gamma(c) \\ hv \in \sigma^{ewait}_{\text{RIGC}} \qquad ligc(hv.m(a_{ligc}), c) = (s, c') \qquad cpol(s) = (\gamma(c'), g) \\ hv'.C.pc = a_{rncw} \qquad hv'.u2c = \text{None} \qquad hv'.c2u = \text{R}\ a^{IGC}_{LR}(s)\ 4 \end{array}}{hv \rightsquigarrow_c hv'}\ \text{RIGC-Check}$$

Note that this action is only enabled if $g' = \gamma(c')$ is allowed to communicate with g according to the HASPOC communication policy. However, as we never send any IGC notifications from guest g' where this is not the case, the action can not block progress of the hypervisor transition system indefinitely. The program counter is set to a_{rncw} to denote the states where the hypervisor expects a reply from the GIC.

The states $hv \in \sigma^{cwait,c,s,c',l}_{\text{RIGC}}$ of core c where this read answer l is received for channel s and sender core c'. The possible next steps can be taken if the following properties hold.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{rncw}$ – The program counter is pointing to the address were we stopped in the hypervisor in order to request the read to the list register corresponding to the requested IGC virtual interrupt.
- $ligc(hv.m(a_{ligc}), c) = (s, c')$ – The interrupt requested ties to channel s.
- $hv.C.u2c = rplR\ a^{IGC}_{LR}(s)\ l$ – The value l of the requested list register is returned to the core from the GIC.

Again we introduce auxiliary definitions for the two possible actions originating from these states. First we define a predicate that determines when the interrupt can be injected. This is simply determined by bit 28 of the list register which is 1 if the interrupt is pending (or active-and-pending) and we set $inj_{IGC}(l) = \neg l[28]$.

As in the IRQ virtualization, the request to be sent is determined by bit 29 of the list register, denoting an active interrupt. Function $injlr_{IGC} : \mathbb{N}_{ns} \times \mathbb{B}^{32} \to \mathbb{R}$ computes this write request for a given channel index and the read list register

value. Moreover $prio_{IGC} \in \mathbb{B}^5$ denotes the 5-bit encoding of the fixed priority assigned to virtual IGC interrupt which is the same for all channels on the HASPOC platform and $id_{IGC} : \mathbb{N}_{ns} \to \mathbb{B}^{10}$ is the binary virtual interrupt ID reserved for a given channel.

$$injlr_{IGC}(s, l) = \begin{cases} \text{W } a_{LR}^{IGC}(s)\ 4\ 0101 \circ prio_{IGC} \circ 0^{13} \circ id_{IGC}(s) & : \quad \neg l[29] \\ \text{W } a_{LR}^{IGC}(s)\ 4\ l[31:30] \circ 11 \circ l[27:0] & : \quad l[29] \end{cases}$$

In case the virtual interrupt is currently inactive we inject a new pending interrupt (bits 29-28 = 01) with the virtual ID corresponding to channel s (bits 9-0). The hardware bit (bit 31) is set to zero, denoting an interrupt generated by software with no corresponding active physical interrupt present. If the virtual interrupt is currently active (and not active-and-pending), the interrupt state of the virtual IGC interrupt becomes active-and-pending. Note that this behaviour is similar to the injection of physical interrupts as is the corresponding injection transition.

$$\frac{hv \in \sigma_{\text{RIGC}}^{cwait,c,s,c',l} \qquad inj_{IGC}(l) \qquad hv'.C.pc = a_{rniw} \qquad hv'.u2c = \text{None} \qquad hv'.c2u = injlr_{IGC}(s,l)}{hv \rightsquigarrow_c hv'} \text{ RIGC-Inject}$$

After the transition the program pointer points to a_{rniw} to denote the waiting states $hv \in \sigma_{\text{RIGC}}^{iwait,c,s,c'}$ of the hypervisor handler for channel s on core c. Core index c' denotes the sender of the initial request. In these waiting states the hypervisor receives the write confirmation from the GIC and we have:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{rniw}$ – The program counter is pointing to the address where the hypervisor stopped, waiting for the list register write confirmation.
- $ligc(hv.m(a_{ligc}), c) = (s, c')$ – The interrupt requested ties to channel s.
- $\exists d \in \{4, 8\}, v \in \mathbb{B}^{8d}.\ hv.C.u2c = rplW\ a_{LR}^{IGC}(s)$ – the input message box to the core contains the confirmation of the write to the list register.

In these states and the ones in $\sigma_{\text{RIGC}}^{cwait,c,s,c',l}$ where the interrupt may not be injected, the physical interrupt needs to be deactivated.

$$\frac{hv \in \sigma_{\text{RIGC}}^{iwait,c,s,c'} \vee hv \in \sigma_{\text{RIGC}}^{cwait,c,s,c',l} \wedge \neg inj_g(l) \qquad hv'.C.pc = a_{rndw} \qquad hv'.u2c = \text{None} \qquad hv'.c2u = \text{W } a_{DIR}\ 4\ (0^{19} \circ \text{bin}_3(c') \circ 0^6 1^4)}{hv \rightsquigarrow_c hv'} \text{ RIGC-Deact}$$

The SGI interrupt is deactivated by a write to the *GICC_DIR* register. We use the same value already written to the end-of-interrupt register, as both have the same layout. When the confirmation of the write is sent back from the GIC, the hypervisor is in a state $hv \in \sigma_{\text{RIGC}}^{dwait}$ identified as follows.

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = a_{rndw}$ – The program counter is pointing to the address were we halted in the hypervisor in order to request the write of the deactivate-interrupt register.
- $\exists v \in \mathbb{B}^{32}.\ hv.C.u2c = rplW\ a_{DIR}$ – the input message box to the core contains the confirmation of the write to *GICC_DIR*.

The final transition of the IGC notification reception handler restores the guest and the hypervisor unlocks the SGI tied to the IGC notification request. Again we set bit 7 of the hypervisor control register to 1 in order to signal a virtual interrupt as pending to the guest.

$$\frac{\begin{array}{c} hv \in \sigma^{dwait}_{\text{RIGC}} \qquad cpol(s) = (g', g) \qquad mbox(hv', g', g) = 0 \\ ligc(hv'.m(a_{ligc}), c) = \bot \qquad hv'.C.gpr = ctx(hv.m(a^c_{ctx}), 0, c)).gpr \\ hv'.C.pc = ctx(hv.m(a^c_{ctx}), 0, c)).pc \qquad hv'.C.ps = ctx(hv.m(a^c_{ctx}), 0, c)).ps \\ hv'.C.spr(HCR\_EL2)[7] = 1 \qquad hv'.mmu.active = 1 \qquad hv'.u2c = \text{None} \end{array}}{hv \rightsquigarrow_c hv'} \text{ RIGC-Unlock}$$

Subsequently the pending virtual IGC interrupt may be delivered to the guest by the GIC. Before its handling completes, the hypervisor may, however, still be interrupted by further IGC notification requests as the lock bit in the message box for the sender is cleared. Such redundent interrupts are are either discarded or they change the virtual interrupt state to active-and-pending. However, notification requests sent by different guests may still be delivered to the receiver even if it has pending virtual IGC interrupts for a different IGC channel. This is due to the fact that the hypervisor distinguishes the SGIs implementing the request by the sender core and that each IGC channel has its own lock bit as well as its own dedicated virtual IGC interrupt.

3.4.9 MMU Fault Injection

Whenever the second-stage MMU raises an exception, that is not connected to a memory access targeting the GIC distributor, the guest is trying to access an address outside its virtual memory. In these cases we inject a data abort into the guest. More specifically, we need to jump to the corresponding handler in EL1, depending on which mode and stack pointer the guest was using before the exception. This is similar to transition GICD-Emu-F.

We first define the set of hypervisor entry states $\sigma^{entry,g}_{\text{MMUF}}$ that enable the MMU fault injection handler for guest g. For every such state hv we have:

- $hv.C.ps.mode = 2$ – The hypervisor is executing in EL2.
- $hv.C.pc = hv.C.spr(VBAR\_EL2) +_{64} 0^{53}10^{10}$ – The program counter is pointing to the exception vector for synchronous exceptions from a lower exception level executing in AArch64 mode (exception vector base address + offset 0x400).

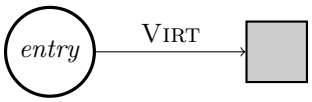

Figure 11: Transition diagram for the MMU fault handler.

- $hv.C.spr(ESR\_EL2)[31:26] \in \{100000, 100100\}$ – The exception syndrome register holds exception codes 0x20 or 0x24, signalling an Instruction or Data Abort from a lower exception level.
- $hv.C.spr(ESR\_EL2)[24] = 1$ – The instruction specific syndrome field holds valid information. This field $ISS = hv.C.spr(ESR\_EL2)[24:0]$ needs to be set up as follows:
  - $ISS[5:3] \in 000$ – The attempted access caused a second-stage Translation Fault or an Address Size Fault at any level of translation.
- $hv.C.spr(HPFAR\_EL2)[39:4] \in A^g_{fault}$ – The Hypervisor IPA Fault Address Register contains an intermediate physical page address that is outside guest g's virtual memory. This region is defined by

  $$A^g_{fault} = \{a \in \mathbb{B}^{36} \mid a \notin \mathbb{A}_g \vee a = a^g_{GICD}\},$$

  i.e., either the address exceeds the guest address space or it is pointing to the guest's GICD interface. Note, however, that GICD accesses are only treated by the MMU fault handler for guest accesses that are not supported by the GICD virtualization handler, as identified by the instruction syndrome field, i.e., if $hv.C.spr(HPFAR\_EL2)[39:4] = a^g_{GICD}$ then

  $$\forall hv' \in \sigma^{entry,g,r}_{\text{GICD}}.\ hv'.C.spr(ESR\_EL2)[24:0] \neq ISS\,.$$

- $hv.C.u2c = \text{None}$ – There are no outstanding replies from the GIC.

In such a state the fault injection as depicted in Fig. 11 is defined as follows.

$$\frac{\begin{array}{c} g = \gamma(c) \\ hv \in \sigma^{entry,g}_{\text{MMUF}} \qquad hv'.C.spr(SPSR\_EL1) = spsr2 = hv.C.spr(SPSR\_EL2) \\ hv'.C.spr(ELR\_EL1) = hv.C.spr(ELR\_EL2) \\ hv'.C.spr(FAR\_EL1) = hv.C.spr(FAR\_EL2) \\ hv'.C.spr(ESR\_EL1) = hv.C.spr(ESR\_EL1)[31:27] \circ spsr2[2] \circ 1 \circ 0^{25} \\ hv'.C.pc = hv.C.spr(VBAR\_EL1) +_{64} 0^{52} \circ ghoff(spsr2) \\ hv'.C.ps = hv'.C.ps[mode \mapsto 1] \end{array}}{hv \rightsquigarrow_c hv'} \text{ MMUF-VIRT}$$

We enter EL1, setting the program counter to the correct exception vector. Moreover, the exception syndrome register holds the encoding of an INstruction or Data abort that was caused by EL1 or EL0, depending on the previous mode

of the guest. In addition the saved processor state and link address is transfered to the corresponding EL1 registers together with the faulting address.

This finishes the definition of the hypervisor transition system. We collect all hypervisor entry point addresses in the set $A_{hve} = \{hv.C.pc \mid hv \in \Sigma^{entry}\} \subset A_{HV}$ with

$$\Sigma^{entry} = \sigma^{c}_{\text{reset}} \cup \sigma^{entry}_{\text{SMC}} \cup \bigcup_{g} \sigma^{entry,g,r}_{\text{GICD}} \cup \bigcup_{v} \sigma^{entry,v}_{\text{SIGC}} \cup \sigma^{entry}_{\text{IRQ}} \cup \bigcup_{g} \sigma^{entry,g}_{\text{MMUF}} .$$

Note that these are the only entry points to the hypervisor, i.e., under no circumstances than the ones described above, the hypervisor is entered. This means in particular that we mask FIQs and SErrors and need to configure the second-stage address translation in a way that avoids Instruction and Data Aborts for other reasons than the ones covered here. For example, we assume that all Access flags in the second-stage page tables are enabled, making the occurence of second-stage Access Flag Faults impossible.

In fact there may still be unexpected faults due to hardware failure and bus parity errors, but we omit a formal treatment here for simplicity. In these cases, the hypervisor should halt execution and perform an emergency erase (which we also do not cover in this model).

To conclude, we list some core observations about the hypervisor handlers (excluding the initialization phase).

- The handlers never read or write guest memory. Also handlers only access the guest context of the core they are running on. Thus, there is no information flow between guests through the handlers.
- The saved guest context only becomes visible in hypervisor memory if handlers are non-atomic, i.e., in intermediate hypervisor steps.
- The handlers as specified above only manipulate a small set of the state. For instance they leave large portions of the registers and hypervisor memory unchanged. This is different in the actual machine code implementation, but since we are proving a refinement of the ideal model where these effects are invisible as well, we can ignore them. Similarly, any refinement to the machine code level only needs to cover components that are modified by the refined model.
- The hypervisor does not touch peripherals or meddle with peripheral interrupts. All interrupt virtualization is handled via the GIC interface.
- All page tables are unmodified by the hypervisor handlers. Similarly SMMU configurations are unchanged. The second-stage MMUs are turned off during communication with the GIC, however the remainder of their configuration stays the same as well.
- Second-stage faults which do not correspond to virtualized GIC distributor accesses lead to the injection of Data Aborts into the guest. Thus, if the second-stage is configured correctly, there is no way for the guest to read or write another guest's memory.

- Moreover, we only distinguish the guest's exception level when selecting the exception vector for the fault injection. Other than that, all memory faults are treated independent of the running guest's exception level. This implies that a guest kernel may allow its user processes to access the GIC distributor, e.g., as a user mode driver.

Finally, the refined model is complete. Admittedly, we made some simplifications and omitted certain details compared to the actual hypervisor implementation. We believe these simplifications are justifiable in the context of this case study.

4 Ideal-Refined Bisimulation

In order to conduct the refinement bisimulation proof between the ideal and the refined model we need to take the following steps.

1. We introduce projections that map the guest-specific parameters like core numbers and address spaces of the ideal model to global parameters of the refined model.
2. We use these projections to define the bisimulation relation between the two models.
3. We define an additional *implementation invariant* that treats all the refined model components not covered by the bisimulation. This invariant needs to hold on all traces of the refined model after initialization and supports establishing the isolation properties of the ideal model as well as proving the bisimulation.
4. We introduce initial configurations for the ideal model and the refined model after reset.
5. We state our refinement theorem and decompose it into three separate proof goals.
6. We formulate assumptions on system functionality like the GIC or second-stage MMU. These are verification conditions on the particular model of each component, but we do not go into the detailed proofs here.
7. We sketch the four proofs that establish the bisimulation.

4.1 Projections

The following projections between refined and ideal world are needed to define the bisimulation relation. They have been partly introduce before, as the information they carry was also needed to define the hypervisor transition relation. We now complete their definition.

- $\pi_C : \mathbb{N}_{nc} \to \mathbb{N}_{ng} \times \mathbb{N}_{nc}$ – maps a core c to its owning guest and the core index within that guest, i.e., if $\pi_C(c) = (g, c')$ then core c in the refined model is core $c' \in \mathbb{N}_{nc_g}$ of guest g in the ideal model. We allow to address the guest index of the projection by γ and the core index by κ. Also we require that the mapping is injective for cores belonging to the same guest, i.e., $\forall c \neq c' \in \mathbb{N}_{nc}.\ \gamma(c) = \gamma(c') \Rightarrow \kappa(c) \neq \kappa(c')$.
- $\pi_P : \mathbb{N}_{np} \to \mathbb{N}_{ng} \times \mathbb{N}_{np}$ – maps a peripheral p to its owning guest and the peripheral index within that guest, i.e., if $\pi_P(p) = (g, p')$ then peripheral p in the refined model is peripheral $p' \in \mathbb{N}_{np_g}$ of guest g in the ideal model. We allow to address the guest index of the projection by γ and the peripheral index by φ. Again we require that the mapping is injective

for peripherals belonging to the same guest, i.e., $\forall p \neq p' \in \mathbb{N}_{np}.\ \gamma(p) = \gamma(p') \Rightarrow \varphi(p) \neq \varphi(p')$.

Using them we restrict the parameters of both models to obey the following rules.

- $nc_g = \#\{c \in \mathbb{N}_{nc} \mid \gamma(c) = g\}$ – The number of cores for each guest agree with the projection function π_C.
- $np_g = \#\{p \in \mathbb{N}_{np} \mid \gamma(p) = g\}$ – The number of peripherals for each guest agree with the projection function π_P.
- $\forall p \in \mathbb{N}_{np}.\ \mathcal{P}_{\gamma(p),\varphi(p)} = \mathcal{P}_p$ – the types of all peripherals agree between the ideal and refined model.
- $\forall p \in \mathbb{N}_{np}.\ \delta_P^{\gamma(p),\varphi(p)} = \delta_P^p$ – We also use the same transition functions in both models for peripherals.
- $\forall p \in \mathbb{N}_{np}.\ PIRQ_{\gamma(p),\varphi(p)} = PIRQ_p$ – each peripheral sends the same interrupts identifiers in the ideal and refined model.
- $\forall g \in \mathbb{N}_{ng}.\ \#A_G(g) = 2^{as_g} - 1$ – in the refined model, the memory reserved for each individual guests is one page less than the amount of memory allocated to them in the ideal model. The missing page refined page is representing the virtual GIC distributor of the guest.
- $\forall p \in \mathbb{N}_{np}.\ \#A_P^{\gamma(p),\varphi(p)} = \#A_P(p)$ – in both ideal and real model the individual peripherals have the same amount of memory allocated to them. Let this size be denoted by $sz(p) = \lceil \log \#A_P(p) \rceil$ counted in page address offset bits needed to address them wrt. their base address, which should be aligned in physical memory, i.e., $(\min A_P(p))[sz(p) - 1 : 0] = 0^{sz(p)}$.
- $\forall p \in \mathbb{N}_{np}.\ A_P(p) \subseteq A_G(\gamma(p))$ – the memory of each peripheral is allocated to the guest that owns it.
- $\#A^g_{GICC} = \#A_{GICV}$ – the number of page addresses for the GIC CPU interfaces in the ideal model, denoted by set $A^g_{GICC} \subset A_P^{g,np_g}$ is the same as for the virtual GIC interface in the refined model.

Note, that the requirement on the peripheral types and transition functions has non-trivial implications. This is due to the fact that peripherals receive requests with intermediate physical addresses in the ideal model and with physical addresses in the refined model. If we want to use the same models for peripherals in the ideal model as in the refined one, we need that peripherals are independent of their allocated memory-mapped I/O region base address, i.e., they ignore the upper parts of addresses that do not contribute to peripheral register offsets within that region. If this is not the case the second-stage address translation functions need to be introduced for memory-mapped I/O accesses already in the ideal model, or modified peripherals need to be introduced in the ideal model that work on intermediate physical addresses.

In fact, this issue highlights a design choice we made in our models. While in the real world cores and peripherals communicate with memory and each other over shared buses that need to be accessed exclusively, here, for reasons of modularity, we represent this communication by a message passing protocol, thus allowing for much more interleaving between accesses from different actors. In addition, we need to keep track of the senders and receivers when handling memory-mapped I/O, while in real systems peripherals are listening on the bus for requests to their base address. The message box mechanism used here is thus just an abstraction of the memory bus logic. Therefore it is reasonable to assume that the remaining part of a peripheral, to which a peripheral model here is actually corresponding, is independent of the actual base addresses used.

Below we introduce the second-stage address translation function between intermediate physical addresses in the ideal and physical address in the refined model for each guest g. It should be an injective mapping of pages

$$\theta_g : \mathbb{A}_g \to A_G(g) \cup \{a_{GICD}\}$$

from the consecutive intermediate physical address space of each guest to the corresponding physical pages, or a_{GICD} for the address of the GIC distributor, i.e., $\theta_g(a^g_{GICD}) = a_{GICD}$. Note that we do not actually set up the GIC distributor mapping in the second-stage page tables, as any guest access to this region should be intercepted by the hypervisor. Still we include the mapping in the definition of θ_g so we can map the hypervisors GIC distributor accesses to the simulated access in the ideal model.

We assume that the page mapping θ_g is statically defined for a given hypervisor configuration and guest, i.e., it is not determined first during run-time. Note that domain and codomain of θ_g have the same size, therefore the mapping is also bijective for each guest and θ_g^{-1} exists. Moreover, since the codomains $A_G(g)$ are disjoint for all guests modulo IGC channels and the GIC distributor, we also know for the translation functions of different guests that they map to different images for all other addresses, i.e.:

$$\begin{aligned}\forall s, \in \mathbb{N}_{ns}, g, g' \in \mathbb{N}_{ng}, a, b.\ cpol(s) = (g, g') \ \Rightarrow \\ a \in \operatorname{dom} \theta_g \setminus \{IGCa(s)_1, a^g_{GICD}\} \ \Rightarrow \\ b \in \operatorname{dom} \theta_{g'} \setminus \{IGCa(s)_2, a^{g'}_{GICD}\} \ \Rightarrow \ \theta_g(a) \neq \theta_{g'}(b)\end{aligned}$$

Concerning IGC channels we can now also demand that they are allocated in physical memory according to the HASPOC communication policy. For all IGC channels $s \in ns$, intermediate physical page addresses $a, b \in \mathbb{B}^{36}$ and guests $g, g' \in \mathbb{N}_{ng}$, we require:

$$IGCa(s) = (a, b) \wedge cpol(s) = (g, g') \Rightarrow A_{IGC}(g, g') = \{\theta_g(a)\} = \{\theta_{g'}(b)\}$$

$$(\nexists s.\ cpol(s) = (g, g')) \Rightarrow A_{IGC}(g, g') = \emptyset$$

Thus the potentially different intermediate physical addresses allocated for IGC channels in the ideal world are mapped to the same physical address for sender and receiver. Note also that the above requirement restricts the size of shared

pages between guests to one per direction in the refined model, just as in the ideal model. Additionally, if there is no channel in the ideal model then there is also no channel in the refined one. This also implies $A_{IGC}(g, g) = \emptyset$ due to restrictions on the communication policy.

For all peripherals we demand that their intermediate physical addresses used by guests are mapped correctly to the peripheral's physical address region. The address regions of each guest's GIC CPU interface, located at pages A^g_{GICC}, are mapped to the virtual GIC interface. The mapping of the peripheral and GIC interfaces should also preserve the order of page addresses.

$$\begin{array}{l} \forall p \in \mathbb{N}_{np}, a, a' \in A_P^{\gamma(p),\varphi(p)}. \quad \theta_{\gamma(p)}(a) \in A_P(p) \land a < a' \Rightarrow \theta_{\gamma(p)}(a) < \theta_{\gamma(p)}(a') \\ \forall g \in \mathbb{N}_{ng}, a, a' \in A^g_{GICC}. \quad \theta_g(a) \in A_{GICV} \land a < a' \Rightarrow \theta_g(a) < \theta_g(a') \end{array}$$

Finally, since we require peripheral models to be independent the peripherals base address, we need to demand that the second stage page tables for each guest are set up in a way such that they preserve the lower bits which are not part of the base address.

$$\forall p \in \mathbb{N}_{np}, a \in A_P^{\gamma(p),\varphi(p)}. \quad a[sz(p) - 1 : 0] = \theta_{\gamma(p)}(a)[sz(p) - 1 : 0]$$

Note that we only need to constrain the sz lowest bits of the page addresses. All lower address bits within a page are already identical in intermediate physical and physical addresses due to the address translation algorithm.

4.2 Bisimulation Outline

Before we introduce the bisimulation relation formally, we give an outline of its definition. We also describe our strategy to prove that the relation is preserved by any step of the ideal or refined model for both directions of the bisimulation theorem. The stepping of both models is of particular interest for the hypervisor handlers, as it directly influences the formulation of the bisimulation relation.

This relation formally links the ideal and the refined model. If we want to be able to transfer all kinds of properties stated about the ideal model "down" to the refined model, then the relation needs to cover all components of the ideal model. However it does not need to cover all components of the refined model. This lies in the nature of abstraction and refinement. Not all behaviour in the refined model, e.g., the work of the second stage MMU, is invisible in the ideal model due to the restrictions the hypervisor imposes on the refined model. These restrictions are not covered by the bisimulation relation but by an additional implementation invariant that will be defined later. Here we focus on the the components of the refined model that correspond to ideal model components. Below we give a rationale how each component of the ideal model is implemented in the refined model and introduce some auxiliary notation that will support our definitions.

- G – the partitioning of guest configurations is not directly visible in the refined model, which is the intuition for why to introduce an ideal model

in the first place. However each component of the guest is mapped to disjoint portions of the refined model, except for resources related to GIC, IPI, and IGC. For each guest we map their components as follows.

- C – the core components of a guest are only coupled while the core is active (i.e., powered on) in the ideal model. Then they are mapped one to one to the corresponding refined model while the guest is running on the particular core. If all hypervisor calls were executed atomically (without revealing any intermediate state to other threads) this would be sufficient, since each core is owned exclusively by a guest. However, several handlers are not atomic, thus we need to expose the saved core context and couple it with the ideal core state while the hypervisor is executed and in an intermediate state. Below we give the definitions for each core component
  - *active* – cores in the ideal model are active whenever their corresponding core in the refined model are active and initialized, and vice versa
  - *ps* – the processor state of the ideal model is coupled with the state of the corresponding refined core either one-to-one (while the guest is executed), with the *SPSR_EL2* register (directly at hypervisor entry), or with the stored guest context in hypervisor memory (in intermediate states of hypervisor handlers). We require here that the hypervisor stores the context of the guest completely in memory and does not rely on special purpose registers to keep this information while in intermediate states that are observable in the refined model.
  - *pc* – the program counter is coupled either with the ideal PC register during execution of EL1 and EL0, or with the *ELR_EL2* register at hypervisor entry, or with the stored guest context in intermediate hypervisor states. The value saved in the *ELR_EL2* register and memory is either the same as in the ideal model PC (for all interruptions except system calls), or (in case of system calls) the address of the next instruction in the ideal model.
  - *gpr* – the general purpose registers are coupled with the ideal model registers while the guest is executing as well as at hypervisor entry, or with the saved guest context in intermediate hypervisor states.
  - *spr* – There is no need to save SPRs of the guest in the guest context since hypervisor and secure boot have their own set of SPRs for EL2 and EL3. Therefore the SPRs of the ideal model are always identical to their counterparts in the refined model.
- m – the virtual memory pages of each guest are mapped to the refined physical memory according to the address translation function
- *P.st* – the peripherals of each guest are mapped to the corresponding peripherals in the refined model using the projection functions

defined above. Note that this mapping is unique and complete due to the assumptions we made.

- *cif* – the message channels from guest core to its virtual memory are mapped to the corresponding *c2u* channel in the refined model according to the core projection function π_C. The work of the second-stage MMU in the refined model to handle and produce these messages are invisible in the ideal world. However, since guest GIC accesses are virtualized by the hypervisor and the GIC virtual interface, there is a special case for ideal messages of the guest to the GIC. They have to be converted into refined model messages sent by the hypervisor, using the address translation function for all GIC accesses and the conversion function ρ_g for ideal distributor accesses, as defined for the GICD virtualization handlers.
- *m2c* – the message channels from each guest's virtual memory to its cores are mapped to the corresponding *u2c* channels in the refined model. Again there are special cases for replies from the GIC distributor. Write replies and emulated read replies from the ideal model are mapped to the refined model using the address translation function and conversion function ρ_g. Read replies from the distributor in the refined model, however, are mapped to a ideal model reply through the ρ_g conversion function. Additionally memory faults for accesses to the GIC distributor in the refined model are not visible in the ideal model.
- *pif*, *m2p* – the peripheral message channels of each guest are mapped to *p2v* and *v2p* respectively, hiding the actions of the SMMU. Again the physical addresses in the refined model memory-mapped I/O messages are converted to intermediate physical addresses used in ideal model using the inverse address translation function. Accesses to the GIC distributor registers in the ideal model are coupled differently as they are virtualized by the hypervisor. If there are messages to the GIC distributor pernding from a given ideal core, the corresponding refined core is in the entry or exit state of the GICD virtualization handler for the corresponding guest request.
- *PI* – the active interrupts of each guest are mapped to the active interrupt state in the refined model using projection function π_P
- *Q* – the interrupts that are pending, active, or active-and-pending in each guest are mapped from the virtual interrupts of the corresponding core wrt. π_C. We exclude the virtual IGC interrupts as they are represented differently in the ideal model. We also need to translate back the core indices in outstanding intra-guest SGIs. Moreover there is a special case for newly injected interrupts. While the hypervisor is injecting a physical interrupt into one core of a guest, other cores of the same guest may turn off a peripheral interrupt that caused the injection in the first place. Therefore in the bisimulation we need

to schedule the corresponding GIC step that delivers the peripheral interrupt to the guest's core as soon as we know that the hypervisor will eventually inject it as a virtual interrupt.

  - $rIGC, sIGC$ – the outstanding IGC interrupts are coupled with entries in IGC mailbox of the hypervisor implementation and outstanding virtual IGC interrupts.
  - E – the guest's event sequence in the ideal model is extracted from the refined model event sequence for each guest. In other words, the event sequence of the refined model is an arbitrary (but local-order-preserving) interleaving of all the guest's event sequences.
  - GIC – the configuration of a guest's GIC CPU interface is coupled directly with the virtual interrupt interface for each core. For the distributor registers, both models are coupled by converting the values stored in both models back and forth, depending on how a particular GIC register is virtualized. For registers were a read is performed in the refined model, the ideal register holds a version of the refined register value, filtered for the particular guest. For registers that are written by the hypervisor the refined register is a filtered version of the ideal register. For the latter case, and registers which are emulated without touching the GIC, the ideal registers are also identical with their local copy in the hypervisor for each guest. We do not couple the refined CPU interfaces or the virtual interrupt control registers.

- S – the inter-guest communication channels are mapped to the respective shared memory regions in physical guest memory according to the A_{IGC} parameter to the implementation of HASPOC platform communication policy described above.

- tgt – the target of the IGC interrupt corresponds to a pending or soon-to-be pending SGI from the sender core to the corresponding targeted core.

This relation is the central part of our bisimulation theorem. Intuitively it states on one hand that for every refined model computation that is starting in a state that is in the relation with a corresponding ideal model state, we can find a matching ideal model computation such that each state in the refined model computation is matched with a (consecutive) state in the ideal computation. The same property holds vice versa in the simulation of the ideal model by the refined one. Note that one step in the refined model may in principle correspond to an arbitrary number of steps in the ideal model such that the relation holds between the starting and resulting states. Similarly, one ideal step may correspond to any number of refined steps in the same way. However, we aim for a common strategy of scheduling steps in the refined and ideal model for both directions of the bisimulation. This strategy is explained for different kinds of transitions in the following sub-sections.

4.2.1 Guest Core and Peripheral Steps

Whenever a guest is making a core step, i.e., performing a core step in EL1 or EL0 such that EL2 is not entered, exactly the same step is executed in both models. Note that this excludes the execution of hypercalls and SMCs as well as taking exceptions that are trapped to EL2. These cases are handled separatedly by the hypervisor and are explained below.

Similarly, whenever a peripheral performs a step it is scheduled in both models producing the same effect. Note that we need here that the peripheral step has the same effect in the ideal and refined model, i.e., the same internal non-deterministic choice of an action must be possible in both models and the result of the step is not affected by differences in the upper bits of intermediate-physical and physical addresses of memory-mapped I/O requests, handled by the respective peripheral.

Note that this kind of stepping is possible because there is a one-to-one mapping of cores and peripherals in the refined model to the corresponding components in the guest configurations of the refined model.

4.2.2 MMU, SMMU, and Memory Steps

The second-stage memory management units as well as the SMMUs are only visible in the refined model. While they are translating a memory request, the same request is present in the outgoing memory request message box of the corresponding core until either a fault is produced or the final memory request is processed by the shared memory. In both cases we schedule a memory step in the ideal model which produces either an identical memory fault or processes the request in the same way as in the refined model.

For core requests to actual physical memory and peripherals' DMA requests this means that the memory request is consumed by the refined memory and the ideal memory of the corresponding guest. Memory-mapped I/O requests and replies moved to and from the peripheral and GIC input and output channels in a similar fashion, as we step the memory in both models simultaneously. This also implies that replies for outstanding memory requests are produced in lock-step synchronicity by the ideal and refined models.

4.2.3 GIC Steps

Similar to memory steps we step the GIC in the ideal model whenever the GIC is stepped in the refined model, resulting in equivalent behaviour. However, there is an exception for emulated guest accesses to the GIC distributor. Here the GIC step that executes the request is performed whenever the emulation step reads or writes the local copy of the accessed register. More details about the GICD virtualization will be given in the discussion of the virtualization handler.

The responses of the GIC to memory-mapped I/O requests of guests to the ideal CPU interface registers are produced in simultaneously in any case, while in the refined model these requests are mapped to the virtual interrupt interface. Here, the bisimulation relation needs to guarantee that in both models the

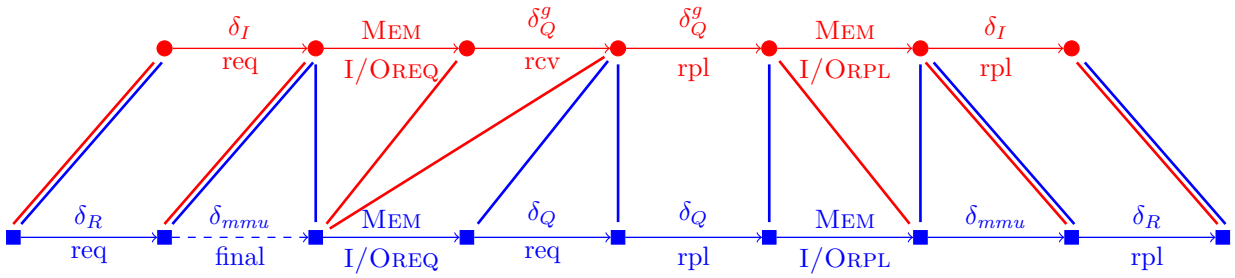

Figure 12: Bisimulation of an access to the ideal GIC CPU interface. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. The dashed arrow subsumes several steps of the MMU.

same behaviour of the GIC is observed. This is also necessary for autonomous GIC steps that deliver interrupts to the cores. It is necessary to establish the bisimulation in a way that whenever an interrupt can be delivered to the GIC in the ideal model, this is also possible in the refined model and vice versa. More details about this will be given in the context of the IRQ reception handler.

There is a particularity about the stepping of the bisimulation when guests access their GIC CPU interface, as shown in Fig. 12. Since there are several ideal GICs in the ideal model but only one in the refined model, problems arise when cores from different guests access the GIC simulataneously in the ideal model. We need to make sure in the stepping of the refined model that the GIC is always free to receive requests and produce replies.

Therefore in the simulation of the ideal model by the refined model we need to establish a *simulation invariant* that guarantees that the channels to and from the GIC are always empty. In order to maintain this invariant we need to step the refined and ideal models "out of sync" so that requests arrive in the refined GIC and the results are produced and sent back to the requesting core in a single atomic steps.

Similarly, for the simulation of the refined by the ideal model, we need to make sure that any ideal GIC step that produces an I/O reply can also be scheduled on the ideal model. To this end another simulation invariant is established on the ideal model that guarantees that in each guest the input and output channels of its GIC are empty. Then the reception of requests from the core and the production and delivery of replies to the core are simulated by two atomic steps.

Note, that this setting leads to the coupling for the two directions not to be "symmetric", in the sense that the coupled states in the simulation of the ideal by the refined model is not a subset of the coupled states in the simulation of the other direction. This is no problem as long as we can find a bisimulation relation that fits both directions. However, observe that this issue is purely synthetic as it arises due to the existence of the message channels between memory and the GIC. If we could eliminate them, or increase their capacity, the stepping

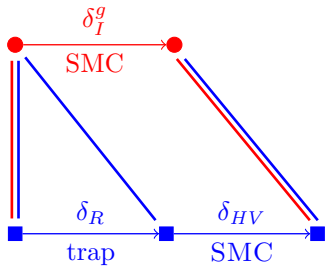

Figure 13: Bisimulation of the power control SMC handler. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red.

bisimulation would be symmetric.

4.2.4 Boot and Initialization

While the HASPOC platform is being booted and initialized, no steps are performed in the ideal model. However the bisimulation relation is maintained with the initial guest states and the implementation invariant is established step by step. Note that the guests are launched concurrently, i.e., while one guest is still in preparation, another one may already be executing, due to the non-deterministic scheduling of the refined model. In any case, the stepping of the ideal model is straight-forward. Whenever a guest starts executing on a core of the refined model, the corresponding core in the ideal model performs its first step. Conversely, in order to simulate the first step of a core in a guest of the ideal model, all refined steps necessary to initialize that guest and the core need to be executed sequentially.

4.2.5 SMC Handler

The coupling of the steps of the power control SMC handlers with the ideal model is depicted in Fig. 13. In the refined model first a core step (denoted by δ_R) executes the SMC instruction which causes a trap into the hypervisor. There the power control request is either granted or denied, as described in the defintion of the SMC handler. In both cases the handler just takes one atomic step and immediately returns to the guest. These two steps are coupled with one ideal core step (denoted by δ_I^g) which has a special semantics for the SMC instruction that reflects the effect of the SMC, i.e., either issuing the power control command or denying the request.

Afterwards both models are in sync again. Note, that the exception step in the refined model is not visible in the ideal model. Instead the ideal computation is stuttering and the guest's core configuration is coupled with the exception context of that core in the refined model. Then both the idealized core transition and the hypervisor handler are executed in lock-step.

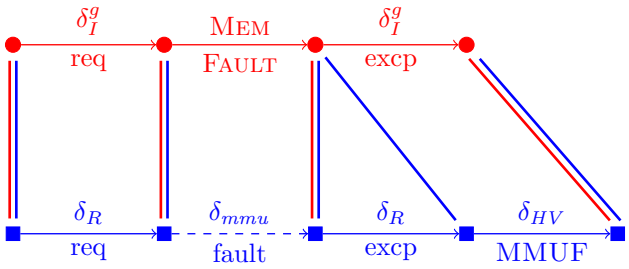

Figure 14: Bisimulation of the MMU fault handler. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. The dashed arrow subsumes several steps of the MMU and memory. All intermediate refined states are coupled with the ideal state before the memory step.

Note for Fig. 13 and the following ones that in any intermediate state of the ideal and refined model any other core, MMU, SMMU, peripheral, the GIC or the memory may be stepped, given that such a step is enabled. We leave out these possible interleavings for clarity. In any case they need to preserve the bisimulation for all components in the system.

4.2.6 MMU Fault Handler

Figure 14 depicts the coupling of the MMU fault handler including preceding steps. Note that the MMU steps leading to an MMU fault are hidden in the ideal model, the corresponding message is just stalled in the message channel to memory. In the final MMU step the fault message is produced. We couple this step with an ideal memory step that produces the same fault message in case the targeted message is out of bounds of the corresponding guests memory. Note that the MMU fault handler captures only these kinds of memory faults. Accesses to the GIC distributor are handled by the handler described next.

Once the fault message is consumed by the refined core, an exception into EL2 is taken. There the MMU fault handler decodes the fault and injects it into the guest at exception level EL1. In the ideal model these two steps are mapped to the exception step of the ideal core that enters EL1 directly.

In the simulation of the ideal model by the refined model all MMU steps need to be executed sequentially whenever the memory transition produces the fault message. Similarly the exception step of the ideal core is mapped to the refined exception step and the complete handler execution.

4.2.7 GIC Distributor Virtualization

The coupling for the GIC distributor virtualization handler is depicted in Fig. 15. In the beginning, it is similar to the memory fault handler until in the refined model the Data Abort is generated. This memory fault is not visible in the ideal model. Instead the ideal model is stalled with the GICD request still in

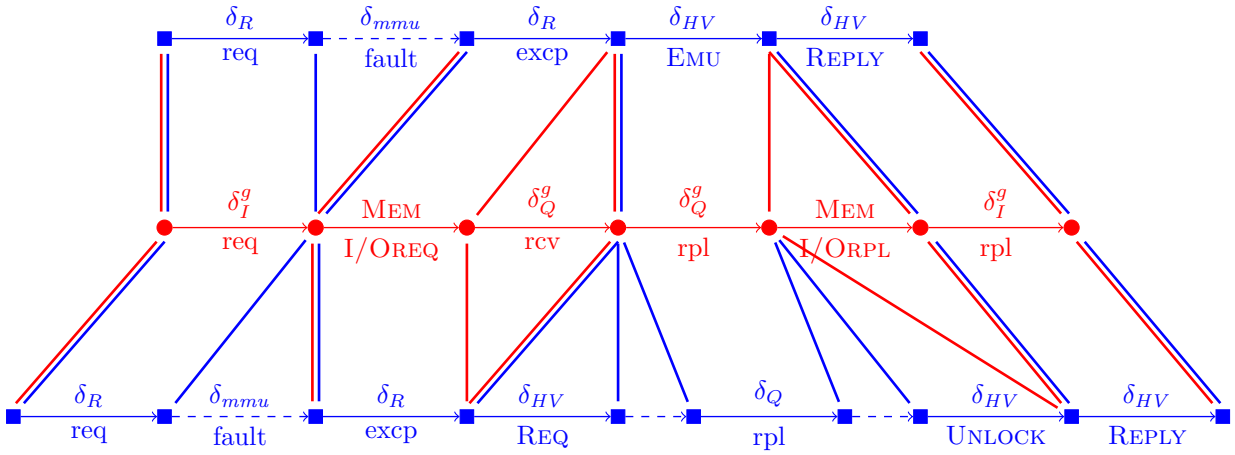

Figure 15: Bisimulation of the GIC distributor virtualization handler. The upper execution trace describes an emulated request, in the lower trace the GIC request is actually performed, locking the distributor for concurrent access. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. The dashed arrow subsumes several steps of the MMU, memory, and GIC that move the request to and from the interrupt controller as well as to translate the initial request.

the channel to memory. When the exception occurs and he hypervisor handler is entered in the refined model, we perform a memory step and a GIC step in the corresponding guest of the ideal model such that the GICD request is now pending in the GIC. This is possible due to the simulation invariant stating that all ideal GICs are always ready to receive a request as their inputs channels are empty.

During the execution of the handler the ideal model is stalled until the GIC step that produces the reply is occuring on the handler. We perform the same transition on the ideal GIC but also schedule a memory step in order to preserve the simulation invariant on the GIC output channel. Afterwards the ideal model is again stalled until the final step of the handler where execution returns to the guest. The ideal step that receives the GIC reply is scheduled on the ideal model such that read result is injected into the resulting guest configuration.

For the simulation of the emulated request the stepping is similar since the emulated ideal steps also need to preserve the simulation invariant on the GIC input and output channels.

The simulation of the ideal model by the refined model then follows a similar approach. While the request is outstanding in the ideal GIC, the corresponding refined core is still in the entry point of the GICD virtualization handler. When the reply is produced in the ideal model, the complete handler is executed until its last step, including all the necessary MMU, memory, and GIC steps.

Note that in the refined model the handler needs to acquire the lock on the

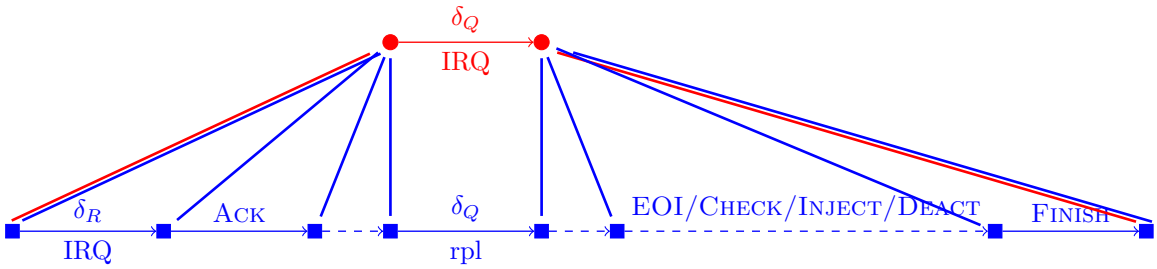

Figure 16: Bisimulation of the injection of a peripheral or an inter-processor interrupt. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. Dashed arrows subsume several steps of the MMU, GIC, and the handler.

distributor. As we schedule accesses to the refined model interrupt controller in an atomic fashion, we can establish another simulation invariant stating that in the refined model the lock is always free. This allows the refined model to simulate arbitrarily interleaved accesses to different GICs in the ideal model. Observe that this invariant obviously only holds when simulating the ideal model by the refined model. In the other direction it does not hold, but this is precisely the nature of a simulation invariant: it only holds because of the way we schedule the refined model in the simulation proof.

Observe as well again that the asymmetries in the coupling for both direction solely occur because of the buffering of messages in the message channels to and from the GIC. If the delivery from the MMU output into the GIC consisted of only a single step, the coupling would be symmetric.

4.2.8 IRQ Handler

Interrupt delivery to the core interface is performed as a single step by the GIC in the ideal model. On the contrary, it requires the execution of the complete IRQ reception handler in the refined model, as shown in Fig. 16. Note that the ideal step can by construction only be performed if the corresponding core has no outstanding memory requests in the system. Then also the refined core is ready to take a pending peripheral interrupt or an SGI. The handler is then taking its steps to inject the interrupt as a pending virtual interrupt into a core of the guest.

The particularity here is the question when to step the ideal model during the execution of the IRQ reception handler. Intuitively, one would probably choose to step the ideal GIC simultaneously with the refined GIC step that updates the list register to inject the virtual interrupt. However, in order to make a peripheral interrupt pending in the ideal model, it must still be raised by the peripheral at the time we step the ideal GIC. Since the hypervisor has no control over how a guest configures its peripherals, it cannot be guaranteed that this condition is met at the moment the virtual interrupt is injected into

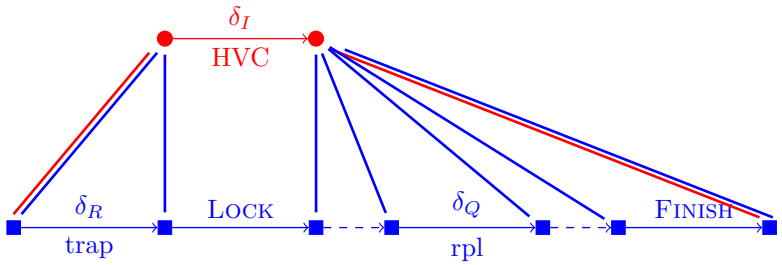

Figure 17: Bisimulation of a successful IGC notification request. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. Dashed arrows subsume several steps of the MMU and the GIC.

the guest's core.

Therefore we perform the ideal GIC step as soon as possible, i.e., at the earliest point during the handler execution where we know the peripheral interrupt and we have determined that it will eventually be injected into the guest. This is the case after the GIC produces the reply for the read of the acknowledgement register. The GIC semantics guarantee that any peripheral interrupt mentioned in the reply from the GIC is still raised by the corresponding peripheral. If the interrupt is not already pending in the virtual interface for the current core, it will eventually be injected, thus we can safely step the ideal GIC already.

All the remaining steps of the handler are then coupled with the resulting state in the ideal after the GIC made the interrupt pending at the ideal core's CPU interface. We need to define the coupling of the refined model virtual interrupts with the pending physical interrupts of the ideal model in such a way that it takes into account soon-to-be-pending interrupts. After the virtual interrupt is finally injected, the virtual and physical interrupts of both models are in sync again directly.

4.2.9 IGC Requests

In the ideal model, guest use a hypercall mechanism to request notification interrupts for a given IGC channel. The semantics of the hypercall in the ideal core transition relation creates such a notification interrupt (in the successful case) by raising the corresponding send-IGC flag. In the refined model, on the other hand, the hypervisor is invoked and it has to request an inter-processor interrupt (SGI) from the GIC distributor.

In the simulation of the refined by the ideal model, we perform the hypercall step of the corresponding core as soon as the hypervisor locks the message box bit for the requested channel, as depicted in Fig. 17. Note that the SGI is only raised later, however in the other direction of simulation both the interrupt lock bit and the SGI are activated simultaneously, as the complete handler and corresponding GIC steps are executed as one atomic block. Note that in the simulation of the ideal by the refined model, the GIC distributor is always ac-

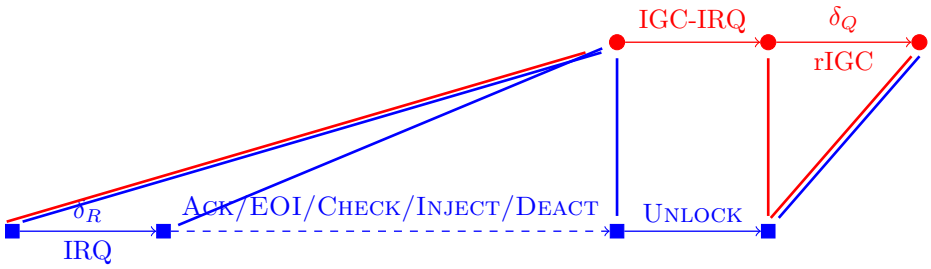

Figure 18: Bisimulation of the reception and injection of an IGC notification interrupt. Refined steps and their coupling are coloured blue, ideal steps and their coupling are cloured red. Dashed arrows subsume several steps of the MMU, GIC, and the handler.

cessible, therefore the request cannot block due to another request being present in the GIC input channel.

4.2.10 IGC Reception

The IGC notification reception handler is similar to the IRQ reception handler, however the coupling is quite different. The delivery of IGC notification interrupts to the targeted core in the ideal model happens in two steps: the inter-guest interrupt step which raises the receive-IGC flag of the channel for that core and a guest-local GIC step that makes the corresponding interrupt pending at the core's CPU interrupt interface.

Figure 18 shows that that both steps are executed in one block in the simulation of the refined by the ideal model, when the hypervisor finishes the reception handler by unlocking the IGC message box entry for the particular channel. In the other direction of simulation this step is coupled with the inter-guest interrupt step and the following ideal GIC step is coupled with a stuttering step of the refined model. Observe that this is possible, as in the refined model the corresponding virtual interrupt is already pending at the core.

The "delayed" stepping of the ideal model is necessary because as soon as the send-IGC flag in the ideal model is lowered a new notification request may be attempted by the sending core. Such a request can only be simulated by the refined model if the message box is empty, therefore the inter-guest interrupt step in the ideal model may only be performed when the message box is cleared.

On the other hand the message box may not be cleared before the inter-processor interrupt that implemented the notification is deactivated. Otherwise a new notification request may be deactivated by a "stale" deactivation request, breaking software conditions imposed by the GIC. Moreover, the message box should also not be cleared before the virtual interrupt is injected into the ideal core, so that the receive-IGC flag is coupled with the pending virtual interrupt. These considerations explain the order in which the IGC reception handler performs its different operations.

Note, that the stepping of the ideal model is the same in both the successful and unsuccessful injection case. If the notification interrupt could not be injected because the corresponding virtual interrupt is already pending at the targeted core, the inter-guest interrupt and ideal GIC steps are benign, as they simply lower both the send- and receive-IGC flags but leave the IGC interrupt pending at the core.

4.3 Auxiliary Definitions

In order to formalize these notions concisely, we need to introduce some auxiliary notation. First of all in order to reference outstanding requests within cores, GIC, memory, and peripherals, we also introduce history variables *curr* (in addition to the ones defined earlier) to each component in the ideal and refined model, recording the outstanding memory request for cores ($curr \in \mathbb{R}$), outstanding received requests for memory and GIC ($curr \subset \mathbb{R} \times \mathbb{S}$), as well as the last sent DMA request and outstanding received memory-mapped I/O requests for peripherals ($curr \subset \mathbb{R} \times \mathbb{S} \cup \{\epsilon\mathbb{R}\}$). The semantics for these history variables is simple. When ever a request is sent, or received respectively, it is saved in *curr*. Whenever a reply is returned for an outstanding request it is removed from *curr*. We omit a formal definition here.

Now we define a number of shorthands to collect requests and replies for a core c and peripheral p present in different parts of a refined model state M_R.

$$
\begin{aligned}
CR^c(M_R) &= \{M_R.c2u(c)\} \setminus \{\text{None}\} \\
CQ^c(M_R) &= \{M_R.u2c(c)\} \setminus \{\text{None}\} \\
UR^c(M_R) &= \{M_R.u2m(c)\} \setminus \{\text{None}\} \\
UQ^c(M_R) &= \{M_R.u2c(c)\} \setminus \{\text{None}\} \\
MR^c(M_R) &= \{r \mid (r, \text{C } c) \in M_R.m.curr\} \\
PR^c(M_R) &= \{r \mid \exists p.\ (r, \text{C } c) \in \{M_R.v2p(p)\} \cup M_R.P.st(p).curr\} \\
PQ^c(M_R) &= \{q \mid \exists p.\ M_R.p2v(p) = (q, \text{C } c)\} \\
PR^p(M_R) &= \{r \mid \exists c.\ (r, \text{C } c) \in \{M_R.v2p(p)\} \cup M_R.P.st(p).curr\} \\
PQ^p(M_R) &= \{q \mid \exists c.\ M_R.p2v(p) = (q, \text{C } c)\} \\
GR^c(M_R) &= \{r \mid (r, \text{C } c) \in \{M_R.v2p(np)\} \cup M_R.GIC.curr\} \\
GQ^c(M_R) &= \{q \mid M_R.p2v(np) = (q, \text{C } c)\} \\
SR^p(M_R) &= (\{M_R.p2v(p), M_R.v2m(p)\} \cap \mathbb{R}) \setminus \{\text{None}\} \\
SQ^p(M_R) &= (\{M_R.m2v(p), M_R.v2p(p)\} \cap \mathbb{Q}) \setminus \{\text{None}\}
\end{aligned}
$$

Note that we do not include the current (DMA) requests of the cores, MMUs, and peripherals here. These will be coupled with the messages in the sets above by an implementation invariant later. For the request and replies for core c on the way to and from the GIC we define:

$$
\begin{aligned}
C2GR^c(M_R) &= CR^c(M_R) \cup UR^c(M_R) \cup GR^c(M_R) \\
G2CQ^c(M_R) &= GQ^c(M_R) \cup UQ^c(M_R) \cup CQ^c(M_R)
\end{aligned}
$$

We allow sets $\mathrm{x}R^{\mathrm{y}}(M_R)$ and $\mathrm{x}Q^{\mathrm{y}}(M_R)$ to be abbreviated by $\mathrm{x}R^{\mathrm{y}}_R$ and $\mathrm{x}Q^{\mathrm{y}}_R$. Moreover we define similar shorthands for the ideal model.

$$
\begin{aligned}
CR^c_I &= \{M_I.G(\gamma(c)).c2u(\kappa(c))\} \setminus \{\text{None}\} \\
CQ^c_I &= \{M_I.G(\gamma(c)).u2c(\kappa(c))\} \setminus \{\text{None}\} \\
PR^c_I &= \{r \mid \exists p.\ (r, \mathrm{C}\ \kappa(c)) \in \{M_I.G(\gamma(c)).v2p(p)\} \cup M_I.G(\gamma(c)).P.st(p).curr\} \\
PQ^c_I &= \{q \mid \exists p.\ M_I.G(\gamma(c)).p2v(p) = (q, \mathrm{C}\ \kappa(c))\} \\
PR^p_I &= \{r \mid \exists c.\ (r, \mathrm{C}\ c) \in \{M_I.G(\gamma(p)).v2p(\varphi(p))\} \cup M_I.G(\gamma(p)).P.st(\varphi(p)).curr\} \\
PQ^p_I &= \{q \mid \exists c.\ M_I.G(\gamma(c)).p2v(p) = (q, \mathrm{C}\ c)\} \\
GR^c_I &= \{r \mid (r, \mathrm{C}\ \kappa(c)) \in \{M_I.G(\gamma(c)).v2p(np_{\gamma(c)})\} \cup M_I.G(\gamma(c)).GIC.curr\} \\
GQ^c_I &= \{q \mid M_I.G(\gamma(c)).p2v(np_{\gamma(c)}) = (q, \mathrm{C}\ \kappa(c))\}
\end{aligned}
$$

The context of a guest core on hypervisor entry is encoded completely in registers. We define the function *ectx* which takes a refined core state $C \in \mathcal{C}$ and reconstructs the guest context from it. We use predicate $sc : \mathcal{C} \to \mathbb{B}$ to determine whether the hypervisor was entered due to a system call, i.e., an SMC or HVC instruction, from the guest. The decision is based on the exception status register but we omit further details here for brevity.

$$
\begin{aligned}
ectx(C).ps &= PS(C.spr(SPSR\_EL2)) \\
ectx(C).pc &= \begin{cases} C.spr(ELR\_EL2) -_{64} 4_{64} & : \quad sc(C) \\ C.spr(ELR\_EL2) & : \quad \text{otherwise} \end{cases} \\
ectx(C).gpr &= C.gpr
\end{aligned}
$$

While a hypervisor handler is in an intermediate state, the guest context is stored in hypervisor memory. However, when coupling this saved context with the ideal guest core context there is a special case for the IGC notification request handler. When the hypervisor sends the request to activate the corresponding SGI to the GIC it does not immediately return to the guest but waits until it receives the write confirmation reply from the GIC. In between these to steps the GIC activates the SGI to the receiver of the IGC notification.

In the ideal model the initial refined model step that locks the IGC message box and requests the SGI corresponds to raising the *sIGC* flag for the corresponding channel, however this syscall steps also forwards the program counter to the next instruction, the same value that is already stored in the saved context in case of system calls. Thus in the bisimulation relation for the core state we need to distinguish whether the handler already took effect or not. To this end we define the following predicate for core c and refined state M_R.

$$done_{IGC}(M_R, c) \equiv M_R.C(c).pc = a_{snsgi}$$

Then we introduce an extended system call predicate to denote all other cases where the context needs to be coupled during a system call.

$$sc_c(M_R) \equiv sc(M_R.C(c)) \wedge \neg done_{IGC}(M_R, c)$$

For converting memory-mapped I/O messages (to/from GIC and other peripherals) from the refined to those in the ideal model we extend bijective translation functions $\theta : \mathbb{B}^{36} \rightleftharpoons \mathbb{B}^{36}$ to be applicable to requests and replies $x \in \mathbb{R} \cup \mathbb{Q}$. The intuitive idea is that $\theta(x)$ takes any address $a \in \mathbb{B}^{48}$ from x (if present) and replaces it with $\theta(a[47:12]) \circ a[11:0]$, returning the translated request or reply. We omit a detailed definition here.

Based on the address translation of requests and replies we define the conversion of the messages between peripherals and the SMMU in the refined model, or memory in the ideal model respectively. For $msg \in \mathbb{R} \cup \mathbb{Q} \cup (\mathbb{R} \times \mathbb{S}) \cup (\mathbb{Q} \times \mathbb{S})$ to or from peripherals we define the following function that converts memory-mapped I/O requests or replies from or to the cores.

$$conv_{IO}(msg) = \begin{cases} msg & : \quad msg \in \mathbb{R} \cup \mathbb{Q} \\ (\theta^{-1}_{\gamma(c)}(r), \mathrm{C}\ \kappa(c)) & : \quad msg = (r, \mathrm{C}\ c) \in (\mathbb{R} \cup \mathbb{Q}) \times \mathbb{S} \end{cases}$$

The conversion is ignoring DMA requests and replies and just targets memory-mapped I/O requests. In particular it translates the contained physical address to an intermediate physical one according to the inverse translation function for the guest that owns the sending core, and it adjusts the sender identifier to the right core index within that guest. Note that the conversion is applied also to the channels between GIC and memory, resembling the MMU channel conversion defined above.

Messages between cores and GIC are translated similary. However we also need to mix in effects of the GICD virtualization. The definition becomes complicated since in most of the cases we can derive the contexts of the refined message channels from the ideal ones, but there are two cases where this is not possible, namely for memory faults upon GICD guest access and non-emulated read replies from GIC distributor. In order to distinguish read and write requests $r \in \mathbb{R}$ and replies $q \in \mathbb{Q}$ to certain address regions $A \subseteq \mathbb{B}^{36}$, we introduce the following shorthands where $o \in \mathbb{B}^{12}$ is the offset within a page and $d \in \mathbb{N}$ the size of an access. Predicate $bchk_d(a, A)$ checks if a d-byte access to address $a \in \mathbb{B}^{48}$ is still within the upper bound of set A.

$$\begin{aligned}
bchk_d(a, A) &\equiv \mathrm{bin}_{48}(\langle a \rangle + d)[47:12] \in A \\
Rreq(r, A) &\equiv \exists a \in A, o, d.\ r = \mathrm{R}\ (a \circ o)\ d \wedge bchk_d(a \circ o, A) \\
Wreq(r, A) &\equiv \exists a \in A, o, d, v \in \mathbb{B}^{8d}.\ r = \mathrm{W}\ (a \circ o)\ d\ v \wedge bchk_d(a \circ o, A) \\
req(r, A) &\equiv Rreq(r, A) \vee Wreq(r, A) \\
PTreq(q, A) &\equiv \exists a \in A, o.\ q = \mathrm{PTW}\ (a \circ o) \wedge bchk_8(a \circ o, A) \\
Rrpl(q, A) &\equiv \exists a \in A, o, d, v \in \mathbb{B}^{8d}.\ q = rplR\ (a \circ o)\ v \wedge bchk_d(a \circ o, A) \\
Wrpl(q, A) &\equiv \exists a \in A, o, d, v \in \mathbb{B}^{8d}.\ q = rplW\ (a \circ o) \wedge bchk_d(a \circ o, A) \\
rpl(q, A) &\equiv Rrpl(q, A) \vee Wrpl(q, A) \\
PTrpl(q, A) &\equiv \exists a \in A, o, v \in \mathbb{B}^{64}.\ q = rplPTW\ (a \circ o)\ v \wedge bchk_8(a \circ o, A)
\end{aligned}$$

Another special case occurs, when the guest is requesting a read or write to a GICD register in the ideal model but the corresponding second-stage fault

has not reached the hypervisor yet in the refined one. For request $r \in \mathbb{R}$ and corresponding core state $C \in \mathcal{C}$ this *pre-GICD-handler* case is identified by the following predicate.

$$pgh_g(C, r) \equiv req(r, \{a^g_{GICD}\}) \wedge C.ps.mode < 2$$

To support the definition of the coupling of message boxes in case of accesses to the GIC, we introduce θ^{GICD}_g which is the address translation function that only translates accesses to the GIC CPU distributor registers. It is applied to couple ideal messages in the message channel between cores and memory with refined messages in the channel between core and second-stage MMU.

$$\theta^{GICD}_g(a) = \begin{cases} \theta_g(a) & : \quad a \in A^g_{GIC} \\ a & : \quad \text{otherwise} \end{cases}$$

For coupling a guests outstanding interrupts with the outstanding virtual interrupts in the refined model, we need to be able to extract and translate the relevant interrupts for a given guest g. To this end we define function $xtvi_g : 2^{VIRQ} \to 2^{IRQ_g}$ as follows.

$$\begin{aligned} xtvi_g(V) &= \{q \in PIRQ_g \cup IGCQ_g \mid virq_q \in V\} \\ &\cup \{\mathrm{sgi}^{id}_{\kappa(c),\kappa(c')} \mid \gamma(c) = \gamma(c') = g \wedge virq_{\mathrm{sgi}^{id}_{c,c'}} \in V\} \end{aligned}$$

However we cannot simply couple a core's outstanding interrupts in the ideal model directly with the extracted virtual interrupts in the refined model for that core. The reason for this is the scenario explained earlier, where other cores in the same guest disable a peripheral interrupt while the hypervisor is handling it but before it has updated the virtual interrupt interface.

Therefore, in addition to the already present virtual interrupts in the system, we also need to add new virtual interrupts that will be injected by the hypervisor shortly. To be precise these new virtual interrupts are visible in the coupling relation during the IRQ handler is running on a given core from the state after the GIC produces the reply to a read of the acknowledgement register (and the current interrupt is allowed to be eventually injected) until the GIC step that actually updates the virtual interrupt list register for the handled peripheral interrupt.

In order to formalize this notion, we first introduce notation for accessing the list register for a peripheral interrupt $q \in PIRQ$, that guest $g \in \mathbb{N}_{ng}$ is allowed to receive in refined model state $M_R \in \mathcal{M}_R$. We need a function

$$r_{GIC} : \mathbb{B}^{48} \times \mathbb{N} \rightharpoonup GICC \cup GICD \cup GICH$$

that maps a physical address and an access size to the name to the corresponding GIC register (or $\perp$ if the input is invalid). Then

$$lr^c_g(M_R, q) = M_R.GIC.gich(r_{GIC}(a^g_{LR}(q), 4), c)$$

gives back the current content of the list register allocated for interrupt q on guest g. Now we define a predicate to determine whether a peripheral interrupt q can currently be injected to core c as a pending or active-and-pending interrupt, distinguished by flag $pend \in \mathbb{B}$.

$$inj_g^c(M_R, q, pend) \equiv q \in [16:1019] \wedge lr_g^c(M_R, q)[29:28] = \begin{cases} 00 & : \quad pend \\ 10 & : \quad \neg pend \end{cases}$$

As a reminder, peripheral interrupts have IDs 16 to 1019, bits 29-28 of the list register denote that an interrupt is currently inactive by bit code 00, and that it is active (and not pending) by code 10.

Now a new peripheral interrupt q is about to be injected into core c in state M_R if 1. The hypervisor is in a state "before" $\sigma_{\text{IRQ}}^{await,v,g}$, signified by the PC pointing to a_{irqaw}, where the GIC has replied to the read request of the $GICC\_AIAR$ register, i.e., there is a corresponding reply in one of the channels from GIC to core c, and the returned value states an interrupt that can be injected, or 2. the hypervisor is in a state before or in $\sigma_{\text{IRQ}}^{cwait,c,l}$, i.e., the PC is pointing to a_{irqcw}, and the last interrupt received, as recorded in the corresponding data structure after step IRQ-CHECK, can be injected, or 3. the hypervisor is before state $\sigma_{\text{IRQ}}^{iwait}$ and the list register of the virtual interrupt interface has not yet been updated to inject the interrupt.We capture these cases in the following predicate, where $pend \in \mathbb{B}$ distinguishes again whether the interrupt is injected in pending or active-and-pending state.

$$\begin{aligned} injnu_g^c(M_R, q, pend) \quad = \quad & M_R.C(c).pc \in \{a_{irqaw}, a_{irqcw}, a_{irqiw}\} \\ \wedge \quad & (\exists v \in \mathbb{B}^{22}.\ rplR\ a_{IAR}\ (v \circ \text{bin}_{10}(q)) \in G2CQ_c(M_R)\ \vee \\ & \quad q = lirq(M_R.m(a_{lirq}), c) \wedge M_R.C(c).pc \neq a_{irqaw}) \\ \wedge \quad & inj_g^c(M_R, q, pend) \\ \wedge \quad & \nexists v \in \mathbb{B}^{32}.\ rplW\ a_{LR}^g(q) \in G2CQ_c(M_R) \end{aligned}$$

Hence we can define the set of new virtual interrupts to be injected shortly into core c of guest g. Note that there is at most one such interrupt, as the hypervisor does not batch-process outstanding interrupts.

$$nuvi_g^c(M_R, pend) = \begin{cases} \{virq_q\} & : \quad injnu_g^c(M_R, q, pend) \\ \emptyset & : \quad \text{otherwise} \end{cases}$$

For the sending of IGC notification interrupts, there is a similar situation where the ideal model is stepped, setting the send-IGC flag and the target for the interrupt, but the corresponding inter-processor interrupt has not yet been made pending by the GIC. We denote this situation by the following predicates for IGC notifications sent from refined core c to c'.

$$\begin{aligned} nuigc_{c,c'}(M_R) \quad \equiv \quad & mode_R^c = 2 \wedge C_R^c.pc = a_{snsgi} \\ \wedge \quad & \exists q \in C2GR_R^c.\ q = \text{W}\ a_{SGIR}\ 4\ 0^8 \circ 0^{7-c'} 10^{c'} \circ 0^{12} 1^4 \end{aligned}$$

Moreover, we also need a way to extract the peripheral event sequences of each guest from the global event sequence in the refined model. For a sequence $es \in \mathcal{E}^*$ and a guest $g \in \mathbb{N}_{ng}$ this is achieved by the following recursive function.

$$es|_g = \begin{cases} \varepsilon & : \quad es = \varepsilon \\ e \circ (tl(es)|_g) & : \quad hd(es) = e \in \mathcal{E}_g \\ tl(es)|_g & : \quad \text{otherwise} \end{cases}$$

Finally, in order to obtain compact formulas, we use abbreviations like C_R^c for $M_R.C(c)$ and C_I^c for $M_I.G(\gamma(c)).C(\kappa(c))$ and similarly for all other components of the refined and ideal model. By ctx_R^c we denote $ctx(M_R.m(a_{ctx}), sc_c(M_R), c)$ and $mode_R^c = C_R^c.ps.mode$. We use function $hv_c : \mathcal{M}_R \to \mathbb{H}$ to extract the hypervisor state for a given core c. At last, let

$$Q_c = Q(c).pend \cup Q(c).act \cup Q(c).ap$$

denote the union of pending, active, and active-and-pending interrupts of core c for a given interrupt state.

4.4 Bisimulation Relation

Now, the bisimulation relation between the ideal and refined model is denoted by the symbol $\sim \subseteq \mathcal{M}_I \times \mathcal{M}_R$. For ideal system states we use identifier M_I while we use M_R for refined states. Following the ideas sketched above we define the bisimulation relation for all components of the ideal model.

The relation $M_I \sim M_R$ holds iff all of the following statements hold. Note that the coupling ignores the history variables *curr* unless they are explicitly mentioned.

Context The ideal guest register context is equal to the refined register context while the guest is running in EL1 or EL0, to the exception context at hypervisor entry points, or with the saved context in intermediate hypervisor states with the special cases for system calls.

$$\begin{aligned} \forall c \in \mathbb{N}_{nc} \quad . \quad & C_I^c.active = C_R^c.active \wedge C_R^c.init \\ \wedge \quad & C_I^c.active \Rightarrow \forall \mathrm{x} \in \{ps, pc, gpr\}.\ C_I^c.\mathrm{x} = \\ & \qquad \begin{cases} C_R^c.\mathrm{x} & : \quad mode_R^c < 2 \\ ectx(C_R^c).\mathrm{x} & : \quad mode_R^c = 2 \wedge C_R^c.pc \in A_{hve} \\ ctx_R^c.\mathrm{x} & : \quad \text{otherwise} \end{cases} \\ \wedge \quad & C_I^c.active \Rightarrow \forall r \in SPR_{\mathbb{G}}.\ C_I^c.spr(r) = M_R.C.spr(r) \end{aligned}$$

Observe that for each core the special purpose registers for EL1 are always coupled directly.

Memory For each guest, its memory in the ideal model is equal to the corresponding guest memory in the refined model, where pages are mapped according to the guest's translation function.

$$\forall g \in \mathbb{N}_{ng}, a \in \mathbb{A}_g \setminus A_P^g.\ M_I.G(g).m(a) = M_R.m(\theta_g(a))$$

We exclude the peripheral region $A_P^g = \bigcup_{p=0}^{np} A_P^{g,p}$ from this region as memory is not updated for memory-mapped I/O accesses to it.

PERIPH Peripherals are mapped one-to-one using the projection functions.

$$\forall p \in \mathbb{N}_{np}.\ M_I.G(\gamma(p)).P.st(\varphi(p)) = M_R.P.st(p)$$

CORE2MEM For a request stored in the ideal model message channel from a core to memory there are several cases to handle.

1. A request to the GIC distributor — Then we require that the GICD handler is not yet running and there are three cases:
   (a) There is a corresponding guest request in the refined model channel from core to MMU — then the two requests are identical.
   (b) The second stage MMU is busy — then it is currently translating the memory request and the input channel in the refined model is empty.
   (c) A second-stage fault has been issued in the refined model, present in the channel from MMU to the core — then the refined model channel from core to MMU is empty.
2. A request to any other address — then the request is either contained identically in the refined model channel from core to MMU, or its currently being processed by the second-stage MMU (it is the current request *curr* in the walk component of the MMU), or its address-translated version is contained in the channel from MMU to memory for that core.

We summarize these cases in the following statement.

$$\begin{aligned}
&\forall g \in \mathbb{N}_{ng}, c \in \mathbb{N}_{nc}, r \in \mathbb{R}.\ \gamma(c) = g \wedge cif_I^c = r \neq \text{None} \ \Rightarrow \\
&\qquad c2u_R^c = \begin{cases} \text{None} & : \quad pgh_g(C_R^c, r) \wedge u2c_R^c = \text{F}\ a_{GICD} \\ & \qquad\quad \vee\ mmu_R^c.active \wedge mmu_R^c.walk.curr = r \\ r & : \quad \text{otherwise} \end{cases} \\
&\wedge \quad u2m_R^c \in \{\text{None}, \text{PTW}\ a, \theta_g(r) \mid a[47:12] \in A_{PT}(g)\}
\end{aligned}$$

Note, that any corresponding message is exclusively in one of the two refined model channels on the way to memory. This fact will be covered by an implementation invariant later.

CORE2MMU2MEM In order to complete the coupling for the channels from core to memory, we need to state when the ideal model channel contains a message or is empty. The coupling of the values can then be determined from the statement above. We require the following conditions for the outgoing channel belonging to core c:

1. When core c is running in guest mode then the ideal channel from core to memory is empty iff both the refined channel from core to second-stage MMU and the one from MMU to memory are empty and the MMU is idle or it has sent its final request to memory. This final request, i.e., the actual physical memory request into which it has translated the intermediate-physical memory request, is recorded in *walk.final*.

$$\begin{gathered}\forall c \in \mathbb{N}_{nc}.\ mode_R^c < 2 \Rightarrow \\ cif_I^c = \text{None} \\ \Leftrightarrow \\ c2u_R^c = u2m_R^c = \text{None} \\ \wedge\ (mmu_R^c.walk = \bot \vee mmu_R^c.walk.final \neq \text{None})\end{gathered}$$

   Note that the second-stage MMU is always active during the execution of the guest.

2. When the second-stage MMU is busy translating a request, its current request is equal to the request in the ideal model channel from core to memory. An MMU is translating a request while predicate $trans(mmu) \equiv mmu.active \wedge mmu.walk \neq \bot \wedge \neg mmu.walk.complete$ holds.

$$trans(mmu_R^c) \ \Rightarrow\ cif_I^c = mmu_R^c.walk.curr$$

3. When core c is running the hypervisor, the corresponding ideal model channel is empty.

$$\forall c \in \mathbb{N}_{nc}.\ mode_R^c \geq 2 \ \Rightarrow\ cif_I^c = \text{None}$$

4. However, if the GICD virtualization handler is running and the GIC request was not processed by the GIC or emulated yet, there is a correspondning request in outstanding in the ideal GIC.

$$\begin{gathered}\forall r.\ mode_R^c = 2 \wedge hv_c(M_R) \in \sigma_{\text{GICD}}^{entry,g,r} \ \Rightarrow \\ (r, \text{C}\ \kappa(c)) \in M_I.G(\gamma(c)).GIC.curr\end{gathered}$$

$$\begin{gathered}\forall r'.\ mode_R^c = 2 \wedge C_R^c.pc = a_{gdw} \wedge r' \in CR_R^c \cup UR_R^c \ \Rightarrow \\ \exists r.\ (r, \text{C}\ \kappa(c)) \in M_I.G(\gamma(c)).GIC.curr \wedge r' = \rho_{\gamma(c)}(\theta_{\gamma(c)}(r))\end{gathered}$$

5. While an access to the GIC virtual interrupt interface is in the MMU output channel, the corresponding request is either in the ideal core output channel or outstanding at the GIC. For all $r = u2m_R^c$ and $g = \gamma(c)$:

$$req(r, A_{GICV}) \ \Rightarrow\ cif_I^c = r \vee \theta_g^{-1}(r) \in GR_I^c$$

Note, that during hypervisor mode only requests to the GIC are sent. These are mostly invisible in the ideal model except for virtualized accesses to the GIC distributor. In the bisimulation we will step the models such that corresponding ideal requests to GIC distributor registers are already outstanding in the GIC of the ideal model.

Similarly, guest accesses to the GIC CPU interface must be scheduled such that they are already moved into the GIC while they are contained in the MMU output channel of the refined model.

MEM2MMU2CORE For the coupling of the refined reply channels from Memory to the MMU and from there to the core with the corresponding ideal channel from memory to core, we distinguish the following cases on the content of the refined channel from MMU to core.

1. It contains a second-stage fault for the GIC distributor page — then the corresponding reply channel in the ideal model is empty.
2. It contains a reply from the GIC while the hypervisor is executing — then the ideal model channel contains a corresponding non-empty reply from the ideal GIC distributor.
3. It is empty but the corresponding core has entered the hypervisor MMU fault handler — then the corresponding memory fault is still present in the ideal model channel. The address that triggered the fault is $a^c_{hpf} = C^c_R.spr(HPFAR\_EL2)[39:4]$.
4. It is empty and the GICD virtualization handler is running in a state from the set $\sigma^{\mathit{fltr},q'}_{\mathrm{GICD}}$— then the ideal channel contains reply q' from the GIC distributor or it is empty, as the message may still be in the output channel of the GIC in the ideal model.
5. The guest is running and there is no second-stage fault for the GIC distributor page — then there are more cases:
   (a) The channel from MMU to core is not empty — then the ideal channel contains the same reply message.
   (b) The channel from MMU to core is empty and the MMU is in its final phase — then the ideal message is coupled with the reply channel from memory through the address translation function.
   (c) The channel from MMU to core is empty and the MMU is idle or in its translation phase — then the ideal message channel is empty.

These cases are subsumed in the following statement.

$$\forall g \in \mathbb{N}_{ng}, c \in \mathbb{N}_{nc}, q \in \mathbb{Q}.\ \gamma(c) = g \wedge u2c_R^c = q \ \Rightarrow$$
$$m2c_I^c \in \begin{cases} \{q\} & : \ mode_R^c < 2 \wedge q \notin \{\text{None}, \text{F } a_{GICD}^g\} \\ \{\theta_g^{-1}(m2u_R^c)\} & : \ q = \text{None} \wedge mmu_R^c.walk.complete \\ \{\text{F } a_{hpf}^c\} & : \ q = \text{None} \wedge hv_c(M_R) \in \sigma_{\text{MMUF}}^{entry,g} \\ \{\text{None}, q'\} & : \ q = \text{None} \wedge hv_c(M_R) \in \sigma_{\text{GICD}}^{fltr,q'} \\ \mathbb{Q} \setminus \{\text{None}\} & : \ mode_R^c = 2 \wedge C_R^c.pc = a_{gdw} \wedge rpl(q, A_{GICD}) \\ \{\text{None}\} & : \ \text{otherwise} \end{cases}$$

Observe that the cases above are disjoint since the second-stage MMU is idle while the core is executing hypervisor code and the output channel of the MMU to the core is empty while the MMU is busy.

To clarify the case of replies from the GIC distributor we add the following condition. When a GICD reply is contained in one of the refined reply channels for core c for which the hypervisor is waiting in the GICD virtualization handler, then the corresponding reply is contained in the input channel to the core in guest $g = \gamma(c)$.

$$\forall q \in UQ_R^c \cup CQ_R^c.\ mode_R^c = 2 \wedge C_R^c.pc = a_{gdw} \wedge rpl(q, A_{GICD}) \ \Rightarrow$$
$$\exists q'.\ q' \in CQ_I^c \wedge \begin{cases} q' = \rho_g(\theta_g^{-1}(q)) & : \ Rrpl(q, A_{GICD}) \wedge q \in UQ_R^c \\ q' = \rho_g(q) & : \ Rrpl(q, A_{GICD}) \wedge q \in CQ_R^c \\ \theta_g(\rho_g(q')) = q & : \ Wrpl(q, A_{GICD}) \wedge q \in UQ_R^c \\ \rho_g(q') = q & : \ \text{otherwise} \end{cases}$$

MEM2CORE To complete the coupling of memory replies, we also need to state how the refined reply channels are computed from the ideal one. It is sufficient to state what kind of replies are contained in them. We distinguish the following cases for all $g \in \mathbb{N}_{ng}$, $c \in \mathbb{N}_{nc}$, and $q \in \mathbb{Q}$ with $\gamma(c) = g$ and $m2c_I^c = q$:

1. The ideal channel contains a reply from the GIC distributor — then the corresponding core in the refined model is in the hypervisor state of the GICD virtualization handler before returning to the guest and the refined channels are empty. The reply corresponds to the reply that is virtualized by the handler. Formally, if $rpl(q, \{a_{GICD}^g\})$, then:

$$u2c_R^c = \text{None} \ \wedge \ m2u_R^c = \text{None} \ \wedge \ hv_c(M_R) \in \sigma_{\text{GICD}}^{fltr,q}$$

2. The ideal channel contains a memory fault and the core is in the MMU fault handler entry point or in the GICD handler — then both channels are empty and the address in the ideal fault message corresponds to the faulting address that is treated by the handler. If

there exists an address a such that $q = \mathrm{F}\ a$ and $hv_c(M_R) \in \sigma^{entry,g}_{\mathrm{MMUF}} \cup \sigma^{entry,g,r}_{\mathrm{GICD}}$, then:

$$u2c^c_R = \mathrm{None}\ \wedge\ m2u^c_R = \mathrm{None}$$

$$hv_c(M_R) \in \sigma^{entry,g,r}_{\mathrm{GICD}}\ \Rightarrow\ adr(r) = a$$

$$hv_c(M_R) \in \sigma^{entry,g}_{\mathrm{MMUF}}\ \Rightarrow$$
$$a = C^c_R.spr(HPFAR\_EL2)[39:4] \circ C^c_R.spr(FAR\_EL2)[11:0]$$

3. The ideal channel contains a memory fault but the core is running in guest mode — then the refined channel from MMU to core contains the same fault. The channel from memory to MMU is empty. If $q = \mathrm{F}\ a$ and $mode^c_R < 2$, then:

$$m2c^c_R = \mathrm{F}\ a \wedge m2u^c_R = \mathrm{None}$$

4. The ideal channel is empty but the hypervisor is running — then either refined channel could be empty or contain a reply from the GIC distributor or the GICC and GICH interfaces. Note that while the hypervisor is running, only GICD messages and memory faults may be present in the reply channel of the ideal model. If $q = \mathrm{None}$ and $mode^c_R \geq 2$, then:

$$(rpl(u2c^c_R, A_{GICC} \cup A_{GICD} \cup A_{GICH}) \vee u2c^c_R = \mathrm{None}) \wedge$$
$$(rpl(m2u^c_R, A_{GICC} \cup A_{GICD} \cup A_{GICH}) \vee m2u^c_R = \mathrm{None})$$

5. Any other case where the guest is running — then at least one of the channels is empty. Note that the channel from memory to second-stage MMU may contain replies for page table walks in this case.

$$u2c^c_R = \mathrm{None} \vee m2u^c_R = \mathrm{None}$$

MEM2GIC The coupling of the channels from memory to the GIC distringuishes three cases. For each core $c \in \mathbb{N}_{nc}$ and its corresponding ideal message channel $M_I.G(g).m2p(np_g) = (r', \mathrm{C}\ \kappa(c))$ with $\gamma(c) = g$ we have:

1. A message to the GIC distributor is present in the ideal model — then the hypervisor is in the entrypoint to the GICD virtualization handler for that message and the corresponding refined channel does not contain any message from the same core.

$$req(r', A^g_{GICD})\ \Rightarrow\ \nexists r.\ v2p^{np}_R = (r, \mathrm{C}\ c) \wedge hv_c(M_R) \in \sigma^{entry,g,r'}_{\mathrm{GICD}}$$

2. A message in the ideal model is present that is targeting the physical CPU interrupt interface — then the corresponding address translated message is still in the output channel of the corresponding second-stage MMU.

$$req(r', A^g_{GICC})\ \Rightarrow\ \nexists r.\ v2p^{np}_R = (r, \mathrm{C}\ c) \wedge \theta_g(r') \in UR^c_R$$

3. The ideal message channel does not contain a message for core c — then while the corresponding refined core is running in guest mode the refined GIC input channel is either empty or it contains a request for the GIC virtual interrupt interface.

$$\nexists r'.\ M_I.G(g).m2p(np_g) = (r', \mathrm{C}\ \kappa(c)) \wedge mode_R^c < 2\ \Rightarrow \\ \forall r.\ v2p_R^{np} = (r, \mathrm{C}\ c) \Rightarrow req(r, A_{GICV})$$

Note that these are the only possible cases for messages to the ideal GICs. The coupling in the other direction is similar. We have the following cases:

1. The hypervisor is in the entry point of the GICD handler — then either the corresponding ideal channel contains the request from the corresponding core or the message is outstanding in the ideal GIC.

$$hv_c(M_R) \in \sigma_{\mathrm{GICD}}^{entry,g,r}\ \Rightarrow\ r \in GR_I^c$$

2. The hypervisor is running in the GICD handler and waiting for the GIC reply that has not been produced yet — then the corresponding ideal channel does not contain any message from the corresponding core. Instead the message is outstanding in the GIC and coupled through the GIC message conversion and address translation functions.

$$mode_R^c = 2 \wedge C_R^c.pc = a_{gdw} \wedge G2CQ_R^c = \emptyset\ \Rightarrow \\ \nexists r'.\ M_I.G(\gamma(c)).v2p(np_g) = (r', \mathrm{C}\ \kappa(c)) \\ \wedge\ \exists r'.\ (r', \mathrm{C}\ \kappa(c)) \in M_I.G(\gamma(c)).GIC.curr \wedge r = \rho_{\gamma(c)}(\theta_{\gamma(c)}(r'))$$

3. The hypervisor is running in the GICD handler and waiting for the GIC reply after it has been produced — then neither the GIC nor its output channel contain contain any message from the corresponding core.

$$mode_R^c = 2 \wedge C_R^c.pc = a_{gdw} \wedge G2CQ_R^c \neq \emptyset\ \Rightarrow\ GR_I^c = \emptyset$$

   Instead the message is outstanding in the core's input channel as defined earlier.

4. The hypervisor is running in any other point of the hypervisor — then the corresponding ideal channel does not contain any message from the corresponding core.

$$mode_R^c = 2 \wedge hv_c(M_R) \notin \sigma_{\mathrm{GICD}}^{entry,g,r} \wedge C_R^c.pc \neq a_{gdw}\ \Rightarrow \\ \nexists r'.\ M_I.G(\gamma(c)).v2p(np_g) = (r', \mathrm{C}\ \kappa(c))$$

5. Core c is running in guest mode — For any request of core c present in the refined channel $v2p_R^{np} = (r, \mathrm{C}\ c)$, the corresponding ideal channel is empty and the same request subject to the address translation function is outstanding in the GIC.

$$\begin{aligned} mode_R^c < 2\ \ &\Rightarrow\ \ \nexists r'.\ M_I.G(g).m2p(np_g) = (r', \mathrm{C}\ \kappa(c)) \\ &\wedge\ \ (\theta_{\gamma(c)}^{-1}(r), c) \in M_I.G(\gamma(c)).GIC.curr \end{aligned}$$

GIC2MEM For the channels from GIC to memory we again distinguish the two directions of the coupling, mapping the ideal to the refined channel and vice versa. For messages in the ideal channel $M_I.G(\gamma(c)).pif(np_g) = (q', \mathrm{C}\ \kappa(c))$ from the core corresponding to refined core $c \in \mathbb{N}_{nc}$ with $g = \gamma(c)$ we have the following cases:

1. The channel contains a reply from the GIC distributor — then the refined model is in one of the hypervisor states $\sigma^{fltr,q'}_{\mathrm{GICD}}$ as in the simulation of GIC distributor steps by the refined model all corresponding handler and GIC steps are executed in one atomic block. Similarly, in the other direction the ideal model is stepped in a way such that the GIC reply channels are always empty, thus the following statement always holds vacuously.

$$rpl(q', A^g_{GICD}) \ \Rightarrow\ hv_c(M_R) \in \sigma^{fltr,q'}_{\mathrm{GICD}}$$

   Note that this is the only kind of reply messages visible in the ideal model while the hypervisor is running.

2. The channel contains a reply from the GIC CPU interface — then the corresponding core is running in guest mode and the refined output channel of the GIC does not contain a message for core c. Instead the corresponding reply message is stored in the channel from memory to the second-stage MMU of core c.

$$rpl(q', A^g_{GICC}) \ \Rightarrow\ GQ^c_R = \emptyset \wedge m2u^c_R = \theta_g(q')$$

3. The channel does not contain a message for core $\kappa(c)$ and core c is running in guest mode — then the refined GIC output channel either does not contain a reply for core c or it is a reply from the GIC virtual interface.

$$\nexists q'.\ M_I.G(\gamma(c)).pif(np_g) = (q', \mathrm{C}\ \kappa(c)) \wedge mode^c_R < 2\ \Rightarrow \\ \forall q \in GQ^c_R.\ rpl(q, A_{GICV})$$

When mapping the refined channel to the ideal one, we have the following cases for $M_R.p2v(np) = (q, \mathrm{C}\ c)$ and $g = \gamma(c)$.

1. The channel contains a reply from the GIC distributor for which the hypervisor is waiting on the corresponding core — then the corresponding reply is contained in the ideal channel from memory to the requesting core in the corresponding guest g.

$$mode^c_R = 2 \wedge C^c_R.pc = a_{gdw} \wedge rpl(q, A_{GICD})\ \Rightarrow \\ \exists q'.\ q' \in CQ^c_I \wedge \begin{cases} q' = \theta_g^{-1}(\rho_g(q)) & : \quad Rrpl(q, A_{GICD}) \\ \rho_g(\theta_g(q')) = q & : \quad \text{otherwise} \end{cases}$$

2. The refined channel contains an arbitrary message while on core c the hypervisor is in one of the states in $\sigma^{fltr,q'}_{\mathrm{GICD}}$ — then there the

corresponding GIC output channel of guest g either contains q' or it is empty.

$$mode_R^c = 2 \wedge hv_c(M_R) \in \sigma_{\text{GICD}}^{fltr,q'} \;\Rightarrow\; GQ_I^c \subseteq \{q'\}$$

3. The refined channel contains an arbitrary message while the hypervisor is in the entry state of the GICD virtualization handler — then the only possible reply in the ideal GIC output channel of guest g is a fault message.

$$mode_R^c = 2 \wedge hv_c(M_R) \notin \sigma_{\text{GICD}}^{entry,g,r} \;\Rightarrow\; GQ_I^c \subseteq \{\text{F } adr(r)\}$$

4. The refined channel contains an arbitrary message while the hypervisor is running on core c outside of the aformentioned states of the GICD virtualization handler — then there is no reply for the corresponding core in the GIC output channel of guest g.

$$mode_R^c = 2 \wedge C_R^c.pc \neq a_{gdw} \wedge hv_c(M_R) \notin \sigma_{\text{GICD}}^{fltr,q'} \cup \sigma_{\text{GICD}}^{entry,g,r} \;\Rightarrow\; GQ_I^c = \emptyset$$

5. The guest is running and the refined channel contains a reply for core c from the virtual GIC interface — then the corresponding reply from the ideal GIC CPU interface is present in the guest's corresponding core input channel and the ideal GIC output channel does not contain a message for that core.

$$mode_R^c < 2 \wedge rpl(q, A_{GICV}) \;\Rightarrow\; \theta_g^{-1}(q) \in CQ_I^c \wedge GQ_I^c = \emptyset$$

6. The guest is running and there are no replies for core c — then all replies for the corresponding core in guest g come from the guest's CPU interrupt interface.

$$mode_R^c < 2 \wedge GQ_R^c = \emptyset \;\Rightarrow\; \forall q' \in GQ_I^c.\; rpl(q', A_{GICC}^g)$$

PerChan The channels between memory and peripherals are mapped from the refined to the ideal model via the address translation function, adjusting sender IDs using the core projection function. We need to make a case split on the state of the corresponging SMMU for DMA messages to determine which component is currently coupled with the ideal channel. For all $p \in \mathbb{N}_{np}$, let

$$pif_I^p = M_I.G(\gamma(p)).pif(\varphi(p)) \quad \text{and} \quad m2p_I^p = M_I.G(\gamma(p)).m2p(\varphi(p))\,,$$

then we have:

$$pif_I^p \;=\; \begin{cases} smmu_R^p.walk.curr & : \quad trans(smmu_R^p) \\ \theta_{\gamma(p)}^{-1}(v2m_R^p) & : \quad smmu_R^p.walk.complete \\ conv_{IO}(p2v_R^p) & : \quad \text{otherwise} \end{cases}$$

$$m2p_I^p \;=\; \begin{cases} \theta_{\gamma(p)}^{-1}(m2v_R^p) & : \quad smmu_R^p.walk.complete \\ conv_{IO}(v2p^p) & : \quad \text{otherwise} \end{cases}$$

PerIRQ The same peripheral interrupts are active in both models. For each peripheral its interrupt state is mapped to the corresponding guest using the peripheral projection function.

$$\forall p \in \mathbb{N}_{np}.\ M_I.G(\gamma(p)).PI(\varphi(p)) = M_R.PI(p)$$

VirtIRQ Similarly the (current and soon to be injected) virtual interrupts are mapped for each refined core to the corresponding guest core's interrupt state. We use function $xtvi_g$ to extract the pending, active, and active-and-pending interrupts for guest g, while also translating core indices for SGIs. For all $c \in \mathbb{N}_{nc}$, with $IGCQ_I^c = \{\text{igc}_s \mid M_I.G(\gamma(c)).rIGC(s, \gamma(c)) = 1\}$ we define:

$$\begin{aligned} Q_I^c.pend &= xtvi_{\gamma(c)}(M_R.VI(c).pend \cup nuvi^c_{\gamma(c)}(M_R, 1)) \setminus IGCQ_I^c \\ Q_I^c.act &= xtvi_{\gamma(c)}(M_R.VI(c).act \setminus nuvi^c_{\gamma(c)}(M_R, 0)) \\ Q_I^c.ap &= xtvi_{\gamma(c)}(M_R.VI(c).ap \cup nuvi^c_{\gamma(c)}(M_R, 0)) \end{aligned}$$

Note that soon to be pending virtual interrupts are added to the pending set and similarly for active-and-pending interrupts. However, soon to become active-and-pending interrupts are removed from the active set according to the GIC semantics that demands these sets to be disjoint. In addition we need to exclude IGC interrupts that are already pending in the refined model but still await delivery by the GIC in the ideal model.

SndIGC For each IGC channel, the send-IGC notification interrupt flag of the sending guest is directly coupled with the interrupt lock bit in the IGC message box data structure of the hypervisor. The corresponding inter-processor interrupt targets the same core as in the ideal model. However, this is only the case after initialization, therefore we introduce an $init \in \mathbb{B}$ flag in each guest state, such that $start(M_I)$ implies $M_I.G(g).init = 1$ for all guests $g \in \mathbb{N}_{ng}$. Then the coupling is defined as follows:

$$\begin{gathered} \forall g, c', g' \in \mathbb{N}_{ng}, s \in \mathbb{N}_{ns}. \\ cpol(s) = (g, g') \wedge \gamma(c') = g' \wedge M_I.G(g).init = 0 \ \Rightarrow \\ M_I.G(g).sIGC(s) = 1 \ \Leftrightarrow \ mbox(M_R, g, g') = 1 \\ \wedge \\ \text{sgi}^{15}_{c,c'} \in M_R.Q_{c'} \vee nuigc_{c,c'}(M_R) \ \Leftrightarrow \ M_I.tgt(s) = \kappa(c') \end{gathered}$$

That means that for each channel the message box is locked iff in the ideal model a notification interrupt was requested for the same channel. The corresponding interrupt is (soon-to-be) pending, or active, or active-and-pending at the targeted core.

RcvIGC If for any given core in the ideal model the receive-IGC notification interrupt flag is active, then a virtual IGC interrupt is pending or active and pending at the corresponding core in the refined model. For all $g, g' \in$

$\mathbb{N}_{ng}$, $s \in \mathbb{N}_{ns}$ with $cpol(s) = (g, g')$ and $c \in \mathbb{N}_{nc}$ such that $\gamma(c) = g'$ we have:

$$M_I.G(g').rIGC(s, \kappa(c)) = 1 \;\Rightarrow$$
$$virq_{\mathrm{igc}_s} \in M_R.VI(c).pend \cup M_R.VI(c).ap$$

Moreover, while the ideal model IGC notification is pending or active-and-pending at a core, so is the refined model interrupt.

$$\forall \mathrm{x} \in \{pend, ap\}.\ \mathrm{igc}_s \in M_I.G(g').Q(\kappa(c)).\mathrm{x} \;\Rightarrow\; virq_{\mathrm{igc}_s} \in M_R.VI(c).\mathrm{x}$$

While a core is running in the end of the IGC reception handler, then the corresponding receive-IGC flag is raised and the state of IGC interrupt is related to the virtual interrupt state as follows.

$$mode^c_R = 2 \wedge C^c_R.pc \in \{a_{rniw}, a_{rndw}\} \;\Rightarrow$$
$$M_I.G(g').rIGC(\kappa(c)) = 1$$
$$\wedge\ (virq_{\mathrm{igc}_s} \in M_R.VI(c).pend \Rightarrow \mathrm{igc}_s \notin M_I.G(g').Q_{\kappa(c)})$$
$$\wedge\ (virq_{\mathrm{igc}_s} \in M_R.VI(c).ap \Rightarrow \mathrm{igc}_s \in M_I.G(g').Q_{\kappa(c)})$$

When a core is running outside of the aforementioned states and the virtual IGC interrupt is pending or active-and-pending at the refined core, then the receive-IGC flag may be raised, or the ideal model IGC notification interrupt is also pending or active-and-pending at the corresponding ideal core, or both. For all $\mathrm{x} \in \{pend, ap\}$:

$$\neg(mode^c_R = 2 \wedge C^c_R.pc \in \{a_{rniw}, a_{rndw}\}) \wedge virq_{\mathrm{igc}_s} \in M_R.VI(c).\mathrm{x} \;\Rightarrow$$
$$M_I.G(g').rIGC(s, \kappa(c)) = 1 \vee \mathrm{igc}_s \in M_I.G(g').Q(\kappa(c)).\mathrm{x}$$

Note that the coupling as described above is somewhat intricate as in the bisimulation the IGC interrupt is injected as a virtual interrupt before it is delivered by the GIC in the ideal model. Also we rely on the fact here, that interrupt injection may only change an interrupt from inactive to pending, or from active to active-and-pending.

Ext The external event sequence for each guest in the ideal model is extracted from the global external event signal in the refined model.

$$\forall g \in \mathbb{N}_{ng}.\ M_I.G(g).E = M_R.E|_g$$

GICcpu Each core's GIC CPU interface in the ideal model is directly coupled with the corresponding core's virtual interrupt interface.

$$\forall c \in \mathbb{N}_{nc}.\ M_I.G(\gamma(c)).GIC.gicc(\kappa(c)) = M_R.GIC.gicv(c)$$

GICdist We also need to couple the GIC distributors of each guest with the one in the refined model. For each register the coupling depends on the way how guest accesses are simulated by the hypervisor. We distinguish three cases:

1. The register may not be accessed by the guest — then no coupling is needed.
2. Read accesses to the register require a GIC access by the hypervisor — then we need to couple the ideal GIC distributor registers with the refined model register state. We apply function η_g^r to such registers r in the refined model in order to remove any readable information or writable settings that are not available to guest g because the corresponding interrupts are allocated to a different guest.
3. Read accesses to the register are emulated by the hypervisor — then we couple the ideal GIC distributor registers with the local copy of the register in hypervisor memory. However, this coupling is only maintained while there is no GICD virtualization handler operating on the GIC distributor. This is the case while any core of the corresponding guest has an outstanding write request to the GICD virtualization handler that was not processed by the GIC, yet.

Moreover, we only couple the GICD control register in this way while the ideal distributor is enabled. We define the following predicate to determine when to couple ideal and refined GICD registers.

$$\begin{aligned} cpl^g_{GICD}(M_I, r) \quad &\equiv \quad M_I.G(g).GIC.gicd(GICD\_CTLR)[1] \\ &\vee \quad r \neq GICD\_CTLR \end{aligned}$$

Formally the coupling is captured by the following statement, where we use $a_{GICD}(r) \in \mathbb{B}^{48}$ to denote the physical address of a GICD register. Note that we only need to enforce it once the guest starts executing.

$$\begin{aligned} &\forall g \in \mathbb{N}_{ng}, r \in GICD. \\ &\quad cpl^g_{GICD}(M_I, r) \wedge (\exists B \in \mathbb{B}.\ \neg failgicd(r, B)) \wedge \neg M_I.G(g).init \Rightarrow \\ &\qquad reqgicd(r, 0) \Rightarrow M_I.G(g).GIC.gicd(r) = \eta^r_g(M_R.GIC.gicd(r)) \\ &\quad \wedge \quad (\quad emugicd(r, 0) \wedge \nexists c, q.\ \gamma(c) = g \wedge q \in C2GR^c_R \wedge \\ &\qquad\qquad Wreq(q, A_{GICD}) \wedge adr(q) = a_{GICD}(r) \qquad) \Rightarrow \\ &\qquad\qquad M_I.G(g).GIC.gicd(r) = gcpy_g(M_R.m(a^g_{gcpy}), r) \end{aligned}$$

While the ideal distributor is disabled, the forwarding of all peripheral interrupts to cores of this guest are disabled. This is formalized in the statement below using the distributor mask function *dmsk* that determines which interrupts can be forwarded to a specific core by the distributor.

$$\begin{aligned} &\forall g \in \mathbb{N}_{ng}.\ M_I.G(g).GIC.gicd(GICD\_CTLR)[1] = 0 \ \Rightarrow \\ &\quad dmsk(M_R.GIC) \supseteq \{(q, c) \mid q \in PIRQ \wedge \gamma(c) = g\} \end{aligned}$$

IGCchan Each IGC communication channel contains the refined model memory content of corresponding shared memory page.

$$\forall s \in \mathbb{N}_{ns}, g, g' \in \mathbb{N}_{ng}.\ cpol(s) = (g, g') \Rightarrow M_I.S(s) = M_R.m(\epsilon A_{IGC}(g, g'))$$

Note that $A_{IGC}(g, g')$ is a singleton set for guests g and g' that are allowed to communicate, as noted earlier. Therefore the Hilbert choice operator ϵ is deterministic and selects the single IGC shared memory page address that is reserved for communication from g to g'.

CURR We couple the history variables in the ideal and refined model as follows. While the guest is running on a given refined core, the outstanding memory request is the same in refined and ideal model.

$$\forall c \in \mathbb{N}_{nc}.\ mode_R^c < 2 \Rightarrow C_I^c.curr = C_R^c.curr$$

While the hypervisor is running, the current core request is either empty, or coupled with the current refined core request through the GIC request conversion function, in case of the GIC distributor handler.

$$\forall c \in \mathbb{N}_{nc}.\ mode_R^c \geq 2 \Rightarrow$$
$$\begin{cases} C_I^c.curr = r & : \quad hv_c(M_R) \in \sigma_{\text{GICD}}^{entry,g,r} \\ \rho_{\gamma(c)}(C_I^c.curr) = C_R^c.curr & : \quad C_R^c.pc \in \{a_{gdw}, a_{gdfltr}\} \\ C_I^c.curr = \text{None} & : \quad \text{otherwise} \end{cases}$$

The outstanding memory requests in the refined model are the union of all outstanding memory requests in the ideal model taking the second-stage address translation and the renumbering of sender cores and peripherals into account. For all requests $r \in \mathbb{R}$, $c \in \mathbb{N}_{nc}$, and $p \in \mathbb{N}_{np}$:

$$(r, \text{C } c) \in M_R.m.curr \quad \Leftrightarrow \quad (\theta_{\gamma(c)}^{-1}(r), \text{C } \kappa(c)) \in M_I.G(\gamma(c)).m.curr$$
$$(r, \text{P } p) \in M_R.m.curr \quad \Leftrightarrow \quad (\theta_{\gamma(p)}^{-1}(r), \text{C } \varphi(p)) \in M_I.G(\gamma(p)).m.curr$$

Any outstanding memory request in one of the peripherals in the refined model also outstanding in the corresponding ideal peripheral, and vice versa, after applying address translation and index projection as necessary. For all $r \in \mathbb{R} \cup \mathbb{R} \times \mathbb{S}$:

$$r \in M_R.P.st(p).curr \quad \Leftrightarrow \quad conv_{IO}(r) \in M_I.G(\gamma(p)).P.st(\varphi(p)).curr$$

For the GIC we demand that all outstanding requests to the GIC virtual interface are also outstanding in the ideal model, with addresses and core indices subject to translation.

$$(r, \text{C } c) \in M_R.GIC.curr \wedge req(r, A_{GICV})$$
$$\Leftrightarrow$$
$$conv_{IO}(r) \in M_I.G(\gamma(c)).GIC.curr$$

Finally, for requests to the GIC distributor we require that a request is pending in the ideal GIC iff either the requesting refined core is in the entry point of the GICD virtualization handler, or in the state of that handler where the hypervisor is waiting for a reply from the GIC. In the

latter case the GIC lock is taken for that core and there is an outstanding request on the way to or in the GIC that is a filtered and address translated version of the ideal request. For all $r' \in \mathbb{R}$, $c \in \mathbb{N}_{nc}$:

$$(r', \kappa(c)) \in M_I.G(\gamma(c)).GIC.curr \wedge req(r', A^g_{GICD}) \\ \Leftrightarrow \\ hv_c(M_R) \in \sigma^{entry,g,r'}_{\text{GICD}} \vee \\ mode^c_R = 2 \wedge C^c_R.pc = a_{gdw} \wedge Glock(M_R.m(a_{glk})) = c \wedge \\ \exists r \in C2GR^c_R.\ req(r, A_{GICD}) \wedge r = \rho_g(\theta_g(r'))$$

Note that the projection functions γ, κ, and φ are covering all cores and peripherals of all guests, since they are injective and guests do not share cores or peripherals. Thus the complete ideal model is covered by the simulation relation, however we could not write it as a projection function from refined to ideal configurations, as the complexity of the GIC virtualization affects the coupling of the message boxes and GICD registers.

The bisimulation theorem itself is stated and proven separately for both directions. In the simulation of the refined model by the ideal model, every step of a given refined model computation has to be in the simulation relation with a number of (consecutive or empty) steps of a corresponding ideal model computation, and vice versa for the other direction. That this works for the given bisimulation relation is not immediately obvious, however we defined the coupling in a way that captures the intended stepping of the models during the bisimulation proof.

For instance, the intricate coupling of the channels between GIC and memory shows that we need to step the refined and ideal GIC simultaneously to process any guest request to the ideal GIC at the same time as the (possibly simulated) request in the refined model is processed.

Another noteworthy observation is that the interrupt signals are only coupled for interrupts between cores and peripherals belonging to the same guest. This is by design as interrupts between cores of different guests should only occur through IGC signals in the ideal model, while interrupts from peripherals belonging to one guest should never be sent to a core of another guest. However the constaints that the bisimulation relation itself puts on the refined model do not guarantee this. Instead such properties are ensured by parts of the hypervisor implementation that we need to cover with an additional *implementation invariant* which must hold at all times after initialization of the HASPOC platform.

4.5 Implementation Invariant

As explained above the implementation invariant is needed to capture properties of the hypervisor implementation that are not implied by the bisimulation relation connecting the ideal and refined model, but are needed to establish the simulation and make the inductive proof go through. As one might guess the implementation invariant constrains the parts of the refined model that are not covered by the bisimulation. These are:

- the second-stage page tables for the guests, located in hypervisor memory,
- the page tables used by the SMMUs of each peripheral,
- the configurations of second-stage MMUs and SMMUs,
- other EL2 and EL3 system level registers that support the security of the system and control the trapping mechanism from EL1 and EL0,
- the GIC configuration for inter-processor interrupts between cores of different guests, as well as peripheral interrupts to cores that are not allowed to receive them,
- the interrupt state for inter-processor interrupts between cores of different guests,
- the contents of IGC message boxes that belong to disallowed channels,
- properties of the refined model transition system, in particular for content of the message channels, arising from the refined model semantics,
- additional properties of the hypervisor data structures,
- the state of the memory controller, and
- the integrity of the boot images.

The implementation invariant does not hold completely from the start of the bisimulation. Some clauses are established by the hypervisor initialization sequence that is executed by the primary core, however the clauses that are particular to certain guests are established concurrently by the cores that are allocated to each guest. The former consider mainly the initialization of internal hypervisor data structures. The latter deal with the guest-specific data structures, e.g., the page tables and virtualization extension registers. Thus they are not guaranteed to hold for one guest or core before that guest (or core of the guest, respectively) is started.

We denote the implementation invariant on a refined system state M_R by $Inv(M_R)$ which is a predicate that ensures the following statements.

4.5.1 Invariants that always hold

Directly after boot the following invariants can be relied on.

Inv-Img The images loaded during boot are still located in flash memory and not modified. For all $img \in IMG_H, a \in Ra_i(img)$:

$$M_R.m(a) = \langle img \rangle[\langle a \rangle - \langle img.adr \rangle + 4096 \cdot 8 - 1 : \langle a \rangle - \langle img.adr \rangle]$$

Inv-GICPol For each guest the GIC distributor is configured such that all peripheral and IGC interrupts not belonging to that guest are never forwarded to the guest's cores. We call such a distributor mask the *Golden Mask* for guest g, denoted by GM_g that contains the following pairs of interrupts and cores.

$$\{(q, c) \mid \gamma(c) = g \wedge q \in [16 : 1019] \setminus (PIRQ_g \cup \{\langle id_{IGC}(s)\rangle \mid s \in IGCin_g\})\}$$

Then the masked interrupts for each guest must contain at least its Golden Mask, i.e., for all $g \in \mathbb{N}_{ng}$ we have $dmsk(M_R.GIC) \supseteq GM_g$. To this end we assume that there exists a set of *Golden GIC distributor* configurations $GICD_{GM}$ that implement the Golden Mask for each guest independent of the configuration of the CPU and virtual interrupt interfaces. We require that the GIC distributor is always in a Golden configuration.

$$M_R.GIC.gicd \in GICD_{GM}$$

Note, that we leave the definition of this set of configurations implicit, as to not introduce the specific GIC model with all its complexity. The only setting we require here is that the GIC distributor is always enabled at least for the non-secure interrupts.

$$\forall gicd \in GICD_{GM}.\ gicd(GICD\_CTLR)[1] = 1$$

Note also that we do not enforce the masking of inter-processor interrupts as they are always enabled in the GIC distributor.

However, there is an invariant that (non-secure) inter-processor interrupts from cores of different guests are never pending for the cores of a given guest, unless they have ID 15 which is used for IGC interrupts. Moreover secure SGIs (with IDs 0-7) are never pending in the system as they are not used by the hypervisor or boot code. This implies that there are only interrupts present for a given core c that the corresponding guest is allowed to receive. Let Q_R^c denote the set of all pending, active, or, active-and-pending interrupts for core c, i.e., $Q_R^c = M_R.Q_c$. Moreover, let

$$\begin{aligned} IRQ_c^- &= PIRQ_{\gamma(c)} \cup \{\mathrm{sgi}_{c',c}^{id} \mid id \in [8 : 15] \wedge \gamma(c) = \gamma(c')\} \\ IRQ_c &= IRQ_c^- \cup \{\mathrm{sgi}_{c',c}^{15} \mid \exists s \in \mathbb{N}_{ns}.\ cpol(s) = (\gamma(c'), \gamma(c))\} \end{aligned}$$

Then for all $c \in \mathbb{N}_{nc}$ we require:

$$Q_R^c \subseteq IRQ_c$$

Observe that the invariants on the GIC distributor hold already initially for all guests as interrupt forwarding by the GIC is initially disabled for all interrupts and interrupts are only enabled incrementally for each guest. Moreover no interrupts are pending or active initially.

Inv-GICInterface We demand a setting of the GIC CPU interface that controls how interrupts are being signalled to the hypervisor. First of all, the interface should always be enabled for signalling non-secure interrupts and disabled for the signalling of secure interrupts, as the hypervisor and the guests make no use of them, or do not have access to them, respectively. For all $c \in \mathbb{N}_{nc}$ we require:

$$M_R.GIC.gicc(GICC\_CTLR, c).EnableGrp1 = 1$$
$$M_R.GIC.gicc(GICC\_CTLR, c).EnableGrp0 = 0$$

Secondly, the non-secure Group 1 interrupts should be acknoledged by a read of the *GICC_AIAR* register, and not by reading the *GICC_IAR* register, as recommended by ARM. Use of the latter register should be reserved for secure Group 0 interrupts only. To this end, we require that the interface control registers are set up accordingly, namely the flag *AckCtl* needs to be cleared.

$$M_R.GIC.gicc(GICC\_CTLR, c).AckCtl = 0$$

Finally, we require that the CPU interface is configured to implement separate priority drop and interrupt deactivation.

$$M_R.GIC.gicc(GICC\_CTLR, c).EOIModeNS = 1$$

We collect these invariants in the predicate $Inv_{GICC}(M_R, c)$ for each core. In our model we assume for simplicity that they hold after reset. However in the real hardware the initialization code may need to establish them by communicating with the GIC before the guest is started on each given core. We leave these potential configuration steps implicit here in order not to complicate further the boot code specification as well as the bisimulation relation and proof.

4.5.2 Invariants established by the Boot Loader

Only the boot loader executes in mode EL3, therefore here is the only chance to configure the security control registers and other control registers of the boot loader. Moreover it copies all images to their correct location, however this cannot be expressed as an invariant of the system as contained variables may be overwritten by the hypervisor during runtime. We require the following properties that should hold after the execution of the boot loader, i.e., when a core enters EL2 for the first time at address a_{iep} or a_{ies} (depending on whether core c is the primary or a secondary core).

Inv-RegEl3 Registers *SCR_EL3* and *VBAR_EL3* are set up as in the initialization steps Init-PCold, or Init-SCold respectively.

$$C^c_R.spr(SCR\_EL3) = scr_H \wedge C^c_R.spr(VBAR\_EL3) = 0^{16} \circ vbar_{psci} \circ 0^{12}$$

4.5.3 Hypervisor Invariants established by the Primary Core

The following statements are holding after the primary core finishes the initialization of global hypervisor data structures in transition Init-PInit. This state is identified by a history variable $hvinit \in \mathbb{B}$ that is set to true as soon as that step is executed by the primary core, i.e., $hv_0(M_R) \in \sigma^{prim}_{\text{INIT}}$. If $M_R.hvinit$, then:

Inv-Pow The content of the power state data structure reflects the power state of each core.

$$\forall c \in \mathbb{N}_{nc}.\ pow(M_R.m(a_{pow}), c) = C^c_R.active$$

Inv-MsgBox The IGC message box entries for channels that are disallowed according to the communication policy are always zero.

$$\forall g, g' \in \mathbb{N}_{ng}.\ (\nexists s \in \mathbb{N}_{ns}.\ cpol(s) = (g, g')) \Rightarrow mbox(hv_0(M_R), g, g') = 0$$

Inv-IgcSgi An SGI interrupt with ID 15 between cores of different guests is pending or active only if the corresponding message box bit is set to one. For all cores $c, c' \in \mathbb{N}_{nc}$ such that $\gamma(c) \neq \gamma(c')$ we have:

$$\text{sgi}^{15}_{c,c'} \in M_R.Q(c').pend \cup M_R.Q(c').act \Rightarrow mbox(hv_0(M_R), g, g') = 1$$

Such an SGI is never active-and-pending at any core, as the hypervisor locks the IGC notification channel until any pending notification is received, acknowledged, and deactivated.

$$\text{sgi}^{15}_{c,c'} \notin M_R.Q(c').ap$$

Note, that Inv-GICPol guarantees that SGIs with ID 15 from cores of different guests are only pending or active at a core's CPU interface if they belong to an allowed IGC channel.

4.5.4 Invariants established for each Guest

The following invariants are holding when the first core of a given guest is started, i.e., for each guest g when

$$\pi_C(c) = (g, 0) \wedge C^c_R.init$$

Inv-PT The golden image page tables are located at $A_{PT}(g)$ for each guest g.

$$\forall a \in A_{PT}(g).\ M_R.m(a) = GI_g(a)$$

Inv-PPT The golden image SMMU page table are located at $A_{PPT}(p)$ for each peripheral p.

$$\forall a \in A_{PPT}(g).\ M_R.m(a) = GIP_g(a)$$

Inv-SMMU All SMMUs are configured according to the golden image of the peripheral page tables.

$$\forall p \in \mathbb{N}_{np}.\ \gamma(p) = g \Rightarrow M_R.smmu(p) \in SMMU^g_{GI}$$

Note that all of these invariants are established by initialization step Init-Ginit.

4.5.5 Invariants established for each Core

Similarly there are invariants for each core that has been initialized, i.e., for all cores $c \in \mathbb{N}_{nc}$ with $C_R^c.init$ we have:

Inv-RegEl2 Registers *HCR_EL2*, and *VBAR_EL2* are set up as in the Init-Cinit step. There exists some $vi \in \mathbb{B}$ such that:

$$C_R^c.spr(HCR\_EL2) = hcr_H^{vi} \wedge C_R^c.spr(VBAR\_EL2) = 0^{16} \circ vbar_H \circ 0^{12}$$

Inv-MMU While the core is running in guest mode or in a hypervisor entry state, its MMU is set up according to the golden image page table configuration.

$$mode_R^c < 2 \vee C_R^c.pc \in A_{hve} \;\Rightarrow\; M_R.mmu(c) \in MMU_{GI}^{\gamma(c)}$$

While the core is running in hypervisor mode and it is not in an entry point, the MMU is turned off but still configured according to the golden image.

$$\begin{array}{rll} mode_R^c = 2 \wedge C_R^c.pc \notin A_{hve} & \Rightarrow & M_R.mmu(c).active = 0 \\ & \wedge & M_R.mmu(c).pto = MMU_{GI}^{\gamma(c)}.pto \\ & \wedge & M_R.mmu(c).cfg = MMU_{GI}^{\gamma(c)}.cfg \\ & \wedge & M_R.mmu(c).walk = \bot \end{array}$$

4.5.6 Intrinsic Invariants of the Transition System

There are a number of invariants that are intrinsic properties of the refined model and its transition system. These properties hold throughout each computation of the refined model and are not depending on the hypervisor implementation.

Inv-HVEntry When the hypervisor is in one of its entry states on any given core, then the message channels for this core are empty and there are no outstanding requests for this core.

$$\begin{array}{rl} \forall c \in \mathbb{N}_{nc}.\ hv_c(M_R) \in \Sigma \wedge C_R^c.pc \in A_{hve} \;\Rightarrow & \\ \quad C2GR_R^c \cup MR_R^c \cup PR_R^c \cup G2CQ_R^c \cup PQ_R^c = \emptyset & \\ \wedge \quad C_R^c.curr = \text{None} \wedge mmu_R^c.walk = \bot & \end{array}$$

Inv-Msg For all cores $c \in \mathbb{N}_{nc}$ there is at most one outstanding request or reply message in the system:

$$\begin{array}{rl} & \#CR_R^c + \#UR_R^c + \#MR_R^c + \#GR_R^c + \#PR_R^c \\ + & \#CQ_R^c + \#UQ_R^c + \#GQ_R^c + \#PQ_R^c \quad \leq\ 1 \end{array}$$

Inv-Walk There are only table walk requests and replies in channels *u2m* and *m2u* if the MMU is in its translating phase. Then also peripheral channels

never contain a message tied to a core while its MMU is translating a request. Moreover, the current request in the MMU corresponds to the last request sent by the core. If there is no current memory request in the core or it was already answered, the MMU is not busy. Let

$$\mathbb{R}_{ptw} = \{\text{None}, \text{PTW } a \mid a \in \mathbb{B}^{48}\}$$
$$\mathbb{Q}_{ptw} = \{\text{None}, \mathit{rplPTW}\ a\ v \mid a \in \mathbb{B}^{48}, v \in \mathbb{B}^{64}\}$$

then we require:

$$\begin{aligned}
\forall c \in \mathbb{N}_{nc} \quad . \quad & \mathit{trans}(\mathit{mmu}_R^c) \ \Rightarrow\ \mathit{u2m}_R^c \in \mathbb{R}_{ptw} \wedge \mathit{m2u}_R^c \in \mathbb{Q}_{ptw} \\
& \qquad \wedge\ \mathit{PR}_R^c \cup \mathit{GR}_R^c \cup \mathit{PQ}_R^c \cup \mathit{GQ}_R^c = \emptyset \\
\wedge \quad & \mathit{trans}(\mathit{mmu}_R^c) \vee \mathit{mmu}_R^c.\mathit{walk}.\mathit{complete} \ \Rightarrow \\
& \qquad \mathit{mmu}_R^c.\mathit{walk}.\mathit{curr} = C_R^c.\mathit{curr} \neq \text{None} \\
\wedge \quad & C_R^c.\mathit{curr} = \text{None} \vee \mathit{u2c}_R^c \neq \text{None} \Rightarrow \mathit{mmu}_R^c.\mathit{walk} = \bot
\end{aligned}$$

Inv-Busy Whenever there is an outstanding request for core c in memory, GIC, or the peripherals, or in the channels between them and the MMU of that core, and the MMU is active, then it is also busy, i.e., either in its translation of final phase.

$$\begin{gathered}
\mathit{UR}_R^c \cup \mathit{MR}_R^c \cup \mathit{GR}_R^c \cup \mathit{PR}_R^c \cup \mathit{UQ}_R^c \cup \mathit{GQ}_R^c \cup \mathit{PQ}_R^c \neq \emptyset \ \Rightarrow \\
\mathit{mmu}_R^c.\mathit{active} \Rightarrow \mathit{mmu}_R^c.\mathit{walk} \neq \bot
\end{gathered}$$

Inv-PWalk For the channels between SMMUs and memory we have a similar condition as for the second-stage MMU channels, restricting DMA accesses and their corresponding SMMU page table walks as well as the current request handled by the SMMU.

$$\begin{aligned}
\forall p \in \mathbb{N}_{np} \quad . \quad & \mathit{trans}(\mathit{smmu}_R^p) \Rightarrow \mathit{v2m}_R^p \in \mathbb{R}_{ptw} \vee \mathit{m2v}_R^p \in \mathbb{Q}_{ptw} \\
\wedge \quad & \mathit{trans}(\mathit{smmu}_R^p) \vee \mathit{smmu}_R^p.\mathit{walk}.\mathit{complete} \ \Rightarrow \\
& \qquad \text{None} \neq \mathit{smmu}_R^p.\mathit{walk}.\mathit{curr} \in \mathit{P.st}_R^p.\mathit{curr} \cap \mathbb{R} \neq \emptyset \\
\wedge \quad & \mathit{P.st}_R^c.\mathit{curr} \cap \mathbb{R} = \emptyset \vee \mathit{v2p}_R^p \neq \text{None} \Rightarrow \mathit{smmu}_R^p.\mathit{walk} = \bot
\end{aligned}$$

Inv-Request The remaining requests we cover here are sent by the cores, MMUs, and SMMUs respectively. MMU component $\mathit{lookup} \in \mathbb{R}_{ptw}$ contains the last requested page table lookup of an MMU, while component $\mathit{final} \in \mathbb{R}$ contains the final memory request sent by an MMU. For all $c \in \mathbb{N}_{nc}$, $p \in \mathbb{N}_{np}$ we have:

$$\mathit{c2u}_R^c \neq \text{None} \quad \Rightarrow \quad \mathit{c2u}_R^c = C_R^c.\mathit{curr}$$

$$\mathit{u2m}_R^c \neq \text{None} \quad \Rightarrow \quad \mathit{u2m}_R^c = \begin{cases} \mathit{mmu}_R^c.\mathit{walk}.\mathit{lookup} & : \quad \mathit{trans}(\mathit{mmu}_R^c) \\ \mathit{mmu}_R^c.\mathit{walk}.\mathit{final} & : \quad \mathit{mmu}_R^c.\mathit{walk}.\mathit{complete} \\ C_R^c.\mathit{curr} & : \quad \text{otherwise} \end{cases}$$

$$p2v_R^p \neq \text{None} \quad \Rightarrow \quad p2v_R^p \notin \mathbb{Q} \times \mathbb{S} \quad \Rightarrow \quad p2v_R^p \in P.st_R^p.curr$$

$$v2m_R^p \neq \text{None} \quad \Rightarrow \quad v2m_R^p = \begin{cases} smmu_R^p.walk.lookup & : \quad trans(smmu_R^p) \\ smmu_R^p.walk.final & : \quad \text{otherwise} \end{cases}$$

Additionally we require that all memory-mapped I/O requests in peripherals and the GIC are sent by the cores, i.e., we forbid peripherals to perform memory-mapped I/O requests . For all requests $r \in \mathbb{R}$ and sender IDs $s \in \mathbb{S}$ we demand:

$$(r, s) \in \{v2p_R^{np}\} \cup M_R.GIC.curr \cup \bigcup_{p \in \mathbb{N}_{np}} PR_R^p \;\Rightarrow\; \exists c.\ s = \text{C } c$$

Inv-MemReq For each such outstanding request there exists a sent request in one of the MMUs, SMMUs, or cores. The latter case occurs only for hypervisor accesses where the second-stage MMU is inactive. For all $r \in \mathbb{R}$, $c \in \mathbb{N}_{nc}$, and $p \in \mathbb{N}_{np}$:

$$(r, \text{C } c) \in M_R.m.curr \;\Rightarrow$$
$$r = \begin{cases} mmu_R^c.walk.lookup & : \quad trans(mmu_R^c) \\ mmu_R^c.walk.final & : \quad mmu_R^c.walk.complete \\ C_R^c.curr & : \quad \text{otherwise} \end{cases}$$
$$(r, \text{P } p) \in M_R.m.curr \;\Rightarrow$$
$$r = \begin{cases} smmu_R^p.walk.lookup & : \quad trans(smmu_R^p) \\ smmu_R^p.walk.final & : \quad \text{otherwise} \end{cases}$$

Note that this invariant implies that for each core or peripheral there is at most one outstanding request in the memory, thus excluding memory models that are weakly consistent and out-of-order execution in the cores.

Inv-Reply Any outstanding reply in one of the channels between cores and memory matches an outstanding request in the MMU or the core. Note that matching replies can also be faults for the requested address. We capture the notion of matching requests $r \in \mathbb{R}$ and replies $q \in \mathbb{Q}$ with $r, q \neq \text{None}$ by the following predicate.

$$match(r, q) \equiv q \in \begin{cases} \{rplR\ a\ v, \text{F } a\} & : \quad r = \text{R } a\ d \wedge v \in \mathbb{B}^{8d} \\ \{rplW\ a, \text{F } a\} & : \quad r = \text{W } a\ d\ v \\ \{rplPTW\ a\ v, \text{F } a\} & : \quad r = \text{PTW } a \wedge v \in \mathbb{B}^{64} \end{cases}$$

Then the invariant for the channel from memory to second-stage MMU is defined as follows for all $c \in \mathbb{N}_{nc}$ and q as above.

$$m2u_R^c \neq \text{None} \Rightarrow \begin{cases} match(mmu_R^c.walk.lookup, m2u_R^c) & : trans(mmu_R^c) \\ match(mmu_R^c.walk.final, m2u_R^c) & : mmu_R^c.walk.complete \\ match(C_R^c.curr, m2u_R^c) & : \text{otherwise} \end{cases}$$

For the channel from second-stage MMU to core we have:

$$u2c_R^c \neq \text{None} \Rightarrow match(C_R^c.curr, u2c_R^c)$$

For the channels from memory to the SMMUs we have for all $p \in \mathbb{N}_{np}$:

$$m2v_R^p \neq \text{None} \Rightarrow \begin{cases} match(smmu_R^p.walk.lookup, m2v_R^p) & : trans(smmu_R^p) \\ match(smmu_R^p.walk.final, m2v_R^p) & : \text{otherwise} \end{cases}$$

Replies in the input channels to peripherals are DMA replies.

$$v2p_R^p = q \quad \Rightarrow \quad match(P.st_R^p.curr, v2p_R^p)$$

Moreover, memory-mapped I/O replies in the GIC and peripheral output channels are never targeted at other peripherals. For all $s \in \mathbb{S}$ we have:

$$(q, s) \in \{p2v_R^p \mid p \in \mathbb{N}_{np+1}\} \;\Rightarrow\; \exists c.\; s = \text{C } c$$

Inv-Dma While there are outstanding DMA requests in the SMMUs or memory, the message channels to and from the corresponding sender are empty. For all cores $c \in \mathbb{N}_{nc}$ and $p \in \mathbb{N}_{np}$ as well as requests $r \in \mathbb{R}$ with $r \neq \text{None}$ we have:

1. While a DMA request from peripheral p is outstanding in memory, no DMA requests are sent from this peripheral or its SMMU and no DMA replies are present in the corresponding channels.

$$\begin{aligned}(r, \text{P } p) \in M_R.m.curr \;\Rightarrow\; & v2m_R^p = m2v_R^p = \text{None} \\ \wedge\; & p2v_R^p \in \mathbb{Q} \times \mathbb{S} \cup \{\text{None}\} \wedge v2p_R^p \in \mathbb{R} \times \mathbb{S} \cup \{\text{None}\}\end{aligned}$$

2. While an SMMU is busy with a DMA request from peripheral p, no new DMA requests are sent from this peripheral no DMA replies are sent back.

$$\begin{gathered}smmu_R^p.walk \neq \bot \;\Rightarrow \\ p2v_R^p \in \mathbb{Q} \times \mathbb{S} \cup \{\text{None}\} \wedge v2p_R^p \in \mathbb{R} \times \mathbb{S} \cup \{\text{None}\}\end{gathered}$$

3. Additionally, the GIC never performs DMA requests, i.e.:

$$p2v_R^{np} \cap \mathbb{R} = \text{None} \qquad v2p_R^{np} \cap \mathbb{Q} = \text{None}$$

Inv-Pirq The physical interrupt signals of the GIC in the refined model are either pending, or active, or active-and-pending or active. For all $c \in \mathbb{N}_{nc}$:

$$\begin{gathered}M_R.Q(c).pend \cap M_R.Q(c).act = \emptyset \qquad M_R.Q(c).act \cap M_R.Q(c).ap = \emptyset \\ M_R.Q(c).pend \cap M_R.Q(c).ap = \emptyset\end{gathered}$$

Note that many of these invariants depend on the transition relations of cores, (S)MMUs, peripherals, memory, and the GIC. For instance, invariant Inv-Reply relies on the fact that the memory only produces replies that have been actually requested. This and other constaints on the transition relation will be formulated in Sect. 4.9.

4.5.7 Specific Run-time Invariants of the HASPOC platform

In addition to the intrinsic invariants of the transition system, there are other invariants that only hold due to the hypervisor implementation. They are mainly related to the GICD and interrupt virtualization handlers.

INV-HYP-GCPY For each guest, the local copy of GICD registers r for which guest read accesses are only emulated are in sync with the actual GIC distributor registers. Again we use function η_g^r to filter out any settings or information in register r that are not accessible to the guest. Then for each register, the copy is only coupled if there is no outstanding write to the register by a GICD virtualization handler on a core that belongs to the same guest, since the copy is updated before the GIC distributor.

Note that we do not need to couple the GICD control register with the copy, since the distributor is always enabled for non-secure interrupts in the refined model, and the guest may not configure secure interrupts. Moreover, the registers to enable and disable the distribution of interrupts are only coupled while the GIC distributor of the corresponding guest is enabled. We overload predicate cpl^g_{GICD} to denote which registers to couple between the GIC distributor and its local copy for guest g in the refined model.

$$\begin{aligned} cpl^g_{GICD}(M_R, r) \quad \equiv \quad & r \notin \{GICD\_CTLR, GICD\_SGIR\} \\ \wedge \quad & (\exists B \in \mathbb{B}.\ \neg failgicd(r, B)) \\ \wedge \quad & gcpy_g(M_R.m(a^g_{gcpy}), GICD\_CTLR)[1] = 0 \ \Rightarrow \\ & r \notin \{GICD\_ISENABLERn, GICD\_ICENABLERn\} \end{aligned}$$

Then for all $g \in \mathbb{N}_{ng}$, $r \in GICD$ and its corresponding physical address $a_{GICD}(r) \in \mathbb{B}^{48}$ we have:

$$\begin{gathered} emugicd(r, 0) \wedge cpl^g_{GICD}(M_R, r) \wedge \\ \nexists c, d, v.\ \gamma(c) = g \wedge mode^c_R = 2 \wedge \mathrm{W}\ a_{GICD}(r)\ d\ v \in C2GR^c_R \ \Rightarrow \\ gcpy_g(M_R.m(a^g_{gcpy}), r) = \eta^r_g(M_R.GIC.gicd(r)) \end{gathered}$$

Note that registers $GICD\_ISENABLERn$ and $GICD\_ICENABLERn$ do not need to be coupled with the GIC distributor, as all interrupts for guest g are disabled in the actual distributor while the guests ideal distributor is disabled, according to bisimulation relation GICDIST.

INV-HYP-GWRITE If there is an update to the GICD distributor by the hypervisor, then the filtered new value of the GICD register after the prospected update equals the local GICD register copy for any guest g and GICD register r. With $c \in \mathbb{N}_{nc}$, $d \in \mathbb{N}$, and $v \in \mathbb{B}^{8d}$ we have:

$$\begin{gathered} emugicd(r, 0) \wedge cpl^g_{GICD}(M_R, r) \wedge \\ \gamma(c) = g \wedge mode^c_R = 2 \wedge \mathrm{W}\ a_{GICD}(r)\ d\ v \in C2GR^c_R \ \Rightarrow \\ \eta^r_g(\nu(r, M_R.GIC.gicd(r), v)) = gcpy_g(M_R.m(a^g_{gcpy}), r) \end{gathered}$$

Observe that this and the previous invariant, together with the bisimulation relation $M_I \sim M_R$, guarantee

$$M_I.G(g).GIC.gicd(r) = \eta_g^r(M_R.GIC.gicd(r))$$

for all guests and GICD registers while there is no outstanding write to the GIC distributor.

INV-HYP-GREQ The hypervisor (and noone else) sends only memory requests to the GIC CPU interface, the GIC distributor, or the virtual interrupt control interface of the GIC. For all $c \in \mathbb{N}_{nc}$ we have:

$$\forall r \in C2GR_R^c.\ mode_R^c \geq 2 \ \Rightarrow\ req(r, A_{GICC} \cup A_{GICD} \cup A_{GICH})$$
$$\forall r \in UR_R^c \cup GR_R^c.\ req(r, A_{GICC} \cup A_{GICD} \cup A_{GICH}) \Rightarrow mode_R^c = 2$$

For each such outstanding request in the GIC there exists a sent request in one of the MMUs (for guest accesses to the virtual GIC interface) or cores (in case the hypervisor is accessing the GIC). Note that no peripheral DMA accesses or page table look-ups are ever forwarded to the GIC. For all $r \in \mathbb{R}$ and $c \in \mathbb{N}_{nc}$:

$$(r, \mathrm{C}\ c) \in M_R.GIC.curr \ \Rightarrow\ r = \begin{cases} mmu_R^c.walk.final & : \quad req(r, A_{GICV}) \\ C_R^c.curr & : \quad \text{otherwise} \end{cases}$$

INV-HYP-GMSG More specifically, in different hypervisor states we restrict the content of the message channels as follows. If $mode_R^c = 2$ then:

$$C_R^c.pc = a_{gdw} \ \Rightarrow\ \exists r \in \mathbb{R}, q \in G2CQ_R^c.$$
$$\rho_{\gamma(c)}(r) \in C2GR_R^c \wedge req(\rho_{\gamma(c)}(r), A_{GICD}) \vee rpl(q, A_{GICD})$$

This invariant corresponds to hypervisor states $\sigma_{\mathrm{GICD}}^{wait}$.

$$C_R^c.pc = a_{irqaw} \quad \Rightarrow \quad \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.$$
$$r = \mathrm{R}\ a_{IAR}\ 4 \vee q = rplR\ a_{IAR}\ v$$

This invariant corresponds to hypervisor states $\sigma_{\mathrm{IRQ}}^{await,v,g}$.

$$C_R^c.pc = a_{irqew} \quad \Rightarrow \quad \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.$$
$$r = \mathrm{W}\ a_{EOIR}\ 4\ v \vee q = rplW\ a_{EOIR}$$

This invariant corresponds to hypervisor states $\sigma_{\mathrm{IRQ}}^{ewait}$.

$$C_R^c.pc = a_{irqcw} \quad \Rightarrow \quad \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), l \in \mathbb{B}^{32}.$$
$$r = \mathrm{R}\ a_{LR}^{\gamma(c)}(lirq(M_R.m(a_{lirq}), c))\ 4\ \vee$$
$$q = rplR\ a_{LR}^{\gamma(c)}(lirq(M_R.m(a_{lirq}), c))\ l$$

This invariant corresponds to hypervisor states $\sigma_{\text{IRQ}}^{cwait,c,l}$.

$$C_R^c.pc = a_{irqiw} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), d \in \{4,8\}, v \in \mathbb{B}^{8d}.\\ r = \text{W}\ a_{LR}^{\gamma(c)}(lirq(M_R.m(a_{lirq}),c))\ d\ v\ \vee\\ q = rplW\ a_{LR}^{\gamma(c)}(lirq(M_R.m(a_{lirq}),c))$$

This invariant corresponds to hypervisor states $\sigma_{\text{IRQ}}^{iwait}$.

$$C_R^c.pc = a_{irqfw} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.\\ r = \text{W}\ a_{DIR}\ 4\ v \vee q = rplW\ a_{DIR}$$

This invariant corresponds to hypervisor states $\sigma_{\text{IRQ}}^{fwait}$.

$$C_R^c.pc = a_{snsgi} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.\\ (r = \text{W}\ a_{SGIR}\ 4\ v \vee q = rplW\ a_{SGIR}) \wedge \langle v[3:0]\rangle = 15\\ \wedge\quad \forall i \in \mathbb{N}_8.\ v[16+i] \;\Rightarrow\; \exists s.\ cpol(s) = (\gamma(c),\gamma(i))\\ \wedge\ C2GR_c(M_R) \neq \emptyset \Rightarrow mbox(hv_c(M_R),\gamma(c),\gamma(i)) = 1$$

This invariant corresponds to hypervisor states $\sigma_{\text{SIGC}}^{SGI}$. We require additionally that the SGI ID is 15, they are only sent to different guests according to the IGC communication policy, and that the matching bit in the IGC message box is set while the request has not been processed.

$$C_R^c.pc = a_{rnew} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.\\ r = \text{W}\ a_{EOIR}\ 4\ v \vee q = rplW\ a_{EOIR}$$

This invariant corresponds to hypervisor states $\sigma_{\text{RIGC}}^{ewait}$.

$$C_R^c.pc = a_{rncw} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), l \in \mathbb{B}^{32}.\\ r = \text{R}\ a_{LR}^{IGC}(ligc(M_R.m(a_{ligc}),c)_1)\ 4\ \vee\\ q = rplR\ a_{LR}^{IGC}(ligc(M_R.m(a_{ligc}),c)_1)\ l$$

This invariant corresponds to hypervisor states $\sigma_{\text{RIGC}}^{cwait,c,s,c',l}$.

$$C_R^c.pc = a_{rniw} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), d \in \{4,8\}, v \in \mathbb{B}^{8d}.\\ r = \text{W}\ a_{LR}^{IGC}(ligc(M_R.m(a_{ligc}),c)_1)\ d\ v\ \vee\\ q = rplW\ a_{LR}^{IGC}(ligc(M_R.m(a_{ligc}),c)_1)$$

This invariant corresponds to hypervisor states $\sigma_{\text{RIGC}}^{iwait,c,s,c'}$.

$$C_R^c.pc = a_{rndw} \;\Rightarrow\; \exists r \in C2GR_c(M_R), q \in G2CQ_c(M_R), v \in \mathbb{B}^{32}.\\ r = \text{W}\ a_{DIR}\ 4\ v \vee q = rplW\ a_{DIR}$$

This invariant corresponds to hypervisor states $\sigma_{\text{RIGC}}^{dwait}$. In all other cases where the hypervisor is running on core c we have $C2GR_c(M_R) = \emptyset$ and $G2CQ_c(M_R) = \emptyset$. We omit a more detailed formalization for brevity.

Inv-Hyp-LastIGC While the hypervisor is running the IGC reception handler, after it acknowledged the corresponding SGI interrupt, and before disabling it, the message box entry corresponding to channel for the last received IGC interrupt is set to one and the corresponding SGI interrupt is active. Let those states be denoted by the set of PC values $A_{RIGC} = \{a_{rncw}, a_{rnew}, a_{rniw}, a_{rndw}\}$ and let $c' = ligc(M_R.m(a_{ligc}), c)_2$ denote the sender of the IGC interrupt then for all $c \in \mathbb{N}_{nc}$ we have:

$$mode_R^c = 2 \wedge C_R^c.pc \in A_{RIGC} \Rightarrow mbox(hv_c(M_R), \gamma(c'), \gamma(c)) = 1$$

$$mode_R^c = 2 \wedge C_R^c.pc \in A_{RIGC} \wedge \neg(C_R^c.pc = a_{rndw} \wedge G2CQ_R^c \neq \emptyset) \Rightarrow \mathrm{sgi}_{c',c}^{15} \in M_R.Q(c).act$$

Inv-Hyp-AckIGC While the hypervisor is waiting for the reply from the GIC interrupt acknowledgement register and such an outstanding reply is signalling an IGC inter-processor interrupt, the message box entry for the corresponding channel is set to one and the corresponding inter-processor interrupt is active. For all cores $c, c' \in \mathbb{N}_{nc}$ such that $g = \gamma(c) \neq \gamma(c') = g'$ we have :

$$\begin{array}{c} mode_R^c = 2 \wedge C_R^c.pc = a_{irqaw} \wedge \\ (\ \exists v \in \mathbb{B}^{32} \ . \ \ rplR\ a_{IAR}\ v \in G2CQ_c(M_R) \\ \wedge \ \ \langle v[9:0] \rangle = 15 \wedge \langle v[12:10] \rangle = c'\ ) \ \Rightarrow \\ mbox(hv_c(M_R), g', g) = 1 \wedge \mathrm{sgi}_{c',c}^{15} \in M_R.Q(c).act \end{array}$$

Inv-Hyp-Virq For the virtual interrupts we have the same constraints as the for the physical interrupts. For all $c \in \mathbb{N}_{nc}$:

$$M_R.VI(c).pend \cap M_R.VI(c).act = \emptyset \qquad M_R.VI(c).act \cap M_R.VI(c).ap = \emptyset$$
$$M_R.VI(c).pend \cap M_R.VI(c).ap = \emptyset$$

Inv-Hyp-Pmsg The communication with the peripherals is restricted by the hypervisor through the second-stage MMU. First of all, page table walks do not target peripherals and the GIC, as any table look-up to device memory is treated as a permission fault in the second-stage of translation. Therefore page table walks from the MMU never reach the peripherals. Moreover the hypervisor never touches peripheral memory, and the transition system only forwards matching I/O requests to the peripheral, thus the following invariant holds for all $p \in \mathbb{N}_{np}$.

$$(r, \mathrm{C}\ c) \in PR_R^p \quad \Rightarrow \quad r = mmu_R^c.walk.final \wedge req(r, A_P(p))$$

Replies to memory-mapped I/O requests are contained in the peripherals output channel and match the last request sent by the corresponding second-stage MMU of the sending core.

$$p2v_R^p = (q, \mathrm{C}\ c) \quad \Rightarrow \quad match(mmu_R^c.walk.final, q) \wedge rpl(q, A_P(p))$$

For the GIC we also require that its outstanding requests are either reads or writes to the GIC memory area.

$$(r, \mathrm{C}\ c) \in GR_R^c \quad \Rightarrow \quad req(r, A_{GIC}) \\ \wedge \quad r = \begin{cases} mmu_R^c.walk.final & : \quad req(r, A_{GICV}) \\ C_R^c.curr & : \quad \text{otherwise} \end{cases}$$

Also for replies from the GIC we have a similar invariant as for requests to the GIC.

$$p2v_R^{np} = (q, \mathrm{C}\ c) \quad \Rightarrow \quad rpl(q, A_{GIC}) \\ \wedge \quad \begin{cases} match(mmu_R^c.walk.final, q) & : rpl(q, A_{GICV}) \\ match(C_R^c.curr, q) & : \text{otherwise} \end{cases}$$

Inv-Hyp-Iso Finally, there are the invariants that guarantee memory isolation between the guests. For all cores $c \in \mathbb{N}_{nc}$ and peripherals $p \in \mathbb{N}_{np}$ it holds:

1. The second-stage MMU of core c only sends translation table look-up requests to the page table of the corresponding guest.

$$trans(mmu_R^c) \wedge r = mmu_R^c.walk.lookup \neq \text{None} \ \Rightarrow \\ PTreq(r, A_{PT}(\gamma(c)))$$

2. The second-stage MMU of core c receives during its translation phase only replies to page table walks from memory that target the page table of the corresponding guest. Moreover the value returned is the one corresponding to the requested entry in the Golden Image page table for that guest. To extract d consecutive bytes stored at address $a \in \mathbb{B}^{12}$ in a memory page $x \in \mathbb{B}^{48}$ we use notation $x_d[a]$, then:

$$trans(mmu_R^c) \wedge q = m2u_R^c \neq \text{None} \ \Rightarrow \\ \exists a, v \ . \quad q = rplPTW\ a\ v \ \wedge \ bchk_8(a, A_{PT}(\gamma(c))) \\ \wedge \quad v = GI_{\gamma(c)}(a[47:12])_8[a[11:0]]$$

3. The second-stage MMU of core c only sends memory requests to the memory of the corresponding guest. It never sends stage-one table look-ups to device memory or the GIC. Let $A_P(g) = \bigcup_{\gamma(p)=g} A_P(p) \cup A_{GICV}$ and $A_G^-(g) = A_G(g) \setminus A_P(g)$, then:

$$mmu_R^c.walk.complete \wedge r = u2m_R^c.walk.final \neq \text{None} \ \Rightarrow \\ req(r, A_G(\gamma(c))) \vee PTreq(r, A_G^-(\gamma(c)))$$

4. The SMMU of peripheral p only sends translation table look-up requests to the peripheral page table of the corresponding guest.

$$trans(smmu_R^p) \wedge r = smmu_R^p.walk.curr \neq \text{None} \ \Rightarrow \\ PTreq(r, A_{PPT}(\gamma(p)))$$

5. The SMMU of peripheral p receives during its translation phase only replies to page table walks from memory that target the peripheral page table of the corresponding guest. Moreover the value returned is the one corresponding to the requested entry in the Golden Image peripheral page table for that guest.

$$\begin{array}{l} trans(smmu_R^p) \wedge q = m2v_R^p \neq \text{None} \;\Rightarrow \\ \quad \exists a, v \;\; . \quad q = rplPTW \;\; a \;\; v \;\; \wedge \;\; bchk_8(a, A_{PPT}(\gamma(p))) \\ \qquad\quad \wedge \quad v = GIP_{\gamma(p)}(a[47:12])_8[a[11:0]] \end{array}$$

6. The SMMU of peripheral p only sends DMA requests to the memory of the corresponding guest. Here we assume that peripherals never send page table look-up requests.

$$\begin{array}{c} smmu_R^p.walk.complete \wedge r = smmu_R^c.walk.final \neq \text{None} \;\Rightarrow \\ req(r, A_G(\gamma(p))) \end{array}$$

7. Any core request currently present in memory targets the memory of the corresponding guest excluding the peripheral memory and the second-stage page tables for that guest. Peripheral request are only reads or writes or page table look-ups to the SMMU page tables. For all $r \in \mathbb{R}$:

$$\begin{array}{lcl} (r, \text{C } c) \in M_R.m.curr & \Rightarrow & req(r, A_G^-(\gamma(c))) \;\; \vee \\ & & PTreq(r, A_G^-(\gamma(c)) \cup A_{PT}(\gamma(c))) \\ (r, \text{P } p) \in M_R.m.curr & \Rightarrow & req(r, A_G^-(\gamma(p))) \vee PTreq(r, A_{PPT}(\gamma(p))) \end{array}$$

8. Each core running in guest mode only receives memory replies targeting the memory its owning guest is allowed to access as well as MMU faults. Let $F(q) \equiv \exists a.\ q = \text{F } a$ denote that reply q contains a fault, and let $A_P^g = \bigcup_{p=0}^{np_g} A_P^{g,p}$ as well as $\mathbb{A}_g^- = \mathbb{A}_g \setminus A_P^g$, then:

$$\begin{array}{c} mode_R^c < 2 \wedge q = u2c_R^c \neq \text{None} \;\Rightarrow \\ rpl(q, \mathbb{A}_{\gamma(c)}) \vee PTrpl(q, \mathbb{A}_{\gamma(c)}^-) \vee F(q) \end{array}$$

9. Any DMA reply received by peripheral p is targeting the memory of the corresponding guest or it is an SMMU fault.

$$q = v2p_R^p \neq \text{None} \;\Rightarrow\; rpl(q, \mathbb{A}_{\gamma(p)}) \vee F(q)$$

10. memory-mapped I/O requests and replies have a sender ID of a core that belongs to the same guest as the receiving peripheral.

$$(r, \text{C } c) \in PR_R^p \vee p2v_R^p = (q, \text{C } c) \;\Rightarrow\; \gamma(c) = \gamma(p)$$

These are all the implementation invariants needed for the refined model to prove the bisimulation with the ideal model. We state the bisimulation theorem below, starting with the introduction of valid refined and ideal initial states. In the context of the Common Criteria documentation, The implementation invariant defines (a superset of) the secure states *Sec* of the refined model.

4.6 Initial Configurations

The bisimulation between the ideal and the refined model is proven for two executions that start in an initial ideal or refined model states. For the refined model the initial state of the system is defined by the state of the ARMv8 hardware after a reset signal has been received. After such a reset the refined system is configured as follows.

- All core registers are reset in mode EL3, registers are set to default values. We denote the reset state for refined core c by C^c_{reset} without going into details here.
- The content of non-persistent memory (RAM) is arbitrary.
- ROM and Flash memory are assumed to contain the first stage boot loader and the HASPOC images.
- Peripherals p are in their initial state P^p_{reset} which we leave undefined here.
- The GIC distributor is initially configured to completely block all incoming interrupts. We assume here that the distributor and CPU interface registers are configured according to invariants Inv-GICPol and Inv-GICInterface, avoiding to model any GIC reconfiguration during the boot process.
- There are no pending, active, or active-and-pending physical or virtual interrupts. The virtual interrupt control registers and the virtual interrupt interface registers r are in their reset configurations $GICH_{reset}(r)$, and $GICC_{reset}(r)$ respectively.
- MMUs and SMMUs are disabled.
- There are no active peripheral interrupts.
- The message channels in the system are empty.
- The peripheral event sequence is empty.

Since non-persistent memory is initialized in an arbitrary fashion, there is not a single refined model state that counts as *the* initial state to start execution. Therefore we define a predicate *start* on refined states M_R that identifies such initial states according to the notions listed above. First however we introduce an abbreviation for the initial states of components related to cores $c \in \mathbb{N}_{nc}$.

$$\begin{aligned} start_c(M_R) \equiv\ & M_R.C(c) = C^c_{reset} \wedge \neg M_R.mmu(c).active \wedge Inv_{GICC}(M_R, c) \\ & \wedge\ c2u^c_R = u2c^c_R = u2m^c_R = m2u^c_R = \text{None} \\ & \wedge\ M_R.VI(c).pend = \emptyset \wedge M_R.VI(c).act = \emptyset \wedge M_R.VI(c).ap = \emptyset \\ & \wedge\ \forall r \in GICH.\ M_R.GIC.gich(r, c) = GICH_{reset}(r) \\ & \wedge\ \forall r \in GICC.\ M_R.GIC.gicv(r, c) = GICC_{reset}(r) \end{aligned}$$

Then the overall predicate *start* is defined as follows:

$$
\begin{array}{ll}
start(M_R) \equiv & \\
\quad & \forall c \in \mathbb{N}_{nc}.\ start_c(M_R) \\
\wedge & \forall img \in IMG_H. \\
\quad & \quad M_R.m(a) = \langle img \rangle[\langle a \rangle - \langle img.adr \rangle + 4096 \cdot 8 - 1 : \langle a \rangle - \langle img.adr \rangle] \\
\wedge & \forall p \in \mathbb{N}_{np}.\ M_R.P.st(p) = P^p_{reset} \wedge \neg M_R.smmu(p).active \wedge PI(p) = \emptyset \\
\wedge & dmsk(M_R.GIC) = (PIRQ \cup VIRQ) \times \mathbb{N}_{nc} \cup IPIRQ \\
\wedge & M_R.GIC.gicd(GICD\_CTLR)[1] = 1 \\
\wedge & M_R.Q.pend = \emptyset \wedge M_R.Q.act = \emptyset \wedge M_R.Q.ap = \emptyset \\
\wedge & \forall p \in \mathbb{N}_{np+1}.\ p2v^p_R = v2p^p_R = v2m^p_R = m2v^p_R = \text{None} \\
\wedge & M_R.E = \varepsilon
\end{array}
$$

We may define the initial state of the ideal model as we like as long as it is in the bisimulation relation with the initial refined model state. We choose to turn off the cores of each guest initially, since after reset the boot hypervisor initialization code is executed and the cores cannot be coupled with any guest context yet. The initial guest memories are arbitrary (but later coupled to be identical with the corresponding refined model memory regions) besides the IGC message buffers which are identical with the corresponding guest memory pages. All other components are defined in the obvious way. We overload predicate start for ideal model configurations as follows.

$$
\begin{array}{lll}
start(M_I) \equiv \forall g \in \mathbb{N}_{ng}, s \in IGCin_g, s' \in IGCout_g. & & \\
\quad \forall c \in \mathbb{N}_{nc_g} & . & M_I.G(g).C(c) = C_{off} \wedge M_I.G(g).rIGC(s,c) = 0 \\
\quad & \wedge & M_I.G(g).cif(c) = M_I.G(g).m2c(c) = \text{None} \\
\quad & \wedge & \forall r \in GICC.\ M_I.G(g).GIC.gicc(r,c) = GICC_{reset}(r) \\
\wedge & \forall p \in \mathbb{N}_{np_g}.\ M_I.P.st(p) = P^{g,p}_{reset} \wedge PI(p) = \emptyset & \\
\wedge & dmsk(M_I.G(g).GIC) = (PIRQ_g \cup IGCQ_g) \times \mathbb{N}_{nc} \cup IPIRQ_g & \\
\wedge & M_I.G(g).GIC.gicd(GICD\_CTLR)[1] = 0 & \\
\wedge & M_I.G(g).Q.pend = \emptyset \wedge M_I.G(g).Q.act = \emptyset \wedge M_I.G(g).Q.ap = \emptyset & \\
\wedge & \forall p \in \mathbb{N}_{np_g+1}.\ M_I.pif(p) = M_I.m2p(p) = \text{None} & \\
\wedge & M_I.G(g).sIGC(s') = 0 & \\
\wedge & M_I.G(g).E = \varepsilon & \\
\wedge & \forall g'.\ cpol(s') = (g,g') \Rightarrow & \\
\quad & \quad M_I.S(s') = M_I.G(g).m(IGCa(s')_1) = M_I.G(g').m(IGCa(s')_2) &
\end{array}
$$

Note, that from such a configuration with the current definition of the ideal model, no step is possible, as all cores are inactive ($\neg C_{off}.active$). We need to introduce a reset transition in the ideal model that allows a guest to start executing when all its cores are inactive. However, this transition should only be allowed at the beginning of a computation. Then from an initial guest configuration we allow to start the first core of a guest with the following rule, extending the guest transition system.

$$
\frac{G.init \qquad \forall c \in \mathbb{N}_{nc_g}.\ \neg G.C(c).active \qquad G'.C[0 \mapsto C^{g,0}_{reset}] \qquad \neg G'.init}{G \longrightarrow_g G'}\ \textsc{StartUp}
$$

In the bisimulation this step will be coupled with the last step of the hypervisor initialization, when the guest is actually started. Since this initialization is happening on several cores concurrently, we cannot simply assume that all guests are started at the same time, therefore the rule is needed to maintain the bisimulation when some activated cores already perform guest steps while others are still to be initialized.

4.7 Bisimulation Theorem

The bisimulation theorem is stated and proven for computations $\overline{M_R} \in \mathcal{M}_R^*$ and $\overline{M_I} \in \mathcal{M}_I^*$ given as sequences $M_R^0, M_R^1, \ldots$ and $M_I^0, M_I^1, \ldots$ of states that are in the transition relation, starting in a proper initial state. Formally this is captured by the following predicate for refined and ideal state sequences $\overline{M_{\mathrm{X}}}$ with $\mathrm{X} \in \{R, I\}$.

$$\mathit{valid}(\overline{M_{\mathrm{X}}}) \equiv \mathit{start}(M_{\mathrm{X}}^0) \wedge \forall i \geq 0.\ M_{\mathrm{X}}^i \xrightarrow{\mathrm{X}} M_{\mathrm{X}}^{i+1}$$

We can prove the by simulation only for initial states that do not cause the boot loader to fail, i.e., all images need to be correctly signed. For the initial state M_R^0 this failure condition is captured by $\mathit{fail}(M_R^0) = \mathit{fail}(hv_0(M_R^0))$.

Theorem 1 (Ideal-Refined Bisimulation)**.** *The bisimulation theorem between the ideal and real model is stated for both directions separately. 1. For any valid refined model computation $\overline{M_R}$ where images are signed correctly initially, there exists a valid ideal model computation $\overline{M_I}$ and a monotonic step function $s : \mathbb{N} \to \mathbb{N}$ such that every state of the refined computation is in the bisimulation relation with a state of the ideal model computation identified by s.*

$$\forall \overline{M_R}.\ \mathit{valid}(\overline{M_R}) \wedge \neg \mathit{fail}(M_R^0) \implies \exists \overline{M_I}, s.\ \mathit{valid}(\overline{M_I}) \wedge \forall j \geq 0.\ M_I^{s(j)} \sim M_R^j$$

2. For any ideal model computation $\overline{M_I}$ there exists a refined ideal model computation $\overline{M_R}$ and a monotonic step function $s : \mathbb{N} \to \mathbb{N}$ such that every state of the ideal computation is in the bisimulation relation with a state of the refined model computation identified by the step function.

$$\forall \overline{M_I}.\ \mathit{valid}(\overline{M_I}) \implies \exists \overline{M_R}, s.\ \mathit{valid}(\overline{M_R}) \wedge \forall j \geq 0.\ M_I^j \sim M_R^{s(j)}$$

In both cases we have $s(0) = 0$ and $\forall j \geq 0.\ s(j+1) \geq s(j)$.

The theorem is illustrated in Fig. 19. Note that the bisimulation relation is maintained for every step of the refined model, i.e., even intermediate hypervisor steps of the refined model are still coupled with a (possibly zero) number of corresponding ideal steps. Note moreover that both directions are formulated symmetrically, except that the simulation of the refined model by the ideal model requires the images to be signed correctly. This is not needed for the ideal model where the images are not visible at all. In the simulation of the ideal model by the refined model we need to select an initial refined state that does not make the boot fail.

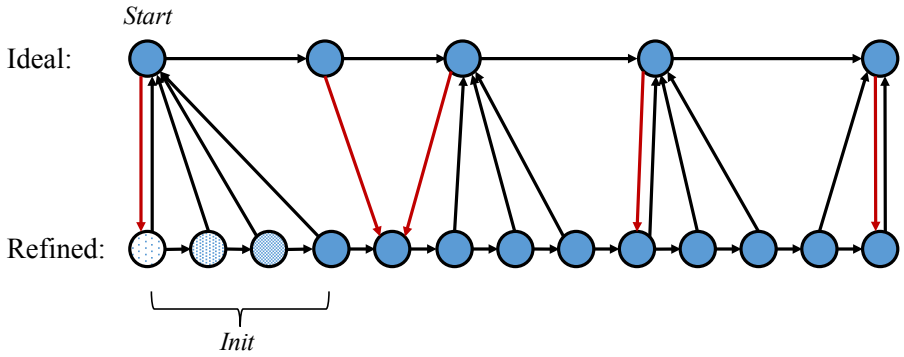

Figure 19: Illustration of the Bisimulation Theorem. Black arrows denote the bisimilar states in the simulation of the ideal by the refined model, red arrows denote bisimilar states in the other direction. Fully blue circles denote states in which the invariant is established for all guests and all cores. In lighter colored states during initialization, the invariant holds only partially

For the case that the images are not signed correctly, we introduce a second theorem, showing that the refined model never enters an insecure state.

Theorem 2 (Secure Failure)**.** *Any valid computation of the the refined model starting in a state where the HASPOC images are not correctly signed will lead into a secure state where only the primary core is running and is stalled in an endless loop.*

$$\begin{aligned}&\forall \overline{M_R}.\ \mathit{valid}(\overline{M_R}) \wedge \mathit{fail}(M_R^0) \implies \\ &\quad \forall j \geq 0.\ \mathit{Inv}(M_R^j) \wedge \exists k.\ j \geq k \Rightarrow \\ &\quad\quad M_R^j.C(0).pc = a_{\mathit{fail}} \wedge \forall c \neq 0.\ M_R^j.C(c) = C_{\mathit{off}}\end{aligned}$$

Proof. If $\mathit{fail}(M_R^0)$, the only possible step for the primary core is Init-Abort which sets the program counter to a_{fail}. From this hypervisor state only stuttering steps are possible on the primary core. The only possible step for the secondary cores is Init-SCold which powers them off. Assuming a fair scheduling of the refined model (every enabled transition rule is eventually applied), there exists a step k in which all cores are powered off and the primary core is stalled in address a_{fail}.

The implementation invariant holds initially, as by $\mathit{start}(M_R^0)$ all HASPOC images are initially stored in flash memory, no interrupts are pending, all of them are masked, and the GIC distributor and interface registers are in the desired initial states. These facts imply invariants Inv-Img, Inv-GICPol, and Inv-GICInterface which are the only ones that need to hold before the hypervisor is started. As neither Init-Abort nor Init-SCold touch memory or the GIC, the implementation invariant is maintained by all further (possibly stuttering) steps of the refined model. □

The observation that in initial states of the refined model the implementation invariant always holds is formalized in the following corollary.

Corollary 1 (Initial Invariant)**.** *For all refined model states M_R we have:*

$$start(M_R) \Longrightarrow Inv(M_R)$$

Theorems 1 and 2 make up the HASPOC security policy P. Note that only Theorem 2 talks about the secure states *Sec* of the system, i.e., states reachable from a valid start configuration where the implementation invariant holds. For the formulation of Theorem 1 the implementation invariant is not needed, however we need it to prove the theorem. Below we explain our proof strategy.

4.8 Decomposition

The proof is split into four lemmas as follows.

Lemma 1 (Induction Start)**.** *For any initial ideal model state there exists a bisimilar initial refined model state and vice versa.*

$$\begin{array}{l} \forall M_I^0 \in \mathcal{M}_I.\ start(M_I^0) \ \Rightarrow\ \exists M_R^0 \in \mathcal{M}_R.\ start(M_R^0) \wedge M_I^0 \sim M_R^0 \\ \forall M_R^0 \in \mathcal{M}_R.\ start(M_R^0) \ \Rightarrow\ \exists M_I^0 \in \mathcal{M}_I.\ start(M_I^0) \wedge M_I^0 \sim M_R^0 \end{array}$$

Lemma 2 (Invariant)**.** *The implementation invariant is maintained by all steps of the refined model.*

$$\forall M_R, M_R'.\ Inv(M_R) \wedge M_R \xrightarrow{R} M_R' \ \Rightarrow\ Inv(M_R')$$

For the induction step, i.e., the proof that the bisimulation relation is maintained by arbitrary steps of the ideal or refined model, we need two different inductive arguments that are specific to the direction of the bisimulation and the way we step the corresponding simulating model. We denote these *simulation invariants* by predicates $sim_I : \mathcal{M}_I \to \mathbb{B}$, and $sim_R : \mathcal{M}_R \to \mathbb{B}$. Their exact definitions will be given later. Here we just demand that they hold in valid initial configurations.

Lemma 3 (Simulation Invariants)**.** *The simulation invariants hold in all initial states of the corresponding models.*

$$\forall M_R, M_I.\ start(M_I) \Rightarrow sim_I(M_I) \ \wedge\ start(M_R) \Rightarrow sim_R(M_R)$$

Then in the induction step we demand that both the bisimulation relation and the simulation invariants are maintained for steps of both models, given that the implementation invariant holds on the refined model.

Lemma 4 (Induction Step)**.** *Given a pair of bisimilar ideal and refined states, where the implementation invariant is holding for the latter one and the ideal*

simulation invariant is holding for the former, for any next refined configuration, there exists a number of ideal steps leading into a bisimilar ideal state maintaining the ideal simulation invariant. The same property holds with ideal and refined model in reversed roles. We state both directions separately for all M_R, M_I such that $Inv(M_R)$ and $M_R \sim M_I$:

$$\forall M'_R.\ M_R \xrightarrow{R} M'_R \wedge sim_I(M_I) \ \Rightarrow\ \exists t.\ M_I \xrightarrow{I}{}^{t} M_I^t \wedge M_I^t \sim M'_R \wedge sim_I(M_I^t)$$
$$\forall M'_I.\ M_I \xrightarrow{I} M'_I \wedge sim_R(M_R) \ \Rightarrow\ \exists t.\ M_R \xrightarrow{R}{}^{t} M_R^t \wedge M'_I \sim M_R^t \wedge sim_R(M_R^t)$$

Above we use a notation $M_{\mathrm{X}} \xrightarrow{\mathrm{X}}{}^{\mathrm{t}} M_{\mathrm{X}}^t$ for $\mathrm{X} \in \{I, R\}$ to state that a configuration M_{X}^t is reached after t computation steps of the ideal or refined model, respectively, starting in configuration M_{X}. It is define recursively as follows:

$$\begin{aligned} M_{\mathrm{X}} \xrightarrow{\mathrm{X}}{}^{0} M_{\mathrm{X}}^0 &\equiv (M_{\mathrm{X}} = M_{\mathrm{X}}^0) \\ \forall t > 0.\ M_{\mathrm{X}} \xrightarrow{\mathrm{X}}{}^{\mathrm{t}} M_{\mathrm{X}}^t &\equiv \exists M'_{\mathrm{X}}.\ M_{\mathrm{X}} \xrightarrow{\mathrm{X}} M'_{\mathrm{X}} \wedge M'_{\mathrm{X}} \xrightarrow{\mathrm{X}}{}^{\mathrm{t\text{-}1}} M_{\mathrm{X}}^t \end{aligned}$$

Note that this lemma requires that whenever a step is possible in one model there needs to be at least one possible simulating step in the other one, i.e., the simulation relation needs to ensure that the same steps are possible in both simulated transition systems.

Moreover, observe that the simulation invariants only needs to hold in the states of the simulating model that are coupled by the bisimulation relation, in intermediate states it may be broken. For example, as explained earlier, in the simulation of the ideal model we step the refined model such that the GIC distributor virtualization handlers are always executed in one block, thus the GIC lock is never taken in the states that are coupled with the ideal model computation. However in intermediate steps the lock is taken (and subsequently released) by the handler, thus the simulation relation (demanding the lock to be never taken) does not hold in the corresponding intermediate states.

Given that these four Lemmas hold we can easily prove Theorem 1. Instead of the original simulations we prove refined versions that add the respective simulation invariant to the claim, i.e., we show in addition that the simulating computations maintain the respective simulation invariant. The original bisimulation theorem is still implied by the existence of the simulating computations.

Proof of Theorem 1. We show both directions separately by induction on the number of steps i of the simulated computation. The induction start in both cases is given by Lemma 1 and $s(0) = 0$, i.e., $M_I^0 \sim M_R^0$ and the respective simulation invariant holds by Lemma 3. Also in both directions the implementation invariant holds in each step of the refined computation $\overline{M_R}$ by Corollary 1 (for the initial state) and inductive application of Lemma 2. Thus in the induction step $Inv(M_R^i)$, and $Inv(M_R^{s(i)})$ respectively, holds. By the induction hypothesis we have $M_I^{s(i)} \sim M_R^i$ and $sim_I(M_I^{s(i)})$, or $M_I^i \sim M_R^{s(i)}$ and $sim_R(M_R^{s(i)})$ respectively. We apply Lemma 4 and conclude with $s(i+1) = s(i) + t$ for both directions. □

4.9 Component Constraints

The implementation invariants, and thus the bisimulation proof, rely on certain assumptions about the underlying hardware model. We formulate these assumptions as constraints on the transition relations of the cores, the memory, the GIC, and the peripherals. Note that these constraints need to be proven for given instantiations of these components, however these proofs are out of the scope of this document.

4.9.1 Core Constraints

In this work we assumes that cores are sequential, i.e., they execute instructions and issue memory requests in the order of the machine code program they are running. Moreover, each core only issues at most memory request at a time. We assume that a core stalls until it receives a reply for an outstanding memory request. Memory requests are outstanding while the *curr* history variable of the core has a defined value.

Assumption 1 (Sequential Core)**.** *While there is an outstanding memory request in a refined model core and a corresponding reply has not yet been delivered to it, any step that is in the transition transition relation for that core state is stuttering.*

$$\begin{array}{l} \forall C, u2c, c2u, q_{in}.\ C.curr \neq \bot \wedge \neg match(C.curr, u2c) \ \Rightarrow \\ \quad \forall C', u2c', c2u'.\ \delta_R(C, u2c, c2u, q_{in}) \ni (C', u2c', c2u') \ \Rightarrow \\ \quad\quad C' = C \wedge u2c' = u2c \wedge c2u' = c2u \end{array}$$

Observe that this property does excludes certain implementations of out-of-order execution and weakly consistent memory behaviour. However core implementations are still allowed to optimize as long as these optimizations do not break the sequential core and memory semantics.

Note also that the ideal model version of the core transition relation needs to exhibit the same behaviour.

Assumption 2 (Sequential Ideal Core)**.** *While there is an outstanding memory request in an ideal model core and a corresponding reply has not yet been delivered to it, any step that is in the transition transition relation for that core state is stuttering.mem*

$$\begin{array}{l} \forall C, m2c, cif, q_{in}.\ C.curr \neq \bot \wedge \neg match(C.curr, m2c) \ \Rightarrow \\ \quad \forall C', m2c', cif', e.\ \delta^g_I(C, m2c, cif, q_{in}) \ni (C', m2c', cif', e) \ \Rightarrow \\ \quad\quad C' = C \wedge m2c' = m2c \wedge cif' = cif \wedge e = \varepsilon \end{array}$$

Note moreover that these assumptions make a core prone to deadlock in case the wrong reply is delivered to it. We put a constraint on the memory to avoid this together with the implementation invariants on the reply channels. Additionally, we require that each core step consumes inputs before producing new memory outputs or changing the core state.

Assumption 3 (Core Messages)**.** *When the core makes a step the input and output channels are updated such that at most one of them contains a message. The core also never modifies already present output messages and only modifies input messages by consuming them. The core always consumes matching input messages for outstanding replies.*

$$\begin{aligned}
&\forall C, u2c, c2u, q_{in}, C', u2c', c2u'.\\
&\quad \delta_R(C, u2c, c2u, q_{in}) \ni (C', u2c', c2u') \;\Rightarrow\\
&\qquad (u2c' = \text{None} \vee c2u' = \text{None}) \wedge\\
&\qquad c2u \neq \text{None} \Rightarrow c2u' = c2u \wedge\\
&\qquad match(C.curr, u2c) \Rightarrow u2c' = \text{None}
\end{aligned}$$

We have a similar constraints for the ideal core.

Assumption 4 (Ideal Core Messages)**.** *When an ideal core of guest g makes a step the input and output channels are updated such that at most one of them contains a message. Moreover, the core never modifies already present output messages and only modifies input messages by consuming them. The core always consumes matching input messages for outstanding replies.*

$$\begin{aligned}
&\forall C, m2c, cif, q_{in}, C', m2c', cif', e.\\
&\quad \delta_I^g(C, m2c, cif, q_{in}) \ni (C', m2c', cif', e) \;\Rightarrow\\
&\qquad (m2c' = \text{None} \vee cif' = \text{None}) \wedge\\
&\qquad cif \neq \text{None} \Rightarrow cif' = c2u \wedge\\
&\qquad match(C.curr, m2c) \Rightarrow m2c' = \text{None}
\end{aligned}$$

For the bisimulation proof, we also require that while the guest is running the same core transitions are possible in the ideal and the refined model, if the guest registers are coupled according to the bisimulation relation.

Assumption 5 (Bisimilar Core Step)**.** *Given an ideal and a refined core state, if all guest-accessable registers have the same values, the core is running in guest mode, the same inputs are pending, and the cores have the same outstanding memory requests, then the same transitions are possible from both states as long as the refined step does not enter EL2 or EL3. We define this notion of core similarity by*

$$\begin{aligned}
C_I \sim_C C_R \equiv\ &\forall r \in GPR_{\mathbb{G}}.\ C_R.gpr(r) = C_I.gpr(r)\\
&\wedge\ \forall r \in SPR_{\mathbb{G}}.\ C_R.gpr(r) = C_I.spr(r)\\
&\wedge\ C_R.pc = C_I.pc \wedge C_R.ps = C_I.ps \wedge C_R.curr = C_I.curr
\end{aligned}$$

Then the bisimilarity property is defined as follows.

$$\begin{aligned}
&\forall g, C_R, C_I, m2c, cif, m2c', cif', q \in IRQ_g.\ C_I \sim_C C_R \wedge C_R.ps.mode < 2 \;\Rightarrow\\
&\quad \forall C_R'.\ \delta_R(C_R, m2c, cif, virq_q) \ni (C_R', m2c', cif') \wedge C_R'.ps.mode < 2 \;\Rightarrow\\
&\qquad \exists C_I'.\ \delta_I^g(C_I, m2c, cif, q) \ni (C_I', m2c', cif', \varepsilon) \wedge C_I' \sim_C C_R'\\
\wedge\quad &\qquad \forall C_I'.\ \delta_I^g(C_I, m2c, cif, q) \ni (C_I', m2c', cif', \varepsilon) \;\Rightarrow\\
&\qquad \exists C_R'.\ \delta_R(C_R, m2c, cif, virq_q) \ni (C_R', m2c', cif') \wedge\\
&\qquad\quad C_R'.ps.mode < 2 \wedge C_I' \sim_C C_R'
\end{aligned}$$

Note that this assumption is trivially fulfilled by our ARMv8 model, as the refined transition relation is just an extension of the ideal transition relation with EL2 and EL3 functionality.

4.9.2 Memory Constraints

We introduce a number of constraints on the memory transition relation, that restrict the memory from doing unwarranted memory updates and sending back arbitrary replies. As the memory is modeled as a page-addressable mapping but accesses may target only a number of bytes we introduce the following shorthand to extract $d \in \mathbb{N}_{4096}$ bytes stored at address $a \in \mathbb{B}^{48}$ out of a page addressable memory $m : \mathbb{B}^{36} \to \mathbb{B}^{4096 \cdot 8}$.

$$m_d(a) = m(a[47:12])_d[a[11:0]]$$

We demand that this access returns bytes in little-endian byte order. We assume that all memory accesses are little endian and the cores reverse the bytes as necessary.

Assumption 6 (Memory Persistence)**.** *Any byte in memory only changes by a memory transition if there was a currently outstanding write request to it that was handled by the memory. For all $a \in \mathbb{B}^{48}$, and $s \in \mathbb{S}$:*

$$\begin{aligned}\forall m, u2m, m', m2u'.\ \delta_M(m, u2m, s) \ni (m', m2u') \land m_1(a) \neq m'_1(a) \Rightarrow \\ \exists r, b, d, v\ .\ r = \mathrm{W}\ b\ d\ v \land \langle a \rangle \in [\langle b \rangle : \langle b \rangle + d - 1] \\ \land\ (r, s) \in m.curr \land (r, s) \notin m'.curr \land m2u' \neq \mathrm{None}\end{aligned}$$

Conversely, this constraint guarantees that memory cells do not change there values unless they are written. Moreover changes in memory are restricted to the steps where a write reply is generated by the memory. As only one reply can be generated at a time this means that the memory sequentializes the handling of memory requests internally.

Assumption 7 (Request Semantics)**.** *If a request is received by the memory it is stored in the set of outstanding requests and no reply is given.*

$$\begin{aligned}\forall m, u2m, m', m2u', s.\\ \delta_M(m, u2m, s) \ni (m', m2u') \land u2m \neq \mathrm{None}\ \Rightarrow \\ (u2m, s) \in m'.curr \land m2u' = \mathrm{None}\end{aligned}$$

Thus we require the memory to first consume a request and produce a reply only in another step.

Assumption 8 (Reply Semantics)**.** *A reply is produced by memory only if there was a corresponding request in memory before the step. Then, by the semantics of the curr history variable, that request is also removed from the set of outstanding memory requests.*

$$\begin{aligned}\forall m, u2m, m', m2u', s.\ \delta_M(m, u2m, s) \ni (m', m2u')\ \Rightarrow \\ m2u' \neq \mathrm{None}\ \Rightarrow\ \exists r \in \mathbb{R}\ .\ & (r, s) \in m.curr \\ \land\ & match(r, m2u') \\ \land\ & m'.curr = m.curr \setminus \{(r, s)\}\end{aligned}$$

Moreover for the different types of requests, the returned value matches the current value of memory. For writes the meomry is updated according to the write value of the request.

$$
\begin{aligned}
&\wedge \quad m2u' = rplR\ a\ v \ \Rightarrow\ v = m'_d(a) \\
&\wedge \quad m2u' = rplW\ a \ \Rightarrow \\
&\qquad \forall a' \in \mathbb{B}^{48}.\ m_1(a') = \begin{cases} v[8o+7:8o] & : \quad o = \langle a' \rangle - \langle a \rangle \in \mathbb{N}_d \\ m_1(a') & : \quad \text{otherwise} \end{cases} \\
&\wedge \quad m2u' = rplPTW\ a\ d \ \Rightarrow\ v = m'_8(a)
\end{aligned}
$$

Note that in combination with Assumption 6 we this implies the memory never changes when the a reply for reads or page table look-ups is produced.

Similar to the bisimilar core step assumption, we require that the same memory steps are possible in two configurations that have the corresponding pending requests to guest memory and the same memory contents wrt. address translation.

Assumption 9 (Bisimilar Memory Step)**.** *Given an ideal and a refined memory state, if all guest-accessable addresses have the same values stored and the same inputs and requests are pending wrt. address translation, then the same transitions are possible from both states. Again we introduce a notion of similarity, here for the memory state of guest g.*

$$
\begin{aligned}
m_I \sim^g_M m_R \equiv\ & \forall a \in A_G(g).\ m_R(a) = m_I(\theta_g(a)) \\
& \wedge\ \forall r \in \mathbb{R}, c \in \mathbb{N}_{nc}.\ \gamma(c) = g \ \Rightarrow \\
& \qquad (r, \mathrm{C}\ c) \in m_R.curr \Leftrightarrow (\theta_g^{-1}(r), \mathrm{C}\ \kappa(c)) \in m_I.curr \\
& \wedge\ \forall r \in \mathbb{R}, p \in \mathbb{N}_{np}.\ \gamma(p) = g \ \Rightarrow \\
& \qquad (r, \mathrm{P}\ p) \in m_R.curr \Leftrightarrow (\theta_g^{-1}(r), \mathrm{P}\ \varphi(p)) \in m_I.curr
\end{aligned}
$$

Then the bisimilarity property is defined as follows.

$$
\begin{aligned}
&\forall m_R, m_I, u2m, q_{in}, m'_R, m'_I, m2u', g \in \mathbb{N}_{ng}, s_R, s_I \in \mathbb{S}.\ m_I \sim^g_M m_R \\
&\quad \wedge \quad s_R = \mathrm{C}\ c \Leftrightarrow s_I = \mathrm{C}\ \kappa(c) \ \wedge \ s_R = \mathrm{P}\ p \Leftrightarrow s_I = \mathrm{P}\ \varphi(p) \\
&\quad \wedge \quad s_R = \mathrm{C}\ c \Rightarrow \gamma(c) = g \ \wedge \ s_R = \mathrm{P}\ p \Rightarrow \gamma(p) = g \\
&\quad \wedge \quad u2m \neq \mathrm{None} \Rightarrow adr(u2m) \in A_G(g) \\
&\quad \wedge \quad m2u' \neq \mathrm{None} \Rightarrow adr(u2m) \in A_G(g) \quad \Rightarrow \\
&\qquad\qquad \forall m'_R.\ \delta_M(m_R, u2m, s_R) \ni (m'_R, m2u') \Rightarrow \\
&\qquad\qquad \delta_M(m_I, \theta_g^{-1}(u2m), s_I) \ni (m'_I, \theta_g^{-1}(m2u')) \wedge m'_I \sim^g_M m'_R \\
&\qquad \wedge \quad \forall m'_R.\ \delta_M(m_I, \theta_g^{-1}(u2m), s_I) \ni (m'_I, \theta_g^{-1}(m2u')) \Rightarrow \\
&\qquad\qquad \delta_M(m_R, u2m, s_R) \ni (m'_R, m2u') \wedge m'_I \sim^g_M m'_R
\end{aligned}
$$

Note that we do not constrain the resulting memories to contain the same values for guest g, as this is already guaranteed by the antecedant and Assumption 8.

The property stated above assumes a notion of locality, i.e., the behaviour of the memory for accesses to a given location does not depend on accesses to

other locations. Moreover, concurrent accesses to the same location by different cores do not restrict the set of possible memory transitions for a given core or peripheral. For a weakly consistent memory with shared caches and a more complex hierarchical structure of storages, these assumptions possibly have to be relaxed.

4.9.3 GIC Constraints

For the GIC we introduce several assumptions focussing how the GIC changes its outputs and register state. First of all, we require that memory-mapped I/O requests are consumed by the GIC immediately without producing a direct reply or changing the interrupt state.

Assumption 10 (GIC I/O Requests)**.** *When the GIC consumes a memory-mapped I/O request, it does not directly produce an output. Also all the interrupt state is unchanged.*

$$\begin{array}{l}\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m'.\\ \quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \;\Rightarrow\\ \qquad m2g \neq \text{None} \;\Rightarrow\; g2m' = \text{None} \wedge GIC'.curr = GIC.curr \cup \{m2g\}\\ \qquad\qquad\quad \wedge\; Q' = Q \wedge VI' = VI\end{array}$$

The property holds similarly for the ideal GIC transition function.

Assumption 11 (Ideal GIC I/O Requests)**.** *When the ideal GIC consumes a memory-mapped I/O request, it does not produce any output or changes the interrupt state except for the prioritized delivery raised IGC interrupts.*

$$\begin{array}{l}\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC'.\\ \quad \delta^g_Q(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \;\Rightarrow\\ \qquad m2g \neq \text{None} \;\Rightarrow\; g2m' = \text{None} \wedge GIC'.curr = GIC.curr \cup \{m2g\}\\ \qquad\qquad\quad \wedge\; \forall c \in \mathbb{N}_{nc_g}, x \in \{pend, act, ap\}.\\ \qquad\qquad\qquad Q'(c).\mathrm{x} \setminus IGCQ = Q(c).\mathrm{x} \setminus IGCQ\end{array}$$

The exception for IGC interrupts makes sure that the ideal model is in sync with the refined model wrt. such interrupts, whenever a memory-mapped I/O request reaches the GIC of the corresponding guest.

Concerning outputs, we demand first of all, that the GIC never accesses memory on its own behalf, i.e., it does not perform DMA accesses.

Assumption 12 (No GIC DMA)**.** *The GIC only sends replies to I/O requests to memory.*

$$\begin{array}{l}\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m'.\\ \quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \;\Rightarrow\; g2m' \in \mathbb{Q} \times \mathbb{S}\end{array}$$

We have a similar property for the ideal model.

Assumption 13 (No Ideal GIC DMA)**.** *In the ideal model, the GIC only sends replies to I/O requests to memory. For all* $g \in \mathbb{N}_{ng}$*:*

$$\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC'.$$
$$\delta_Q^g(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \;\Rightarrow\; g2m' \in \mathbb{Q} \times \mathbb{S}_g$$

For the ideal GIC we already requested that it does not raise IGC interrupts by itself but is only used to acknowledge them. We repeat this assumption here.

Assumption 14 (IGC Reception)**.** *IGC notification interrupts can only be raised through the transition system but not through the GIC itself. Moreover the GIC forwards any received IGC notification to the pending or pending-and-active interrupts immediately, unless it is producing an I/O reply.*

$$\begin{aligned}
&\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC'.\ g2m' = \text{None} \wedge \\
&\quad \delta_Q^g(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \;\Rightarrow \\
&\qquad \forall s \in IGCin_g, c \in \mathbb{N}_{nc_g}.\ rIGC'(s,c) = 0 \\
&\qquad\quad \wedge\ rIGC(s,c) = 1 \wedge \text{igc}_s \notin Q_c \;\Rightarrow\; Q'(c).pend = Q(c).pend \cup \{\text{igc}_s\} \wedge \\
&\qquad\qquad\quad Q'(c).act = Q(c).act \wedge Q'(c).ap = Q(c).ap \\
&\qquad\quad \wedge\ rIGC(s,c) = 1 \wedge \text{igc}_s \in Q(c).pend \cup Q(c).ap \;\Rightarrow\; Q'(c) = Q(c) \\
&\qquad\quad \wedge\ rIGC(s,c) = 1 \wedge \text{igc}_s \in Q(c).act \;\Rightarrow\; Q'(c).ap = Q'(c).ap \cup \{\text{igc}_s\} \wedge \\
&\qquad\qquad\quad Q'(c).act = Q(c).act \setminus \{\text{igc}_s\} \wedge Q'(c).pend = Q(c).pend
\end{aligned}$$

Moreover, the GIC only returns a reply for requests that have been sent to it and. For read and write requests to the GIC, the right registers should be read and updated.

Assumption 15 (GIC I/O Replies)**.** *Once a request is outstanding in the GIC, we demand that it can be served at any time.*

$$\forall GIC, Q, VI, PI, r, c.\ (r, \text{C } c) \in GIC.curr \;\Rightarrow\; \exists GIC', Q', VI', q.$$
$$\delta_Q(GIC, Q, VI, \text{None}, PI) \ni (GIC', Q', VI', (q, \text{C } c)) \wedge match(r, q)$$

If the GIC produces a reply to a core, it is always in respond to a request received earlier.

$$\begin{aligned}
&\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m', q, c. \\
&\quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \;\Rightarrow \\
&\qquad g2m' = (q, \text{C } c) \;\Rightarrow\; \exists r \;.\; (r, \text{C } c) \in GIC.curr \\
&\qquad\qquad\qquad\qquad \wedge\ match(r, q) \\
&\qquad\qquad\qquad\qquad \wedge\ GIC'.curr = GIC.curr \setminus \{(r, \text{C } c\}
\end{aligned}$$

Moreover the GIC state is updated only for writes, according to the request that was handled, and reads return the current register values (or a fault for invalid addresses). All accesses to GIC registers need to be properly aligned and have the right access size, otherwise a fault is returned as well. For all $a \in \mathbb{B}^{48}, d \in \mathbb{N}, v \in \mathbb{B}^{8d}$ *reads have the following effect:*

$$\begin{aligned}
&\wedge\ g2m' = rplR\ a\ v \wedge r_{GIC}(a,d) \in GICC \;\Rightarrow\; v = GIC'_d.gicc(r_{GIC}(a,d), c) \\
&\wedge\ g2m' = rplR\ a\ v \wedge r_{GIC}(a,d) \in GICD \;\Rightarrow\; v = GIC'_d.gicd(r_{GIC}(a,d)) \\
&\wedge\ g2m' = rplR\ a\ v \wedge r_{GIC}(a,d) \in GICH \;\Rightarrow\; v = GIC'_d.gich(r_{GIC}(a,d), c) \\
&\wedge\ g2m' = rplR\ a\ v \wedge r_{GIC}(a,d) \in GICV \;\Rightarrow\; v = GIC'_d.gicv(r_{GIC}(a,d), c)
\end{aligned}$$

For writes we need to use the GIC update function ν. We have:

$$
\begin{array}{ll}
\wedge & g2m' = rplW\ a \wedge r_{GIC}(a,d) \in GICC \ \Rightarrow\ \forall r \in GICC. \\
& \quad GIC'.gicc(r,c) = \begin{cases} \nu(r, GIC.gicc(r,c), v) & : \quad r = r_{GIC}(a,d) \\ GIC.gicc(r,c) & : \quad \text{otherwise} \end{cases} \\
\wedge & g2m' = rplW\ a \wedge r_{GIC}(a,d) \in GICD \ \Rightarrow\ \forall r \in GICD. \\
& \quad GIC'.gicd(r) = \begin{cases} \nu(r, GIC.gicd(r), v) & : \quad r = r_{GIC}(a,d) \\ GIC.gicd(r) & : \quad \text{otherwise} \end{cases} \\
\wedge & g2m' = rplW\ a \wedge r_{GIC}(a,d) \in GICH \ \Rightarrow\ \forall r \in GICH. \\
& \quad GIC'.gich(r,c) = \begin{cases} \nu(r, GIC.gich(r,c), v) & : \quad r = r_{GIC}(a,d) \\ GIC.gich(r,c) & : \quad \text{otherwise} \end{cases} \\
\wedge & g2m' = rplW\ a \wedge r_{GIC}(a,d) \in GICV \ \Rightarrow\ \forall r \in GICV. \\
& \quad GIC'.gicv(r,c) = \begin{cases} \nu(r, GIC.gicv(r,c), v) & : \quad r = r_{GIC}(a,d) \\ GIC.gicv(r,c) & : \quad \text{otherwise} \end{cases} \\
\wedge & g2m' = \mathrm{F}\ a \ \Leftrightarrow\ r_{GIC}(adr(\epsilon(GIC.curr \setminus GIC'.curr))) = \bot
\end{array}
$$

For the ideal model we have an analogous property, restricting only accesses to the GIC CPU interface and the distributor.

Assumption 16 (Ideal GIC I/O Replies)**.** *Once a request is outstanding in the GIC, we demand that it can be served at any time. For all guests $g \in \mathbb{N}_{ng}$ we have:*

$$
\begin{array}{l}
\forall GIC, Q, PI, rIGC, r, c.\ (r, \mathrm{C}\ c) \in GIC.curr \ \Rightarrow\ \exists GIC', Q', rIGC', q. \\
\quad \delta^g_Q(GIC, Q, \mathrm{None}, PI, rIGC) \ni (GIC', Q', (q, \mathrm{C}\ c), rIGC') \wedge match(r,q)
\end{array}
$$

Moreover, if the ideal GIC produces a reply to a core, it is always in respond to a request received earlier.

$$
\begin{array}{l}
\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC', q, c. \\
\quad \delta^g_Q(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \ \Rightarrow \\
\qquad g2m' = (q, \mathrm{C}\ c) \ \Rightarrow\ \exists r\ .\ (r, \mathrm{C}\ c) \in GIC.curr \\
\qquad\qquad\qquad\qquad \wedge\ match(r,q) \\
\qquad\qquad\qquad\qquad \wedge\ GIC'.curr = GIC.curr \setminus \{(r, \mathrm{C}\ c)\}
\end{array}
$$

Same as in the refined model, the GIC state is updated only for writes and reads return the current register values. Function

$$r^g_{GIC} : \mathbb{B}^{48} \times \mathbb{N} \rightharpoonup GICC \cup GICD$$

returns the targeted GIC register for guest g. Writes to the distributor are filtered by the ideal GIC using function η^r_g, allowing only settings for interrupts that belong to guest g. For all $a \in \mathbb{B}^{48}, d \in \mathbb{N}, v \in \mathbb{B}^{8d}$ we have:

$$
\begin{array}{ll}
\wedge & g2m' = rplR\ a\ v \wedge r^g_{GIC}(a,d) \in GICC \ \Rightarrow\ v = GIC'_d.gicc(r_{GIC}(a,d), c) \\
\wedge & g2m' = rplR\ a\ v \wedge r^g_{GIC}(a,d) \in GICD \ \Rightarrow\ v = GIC'_d.gicd(r_{GIC}(a,d))
\end{array}
$$

$$\wedge \;\; g2m' = rplW\ a \wedge r^g_{GIC}(a,d) \in GICC \;\Rightarrow\; \forall r \in GICC.$$
$$GIC'.gicc(r,c) = \begin{cases} \nu(r, GIC.gicc(r,c), v) & : \quad r = r_{GIC}(a,d) \\ GIC.gicc(r,c) & : \quad \text{otherwise} \end{cases}$$
$$\wedge \;\; g2m' = rplW\ a \wedge r^g_{GIC}(a,d) \in GICD \;\Rightarrow\; \forall r \in GICD$$
$$GIC'.gicd(r) = \begin{cases} \eta^r_g(\nu(r, GIC.gicd(r), v)) & : \quad r = r_{GIC}(a,d) \\ GIC.gicd(r) & : \quad \text{otherwise} \end{cases}$$
$$\wedge \;\; g2m' = \mathrm{F}\ a \;\Leftrightarrow\; r_{GIC}(adr(\epsilon(GIC.curr \setminus GIC'.curr))) = \bot$$

In all other cases, some registers might change due to interrupt reception and delivery, however the interrupt masks stay the same.

Assumption 17 (Mask Persistence)**.** *The GIC never changes the interrupt masks unless by responding to a memory-mapped write request.*

$$\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m'.$$
$$\delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \wedge \neg Wrpl(g2m', A_{GIC}) \;\Rightarrow$$
$$msk(GIC') = msk(GIC) \wedge dmsk(GIC') = dmsk(GIC)$$

Again the property holds similarly for the ideal GIC.

Assumption 18 (Ideal Mask Persistence)**.** *The GIC never changes the interrupt masks unless by responding to a memory-mapped write request.*

$$\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC'.$$
$$\delta^g_Q(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \wedge \neg Wrpl(g2m', A^{g,np}_P) \;\Rightarrow$$
$$msk_g(GIC') = msk_g(GIC) \wedge dmsk(GIC') = dmsk(GIC)$$

Similar to the mask, the GIC distributor registers to which read accesses can be emulated by the hypervisor do not change their values unless in a response to a write to them.

Assumption 19 (Distributor Persistence)**.** *If a read to a GICD register can be emulated by the hypervisor, this register only changes its value due to a write request.*

$$\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m', r \in GICD.\ emugicd(r,0)$$
$$\wedge\ \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \;\Rightarrow$$
$$\nexists d, v.\ g2m' = rplW\ a_{GICD}(r) \;\Rightarrow\; GIC'.gicd(r) = GIC.gicd(r)$$

The property holds similarly for the ideal GIC.

Assumption 20 (Ideal Distributor Persistence)**.** *If a read to a GICD register can be emulated by the hypervisor, this register only changes its value in the ideal GIC due to a write request. For all $g \in \mathbb{N}_{ng}$:*

$$\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC', r \in GICD.\ emugicd(r,0)$$
$$\wedge\ \delta^g_Q(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \;\Rightarrow$$
$$\nexists d, v.\ g2m' = rplW\ a^g_{GICD}(r) \;\Rightarrow\; GIC'.gicd(r) = GIC.gicd(r)$$

On the CPU interrupt interface only two registers may change due to non-secure interrupt delivery.

Assumption 21 (Interface Persistence)**.** *The registers of the CPU interrupt interface may only change due to a memory-mapped I/O request, except for the aliased interrupt acknowledgement register, the aliased highest priority pending interrupt register.*

$$\begin{aligned}&\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m', r \in GICC.\\ &\quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', \text{None}) \;\Rightarrow\\ &\qquad r \notin \{GICC\_AIAR, GICC\_AHPPIR\} \Rightarrow GIC'.gicc(r) = GIC.gicc(r)\end{aligned}$$

Note that we ignore changes to the secure aliases of these registers here, as the HASPOC platform does not use secure interrupts.

Similarly, inter-processor interrupts can only be raised or deactivated by memory-mapped I/O accesses to the GIC distributor or the CPU interface.

Assumption 22 (SGI Persistence)**.** *Any SGI interrupt maintains its status unless there is an access to the GIC distributor or the CPU interface.*

$$\begin{aligned}&\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m'.\\ &\quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \;\Rightarrow\\ &\qquad (\forall q, c.\ g2m' = (q, \mathrm{C}\ c) \Rightarrow \neg rpl(q, A_{GICC} \cup A_{GICD})) \;\Rightarrow\\ &\qquad\quad \forall c, \mathrm{x} \in \{pend, act, ap\}.\ Q'_c.\mathrm{x} \cap IPIRQ = Q_c.\mathrm{x} \cap IPIRQ\end{aligned}$$

In fact only a few I/O register accesses may influence the status of the SGIs but we omit the details here.

Furthermore we assumed earlier that there exists a set of *Golden GIC distributor* configurations that implement the Golden Mask for each guest independent of the configuration of the CPU and virtual interrupt interfaces. This is formalized in the constraint below.

Assumption 23 (Golden Mask)**.** *There exists a set of configurations $GICD_{GM}$ for the GIC distributor that sets it up in such a way that for each guest at least its Golden Mask is established.*

$$\forall GIC, g \in \mathbb{N}_{ng}.\ GIC.gicd \in GICD_{GM} \;\Rightarrow\; dmsk(GIC) \supseteq GM_g$$

In the ideal model we assume, that the ideal GIC transition function is hard-coded to never enable the distribution of peripheral interrupts not belonging to a given guest.

Assumption 24 (Ideal Distribution)**.** *A step of the ideal GIC of guest g never enables peripheral interrupts that do not belong to g to be distributed to any of the guests cores. Moreover all peripheral interrupts for cores not belonging to the guest are disabled.*

$$\begin{aligned}&\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC', r \in GICD.\\ &\quad \wedge\ \delta^g_Q(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \;\Rightarrow\\ &\qquad dmsk(GIC') \;\supseteq\; \{(q, c) \mid q \in PIRQ \setminus PIRQ_g, c \in \mathbb{N}_{nc_g}\}\\ &\qquad\qquad\quad \cup\; \{(q, c) \mid q \in PIRQ, c \in \mathbb{N}_8 \setminus \mathbb{N}_{nc_g}\}\end{aligned}$$

Now we restrict the GIC write access conversion function in such a way that it always preserves Golden GIC configurations and only makes interrupts pending or active that the requesting core is allowed to access.

Assumption 25 (Golden GIC Updates)**.** *Whenever a GIC distributor with a Golden configuration is updated for a core using the GICD request conversion function ρ_g, the resulting configuration of the distributor is still Golden. Also the update cannot make pending or active SGIs from cores of other guests or peripheral interrupts that do not belong to it.*

$$\begin{array}{l}\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m', c.\\ \quad GIC.gicd \in GICD_{GM} \wedge Q_c \subseteq IRQ_c\\ \wedge\ \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \ \Rightarrow\\ \quad \forall q.\ Wrpl(q, A_{GICD}) \wedge g2m' = (\rho_{\gamma(c)}(q), \mathrm{C}\ c) \ \Rightarrow\\ \quad\quad GIC'.gicd \in GICD_{GM} \wedge Q'_c \setminus Q_c \subseteq IRQ_c^- \wedge Q'_c \subseteq IRQ_c\end{array}$$

Concerning interrupt signalling, all GIC steps must maintain Inv-Pirq, i.e., the interrupt state needs to be updated correctly.

Assumption 26 (Interrupt Correctness)**.** *Each GIC step updates the interrupt state in a way such that the sets of pending, active, and active-and-pending interrupts are disjoint.*

$$\begin{array}{l}\forall GIC, Q, VI, m2g, PI, GIC', Q', VI', g2m'.\\ \quad \delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m') \ \Rightarrow\\ \quad\quad \forall c \in \mathbb{N}_{ng}, \mathrm{x} \neq \mathrm{y} \in \{pend, act, ap\}.\\ \quad\quad\quad Q'(c).\mathrm{x} \cap Q'(c).\mathrm{y} = \emptyset \wedge VI'(c).\mathrm{x} \cap VI'(c).\mathrm{y} = \emptyset\end{array}$$

Note that the property for virtual interrupts is only guaranteed if the hypervisor obeys the software conditions for programming the list registers of the GICH control interface.

The same property holds for the ideal GIC, excluding the virtual interrupts.

Assumption 27 (Ideal Interrupt Correctness)**.** *Each GIC step in the ideal model maintains invariant* Inv-Pirq *and only makes an IGC notification interrupt pending, if the corresponding notification has been received. For all guests $g \in \mathbb{N}_{ng}$:*

$$\begin{array}{l}\forall GIC, Q, m2g, PI, rIGC, GIC', Q', g2m', rIGC'.\\ \quad \delta_Q^g(GIC, Q, m2g, PI, rIGC) \ni (GIC', Q', g2m', rIGC') \ \Rightarrow\\ \quad\quad \forall c \in \mathbb{N}_{ng}, \mathrm{x} \neq \mathrm{y} \in \{pend, act, ap\}.\ Q'(c).\mathrm{x} \cap Q'(c).\mathrm{y} = \emptyset\\ \quad\quad\quad \wedge\ \forall s \in \mathbb{N}_{np_g}.\ \mathrm{igc}_s \in (Q'(c).pend \setminus Q(c).pend) \Rightarrow rIGC(s, c) = 1\end{array}$$

For simplicity, we assume that a non-I/O GIC step handles one interrupt at a time. This behaviour simplifies the bisimulation proof as only one guest's ideal GIC needs to be stepped to represent a step of the refined GIC. Moreover the GIC only adds an interrupt as pending (or active-and-pending in case the interrupt was active before), i.e., the set of active interrupts is not increased by interrupt signalling.

Assumption 28 (Separate Interrupt Signalling)**.** *Outside of handling I/O requests, each GIC step updates the interrupt state in a way such that there at most one interrupt is affected at a time. The virtual interrupt state and the activation of physical interrupts are only affected by I/O requests.*

$$
\begin{array}{l}
\forall GIC, Q, VI, PI, GIC', Q', VI'. \\
\quad \delta_Q(GIC, Q, VI, \text{None}, PI) \ni (GIC', Q', VI', \text{None}) \Rightarrow \\
\qquad \forall c \in \mathbb{N}_{nc} \;.\; Q(c).pend \subseteq Q'(c).pend \wedge Q'(c).act \subseteq Q(c).act \\
\qquad\qquad \wedge \; Q(c).ap \subseteq Q'(c).ap \subseteq Q(c).ap \cup Q(c).act \\
\qquad\qquad \wedge \; \#((Q'(c).pend \cup Q'(c).ap) \setminus (Q(c).pend \cup Q(c).ap)) \in \{0,1\} \\
\qquad\qquad \wedge \; \#(Q(c).act \setminus Q'(c).act) \in \{0,1\} \wedge VI' = VI
\end{array}
$$

Note that this requirement does not restrict GIC implementations to signal several interrupts at the same time, e.g., to different guests. However, the signalling of each interrupt should be independent of other signalled interrupts so that it can be modeled as a sequence of consecutive GIC steps. We require the same behaviour of each ideal GIC. Due to Assumption 14 we need in addition that there are no outstanding IGC notification interrupts raised.

Assumption 29 (Ideal Interrupt Signalling)**.** *Outside of handling I/O requests and IGC notifications, each ideal GIC step updates the interrupt state in a way such that there at most one interrupt is affected at a time. Again, physical interrupts are only activated by I/O requests.*

$$
\begin{array}{l}
\forall g, GIC, Q, PI, rIGC, GIC', Q', rIGC'. \\
\quad (\forall s \in IGCin_g, c \in \mathbb{N}_{nc_g}.\; rIGC(s,c) = 0) \\
\wedge \; \delta^g_Q(GIC, Q, \text{None}, PI, rIGC) \ni (GIC', Q', \text{None}, rIGC') \;\Rightarrow \\
\qquad \forall c \in \mathbb{N}_{nc} \;.\; Q(c).pend \subseteq Q'(c).pend \wedge Q'(c).act \subseteq Q(c).act \\
\qquad\qquad \wedge \; Q(c).ap \subseteq Q'(c).ap \subseteq Q(c).ap \cup Q(c).act \\
\qquad\qquad \wedge \; \#((Q'(c).pend \cup Q'(c).ap) \setminus (Q(c).pend \cup Q(c).ap)) \in \{0,1\} \\
\qquad\qquad \wedge \; \#(Q(c).act \setminus Q'(c).act) \in \{0,1\} \wedge rIGC' = rIGC
\end{array}
$$

We restrict the interrupt signalling in the refined model further by requiring that a GIC distributor never forwards peripheral interrupts to the core interfaces if they are masked. Moreover it always forwards the peripheral interrupt with the highest priority and never activates SGIs by itself.

Assumption 30 (Masked Interrupt Signalling)**.** *The GIC distributor never forwards peripheral interrupts to a core interface when their forwarding is disabled or there is another unmasked interrupt with higher priority. Inter-processor interrupts are never made pending spontaneously. Let*

$$q_c^{max} = \max\{q' \mid \exists p.\; q' \in PI(p) \wedge (q', c) \notin dmsk(GIC)\}$$

then we demand:

$$
\begin{array}{l}
\forall GIC, Q, VI, PI, GIC', Q'. \\
\quad \wedge \; \delta_Q(GIC, Q, VI, \text{None}, PI) \ni (GIC', Q', VI, \text{None}) \Rightarrow \\
\qquad \forall c \in \mathbb{N}_{nc}, q \in Q'(c).pend \setminus Q(c).pend \cup Q'(c).ap \setminus Q(c).ap.\; q = q_c^{max}
\end{array}
$$

The same property is needed for the ideal GIC.

Assumption 31 (Ideal Masked Interrupt Signalling)**.** *An ideal model GIC distributor never forwards peripheral interrupts to a core interface when their forwarding is disabled or there is another unmasked interrupt with higher priority. Inter-processor interrupts are never made pending spontaneously.*

$$\begin{array}{l}\forall GIC, Q, PI, rIGC, GIC', Q', rIGC'.\\ \quad \delta_Q(GIC, Q, VI, PI, \mathrm{None}, rIGC) \ni (GIC', Q', \mathrm{None}, rIGC') \Rightarrow\\ \quad\quad \forall c \in \mathbb{N}_{nc}, q \in Q'(c).pend \setminus Q(c).pend \cup Q'(c).ap \setminus Q(c).ap.\ q = q_c^{max}\end{array}$$

Note that this assumption implies that the IGC interrupt has a higher priority than any other peripheral interrupt, since it is always delivered first.

For the bisimulation we need that any raised peripheral interrupt can be made pending (or active-and-pending) by both the ideal and the refined GIC at any time, as long they are not disabled in the distributor and they have the highest priority among all non-pending but raised peripheral interrupts.

Assumption 32 (Physical Interrupt Recognition)**.** *Whenever a peripheral interrupt is raised with the highest current priority, if it is not already pending or active-and-pending, and not masked in the distributor, it may be delivered to the interface of a core.*

$$\begin{array}{l}\forall GIC, Q, VI, PI, c \in \mathbb{N}_{nc}, q \notin Q(c).pend \cup Q(c).ap.\ q = q_c^{max} \ \Rightarrow\\ \quad \exists GIC', Q'.\ \delta_Q(GIC, Q, VI, \mathrm{None}, PI) \ni (GIC', Q', VI, \mathrm{None})\\ \quad\quad \wedge\ q \notin Q(c).act \Rightarrow Q'(c).pend = Q(c).pend \cup \{q\}\\ \quad\quad \wedge\ q \in Q(c).act \Rightarrow Q'(c).act = Q(c).act \setminus \{q\} \wedge Q'(c).ap = Q(c).ap \cup \{q\}\end{array}$$

We require the same behaviour of the ideal GIC. Due to Assumption 14 we need in addition that there are no outstanding IGC notifications.

Assumption 33 (Ideal Physical Interrupt Recognition)**.** *Whenever a peripheral interrupt is raised, the GIC may deliver it to the interface of a core as long as it is not masked, not already pending or active-pending, and it has the highest current priority. For all $g \in \mathbb{N}_{ng}$ we have:*

$$\begin{array}{l}\forall GIC, Q, PI, rIGC, GIC', Q', c \in \mathbb{N}_{nc_g}, q \notin Q(c).pend \cup Q(c).ap.\\ \quad (\forall s \in \mathbb{N}_{nc}, c \in \mathbb{N}_{nc_g}.\ rIGC(s, c) = 0) \wedge q = q_c^{max} \ \Rightarrow\\ \quad\quad \exists GIC', Q'.\ \delta_Q^g(GIC, Q, \mathrm{None}, PI, rIGC) \ni (GIC', Q', \mathrm{None}, rIGC)\\ \quad\quad\quad \wedge\ q \notin Q(c).act \Rightarrow Q'(c).pend = Q(c).pend \cup \{q\}\\ \quad\quad\quad \wedge\ q \in Q(c).act \Rightarrow Q'(c).act = Q(c).act \setminus \{q\} \wedge Q'(c).ap = Q(c).ap \cup \{q\}\end{array}$$

We have two more assumptions on the handling of SGIs. Firstly, SGIs are always enabled in the refined and ideal model GIC distributor.

Assumption 34 (SGIs Enabled)**.** *Interprocessor interrupts are always enabled in the GIC distributor. We have the following invariant on GIC configurations $GIC \in \mathcal{G}$ of refined model.*

$$dmsk(GIC) \cap IPIRQ = \emptyset$$

Similarly, for GIC configurations GIC of guest g in the ideal model we include the IGC interrupts which cannot be masked in the distributor as well:

$$\forall g.\ dmsk(GIC) \cap (IPIRQ_g \cup (IGCQ_g \times \mathbb{N}_{nc_g})) = \emptyset$$

Secondly, when an SGI is acknowledged, it becomes active, i.e., it can only become active-and-pending in case an identical SGI is sent for a second time.

Assumption 35 (SGI Acknowledgement)**.** *When a an inter-processor interrupt is acknowledged by a read of the acknowledgement register, the corresponding interrupt was pending before and is active afterwards.*

$$\begin{array}{l}\forall GIC, Q, VI, PI, GIC', Q', g2m', q, c, c'.\\ \quad \wedge\ \delta_Q(GIC, Q, VI, \mathrm{None}, PI) \ni (GIC', Q', VI, g2m') \wedge\\ \qquad \exists v.\ g2m' = (rplR\ a_{IAR}\ v, \mathrm{C}\ c) \wedge q = \langle v[9:0]\rangle \leq 15 \wedge c' = \langle v[12:10]\rangle \Rightarrow\\ \qquad\qquad \mathrm{sgi}^q_{c',c} \in Q(c).pend \wedge \mathrm{sgi}^q_{c',c} \in Q'(c).act\end{array}$$

Finally, we require that the CPU interface and the virtualized interface behave in the same way.

Assumption 36 (Interrupt Virtualization)**.** *Given an ideal and a refined GIC state, if the ideal CPU interface is identical to the virtual interface, and the same I/O requests are pending for a given core, then the corresponding I/O replies have similar effects on both models. An ideal GIC configuration and interrupt state GIC_I, Q_I are considered similar with a refined GIC configuration and virtual interrupt configuration GIC_R, VI_R for core c if the relation $GIC_I, Q_I \sim^c_Q GIC_R, VI_R$ holds that has the following definition.*

$$\begin{array}{c}\forall r \in GICC.\ GIC_I.gicc(r,c) = GIC_R.gicv(r,c) \wedge\\ \forall r \in GICD.\ reqgicd(r,0)\ \Rightarrow\ GIC_I.gicd(r) = \eta^r_{\gamma(c)}(GIC_R.gicd(r)) \wedge\\ Q_I.pend = xtvi_{\gamma(c)}(VI_R.pend) \wedge Q_I.act = xtvi_{\gamma(c)}(VI_R.act) \wedge\\ Q_I.ap = xtvi_{\gamma(c)}(VI_R.ap)\end{array}$$

Then we require the following behaviour of the refined and ideal model GICs for processing requests to the virtual interrupt or CPU interfaces.

$$\begin{array}{l}\forall c, GIC_R, Q_R, VI_R, PI_R, GIC_I, Q_I, PI_I, rIGC_I, g2m'.\\ \quad GIC_I, Q_I \sim^c_Q GIC_R, VI_R \wedge (\forall p.\ \gamma(c) = \gamma(p) \Rightarrow PI_I(\varphi(p)) = PI_R(p)) \wedge\\ \quad \exists q.\ g2m' = (q, \mathrm{C}\ c) \wedge rpl(q, A_{GICV})\ \Rightarrow\\ \qquad \forall GIC'_R, Q'_R, VI'_R.\\ \qquad \delta_Q(GIC_R, Q_R, VI_R, \mathrm{None}, PI_R) \ni (GIC'_R, Q'_R, VI_R, g2m')\ \Rightarrow\\ \qquad \exists GIC'_I, Q'_I, rIGC'_I.\\ \quad \delta^{\gamma(c)}_Q(GIC_I, Q_I, \mathrm{None}, PI_I, rIGC_I) \ni (GIC'_I, Q'_I, conv_{IO}(g2m'), rIGC'_I) \wedge\\ \qquad GIC'_I, Q'_I \sim^c_Q GIC'_R, VI'_R\\ \wedge\quad \forall GIC'_I, Q'_I, rIGC'_I.\\ \quad \delta^{\gamma(c)}_Q(GIC_I, Q_I, \mathrm{None}, PI_I, rIGC_I) \ni (GIC'_I, Q'_I, conv_{IO}(g2m'), rIGC'_I)\ \Rightarrow\\ \qquad \exists GIC'_R, Q'_R, VI'_R.\\ \qquad \delta_Q(GIC_R, Q_R, VI_R, \mathrm{None}, PI_R) \ni (GIC'_R, Q'_R, VI_R, g2m') \wedge\\ \qquad GIC'_I, Q'_I \sim^c_Q GIC'_R, VI'_R\end{array}$$

Note that we assume here that replies from the GIC virtual and CPU interfaces only depend on parts of the distributor registers that reflect the current interrupt state, e.g., the registers listing currently pending or active interrupts. In deed this is a natural assumption, due to the partitioning of the GIC into the different interfaces. The configuration of the distributor only affects the CPU interfaces indirectly by determining which interrupts are forwarded to a given CPU or virtual interface.

Finally we also add an assumption on the GICD request and reply conversion function here, which allows us to maintain the coupling relation for the distributor registers.

Assumption 37 (GICD I/O Message Conversion)**.** *When a write request to the GIC distributor is filtered for a guest g, the resulting values that will be stored in the destination register of the refined model will be equal to that stored in the ideal model wrt. the register filter function for guest g, given that the initial values of the register are equal in the same way.*

$$\begin{gathered}\forall g \in \mathbb{N}_{ng}, a \in \mathbb{B}^{48}, r \in \mathit{GICD}, d, v, w, u_I, u_R \in \mathbb{B}^{8d}.\\ r_{GIC}(a,d) = r \wedge \mathit{reqgicd}(r,1) \wedge u_I = \eta_g^r(u_R) \;\Rightarrow\\ \rho_g(\mathrm{W}\ a\ d\ v) = \mathrm{W}\ a\ d\ w \;\Rightarrow\; \eta_g^r(\nu(r,u_I,v)) = \eta_g^r(\nu(r,u_R,w))\end{gathered}$$

Similarly for read replies we have $\rho_g(\mathit{rplR}\ a\ v) = \mathit{rplR}\ a\ \eta_g^r(v)$ for all r such that $\mathit{rgic}(a,d) = r$ and $\mathit{reqgicd}(r,0)$ holds.

Note that values stored in the GIC distributor of the ideal model are always subject to filtering, as demanded in Assumption 16.

4.9.4 MMU constraints

For the second-stage MMUs and SMMUs we require that the MMUs never turn themselves off or change their configuration.

Assumption 38 (MMU Persistence)**.** *For any MMU with state $mmu \in MMU$ and corresponding inputs and outputs, if the MMU performs a step its configuration is unchanged. Moreover while busy, it never changes its current request.*

$$\begin{gathered}\forall \mathit{mmu}, \mathit{x2w}, \mathit{m2w}, \mathit{mmu}', \mathit{w2x}', \mathit{w2m}'.\\ \delta_{\mathit{mmu}}(\mathit{mmu}, \mathit{x2w}, \mathit{m2w}) \ni (\mathit{mmu}', \mathit{w2x}', \mathit{w2m}') \;\Rightarrow\\ \quad \mathit{mmu}'.\mathit{active} = \mathit{mmu}.\mathit{active} \wedge \mathit{mmu}'.\mathit{pto} = \mathit{mmu}.\mathit{pto} \wedge \mathit{mmu}'.\mathit{cfg} = \mathit{mmu}.\mathit{cfg}\\ \wedge\ \mathit{mmu}.\mathit{walk} \neq \bot \wedge \mathit{w2x}' = \mathrm{None} \;\Rightarrow\; \mathit{mmu}'.\mathit{walk}.\mathit{curr} = \mathit{mmu}.\mathit{walk}.\mathit{curr}\end{gathered}$$

We fix how the MMU cycles through its phases as follows

For the second-stage MMUs and SMMUs we require that the MMUs never turn themselves off or change their configuration.

Assumption 39 (MMU Phase Shift)**.** *For any MMU with state $mmu \in MMU$ and corresponding inputs and outputs, if the MMU is idle and receives a request it enters its translation phase. If it is already translating it completes the walk*

upon receiving a matching memory reply for its last lookup. In both transitions no outputs are generated.

$$
\begin{array}{l}
\forall mmu, x2w, m2w, mmu', w2x', w2m'. \\
\quad \delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m') \;\Rightarrow \\
\qquad mmu.active \wedge mmu.walk = \bot \wedge x2w \neq \text{None} \;\Rightarrow \\
\qquad\quad trans(mmu') \wedge w2x' = \text{None} \wedge w2m' = \text{None} \\
\quad \wedge \quad trans(mmu) \wedge mmu'.walk.complete \;\Rightarrow \\
\qquad\quad match(mmu.walk.lookup, m2w) \wedge w2x' = \text{None} \wedge w2m' = \text{None} \\
\qquad \wedge \quad mmu'.walk.lookup = \text{None} \wedge mmu'.walk.final = \text{None}
\end{array}
$$

Moreover, they only send page table look-ups during there translation phase.

Assumption 40 (MMU Translation Phase)**.** *For any MMU with state $mmu \in MMU$ and corresponding inputs and outputs, if the MMU is in its translation phase, it only produces memory requests that are page table walks which it records in the lookup field of its walk component. Moreover it only produces new walks if it has no outstanding ones, or it consumed a matching memory lookup reply in the same step.*

$$
\begin{array}{l}
\forall mmu, x2w, m2w, mmu', w2x', w2m'. \\
\quad \delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m') \wedge trans(mmu) \;\Rightarrow \\
\qquad w2x' = \text{None} \wedge mmu.walk.final = \text{None} \wedge \neg mmu.walk.complete \\
\qquad \wedge \quad w2m' \neq \text{None} \Rightarrow mmu'.walk.lookup = w2m' \wedge PTreq(w2m', \mathbb{B}^{48})) \\
\qquad \wedge \quad w2m' = \text{None} \;\Leftrightarrow\; mmu'.walk.lookup = \text{None} \vee \\
\qquad\quad \exists r.\; mmu'.walk.lookup = mmu.walk.lookup = r \wedge \neg match(r, m2w)
\end{array}
$$

Observe that this constraint implies, that the MMUs never modify the memory on their own behalf. Furthermore we require that MMUs forward replies for their final memory requests immediately and without modifying them except for the address, translating physical addresses back to intermediate-physical ones.

Assumption 41 (MMU Replies)**.** *For any MMU with state $mmu \in MMU$ and corresponding inputs and outputs, if the MMU receives a reply from memory in its final phase, it forwards it back to the sender of the initial request, changing only the address. If there is no reply from memory it only sends a reply to the core in case of faults. Otherwise it waits for the final reply from memory.*

$$
\begin{array}{l}
\forall mmu, x2w, m2w, mmu', w2x', w2m'. \\
\quad \delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m') \wedge mmu.walk.complete \;\Rightarrow \\
\qquad \neg trans(mmu) \wedge mmu.walk.lookup = \text{None} \\
\quad \wedge\; m2w \neq \text{None} \wedge match(mmu.walk.final, m2w) \;\Rightarrow \\
\qquad\quad \exists a, d, v.\; m2w = rplR\ a\ v \;\Rightarrow\; \exists a'.\; w2x' = rplR\ a'\ v \\
\qquad \wedge \quad \exists a, d, v.\; m2w = rplW\ a \;\Rightarrow\; \exists a'.\; w2x' = rplW\ a' \\
\qquad \wedge \quad \exists a, v.\; m2w = rplPTW\ a\ v \;\Rightarrow\; \exists a'.\; w2x' = rplPTW\ a'\ v \\
\qquad \wedge \quad \exists a.\; m2w = \text{F}\ a \;\Rightarrow\; \exists a'.\; w2x' = \text{F}\ a' \\
\qquad \wedge \quad match(mmu.walk.curr, w2x') \wedge mmu'.walk = \bot \wedge w2m' = \text{None}
\end{array}
$$

$$
\begin{aligned}
&\wedge\ m2w = \text{None} \wedge mmu'.walk.final \neq \text{None}\ \Rightarrow \\
&\qquad w2x' = \text{None} \wedge mmu.walk.final \in \{\text{None}, mmu'.walk.final\} \\
&\quad \wedge\quad w2m' = \begin{cases} mmu'.walk.final & : \quad mmu.walk.final = \text{None} \\ \text{None} & : \quad \text{otherwise} \end{cases} \\
&\wedge\ m2w = \text{None} \wedge mmu'.walk.final = \text{None}\ \Rightarrow \\
&\qquad w2m' = \text{None} \wedge w2x' = \text{F}\ a' \wedge mmu.walk.final = \text{None} \\
&\quad \wedge\quad match(mmu.walk.curr, w2x') \wedge mmu'.walk = \bot
\end{aligned}
$$

If the MMU is stepped while it is idle and there are no requests from the core, it stutters.

Assumption 42 (MMU Idle)**.** *For any MMU with state $mmu \in MMU$ and empty core input, if the MMU is not busy, it does not change its state or produce outputs*

$$
\begin{aligned}
&\forall mmu, x2w, m2w, mmu', w2x', w2m'. \\
&\quad \delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m')\ \Rightarrow \\
&\qquad mmu.walk = \bot \wedge x2w = \text{None}\ \Rightarrow\ mmu' = mmu \wedge w2x' = w2m = \text{None}
\end{aligned}
$$

We also need a constraint for an MMU's behaviour when it is turned off.

Assumption 43 (MMU Forwarding)**.** *For any MMU with state $mmu \in MMU$ and corresponding inputs and outputs, if the MMU is inactive, it only forwards requests from cores and replies from memory, but never generates requests or replies on its behalf.*

$$
\begin{aligned}
&\forall mmu, x2w, m2w, mmu', w2x', w2m'. \\
&\quad \delta_{mmu}(mmu, x2w, m2w) \ni (mmu', w2x', w2m')\ \Rightarrow \\
&\qquad mmu.active = 0\ \Rightarrow\ w2x' = m2w \wedge w2m' = x2w
\end{aligned}
$$

Note that in the real ARMv8 hardware the MMU actually may produce faults while it is inactive. However we moit this detail here as memory requests by the hypervisor implementation are only targeting the GIC and are translated by the second-stage MMU with an identity mapping without giving any faults.

4.9.5 Peripheral Constraints

For peripherals we simplified the model by assuming they never perform page table walks or I/O accesses to other peripherals. This is captured by the following constraint.

Assumption 44 (No Peripheral Walks or I/O)**.** *For any peripheral p with state $P.st \in \mathcal{P}$, whenever the peripheral makes a step for given inputs, it never generates a DMA request that is a page table walk or a memory-mapped I/O request to a different peripheral.*

$$
\begin{aligned}
&\forall P.st, v2p, p2v, E, P.st', v2p', p2v', e, Q_{out}. \\
&\quad \delta^p_P(P.st, v2p, p2v, E) \ni (P.st', v2p', p2v', e, Q_{out})\ \Rightarrow \\
&\qquad p2v' \in \mathbb{R}\ \Rightarrow\ \neg PTreq(p2v', \mathbb{B}^{48}) \\
&\quad \wedge\quad p2v' = (r, s) \in \mathbb{R} \times \mathbb{S}\ \Rightarrow\ \forall p' \in \mathbb{N}_{np+1}.\ s \neq \text{P}\ p'
\end{aligned}
$$

Moreover, we require that they do not send new DMA requests while there is already an outstanding one for them that they have not received a reply for.

Assumption 45 (Sequential DMA)**.** *For any peripheral p with state $P.st \in \mathcal{P}$, whenever the peripheral makes a step for given inputs, it never generates a new DMA request if there is still an outstanding request.*

$$\begin{array}{l}\forall P.st, v2p, p2v, E, P.st', v2p', p2v', e, Q_{out}. \\ \quad \delta_P^p(P.st, v2p, p2v, E) \ni (P.st', v2p', p2v', e, Q_{out}) \;\Rightarrow \\ \qquad v2p' = \text{None} \\ \quad \wedge \quad \exists r \in P.st.curr \cap \mathbb{R} \wedge \neg match(r, v2p) \;\Rightarrow \\ \qquad\qquad p2v' \in \{p2v\} \cup \mathbb{Q} \times \mathbb{S} \wedge r \in P.st'.curr\end{array}$$

Another constraint concerns the replies for memory-mapped I/O requests. All such replies must be sent in response received earlier, moreover the sender ID needs to be preserved.

Assumption 46 (Peripheral I/O Replies)**.** *For any peripheral p with state $P.st \in \mathcal{P}$, whenever the peripheral generates an I/O reply q, it is in reponse to a currently pending I/O request from sender s.*

$$\begin{array}{l}\forall P.st, v2p, p2v, E, P.st', v2p', p2v', e, Q_{out}, q, s. \\ \quad \delta_P^p(P.st, v2p, p2v, E) \ni (P.st', v2p', p2v', e, Q_{out}) \;\Rightarrow \\ \qquad p2v' = (q, s) \;\Rightarrow\; \exists r \;.\quad (r, s) \in P.st.curr \\ \qquad\qquad\qquad\qquad\quad \wedge \quad match(r, q) \\ \qquad\qquad\qquad\qquad\quad \wedge \quad P.st'.curr = P.st.curr \setminus \{(r, s)\}\end{array}$$

As a last constraint, we formalize the requirement that peripheral behaviour should not depend on the base address assigned to them, mentioned in Sect. 4.1. To this end, we introduce an address isomorphism $\approx_p$ on requests and replies sent to and from peripheral p as well as the peripheral state. We omit the formal details here, intuitively two requests r and $\hat{r}$ are isomorphic for peripheral p, i.e., $r \approx_p \hat{r}$, if they agree on request type, size, and written value, and their addresses agree for all bits that are offsets within $A_P(p)$, i.e., for addresses a and a' we have

$$a[11 + sz(p) : 0] = a'[11 + sz(p) : 0]\,.$$

For replies the isomorphism is defined similarly. For two states $P.st$ and $\hat{P.st}$ of peripheral p we require that $P.st \approx_p \hat{P.st}$ implies that for each request r that is in $P.st.curr$ there exists an isomorphic request $\hat{r}$ in $\hat{P.st}.curr$, and vice versa. Then for any step performed in $P.st$ a similar step can be performed in $\hat{P.st}$ that maintains the isomorphism and produces the same outputs wrt. the address isomorphism.

Assumption 47 (Peripheral Base Address Isomorphism)**.** *For each peripheral p, there exists a base address isomorphism that is maintained by any step of the*

peripheral, yielding the same or isomorphic outputs.

$$
\begin{aligned}
&\forall P.st, v2p, p2v, E, \hat{P.st}, \hat{v2p}, \hat{p2v}, P.st', v2p', p2v', e, Q_{out}.\\
&\quad \delta^p_P(P.st, v2p, p2v, E) \ni (P.st', v2p', p2v', e, Q_{out}) \wedge\\
&\quad P.st \approx_p \hat{P.st} \wedge v2p \approx_p \hat{v2p} \wedge p2v \approx_p \hat{p2v} \;\Rightarrow\\
&\qquad \exists \hat{P.st}', \hat{v2p}', \hat{p2v}'.\\
&\qquad\quad \delta^p_P(\hat{P.st}, \hat{v2p}, \hat{p2v}, E) \ni (\hat{P.st}', \hat{v2p}', \hat{p2v}', e, Q_{out}) \wedge\\
&\qquad\quad P.st' \approx_p \hat{P.st}' \wedge v2p' \approx_p \hat{v2p}' \wedge p2v' \approx_p \hat{p2v}'
\end{aligned}
$$

Note that the second-stage address translation is constructed in a way that all offsets within the memory region of a peripheral are subject to an identity mapping, therefore requests using intermediate-physical addresses and their physical counterparts are isomorphic for the targeted peripheral. Using the assumption on the peripheral transition relation from above we can then show that in the ideal model the same peripheral steps are possible as in the refined model, and vice versa.

4.10 Proof of Induction Start

First we prove Lemma 1. It's claim is given as follows.

$$
\begin{aligned}
&\forall M^0_I \in \mathcal{M}_I.\ start(M^0_I) \;\Rightarrow\; \exists M^0_R \in \mathcal{M}_R.\ start(M^0_R) \wedge M^0_I \sim M^0_R\\
&\forall M^0_R \in \mathcal{M}_R.\ start(M^0_R) \;\Rightarrow\; \exists M^0_I \in \mathcal{M}_I.\ start(M^0_I) \wedge M^0_I \sim M^0_R
\end{aligned}
$$

Proof of Lemma 1. We pick two states M^0_R and M^0_I such that $start(M^0_R)$ and $start(M^0_I)$ hold. In what follows we will restrict both states further for each clause of the bisimulation relation, such that finally $M^0_I \sim M^0_R$ holds. Note that all of the following additional constraints are meant to restrict either the ideal or the refined configuration, depending on which direction we are proving. Moreover, these constraints do not contradict the assumptions on the initial states, i.e., they do not invalidate $start(M^0_R)$ and $start(M^0_I)$.

Context In the ideal model we have $\neg C^c_I.active$ and $\neg C^c_R.init$ for all refined cores c, therefore the bisimulation relation holds trivially for all core contexts initially.

Memory The guest memory is only restricted for IGC channels. We require for all guests g

$$\forall a \in \mathbb{A} \setminus A^g_P.\ M^0_I.G(g).m(a) = M^0_R.m(\theta_g(a))\,,$$

which establishes the bisimulation relation for memory. Note that, for all IGC channels s from g to g', $\theta_g(IGCa(s)_1) = \theta_{g'}(IGCa(s)_2)$, thus the restriction above does not contradict the requirement of $start(M^0_I)$ for the IGC memory channels.

Periph In both the refined and the ideal model peripherals are initialized in their reset states. Thus for all $p \in \mathbb{N}_{np}$ we have:

$$M_I^0.G(\gamma(p)).P.st(\varphi(p)) = P_{reset}^{\gamma(p),\varphi(p)} = P_{reset}^{p} = M_R^0.P.st(p)$$

Core2Mem In the ideal model all message channels are empty, therefore the implication in this clause holds vacuously. The second condition on the outgoing core channels holds as well due to the same fact.

Core2MMU2Mem Initially, in both states the message channels are empty, and in the refined model cores are running in EL3. Moreover the MMU is inactive for all cores. Hence this clause's sub-clauses are simplified as follows:

1. No core runs in guest mode, so the implication holds vacuously.
2. All MMUs are inactive, so the implication holds vacuously.
3. All message channels in the ideal model are empty, so the implication holds for all cores.
4. All refined cores are initially running in EL3, so the implications hold vacuously.
5. All message channels in the refined model are empty, so the implication holds vacuously.

Mem2MMU2Core The first part of this clause holds trivially, as only the sixth case is possible initially and all ideal channels are empty.

Similarly the second condition holds vacuously as initially the mode is EL3 on all cores

Mem2Core This clause holds analoguously to the previous one.

Mem2GIC In both models the channels to the GIC are empty, thus the first part of the clause holds trivially. The second part holds as all cores in the refined model are initially running in EL3.

GIC2Mem This clause holds for the same arguments as the previous one.

PerChan The channels in and out of peripherals are empty as well. Moreover the SMMUs are inactive in the initial state. Then the last case of the equivalences hold since function $conv_{IO}$ does not modify empty message channels.

$$pif_I^p = \text{None} = conv_{IO}(\text{None}) = conv_{IO}(p2v_R^p)$$
$$m2p_I^p = \text{None} = conv_{IO}(\text{None}) = conv_{IO}(v2p_R^p)$$

PerIRQ In both models there are no raised peripheral interrupts initially.

$$M_I^0.G(\gamma(p)).PI(\varphi(p)) = \emptyset = M_R^0.PI(p)$$

VirtIRQ Similarly, there are no active virtual interrupts for any core, nor are there any soon-to-be pending interrupts or outstanding IGC interrupts, i.e., $nuvi^c_{\gamma(c)}(M^0_R, p) = \emptyset$ and $IGCQ^c_I = \emptyset$ for all c. For all x $\in \{pend, act, ap\}$ we have:

$$\begin{aligned} & M^0_I.G(\gamma(c)).Q(\kappa(c)).\text{x} \\ &= \emptyset \\ &= xtvi_{\gamma(c)}(\emptyset) \\ &= xtvi_{\gamma(c)}(M^0_R.VI(c).\text{x}) \\ &= (xtvi_{\gamma(c)}(M^0_R.VI(c).pend) \cup nuvi^c_{\gamma(c)}(M^0_R, 1)) \setminus IGCQ^c_I \\ &= (xtvi_{\gamma(c)}(M^0_R.VI(c).pend) \setminus nuvi^c_{\gamma(c)}(M^0_R, 0)) \\ &= (xtvi_{\gamma(c)}(M^0_R.VI(c).pend) \cup nuvi^c_{\gamma(c)}(M^0_R, 0)) \end{aligned}$$

SndIGC Initially no guest has been initialized yet, i.e., for all guests g we have $M^0_I.G(g).init = 1$. Then the implication in this clause holds vacuously.

RcvIGC Initially no virtual interrupts are raised in the refined model. Additionally, the receive-IGC flags are zero initially and there are no IGC interrupts pending in the GIC. All refined cores are running in EL3. Therefore the statement of this clause holds trivially.

Ext In the initial states all external event sequences are empty, thus this clause holds trivially for all guests g as follows:

$$M^0_I.G(g).E = \varepsilon = \varepsilon|_g = M^0_R.E|_g$$

GICcpu Initially, in both system the GIC is in its reset state. We require here that the CPU and virtual interface registers are set up in the same way by the following constraint for all cores c.

$$M^0_I.G(\gamma(c)).GIC.gicc(\kappa(c)) = M^0_R.GIC.gicv(c)$$

As in both interfaces registers have the same format, this can be achieved easily, setting all registers to their reset value.

GICdist Initially all guests are still in their initial state, thus the implication in the first part of this clause holds vacuously. The second part holds trivially, as in M^0_R all peripheral interrupts are masked by $start(M^0_R)$.

IGCchan For each IGC channel s from g to g' this clause is guaranteed by the initialization of the ideal state and our additional requirement for the coupling of the guest memory between refined and ideal model.

$$\begin{aligned} M^0_R.m(\epsilon A_{IGC}(g, g')) &= M^0_R.m(\epsilon\{\theta_g(IGCa(s)_1)\}) \\ &= M^0_R.m(\theta_g(IGCa(s)_1)) \\ &= M^0_I.G(g).m(IGCa(s)_1) \\ &= M_I.S(s) \end{aligned}$$

CURR Initially no requests have been issued yet in both models, the message channels are empty, and all refined cores are running in EL3 ouside the hypervisor code, therefore all statements of this clause hold trivially.

Thus we conclude $M_I^0 \sim M_R^0$ for both directions of the claim. □

Observe from the proof that most clauses of the bisimulation relation directly hold if $start(M_R^0)$ and $start(M_I^0)$. The only further requirements we needed were considering the initial content of guest memory, as well as the contents of the ideal GIC CPU interrupt interface and the refined GIC virtual interrupt interface. However, most clauses held trivially due to the fact that the guests are not active yet and no steps have been performed yet in the refined model.

4.11 Proof of Implementation Invariant

The proof of the implementation invariant is quite complex as there is a large number of invariants and steps of the model to consider. Here, we perform a case distinction over the possible steps of the refined model and only discuss invariants that are affected by a given step. Below we list the claim of Lemma 2 again.

$$\forall M_R, M_R'.\ Inv(M_R) \wedge M_R \xrightarrow{R} M_R' \ \Rightarrow\ Inv(M_R')$$

Then we have the following cases.

4.11.1 Case: Guest

The refined model performs transition GUEST, i.e., an active core c running guest mode is stepped according to transition relation δ_R:

$$\delta_R(M.C(c), M.u2c(c), M.c2u(c), q_{in}) \ni (C', u2c', c2u')$$

The interrupts forwarded to the core by q_{in} is a pending physical or virtual interrupt that is not masked by the GIC and has the highest highest priority among other such interrupts. Note that the only components updated are $M_R'.C(c)$, $M_R'.c2u(c)$, and $M_R'.u2c(c)$, in particular the memory is unchanged. Observe however that this step might lead execution into EL2 by taking synchronous or asynchronous exceptions in the core. For all possible guest steps only the following invariants are affected.

INV-POW The *active* flag of a core may only change by power control commands sent by the hypervisor. We have $M_R'.C(c).active = M_R.C(c).active = 1$. As the memory is not affected by the guest step, the invariant on the power state data structure is preserved.

INV-REGEL2 Registers of the hypervisor cannot be accessed in EL1 or EL0. Therefore the invariant still holds.

INV-REGEL3 Registers of EL3 cannot be accessed in EL1 or EL0. Therefore the invariant still holds.

Inv-MMU The first part of this invariant is maintained simply because guest steps cannot modify the MMU configuration by construction. The only way to break this invariant is to enter into EL2 without jumping into an entry point of the hypervisor. Then the MMU would still be active in our model and thus invalidate the second part of this invariant. However all exceptions to EL2 are rooted in one of the hypervisor entry points, due to the setup of the exception vectors and invariant Inv-RegEl2.

Note that we have not formalized this notion explicitly. Intuitively, the entry points to synchronous exceptions must be allocated at $vbar_H$ plus offset 0x400, while asynchronous interrupts have to be allocated at $vbar_H$ plus offset 0x480, with all of the code residing in A_{HV}.

Inv-HVEntry If the guest step caused an exception then $M'_R.C(c).pc \in A_{hve}$ as argued above. In general there are only cases to consider when a guest step is trapped into EL2. They are identified by the value stored in the Exception Status Register (of EL2) and the Interrupt State Register.

- The guest executed a hypercall instruction (HVC) — then this exception followed an instruction fetch that was consumed from memory, so $M_R.u2c(c) \neq$ None and $M'_R.u2c(c) = M'_R.C(c).curr =$ None. There are no other outstanding requests for core c due to Inv-Msg as well as Inv-Hyp-Pmsg. By the third clause of Inv-Walk we also know that $M_R.mmu(c).curr = \perp$ which is preserved by the guest step not affecting the MMU.
- The guest executed a secure monitor call instruction (SMC) which was trapped to EL2 — then this exception followed an instruction fetch that was consumed from memory, so $M_R.u2c(c) \neq$ None and $M'_R.u2c(c) =$ None. The remainder of the proof is similar to that of the proof is similar to the case above
- the guest caused an Instruction Abort — then this exception was discovered after the core consumed a fault on a fetch request from the MMU channel, thus $M_R.u2c(c) \neq$ None and $M'_R.u2c(c) =$ None. Again we conclude as above.
- The guest caused a data abort — then this exception was discovered after the core consumed a fault on a memory data request from the MMU channel, thus $M_R.u2c(c) \neq$ None and $M'_R.u2c(c) =$ None and the remainder of the proof goes as above.
- An asynchronous exception was received — in this case there should be no outstanding memory requests as the core is stalled otherwise according to Assumption 1. Consequently, either an outstanding reply was received and we conclude as above, or we had $M_R.C(c).curr =$ None and $M_R.u2c(c) =$ None. In this case invariants Inv-Walk, Inv-MemReq, Inv-Hyp-Greq, and Inv-Hyp-Pmsg give us that there are no outstanding requests from core c in MMU, GIC, memory, or the peripherals. Then Inv-Request, Inv-Reply, and Inv-Busy

yield that all message channels for core c are empty, using proof by contradiction.

INV-MSG A core step may change the state of its input and output channels, however by Assumption 3 the updates to the channels are restricted. We have the following cases

1. $M'_R.u2c(c) = \text{None} \wedge M'_R.c2u(c) = \text{None}$ – then the invariant is maintained trivially as it already held before and no new messages where generated.
2. $M'_R.u2c(c) = \text{None} \wedge M'_R.c2u(c) \neq \text{None}$ – then either a new request to the MMU was generated or the previous one is still outstanding and in the output channel and we are done. In the first case, i.e., $M_R.c2u(c) = \text{None}$, we distinguish whether a message was consumed from the input channel or not.
   (a) $M_R.u2c(c) \neq \text{None}$ – the core consumed a reply for an outstanding request and produced a new request for the MMU. This case is not possible due to Assumption 3. Anyway, it maintains the number of messages in the system.
   (b) $M_R.u2c(c) = \text{None}$ – the core produced a new current request and by Assumption 1 we have $M_R.C(c).curr = \text{None}$. Then by the third clause of INV-WALK and INV-MMU, the second-stage MMU of core c is active and not busy. From the contrapositive of INV-BUSY we get that there are no outstanding requests for core c in channels or components beyond the MMU.
3. $M'_R.u2c(c) \neq \text{None} \wedge M'_R.c2u(c) = \text{None}$ – then by Assumption 3 the same request must have been in the channel before the core step but it was not consumed for some reason, i.e., $M_R.u2c(c) \neq \text{None}$. By invariant INV-REPLY however, the memory reply must be matching an outstanding request and is thus consumed by the core step according to Assumption 3. Therefore this case is not possible after a core step and we conclude INV-MSG for M'_R.

INV-WALK Only the second and the third line of this invariant may be affected by the core, in case it changes its currently outstanding request. However by Assumption 1 we know that this can only happen in case $M_R.C(c).curr = \text{None}$, otherwise any core step is stuttering or a previous request is consumed while a new one is produced. In all cases the MMU is not busy, i.e., $M_R.mmu(c).walk = \bot$ as well as

$$\neg trans(M'_R.mmu(c)) \wedge \neg M'_R.mmu(c).walk.complete\,,$$

therefore the invariant still holds in M'_R.

INV-REQUEST Only the first clause of the invariant may be affected by a core step. In case $M'_R.c2u(c) \neq \text{None}$ that new MMU request was either produced by the current step of the core or it was there before, according to Assumption 3.

1. $M_R.c2u(c) = M'_R.c2u(c) \neq$ None – then by invariant Inv-Request for M_R we also know $M_R.C(c).curr = M_R.c2u(c)$, i.e., the core had the same outstandning step before. By Inv-Msg we know that $M_R.u2c(c) =$ None, i.e., the core executed a stuttering step. Hence,

$$M'_R.C(c).curr = M_R.C(c).curr = M'_R.c2u(c)$$

   yields the invariant for M'_R

2. $M_R.c2u(c) =$ None – the step produced a new MMU request and due to the semantics of the *curr* history variable we have $M'_R.C(c).curr = M'_R.c2u(c)$, i.e., the invariant holds for M'_R.

Inv-Reply Only the third clause can be affected by a core step and only if $M'_R.u2c(c) \neq$ None. However, as explained earlier, by Assumption 3 and invariant Inv-Reply on $M_R.u2c(c)$ the case $M'_R.u2c(c) \neq$ None is not possible after a core step.

Inv-Hyp-Gcpy This invariant could be invalidated if a guest core step could change the exception level to EL2 or higher, while there is still an outstanding guest write request to the GIC distributor that breaks the coupling of the GIC with its local copy. However, as proven above, this scenario is impossible and no exceptions are taken before any outstanding memory request is handled completely, in this case by returning a fault.

Inv-Hyp-Gwrite This invariant is maintained for the same arguments as the previous one.

Inv-Hyp-Greq The first clause of this invariant could be invalidated if a guest core step could change the exception level to EL2 or higher, while there is still an outstanding memory request, targeting other addresses than the GIC components mentioned. Again this is not possible as argued before.

A core step could also in principle affect the last clause of this invariant by changing its current request while there is already a pending one in the GIC for that core. We argue as above using Inv-Hyp-Greq, Inv-Walk, and Assumption 1 that this cannot happen.

Inv-Hyp-Gmsg These invariants hold vacuously in the post-state of any guest transition, since if EL2 is entered the program counter points to a hypervisor entry point. None of the PC addresses mentioned in the invariant are such entry point addresses.

Inv-Hyp-LastIGC This invariant is maintained for the same arguments as the previous one.

Inv-Hyp-AckIGC This invariant is maintained for the same arguments as Inv-Hyp-Gmsg.

Inv-Hyp-Pmsg The last two sub-clauses of this invariant can also not be broken by a guest step for the same reasons as those given for Inv-Hyp-Greq.

Inv-Hyp-Iso Only sub-clause eight of this invariant mentions a part of the state that can be modified by a guests step, i.e., $M'_R.u2c(c)$. However any step of the guest can only consume a reply from the MMU or stutter, thus the invariant holds also in M'_R trivially.

4.11.2 Case: MMU

The refined model performs transition MMU, i.e., an active core c in guest mode is stepped according to transition relation δ_{mmu}:

$$\delta_{mmu}(mmu, c2u, m2u) \ni (mmu', u2c', u2m')$$

Note that for all MMU steps the transition system is restricting the inputs such that

$$c2u = \text{None} \vee m2u = \text{None}\,,$$

i.e., there can never exist both a request from a core and a reply from the memory to be handled. Moreover the MMU may only be stepped if its output channels are empty, i.e., $M_R.u2c(c) = \text{None}$ and $M_R.u2m(c) = \text{None}$.

Again we list the invariants that can be affected by an MMU step and why they still hold afterwards. Note also that an MMU step only changes the MMU state and its outputs.

Inv-MMU This invariant can only be broken if the configuration of the MMU changes during one of its steps, however this cannot happen due to Assumption 38 and the fact that an MMU step cannot change the state of its corresponding core.

Inv-HVEntry The MMU cannot change the core state with its step, thus when the hypervisor is in an entry point on the corresponding core, the MMU can only break this invariant by generating outputs or leaving the idle state. However, by Assumption 42 this is not possible.

Inv-Msg We know that the MMU consumes all its outstanding inputs and that there is at most one input present in any MMU step, i.e., $M'_R.c2u(c) = \text{None}$ and $M'_R.m2u(c) = \text{None}$ in any case. Moreover, by Assumptions 41 and 43 it preserves the number of messages in the system in case it has completed a translation and receives its final reply from memory, or when its inactive and just forwarding requests and replies between core and memory. Thus there are two cases left.

1. $trans(M_R.mmu(c))$ – the MMU is in the process of translating a memory request from the core. By the first clause of Inv-Walk also the peripheral channels (including those to the GIC) are empty of messages for core c. Moreover, by Assumption 40 we know that $M'_R.u2c(c) = \text{None}$ and that the MMU only may produce new walk requests in case it has no outstanding ones or it consumed a reply from memory in the same step. In any case, only $M'_R.u2m(c)$ contains a potential message tied to core c in the system.

2. $mmu(c).walk.complete \wedge M_R.m2u(c) = \text{None}$ – the MMU is in its final phase, but it has not received a final reply from memory, then there are two cases.
   (a) $M'_R.mmu(c).walk.final \neq \text{None}$ – the MMU is waiting for its final reply from memory, then $M'_R.u2c(c) = \text{None}$ by Assumption 41.
      i. $M_R.mmu(c).walk.final = \text{None}$ – The MMU has just sent the corresponding request and it is contained in $M'_R.u2m(c)$. By Inv-Hyp-Pmsg we know that there were no former outstanding request or replies for core c in the channels between memory and the peripherals or GIC, and thus $M'_R.u2m(c)$ is the only message of c in the system.
      ii. $M_R.mmu(c).walk.final \neq \text{None}$ – The MMU has sent the final request earlier but not received a request yet, then $M'_R.u2m(c) = \text{None}$ as well and by Inv-Msg there is at most one outstanding message for c in the remaining peripheral/GIC channels that are not changed by the MMU step.
   (b) $M'_R.mmu(c).walk.final \neq \text{None}$ – the MMU is in its final phase but has not sent a final memory request. Then by Assumption 41 a fault was generated instead and returned to the core. This is the only message for c in the system as $M'_R.u2m(c) = \text{None}$ and since, by $M_R.mmu(c).walk.final = \text{None}$, Inv-Hyp-Pmsg yields that there are no outstanding messages for core c in the remaining peripheral/GIC channels of the system.

   As there is at most one message for core c outstanding in the system after the MMU step, Inv-Msg holds also for M'_R

Inv-Walk We prove the three clauses one by one:

1. All inputs to the MMU are consumed by the step, so we only need to focus on $M'_R.u2m(c)$ here during the translation phase. From Assumption 40 we get the desired property directly, i.e., only page table walks are generated here. Moreover while the MMU is in its translation phase there are no outstanding requests in the GIC or peripherals, by Inv-Walk on M_R. If the translation phase was just entered then, $M_R.mmu(c).curr = \perp$ and Inv-Busy guarantees the same. These channels and components are not affected by the MMU step, thus the invariant still holds.
2. There are two cases here. If the MMU is idle and it receives a new request from the core through $M_R.c2u(c)$, the MMU sets its current request accordingly. By Inv-Request we know that this channel contains $M_R.C(c).curr$. In the other cases the MMU is busy, then it never changes its current request by Assumption 38 and the invariant is thus preserved.
3. If there is no outstanding request in the core, i.e., $M_R.C(c).curr = \text{None}$, we know from the same invariant the the MMU is idle. Then

by Assumption 42 the MMU would perform a stutting step and still be idle. If the output channel to the core is not empty, then the MMU step is not enabled and there is nothing to prove. If a step is enabled, from the previous clause we know that the MMU is only busy if there is an outstanding request in the core, so the only way to break this invariant is by producing a reply and not returning to the idle step. However, Assumption 41 forbids this behaviour; the current request is reset whenever an output is returned to the core.

Inv-Busy Just consuming a message from $M_R.m2u(c)$ is benign to this invariant. Thus the only way to break it by an MMU step is to produce a new request in $M'_R.u2m(c)$ or to turn idle while there is an outstanding request. However, to do this an active MMU obviously needs to be busy, i.e., in its translation or final phase. Otherwise by Assumption 42 it performs a stuttering step. By Assumptions 40 and 41 the MMU stays busy until a reply is received for any outstanding request, therefore, when the MMU turns idle again, no outstanding request for core c remains in the system beyond the MMU.

Inv-Request Only the first two sub-clauses of this invariant can be affected by the MMU step changing $M'_R.c2u(c)$ or $M'_R.u2m(c)$. Concerning the first clause, the MMU step always sets $M'_R.c2u(c) = \text{None}$, therefore the invariant holds vacuously. For the second clause we need to consider when the MMU updates its output to memory with a non-empty message.

1. $trans(M'_R.mmu(c)) \wedge M'_R.u2m(c) \neq \text{None}$ – The MMU is translating a request and has produced an output to memory. By Assumptions 39 and 40 we know that the step started in the MMU's translation phase and that the output is a page table walk request. We also know that $M'_R.u2m(c) = M'_R.mmu(c).walk.lookup$, thus the first case of the invariant is proven.
2. $M'_R.mmu(c).walk.complete \wedge M'_R.u2m(c) \neq \text{None}$ – The MMU is in its final phase and produced a memory request. Again by Assumption 39 we know that the step started in the MMU's final phase, as otherwise no memory request would have been produced. By Assumption 41 the only possible value for $M'_R.u2m(c)$ is $M'_R.mmu(c).walk.final$, thus proving the second case of the invariant.
3. $\neg M'_R.mmu(c).active \wedge M'_R.u2m(c) \neq \text{None}$ – The MMU is inactive and just forwards core requests. With Assumption 43 we get $M'_R.u2m(c) = M_R.c2u(c)$ and by Inv-Request on M_R we know that $M_R.c2u(c) = M_R.C(c).curr$, therefore the invariant's third case holds as well.

Inv-MemReq The first part of this invariant is in principle depending on the MMU step as it could switch phase and change the value of the the current lookup or final memory request. However, we know from Assumptions 39 and 41 that the these parts of the MMU can only change when a reply

from memory is received by the MMU. However, Inv-Msg forbids any outstanding messages for core c in the system while a request by it is present in memory. Therefore any step of the MMU is stuttering and the invariant is preserved.

Inv-Reply All MMU steps consume any outstanding reply from memory thus the first part of the invariant holds vacuously. If the MMU produces a reply to the core it might break the second sub-clause of the invariant, however by Assumption 41 we know that all replies match the request that was sent to the MMU. By Inv-Walk this current request is the same as the outstanding request of the core and thus the invariant holds.

Inv-Hyp-Gcpy While hypervisor sends requests to the GIC distributor we have by Inv-HVentry that the hypervisor is not in an entry point. Then we know that the MMU is turned off and only forwards core requests from $M_R.c2u(c)$ to $M'_R.u2m(c)$. Thus each GICD write in $C2GR_c(M_R)$ is still in the set after any MMU step.

Inv-Hyp-Gwrite The proof for this invariant is similar to the one above.

Inv-Hyp-Greq During hypervisor GIC interaction the MMU is inactive and just forwards core requests, therefore the first clause of this invariant is maintained trivially.

The second clause is guaranteed by the fact that, due to the Golden Page Table configuration, in EL1 and EL0 the MMU never sends request outside of the guest memory associated with core c. In particular it follows from Inv-Hyp-Iso and Inv-Request on M'_R. More information about this memory isolation argument will be given in the proof of Inv-Hyp-Iso.

For the third clause, we need to show that steps by the MMU do not change the current final request while a request for the same core is pending in the GIC. The proof is similar to the one for Inv-MemReq. In addition we use the same argument as above to show that no new GIC requests outside the virtual interface are generated by the MMU for guest requests.

Inv-Hyp-Gmsg Again we argue by Inv-Hyp-Greq that the MMU for core c is turned off while the hypervisor is running on this core and accesses a GIC. Then any such outstanding request or reply is just forwarded by the MMU and unchanged, therefore all sub-clauses of this invariant are preserved.

Inv-Hyp-AckIGC This invariant is proven in the same way as the previous.

Inv-Hyp-Pmsg This invariant is proven in a similar way as Inv-MemReq, showing that the MMU does not change it current final request while it is still outstanding. By Inv-Msg we know that there are no replies to the MMU present while there are outstanding requests in the peripheral and GIC channels and components.

Inv-Hyp-Iso The first, third, and eighth sub-clauses of this invariant depend on the MMU translation of guest requests. We know by Inv-MMU that the MMU is always active when guests are running and that it is configured according to the Golden Image setup, making it access only the Golden Image page tables that restrict the memory accesses of a guest to its own memory region. We omit a detailed proof of the exact workings of the MMU as it requires arguing about the detailed MMU model and the format of the page tables. However a formal proof has been conducted in HOL4 to assure that the Golden Image setup indeed confines the guests to their own memory regions as desired.

4.11.3 Case: SMMU

The proof that the implementation invariants are maintained by the SMMUs is similar to the one for the MMU. Many cases are simplified by the fact that the SMMUs are not interfering with core requests, MMU requests, as well as memory-mapped I/O requests in the peripherals and the GIC. Moreover the hypervisor never touches them or their input channels again after initialization. We omit a detailed proof for brevity.

4.11.4 Case: Peripheral

Also for peripherals the proof is slightly easier. The Peripheral rule describes the peripheral step which covers both the handling of memory-mapped I/O requests of cores as well as the sending and receiving of DMA requests to or from the SMMUs. Moreover peripherals may raise their own interrupts and they consume and produce external event signals. The rule is built around the peripheral transition functions for peripherals $p \in \mathbb{N}_{np}$ repeated below.

$$\delta_P^p(P.st, v2p, p2v, E) \ni (P.st', v2p', p2v', e, Q_{out})$$

Note that the rule does not restrict the inputs to the peripheral in any way, however we have defined assumptions above that restrict the behaviour of peripherals. Below we list for each relevant invariant why they are preserved.

Inv-HVEntry This invariant cannot be broken by a peripheral step while the hypervisor is in an entry point on one core. The intuitive reason is that a core first finishes outstanding I/O requests before jumping into the hypervisor and that the hypervisor itself does not touch the peripherals. Thus the only messages generated by a peripheral in absence of I/O requests are DMA requests that are not governed by this invariant. Formally we rely on Assumption 46 which mandates that I/O replies to a given core may only be generated if there is an outstanding and matching request from the same core. By Inv-Hyp-Pmsg such I/O replies are still coupled with the current request outstanding in the MMU which nonetheless is idle, thus the scenario is impossible.

Inv-Msg As explained above, peripherals only generate new I/O replies in case there is an outstanding I/O request of a core c in $M_R.P.st(p)$. Then however, Inv-Msg demands that all channels in the system are free of requests or replies on behalf of core c. Thus the peripheral may at most generate an I/O reply (maintaining the number of outstanding messages) or new DMA requests which are benign to this invariant.

Inv-Walk This invariant is preserved just as in the previous case, as peripherals do not generate spurious I/O replies. If there was an outstanding I/O request by a core c in peripheral p it would need to correspond to a final memory request of its MMU according to Inv-Hyp-Pmsg. However, as the first clause of the invariant demands that the MMU is in translation mode, this case cannot occur since then $M_R.mmu(c).walk.final = \text{None}$ by Assumption 40.

Inv-Busy A peripheral step does not lose or create new I/O requests or replies to a given core by Assumptions 45 and 46. Thus the set of messages memtioned in this invariant is unchanged and the invariant still holds.

Inv-PWalk The second and third clause of this invariant could be broken if the peripheral could drop a current request while it is handled by its SMMU. However, by Assumptions 45 and 46 we know that this only happens when a reply from the SMMU is consumed, but this does not occur while the SMMU is busy according to the third clause of Inv-PWalk on M_R.

Inv-Request The third clause is the only relevant part of this invariant. It holds due to the semantics of the history variable $M'_R.P.st.curr$ for DMA requests. In particular this variable is updated whenever a new DMA request is generated by the peripheral and by Assumption 45 no new request is generated until a matching reply is received.

Inv-Reply The fourth clause of this invariant holds vacuously in M'_R as by Assumption 45 any present DMA reply is consumed immediately by a given peripheral.

The fifth clause follows directly from Assumption 44 that forbids peripheral I/O requests.

Inv-Dma We prove the property on the peripheral output channel for the first two parts separately.

1. While there is a DMA request outstanding in memory, by Inv-MemReq we know that the SMMU is busy and Inv-PWalk guarantees that the current request in the corresponding peripheral p matches its current DMA request. From the second part of Inv-Dma we know that the output channel of the peripheral is empty in M_R. Then with Assumption 45 we conclude the claim for the peripheral output channel, i.e., no new DMA requests are produced while a previous one is outstanding.

2. We assume the corresponding SMMU is busy, then the proof of this part follows the same arguments as above.

Inv-Hyp-Pmsg Only the first two clauses are relevant for peripheral steps. We take them one by one

1. By Assumption 45 all inputs are always consumed directly in a peripheral step. Thus any outstanding I/O request in the peripheral either was their before or it was taken from its input channel. In both cases the claim holds by Inv-Hyp-Pmsg on M_R.
2. This clause follows directly from the first clause on M_R and Assumption 46 which restricts I/O replies to match a previously outstanding request.

Inv-Hyp-Iso Only the last two clauses are relevant for peripheral steps. The first one (clause 9) holds vacuously since any request is immediately consumed by the peripheral. The last one (clause 10) is proven in the same way as Inv-Hyp-Pmsg, deducing each part of the implication from the previous one holding on M_R.

4.11.5 Case: Mem-Core

The rule Mem-Core describes the memory transitions that do not consume memory-mapped I/O requests of the cores. It uses the transition relation δ_M as follows.

$$\delta_M(m, cif, \mathrm{C}\ c) \ni (m', m2c')$$

Note that the input *cif* is not targeting any of the peripherals but the physical memory instead. Moreover, such a memory step consumes the given input request immediately and adds it to the set of currently outstanding requests $m'.curr$ by Assumption 7. On the other, by Assumption 6, memory changes only when a reply for a corresponding write reply is produced.

The proof of the implementation invariant for such steps is simplified by the fact that all invariants only talk about hypervisor or boot memory. By the fifth clause of Inv-Hyp-Iso however, core requests pending in memory only target disjoint guest memory, therefore it is trivial to show that memory updates can never violate the invariants on the hypervisor data structures and images.

Moreover, this rule is only scheduled for handling core requests that do not perform memory-mapped I/O accesses, therefore we can ignore all channels to the GIC and peripherals. The remaining invariants that guarantee the consistency of outstanding core requests in the CPUs, MMU, memory, and the corresponding message channels are proven in an analoguous fashion as their counterparts in the core and MMU step. We therefore omit a formal treatment here for brevity.

Finally, invariants Inv-Hyp-Gcpy, Inv-Hyp-Gwrite, Inv-Hyp-Greq, as well Inv-Hyp-Gmsg only concern hypervisor memory-mapped I/O accesses, hence they are trivially preserved, as the Mem-Core cannot affect them.

The only interesting invariant to be perserved is thus Inv-Hyp-Iso. We give a detailed proof below.

Inv-Hyp-Iso Only the clauses two and the core request part of the seventh clause of this invariant can be affected by a Mem-Core step.

2. When the MMU of core c is in its translation phase and the reply $M'_R.m2u(c)$ from memory is empty, we know by Inv-Reply that the reply matches the MMU's current lookup request. By the first clause of this invariant we obtain that this request r is a page table walk to the Golden Image page table belonging to core c, i.e., $PTreq(r, A_{PT}(\gamma(c)))$. Then by the definitions of *match* and *PTreq* we obtain the first line of the claim.
   When the memory produces a new reply, Assumption 8 gives us that this page table walk request was pending in memory before and is returning value $v = M_R.m_8(a)$ with $a[47:12] \in A_{PT}(\gamma(c))$. From the definitions of byte-addressable memory and byte-addressable pages we see that $M_R.m_8(a) = M_R.m(a[47:12])_8[a[11:0]]$. With Inv-PT we conclude:
   $$v = M_R.m(a[47:12])_8[a[11:0]] = GI_{\gamma(c)}(a[47:12])_8[a[11:0]]$$
7. By Assumptions 7 and 8 as well as the semantics of the *curr* history variable, we know that requests in memory for a given core c stay unchanged except if a new request is received for that core or if a corresponding reply is produced. In the latter case the request is simply removed from $m.curr$, thus the invariant is preserved trivially. In the other case we have $M_R.u2m(c) = (r, \mathrm{C}\ c)$ for some non-I/O request $r \in \mathbb{R}$. By Inv-Request we know that any request sent to memory by a cure is either equal to $M_R.mmu(c).walk.lookup$ (in case the MMU is currently translating), $M_R.mmu(c).walk.final$ (in case the MMU in its final phase), or $M_R.C(c).curr$ (in case the MMU is inactive). Since we can ignore all hypervisor requests according to Inv-Hyp-Greq, as they only target the GIC, we can assume that the MMU is active. Then by the first clause of Inv-Hyp-Iso on M_R we have
   $$PTreq(M_R.mmu(c).walk.lookup, A_{PT}(\gamma(c)))$$
   in case the MMU is translating, and by the third clause we have
   $$req(r, A_G(\gamma(c))) \vee PTreq(r, A_G^-(\gamma(c)))$$
   for $r = M_R.mmu(c).walk.final$ in case the MMU is in its final phase. As the addresses in $A_G(\gamma(c)) \setminus A_G^-(\gamma(c))$ are all in peripheral memory, we can ignore such request for this memory step and get
   $$req(M_R.u2m(c), A_G^-(\gamma(c))) \vee PTreq(M_R.u2m(c), A_G^-(\gamma(c)) \cup A_{PT}(\gamma(c)))$$
   for the remaining ones. By Assumption 7 we know that each such request will be added to $M'_R.m.curr$ and we conclude the invariant.

4.11.6 Case: Mem-Periph

The proof for step Mem-Periph is completely analoguous to the one above. It is even simplified by the fact that it treats only peripheral DMA requests and replies transfered between memory and the SMMUs, so all behaviour of the hypervisor, the core, the MMUs, the peripherals, and the GIC can be ignored. In particular, by construction, the memory never forwards DMA replies to the GIC, i.e., the third part of Inv-Dma holds trivially.

Again we argue that the hypervisor data structures are never modified by memory updates as all peripheral DMA requests are restricted to non-I/O guest memory by Inv-Hyp-Iso and the configuration of the Golden SMMU page tables. The proof that this is preserved by DMA accesses follows exactly the same lines as the one for core memory accesses, therefore we omit a repetition.

4.11.7 Case: Mem-I/Oreq

The transition Mem-I/Oreq moves memory-mapped I/O requests of cores from the MMU output channels to the input channels of the peripherals, bypassing the physical memory. An important feature of this step is that it is only enabled when the targeted *v2p* channel is empty and the source channel *u2m* contains a message to a peripheral or GIC, identified for $p \in \mathbb{N}_{np+1}$ by the following predicate:

$$mmio(u2m, p) \equiv u2m \neq \text{None} \wedge adr(u2m)[47:12] \in A_P(p)$$

Note that the model does not forbid page table walks to peripherals, we will show that the MMU setting forbids these kind of memory-mapped I/O accesses. Note also that *u2m* is always emptied by the step and *v2p* always contains the former value of the source channel.

The proof of the implementation invariant is simplified by the fact that we only need to treat memory-mapped I/O accesses and only need to focus on the channels *u2m* and *v2p*. Therefore the state of the cores, the memory, the SMMUs, and the GIC can largely be ignored. In particular all invariants on hypervisor data structures are maintained. However, there are hypervisor requests to the GIC which need to be treated. Below we list the proofs for all relevant invariants. The others are preserved trivially as *u2m* and *v2p* are not mentioned.

Inv-HVEntry If the hypervisor is in an entry point on core c in M_R, transition Mem-I/Oreq is not enabled for core c as $M_R.u2m(c) = \text{None}$, therefore the invariant is preserved trivially.

Inv-Msg Observe that the I/O request memory step preserves the number of messages in the system. Therefore this invariant is preserved as well.

Inv-Walk This invariant can be broken if, before the step, the MMU is in translation phase and there is a page table walk in the channel from MMU

to memory but the address is pointing to peripheral memory, i.e.,

$$\exists p \in \mathbb{N}_{np+1}.\ PTreq(M_R.u2m(c), A_P(p))\,.$$

In this case the first clause of the invariant may be invalidated. However by Inv-Request we also know that $M_R.u2m(c) = M_R.mmu(c).walk.lookup$ and from the first clause of Inv-Hyp-Iso we get

$$PTreq(M_R.mmu(c).walk.lookup, A_{PT}(\gamma(c)))\,.$$

We deduce $PTreq(M_R.u2m(c), A_{PT}(\gamma(c)))$ and as $A_{PT}(\gamma(c)) \subset A_{HV}$ is disjoint from any $A_P(p) \subset A_G(\gamma(p))$, we know that page table walks are never pointing to peripheral memory. We conclude $\neg mmio(M_R.u2m(c), p)$ thus the I/O request memory step is not enabled.

Inv-Busy This invariant is maintained trivially as the memory I/O request step maintains the set of messages present in the system.

Inv-Request This invariant is maintained trivially as any enabled I/O request memory step will empty the channel from MMU to memory, then the second clause of the invariant holds vacuously.

Inv-Reply The fourth clause of this invariant is preserved trivially as transition Mem-I/Oreq is not enabled while $M_R.v2p(p)$ is not empty. Moreover the step only updates $M'_R.v2p(p)$ with I/O requests that have a different type then the DMA replies that are constrained by this invariant.

Inv-Dma The invariant is preserved trivially as the step only moves I/O requests to the peripheral inputs as explained above. The same argument is used to prove that both clauses are maintained.

Inv-Hyp-Gcpy This invariant couples the local GIC copy of the hypervisor with the GIC configuration for registers that are not currently written by the hypervisor. If there is a memory-mapped I/O write to the GIC in $M_R.u2m(c)$, the memory step will move it into $M'_R.v2p(p)$ unchanged. Therefore $C2GR_c(M'_R)$ is unchanged as well and the invariant is preserved.

Inv-Hyp-Gwrite The proof of this invariant is similar to the case above. As the I/O request memory step does not modify or create write requests to the GIC, the invariant is maintained as well.

Inv-Hyp-Greq The proof for this invariant is identical to the one above as the memory step transfers I/O requests without modification.

Inv-Hyp-Gmsg The same argument applies to all clauses in this invariant mentioning GIC I/O requests.

Inv-Hyp-Pmsg Only the parts of this invariants constraining channels $M'_R.v2p$ need to be considered. We first show that whenever an I/O request $r = M_R.u2m(c)$ is moved to $M'_R.v2p(p)$ for peripheral $p \in \mathbb{N}_{np}$ we have:

$$r = M'_R.mmu(c).walk.final \wedge req(r, A_P(p))$$

We already showed above that any request delivered to the peripheral is not a page table walk. Also by the definition of the transition we know $req(r, A_P(p))$ from $mmio(r, p)$.

From the contrapositive of the first line of Inv-Walk we get that the MMU cannot be in translation phase, i.e., it is either in its final stage, or idle, or inactive. By Inv-Hyp-Greq we obtain that the guest is running, as the hypervisor accesses only the GIC. Invariant Inv-MMU yields that then the MMU is active and according to Inv-Busy the MMU cannot be idle while there is an outstanding request for c in this case. Thus the MMU is in its final stage and we use Inv-Request to deduce our claim. For requests being transferred to the GIC, the poof is similar. We have an additional case for GIC requests by the hypervisor, however it is also covered by Inv-Request.

Inv-Hyp-Iso Here only the last invariant (10) is relevant. We need to show that whatever memory-mapped I/O request is transferred to peripheral p, it is sent by a core c that belongs to the same guest. From the last invariant we know that core c's MMU is in its final phase and:

$$r = M'_R.v2p(p) = M'_R.mmu(c).walk.final \wedge req(r, A_P(p)) .$$

The third clause of Invariant Inv-Hyp-Iso yields

$$req(r, A_G(\gamma(c))) \vee PTreq(r, A_G^-(\gamma(c)))$$

The latter case is not possible as it excludes any peripheral addresses therefore the I/O request MMU step would not have been enabled. Thus we have

$$req(r, A_P(p)) \wedge req(r, A_G(\gamma(c)))$$

and from the definition of predicate *req* we get that $A_P(p)$ and $A_G(\gamma(c))$ are at least overlapping. However, since all peripheral memories are subset of some guest memory we obtain $A_P(p) \subseteq A_G(\gamma(c))$. Moreover, in Sect. 4.1 we required that $A_P(p) \subseteq A_G(\gamma(p))$, hence, as all guest memories are disjoint, we conclude $\gamma(c) = \gamma(p)$.

4.11.8 Case: Mem-I/Orpl

The transition rule Mem-I/Orpl is symmetric to the I/O request case. Thus also the proof is similar. In fact, the hardest cases in the above case are trivial here, since invariants that have been established by delivering the request to the peripheral or GIC are maintained by Inv-Hyp-Pmsg. We therefore omit a detailed proof.

4.11.9 Case: GIC

For GIC steps, modeled by transition GIC, the proof does not need to consider invariants that constrain peripherals, SMMUs, and memory, as well as the corresponding channels. However, cores communicate with the GIC through the

MMU in guest mode, and directly when the hypervisor is accessing the GIC distributor registers. Moreover the GIC step is the only transition affecting the cores' interrupt interfaces. The GIC transition relation in the refined model is given by

$$\delta_Q(GIC, Q, VI, m2g, PI) \ni (GIC', Q', VI', g2m')\,,$$

where $m2g$ represents input channel $M_R.v2p(np)$ and $g2m'$ is used to update the output channel $M'_R.p2v(np)$. Moreover the peripheral interrupt state PI affects the resulting physical and virtual interrupt states Q' and VI'. The GIC register state GIC is affected mainly by memory-mapped I/O accesses of the cores, but certain registers may also be affected by interrupt signals, reflecting the states of interrupts in the core interfaces.

Some of the GIC distributor registers need to be coupled with the local hypervisor copies for each guest. Besides that, the isolation of interupts is the main concern when proving the implementation invariant for GIC steps. Below we go through all the relevant invariants and prove them one by one.

Inv-GICPol The first part of this invariant can only be broken by updates to the GIC distributor according to Assumption 15. If there is such an outstanding write request in the GIC we know by Inv-Hyp-Pmsg that it is equal to the currently outstanding request in $M_R.C(c)$, however such requests are constrained by Inv-Hyp-Gmsg to be processed by the GIC request conversion function. Thus the invariant is preserved by Assumption 25 and the distributor is still in a Golden configuration.

The second part of the invariant concerns the pending and active interrupts of the system. If interrupts are deactivated, the invariant is preserved trivially. Thus we only need to treat the case where new interrupts are made pending or active-and-pending. This can be achieved through GIC distributor register accesses (for all interrupts, including SGIs) or the GIC itself, forwarding peripheral interrupts to the cores' interrupt interfaces.

In the first case we know again that all GIC distributor accesses are filtered through the GIC request conversion function for the requesting guest, thern by Assumption 25 no interrupts may be made pending or active that are not associated with. There is a special case here for writes of the hypervisor that request SGIs being sent as IGC notifications. Invariant Inv-Hyp-Gmsg ensures that the hypervisor only sends SGIs to cores of other guests if this is allowed by the IGC communication policy.

Concerning the delivery of peripheral interrupts, Assumptions 30 and 23 and the definition of the Golden Mask GM_g guarantee that only interrupts from the set $PIRQ_g$ may be signalled to cores of guest g.

Inv-GICInterface According to Assumption 15 and 21 only write requests may update the GIC CPU interface control register. However, by Inv-Hyp-Greq the guest never accesses the GIC CPU interface and by Inv-Hyp-Gmsg the hypervisor never touches the control register after reset, therefore the invariant is preserved trivially.

Inv-IgcSgi In general, SGIs can only change their interrupt status due to I/O requests to the GIC distributor or CPU interface, according to Assumption 22. By invariants Inv-Hyp-Greq we know that only the hypervisor may touch these registers.

For the scenario where an SGI representing an IGC notification interrupt is made pending through a write to the *GICD_SGIR* register, in Inv-Hyp-Gmsg there are only two matching cases. In the first case however, the access is virtualizing a guest access to the distributor and filtered by the GIC request conversion function. According to Assumption 25 it therefore cannot activate an SGI to a different guest.

Thus the only possible case is a write to the *GICD_SGIR* register during the execution of the IGC Notification Request handler. The corresponding invariant yields the claim on the IGC message box directly.

A similar argument holds for making SGIs active. As the hypervisor never activates SGIs directly in the distributor, the only case where an SGI can become active is by reading the acknowledgment register. In this case it was pending before according to Assumption 35, therefore the first part of the invariant is preserved trivially.

The second part of invariant can only be broken if at core c an SGI with ID 15 from a different guest was made active-and-pending directly by the hypervisor, however there is no case in Inv-Hyp-Gmsg that would have such an effect. In particular guest accesses to the distributor are filtered so they cannot manipulate SGIs that to not belong to them.

Inv-HVentry While there are no outstanding requests from core c in the GIC inputs, none can become outstanding by the semantics of the *curr* history variable. The same holds for replies from the GIC by Assumption 15.

Inv-Msg For each core the GIC either consumes a request as an input, or signals interrupts to the core without producing any output, or it generates a reply for an outstanding I/O request. In the two cases the invariant is preserved trivially. Otherwise, by Inv-Msg on M_R we know that no other messages are present in the system for core c thus there is still at most one message for core c in the system by Assumption 15.

Inv-Walk As above the only way to break the invariant by the GIC is to generate an I/O reply. However invariant Inv-Hyp-Pmsg tells us that such a request is either to the virtual GIC interface, in which case the MMU is in its final phase and not translating according to Assumption 40, or to another part of the GIC, in which case Inv-Hyp-Greq yields that the hypervisor is executing. By Inv-Hyp-Gmsg we know that it is not in an entry point and thus the MMU is not active by Inv-MMU. Consequently the MMU is not in its translation phase when the GIC produces an output and the invariant holds vacuously.

Inv-Busy By Assumption 10 and the semantics of the *curr* history variable, the GIC does only add outstanding request when it receives it through its input channel. Similarly, by Assumption 15 whenever a reply is produced the corresponding outstanding request is remuved from the set of current requests, therefore the invariant is preserved trivially.

Inv-Request Only the last part of this invariant is relevant. It is self-preserving, as by Assumption 10 any request in the GICs input channel is saved as-is in the set of current requests.

Inv-Reply Similarly, The last part of this invariant is directly established by Inv-Request and Assumption 15.

Inv-Dma The third part of this invariant holds because the GIC never performs DMA requests, according to Assumption 12.

Inv-Pirq This invariant is preserved directly by Assumption 26.

Inv-Hyp-Gcpy If there is no outstanding write in the GIC to a distributor register r for which $\mathit{emugicd}(r, 0)$ holds, their values are unchanged according to Assumption 19. If such a write W a d v exists and is processed (enabling the invariant), it must stem from the hypervisor by Inv-Hyp-Greq. Invariant Inv-Hyp-Gmsg gives us that it either stems from the SGI notification request handler or the GICD virtualization handler.

In the both cases, Inv-Hyp-Gwrite tells us that the outstanding write preserves the coupling between the local GICD copy and the result of the GIC update function, wrt. the GICD register filter function. g due to the register filtering function. Using $M'_R.m = M_R.m$ as well as Assumption 15 we deduce:

$$\begin{aligned} \mathit{gcpy}_g(M'_R.m(a^g_{\mathit{gcpy}}), r) &= \mathit{gcpy}_g(M_R.m(a^g_{\mathit{gcpy}}), r) \\ &= \eta^r_g(\nu(r, M_R.\mathit{GIC}.\mathit{gicd}(r), v) \\ &= \eta^r_g(M'_R.\mathit{GIC}.\mathit{gicd}(r)) \end{aligned}$$

This proves the invariant.

Inv-Hyp-Gwrite GIC steps cannot break this invariant since either a write access for which the invariant holds is consumed and outstanding afterwards due to Assumption 10, or such an outstanding request is processed by the GIC and removed from the outstanding requests according to Assumption 15, making the invariant hold vacuously for the core whose request was processed. Requests from other cores, that are constrained by this invariant, are not affected and the invariant is preserved trivially.

Inv-Hyp-Greq As above the GIC may only consume inputs or produce replies to outstanding I/O requests. Moreover it cannot affect the state of the core or the MMU, therefore this invariant is preserved in a similar fashion.

Inv-Hyp-Gmsg These invariants are maintained in the same way, using Assumption 10 to transfer properties from the input channels to the outstanding requests, and Assumption 15 for preserving properties of the outstanding requests when they are processed and replies are generated.

Inv-Hyp-LastIGC In order to prove the second part of this invariant, we examine Inv-Hyp-Gmsg for the possible requests that the GIC could process for core c while it is running the hypervisor and the PC points to an address from the set A_{RIGC}.

- $C_R^c.pc = a_{rnew}$ – The hypervisor writes the end-of-interrupt register for the SGI, however by Inv-GICInterface we know that his write only drops the priority due to the setting of the *EOImodeNS* bit in the $GICC\_CTLR$ register. The SGI stays active as given by Inv-Hyp-LastIGC on M_R.
- $C_R^c.pc = a_{rncw}$ – The hypervisor accesses the list register of the virtual interrupt control interface. By Assumption 22, the interrupt state of all SGIs is unchanged.
- $C_R^c.pc = a_{rniw}$ – The same holds for this case.
- $C_R^c.pc = a_{rndw}$ – The hypervisor writes the deactive-interrupt register, however by the antecedent of the invariant the write request that would disable the SGI has not been processed by the GIC, yet. After it does so, the invariant holds vacuously.

Inv-Hyp-AckIGC An inter-processor interrupt representing an IGC notification is acknowledged by the hypervisor. For the corresponding SGI acknowledgement step of the GIC, Assumption 35 guarantees that the corresponding SGI is active in M'_R and that it was pending in M_R. We conclude the property on the message box by Inv-IgcSgi.

Inv-Hyp-Virq This invariant is directly established by Assumption 26

Inv-Hyp-Pmsg The latter two invariants are preserved trivially, by Assumption 10 as well as Assumption 15 and Inv-Hyp-Pmsg on the current GIC requests in M_R.

4.11.10 Case: Ext

For transition Ext there is nothing to show as the peripherals' external event sequence is not constrained by the implementation invariant and the step leaves all other components unchanged.

4.11.11 Case: Hypervisor

The hypervisor step is the biggest part of the proof as it covers the complete hypervisor transition system, i.e., all steps of all handlers as well as the boot and initialization code. Therefore we dedicate a separate section to it.

4.12 Hypervisor Steps

Below we distinguish all hypervisor transitions and show that they preserve the implementation invariant for all relevant clauses, similar to the steps above. Note that the hypervisor transitions only affect the core running the code, its corresponding second-stage MMU, its output channel, and the hypervisor data structures in memory. Additionally the hypervisor may configure the SMMUs but this happens only during the initialization phase, afterwards their configuration is fixed. It also may power up and down other cores

On the other hand, guest memory, the peripherals, and the GIC, the other cores (modulo power management), their interrupt states, and the corresponding channels are not directly affected. The hypervisor always consumes messages from the current core's input channel, but only if it is a reply from the GIC and never in a hypervisor entry state.

Looking at the HYPERVISOR rule and our knowledge about how the hypervisor works, we make a few general observations for hypervisor steps taken on core $c \in \mathbb{N}_{nc}$.

- The hypervisor cannot affect the interrupt state nor the GIC registers directly. Therefore invariants INV-GICPOL, INV-GICINTERFACE, INV-PIRQ, and INV-HYP-VIRQ are preserved trivially.

- The Golden page tables for guest cores and peripherals are set up initially and never touched again. Similarly each SMMU is touched once during initialization. Therefore outside of the initialization steps we do not need to consider invariants INV-PT, INV-PPT, INV-SMMU, INV-PWALK.

- Power management only affects the guest registers and the *active* flag of other processor cores. The system registers keep their values in our model, thus once a core is initialized, INV-REGEL2 and INV-REGEL3 cannot be broken by hypervisor steps on other cores. Similarly on the current core after boot, registers *SCR_EL3* and *VBAR_EL3* are not touched again, therefore INV-REGEL3 is maintained by all other hypervisor transitions.

- Similarly, the MMU is only turned off but not reconfigured during internal hypervisor steps. When the MMU is reactivated at the return to guest mode, it is not busy. Consequently, all invariants restricting the currently active translation and outstanding requests in the MMU hold vacuously. This includes INV-WALK, INV-BUSY, but also part of other invariants to be named below.

- The hypervisor does not touch the peripheral, their input and output channels, or the SMMU state as noted above. Hence, invariant INV-PWALK is preserved trivially.

- Even though the hypervisor modifies memory, it does not do so through the memory interface, but replace the memory contents of its data structures directly. Therefore the currently outstanding request in memory

are not affected and Inv-MemReq cannot be broken by hypervisor steps. The same holds for Inv-Dma.

- While the hypervisor has an outstanding GIC request, it is stalled and does not send any new requests. Its currently outstanding memory request recorded in $C_R^c.curr$ is not changed. Using this fact and the ones listed above we conclude that also invariants Inv-Msg, Inv-Request, Inv-Reply, and Inv-Hyp-Pmsg are maintained directly by steps of the hypervisor.
- Similarly, the hypervisor cannot directly break the memory isolation invariants Inv-Hyp-Iso, as it only constrains the contents of channels and outstanding requests related to MMUs, SMMUs, memory, and the peripherals.
- Neither the hypervisor nor the boot code ever writes to the images in flash memory, therefore their integrity, and invariant Inv-Img, is preserved trivially.

Thus the only relevant invariants left are Inv-PT, Inv-PPT, Inv-SMMU, Inv-RegEl2, and Inv-RegEl3 for the initialization handler, as well as, Inv-Pow, Inv-MsgBox, Inv-IgcSgi, Inv-MMU, Inv-Hyp-Gcpy, Inv-Hyp-Gwrite, Inv-Hyp-Greq, Inv-Hyp-Gmsg, Inv-Hyp-LastIGC, and Inv-Hyp-AckIGC for all other hypervisor transitions after it has been initialized, i.e., $M_R.hvinit$ holds in the refined model. We call these invariants the *hypervisor invariants*.

4.12.1 Case: Boot and Initialization

Below we list the possible transitions of the boot loader and discuss their effect on the implementation invariants.

Init-Abort This step simply jumps into a fail state and does not chang anything besides the program counter, therefore no invariants can be broken by it. In particular those invariants constraining hypervisor states where the PC has a different value hold vacuously, also by the fact that the core is still executing in EL3 and that $M_R.hvinit = 0$.

Init-PCold This step copies the hypervisor images to their target locations and establishes Inv-RegEl3 directly. Afterwards it is trivially preserved as no other transition touches the concerned registers and they are accessible in EL3 only. None of the other hypervisor invariants are concerned as the hypervisor is not initialized yet.

Init-PInit The hypervisor initializes the IGC message box and the power state variable. It also powers up all the other guests' primary cores. After this step we need to establish the following invariants.

Inv-Pow The power-up command is sent for all cores in the set S which comprises the primary cores of all guests, consequently after the step

for all $c' \in S$ we have $M'_R.C(c').active = 1$ and $warm(hv_{c'}(M'_R))$. Additionally, core 0 is still active. The power state variable *pow* is initialized such that it contains ones exactly for the cores in the set $S \cup \{0\}$, and zeroes for all other cores, hence the invariant is established.

INV-MSGBOX The message box is initialized with zeroes for all possible combinations of sender and receiver guests, therefore the invariant holds trivially.

INV-IGCSGI Initially no interrupts are pending as after reset all forwarding of interrupt is disabled. Furthermore, no steps so far touched the GIC, therefore both parts of this invariant hold vacuously.

INIT-GINIT This step directly establishes invariants INV-PT, INV-PPT, and INV-SMMU for the corresponding guest. Afterwards they holds indefinitely since the parts of hypervisor memory and the SMMUs of that guest are never touched again. This uses the fact that the page tables are allocated in disjoint regions of hypervisor memory for each guest. No other hypervisor invariants are to be concerned here.

INIT-CINIT Here the hypervisor directly establishes INV-REGEL2 and INV-MMU for the current core. It also initializes the data structures that record the last received interrupts and IGC notifications, however these updates are not relevant to any of the other invariants. The MMU is active now but all invariants related to it hold trivially since it is not busy. In particular INV-BUSY still holds vacuously after this step as the initialization handler does not send any memory requests.

INIT-LAUNCH The guest is launched in this step. However, the guest state is not constrained by any invariant, therefore all invariants are preserved trivially. In particular, invariants only depend on the program counter during hypervisor execution. Moreover there are no outstanding messages in the system for the running core.

INIT-SCOLD This is the first step after reset for secondary cores. The boot code configures the EL3 registers and powers down the current core, thus enabling INV-REGEL3.

INIT-WARM A secondary core was restarted and enters the hypervisor at address a_{iep}. As this step is only enabled if step INIT-SCOLD was executed before. INV-REGEL3 still holds for this core.

INIT-SPRIM The secondary core is the primary core of its guest. It enters a state where $M_R.hvinit$ already holds, i.e., the primary core already initialized the shared hypervisor structures. Thus are no invariants that need to be established by this step. It is benign to all invariants already holding.

INIT-SSEC The same holds for this step which takes a secondary core of a guest into a state where the guest-specific invariants have already been established in step INIT-GINIT by the primary core of this guest. Note that a secondary core of a guest is only started by the guest itself after its primary core has entered guest mode and therefore already established INV-PT, INV-PPT, as well as INV-SMMU for the guest.

INIT-SOFT Here a guest's core is restarted in a state where it is ready to be launched again. All initialization invariants still hold, in particular INV-REGEL2 and INV-MMU since the hypervisor registers and the second-stage MMU of that core were already initialized by it before the soft reboot.

This finishes the proof for the boot loader. Note that some of the arguments here are somewhat informal. They can be formalized by adding additional history variables recording the progress of the initialization and adding invariants that tie the states of the handler to values of these variables. For instance, when a secondary core is in the warm reset state, the guest initialization step for its guest has already been executed, therefore the guest-specific invariants do not need to be established by the warm boot step. The same holds for cores that are soft-rebooted and the core-specific invariants. We omit a detailed formalization here, in order not to further increase the number of implementation invariants we have to discuss.

4.12.2 Case: Power Control SMC Handlers

In the SMC handler there are only two possible transitions, either the call succeeds and a core of the requesting guest is powered on or off, or the request is denied.

SMC-ISSUE If the request is granted for command cmd and core c' of the requesting guest.

1. $cmd = \text{stop}(c') \wedge C_R^{c'}.active = 1$ – a running core of the requesting guest is powered down, i.e., $M'_R.C(c').active = 0$ and the power state variable pow for the core is set accordingly, preserving INV-POW. Note that this transition is only enabled if core c' has no outstanding memory request, i.e., $C_R^c.curr = \text{None}$ which also holds by default for the powered-down state. Due to the various invariants linking the core's current request with outstanding requests in the rest of the system we know that there is indeed no outstanding requests in the MMU, memory, GIC, peripherals, or message channels for core c'.
2. $cmd = \text{start}(c') \wedge C_R^{c'}.active = 0$ – the core is brought up either as a warm or a soft reboot. In the latter case all initialization invariants already hold. In the first case core c' must be a secondary core of the guest, as the primary cores of each guest are

warm-started only during the initialization phase by step INIT-PINIT. Hence all the guest-specific invariants hold already. Moreover, we have $M'_R.C(c').active = 1$ and the power state variable is updated accordingly, therefore INV-POW is preserved. Note that c' has no outstanding memory requests, i.e., $M'_R.C(c').curr = \text{None}$ as established by the initial state after reset or by an earlier power-down request.

3. otherwise – the command has no effect as it tries to stop a powered-down core or start a core that is already running. All invariants are preserved trivially as nothing is changed for core c'.

SMC-DENY This transition has no effect on the system or its invariants.

In all cases execution returns to guest mode where no memory requests are outstanding by INV-HVENTRY and the fact that the SMC handler does not touch the GIC. Also the memory, hypervisor register, MMUs, SMMUs, as well as the GIC are unchanged, therefore the remaining hypervisor invariants are maintained trivially.

4.12.3 Case: GIC Distributor Virtualization

The virtualization handler for guest GIC distributor accesses affects most of the hypervisor invariant, however the initialization-related invariants are maintained trivially, as it does not modify the page tables, SMMU configurations, nor hypervisor registers. Moreover the data structures related to IGC and the power state are not touched. We prove the remaining invariants INV-HYP-GCPY, INV-HYP-GWRITE, INV-HYP-GREQ, and INV-HYP-GMSG for each transition.

GICD-FAIL The request to the GIC distributor was invalid and thus failed, an exception is injected into the guest and the execution mode is set to EL1. However the guest register state is not restricted by any implementation invariants. As there are also no memory requests generated and the rest of the system is unchanged, the hypervisor invariants are maintained trivially.

GICD-EMU-R In this step the hypervisor reads the local copy of the GIC distributor for the corresponding guest. However no state is updated besides the saved context of the interrupted core and the MMU state. The former is not constrained by the implementation invariant. The MMU is deactivated but otherwise the state is not changed, hence INV-MMU is preserved.

GICD-EMU-W Besides saving the interrupted core context and deactivating the second-stage MMU, this time also the local GIC distributor copy is updated. However this is an emulated request, i.e., the GIC is not accessed, therefore we only need to show that INV-HYP-GCPY is preserved. The registers for which writes can be emulated are *GICD_IPRIORITYn*,

GICD_TYPER, *GICD_IIDR*, and *GICD_ICPIDR2*, but here the written values are ignored. The GICD updated function preserves the previous value of the registers, thus also Inv-Hyp-Gcpy is maintained.

GICD-Req-R Again this step saves the guest core context and deactivates the MMU, preserving Inv-MMU. In addition it acquires the GIC lock and sends a read request to the GIC on behalf of the requesting guest. As the GIC lock is not constrained by any implementation invariant and the request sent is a read, nothing is to show for this step except:

Inv-Hyp-Greq The first part holds obviously as the core is running in EL2 and the read request is targeting the GIC distributor. The other parts hold vacuously as there was no previous outstanding GIC request according to Inv-HVentry and the new request in the core's output channel does not meet the antecedents of these invariants.

Inv-Hyp-Gmsg The PC is set to a_{gdw}, therefore we need to show the first part of this invariant. The request $r = M'_R.c2u(c)$ meets the first part of the claim because for any read request r' we have $\rho_{\gamma(c)}(r') = r'$, i.e., the GIC conversion function does not affect reads.

GICD-Req-W This step is similar to the read request case above, only that we also need to take the update of the local GIC distributor copy and the issued write request into account.

Inv-Hyp-Gcpy Since a write request was issued, this invariant now holds vacuously for guest $g = \gamma(c)$.

Inv-Hyp-Gwrite Let v' be the value to be written as requested by guest g that is taken from the GPR identified by the exception status register, R be the targeted register of the distributor, and v be its former value in the local copy. Obviously we have $\mathit{reqgicd}(R, 1)$ from the precondition of the write request step. From the definition of cpl_{GICD} we know that the same register is written in the copy as by the write request. Moreover, we know that the local copy is updated with the value

$$\mathit{gcpy}_g(M'_R.m(a^g_{gcpy}), R) = \eta^R_g(\nu(R, v, v'))\,.$$

Let $u = M_R.GIC.gicd(R) = M'_R.GIC.gicd(R)$ be the value of R in the GIC distributor, which has not changed due to the hypervisor step. From Inv-Hyp-Gcpy we get $v = \eta^R_g(u)$. Now the sent request has the form

$$M'_R.c2u(c) = \rho_g(\mathrm{W}\ a_{GICD}(r)\ d\ v') = \mathrm{W}\ a_{GICD}(r)\ d\ w$$

and the new value stored in the GIC register according to this write request and Assumption 15 will be $\nu(R, u, w)$. Now with Assumption 37 we deduce

$$\eta^R_g(\nu(R, v, v')) = \eta^R_g(\nu(R, u, w))$$

which proves the invariant.

Inv-Hyp-Greq This invariant is proven just as in the previous case.

Inv-Hyp-Gmsg Again we need to show the first part of this invariant. By definition of the GICD-Req-W rule, the requested GIC access is a translation of the original request by the GIC request conversion function for guest g.

GICD-Unlock-R In this step the hypervisor receives the read reply to its previous request from the GIC. The GIC lock is released, without consequences for any implementation invariants. As the reply is consumed any invariants related to that message hold vacuously now. Moreover, the returned value is stored in the local copy of the distributor after filtering it with the GIC conversion function. However, Inv-Hyp-Gcpy is preserved trivially as $emugicd(r, 0)$ obviously does not hold for the register r that was read from the GIC and updated in the local copy.

GICD-Unlock-W This step only releases the GIC lock and consumes the write confirmation from the core's input channel, thus there is nothing to show here.

GICD-Reply-R Execution returns to the guest restoring its context updated with the read result and no outstanding memory requests. All invariants are preserved trivially.

GICD-Reply-W The proof for the write case is analoguous.

4.12.4 Interrupt Injection

The handler for peripheral and inter-processor interrupts injects phyisical interrupts into the guest cores as virtual interrupts. It touches mainly the GIC CPU interface and the virtual interrupt control interface. The only hypervisor invariants that are affected are Inv-RegEl2, Inv-MMU, Inv-Hyp-Greq, and Inv-Hyp-Gmsg. We prove them below for all transitions of the handler. Note that the handled inter-processor interrupts are only between cores of the same guest. Inter-guest interrupts are treated separately by the IGC notification handler.

IRQ-Ack The hypervisor disables the second stage MMU and sends a read request for the acknowledgment register of the GIC CPU interface. Invariant Inv-MMU is preserved as earlier, since the remaining configuration of the MMU is not changed. The message to the GIC is also in line with the first two parts of Inv-Hyp-Greq. The third part holds vacuously. The PC is set to a_{irqaw} in which case the second part of Inv-Hyp-Gmsg demands that there exists a message to the GIC reading the acknowledgment register. In deed such a message is contained in $M'_R.c2u(c)$.

IRQ-EOI/Drop A priority drop is requested for the received interrupt by writing the end-of-interrupt registers. Invariants Inv-Hyp-Greq and the third part of Inv-Hyp-Gmsg are maintained directly by the generated GIC I/O request.

IRQ-Check The hypervisor reads a list register of the virtual interrupt control interface. Again the definition of the rule directly yields Inv-Hyp-Greq and the fourth part of Inv-Hyp-Gmsg.

IRQ-Inject In this step the list register is updated with a write request to the GIC. Invariant Inv-Hyp-Greq and the fifth part of Inv-Hyp-Gmsg hold as above.

IRQ-Deact The physical interrupt is deactivated by writing the corresponding register of the GIC CPU interface. Invariant Inv-Hyp-Greq and the sixth part of Inv-Hyp-Gmsg hold as above.

IRQ-Finish The MMU is re-enabled and the guest context is restored. Invariant Inv-MMU is preserved since the remaining configuration was not changed. Moreover the hypervisor sets the *VI* bit in the hypervisor control register, signalling the virtual interrupt to the guest. This maintains invariant Inv-RegEl2 as there an arbitrary value is allowed for this bit.

4.12.5 IGC Notification Request

When a guest is requesting an IGC notification request through a hypercall instruction, the hypervisor may send an SGI to a corresponding core of the target guests, in case a notification is not already pending and there is a powered up core to receive the notification. Otherwise the request is dropped.

The main components touched by this handler are the GIC distributor and the IGC message box. We need to show that invariants Inv-MsgBox, Inv-MMU, Inv-Hyp-Gcpy, Inv-Hyp-Gwrite, Inv-Hyp-Greq, and Inv-Hyp-Gmsg are preserved by the following hypervisor transitions.

SIGC-Drop This step only restores the guest in the program position after the HVC instruction without having any other effect. All invariants are maintained trivially.

SIGC-Lock The guest context is saved and the second-stage MMU is deactivated. Invariant Inv-MMU is preserved as in similar steps above. We prove the remaining invariants one by one.

- Inv-MsgBox By the definition of $grant_{g,g'}$ we know that the granted IGC request targets only guests g' for channels s such that $cpol(s) = (g, g')$. For the requested channel the message box bit is set to one, for all other bits the message box invariant is preserved inductively as they do not change.
- Inv-Hyp-Gcpy A write to the send-IGC register is requested, however such requests are benign to this invariant and is preserved trivially.
- Inv-Hyp-Gwrite Again the write to the GIC concerns a register which is not constrained by this invariant, therefore it is preserved trivially.

Inv-Hyp-Greq The write request targets the GIC distributor, thus the first part of this invariant holds directly by definition of the rule. The other ones hold vacuously for core c.

Inv-Hyp-Gmsg We need to preserve the seventh clause of this invariant. By definition, for the written value v and the targeted core t we have $\langle v[3:0]\rangle = 15$ and that $v[23:16] = 0^{7-t}10^t$. Therefore $v[16+t] = 1$ holds and by definition of $grant_{g,g'}$ we know that the targeted core t belongs to $\gamma(t) = g'$ such that there exists an s with $cpol(s) = (g, g')$, where $g = \gamma(c)$. The condition on the message box follows directly by construction of the rule as well.

SIGC-Finish The guest context is restored with empty input and outputhannels and the MMU is re-enabled, maintaining Inv-MMU. There is nothing else to prove.

4.12.6 IGC Notification Reception

The dual handler of IGC notification reception is quite similar to the interrupt injection handler. In fact its initial step is identical as before the read reply from the interrupt acknowledgment register is received, it is impossible to distinguish for the core, which asynchronous interrupt was received.

The remaining steps follow a similar control flow. The invariants touched are Inv-RegEl2, Inv-MsgBox, Inv-MMU, Inv-Hyp-Greq, Inv-Hyp-Gmsg, and Inv-Hyp-LastIGC.

RIGC-EOI The hypervisor receives the reply from the GIC, reading the acknowledgment register, and discovers the reception of an IGC notification, i.e., an SGI from a core of a different guest. It updates the last-IGC variable with the corresponding channel s and sender core c' with $\gamma(c') \neq \gamma(c)$. Moreover a write request is sent to the end-of-interrupt, requesting a priority drop for the corresponding SGI.

Only the following invariants are affected.

Inv-Hyp-Greq The write request that was issue is targeting the GIC CPU interface, therefore the first two clauses of this invariant hold by construction. The third and fourth one hold vacuously.

Inv-Hyp-Gmsg The eighth part of this invariant holds directly by construction.

Inv-Hyp-LastIGC In the transition the PC changes its value from a_{irqaw} to a_{rnew}. Moreover, by definition of $\sigma_{\text{RIGC}}^{await,g',c',g}$ it consumes the read reply from the acknowledgment register which contains a value v for which we have $\langle v[9:0] = 15$ from sender c' belonging to a different guest. Therefore by Inv-Hyp-AckIGC we know that in the prestate the message box for the corresponding channel is set to one and the corresponding SGI is active. The directly proves both parts

of Inv-Hyp-LastIGC for M'_R, since neither components are changed by the transition.

RIGC-Check The hypervisor reads the list register in the virtual interrupt control interface for the last received IGC interrupt. By construction, this read request maintains Inv-Hyp-Greq and the nineth part of Inv-Hyp-Gmsg. No other invariants are affected by this read request, in particular Inv-Hyp-LastIGC is maintained as neither the message box nor the GIC are touched directly.

RIGC-Inject Here the corresponding list register is written by the hypervisor. Again the request directly maintains Inv-Hyp-Greq and the tenth part of Inv-Hyp-Gmsg.

RIGC-Deact The hypervisor generates a write request to the GIC CPU register in order to deactivate the active SGI, directly maintaining Inv-Hyp-Greq and the eleventh part of Inv-Hyp-Gmsg. Invariant Inv-Hyp-LastIGC is again preserved as the message box and the GIC are not touched, yet. In particular the request to deactivate the SGI has not reached the GIC yet, therefore the second part is preserved.

RIGC-Unlock The guest context is restored and the second-stage MMU reactivated, maintaining Inv-MMU. Moreover the *VI* flag is set in the hypervisor control register, preserving Inv-RegEl2 as in the IRQ injection handler. Finally the message box bit for the channel is cleared, preserving Inv-MsgBox trivially, as only setting a message box bit could break the invariant.

4.12.7 MMU Fault Handler

The MMU fault handler injects a data or instruction abort into the guest, by jumping to its corresponding handler in EL1. As only the guest state is affected and no new memory requests are generated, transition MMUF-Virt preserves all implementation invariants trivially. □

4.13 Simulation Invariants

Besides the implementation invariants that hold for all states in computations of the refined model that are starting in a valid initial state, there are two additional sets of invariants that hold during the bisimulation proof on the simulating model, due to the way the simulation is conducted. That means, that when proving the simulation of the refined by the ideal model, we restrict the set of ideal traces we consider for that simulation. In the same way we restrict the set of refined model traces to consider during the simulation of the ideal model.

Note that this restriction is just needed to conduct the proof in an inductive fashion. The actual bisimulation theorem is still holding as stated in Theorem 1,

i.e., for all computations of the refined model there exists a corresponding ideal computation, and vice versa. Moreover the simulation invariants of each model are only used in one direction of the proof, making sure that all computations of the simulated model are covered.

As explained earlier, when proving Lemma 4, it has also to be shown that the simulation invariants for the corresponding simulating model are maintained. We define the simulation invariants for the refined and ideal model below.

4.13.1 Refined Model

The simulation invariant for the refined model is denoted by the predicate $sim_R : \mathcal{M}_R \to \mathbb{B}$. It must hold in all states of computations that simulate a given ideal model computation. We require the following constraints.

SIMR-GLOCK After the global hypervisor initialization, the GIC lock is always free, i.e., we will step the refined system in the simulation of the ideal model in a way that all acquisitions of the lock are always followed by a release of the lock in the same block of steps.

$$M_R.hvinit \;\Rightarrow\; Glock(M_R.m(a_{glk})) = \bot$$

SIMR-IRQPEND In the refined model no physical interrupts are ever pending, active, or active-and-pending in the CPU interface of a core. This means when simulating the ideal model we will deliver peripheral interrupts and SGIs only directly before injecting them as virtual interrupts into the receiving core. This simplifies the proof as peripheral interrupts are always in sync between both models and when injecting virtual interrupts, the interrupts present in the CPU interfaces do not need to be constrained by the bisimulation relation.

$$\forall c \in \mathbb{N}_{nc}.\ M_R.Q_c = \emptyset$$

Proof of Lemma 3 for sim_R. We directly have $sim_R(M_R^0)$ from the definition of $start(M_R^0)$ as $M_R^0.hvinit$ does not hold initially and no interrupts are outstanding in the CPU interface. □

4.13.2 Ideal Model

The simulation invariant for the ideal model is denoted by the predicate $sim_I : \mathcal{M}_I \to \mathbb{B}$. It must hold in all states of computations that simulate a given refined model computation. We require the following constraints.

SIMR-GLOCK The input and output channels of the ideal GICs in all guests are always empty. We achieve this by stepping the ideal model in a way such that ideal GIC I/O requests are always handled in blocks where a request first enters the GICs input channel, is then consumed and processed by

the GIC, and moved out of its output channel immediately. Thus the GIC is always available to receive I/O requests in the ideal model.

$$\forall g \in \mathbb{N}_{ng}.\ M_I.G(g).m2p(np_g) = M_I.G(g).pif(np_g) = \text{None}$$

Proof of Lemma 3 for sim_I. We directly have $sim_I(M_R^0)$ from the definition of $start(M_R^0)$ as all message channels are empty initially. □

A List of Abbreviations and Correspondences

Below we list the abbreviations used in this document and the corresponding HOL4 artifacts (annotated with their types) in the formal development.

$B \xrightarrow{a}_{BUF} B'$	`bif_step(`B`,`a`,`B'`) : bool`
$C \xrightarrow{a}_{\mathcal{C}_{id}} C'$	`idcore_step(`C`,`a`,`C'`) : bool`
$M \xrightarrow{a}_{\mathcal{M}_g} M'$	`mem_step(`M`,`a`,SOME `g`,`M'`) : bool`
$P \xrightarrow{a}_{PSCI_g} P'$	`idpsci_step(`P`,`a`,c,`P'`) : bool`
A_{GICC}	`GICC : AddressRegion`
A_{GICD}	`GICD : AddressRegion`
A_{GICH}	`GICH : AddressRegion`
A_{GICV}	`GICD : AddressRegion`
BufRcvReq	`bif_step_rcv_req_def`
BufRReq r	`RCV (MREQ` r `) : Action`
BufRcvRpl	`bif_step_rcv_rpl_def`
BufRRpl q	`RCV (SRPL` q `id) : Action`
BufSndReq	`bif_step_snd_req_def`
BufSReq r	`SEND (SREQ` r `id) : Action`
BufSndRpl	`bif_step_snd_rpl_def`
BufSRpl q	`SEND (MRPL` q `) : Action`
BufFault	`bif_step_snd_fault_def`
BufFault q	`SEND (MRPL` q `) : Action`
$\mathbb{A}_g$	`A_G` g `: bool[36] set`
$A_P^{g,p}$	`A_gp` g p `: bool[36] set`
as_g	— *skipped, replaced by* `A_G` g `: bool[36] set`
A_{GIC}	`gicAR : AddressRegion set`
$\mathcal{C}_{id}$	`:idcore`
$\mathcal{C}_{id}^{abs}$	`:idcore_config`
CMD	`:event`

$\mathcal{E}^{in}_{g,p}$	`:pevent` *(simplified, one external message type for all g,p)*
$\mathcal{E}^{out}_{g,p}$	`:pevent` *(simplified, one external message type for all g,p)*
$\mathcal{F}$	`:fault_info`
F r f	`Fault` r f `: reply`
$\mathcal{G}_{id}$	`:idgic`
$\mathbb{G}_g$	`:guest` *(simplified, one guest type for all)*
$IGCin_g$	`igcin_g (p:params)` g `: num set`
$IGCout_g$	`igcout_g (p:params)` g `: num set`
$IGCQ_g$	— *skipped, replaced by* `xPIRQ` g `: irqID set`
INTERNAL	`TAU : Action`
$Inv_{\mathcal{C}_{id}}$	`inv_good_idcore : idcore -> bool`
ior	`IOreq_rcvd : request -> senderID option`
$IPIRQ_g$	`ipirq_g (p:params)` g `: irqID set`
IRQ	`:irqID`
IRQ_g	`xguest_irq` g `: irqID set`
$\lceil m \rceil.m$	`mem_abs` m `: bool[36] -> page`
$\lceil m \rceil.QR$	`mem_rpl_rcvd` m `: (reply # senderID) set`
$\lceil m \rceil.RR$	`mem_req_rcvd` m `: (request # senderID) set`
$\lceil m \rceil.RS$	`mem_req_sent` m `: (request # senderID) set`
$\mathcal{M}$	`:memory`
$\mathcal{M}_g$	`:memory` *(simplified, one memory type for all purposes)*
MEMRREQ r id	`RCV (SREQ` r id`) : Action`
MEMRRPL q id	`RCV (SRPL` q id`) : Action`
MEMSREQ r id	`SEND (SREQ` r id`) : Action`
MEMSRPL q id	`SEND (MRPL` q id`) : Action`
MEMINTRNL id	`TAU : Action`
$\mathcal{P}_{g,p}$	`:peripheral` *(simplified, one peripheral type for all)*
pir q	`PIR` q `: irqID`

$PIRQ_g$	`pirq_g (params)` g `: irqID set`
PowIGC s c	`SEND (RCU cn` s c `) : Action`
PowRcv l	`RCV (PSCIL` l `) : Action`
PowSnd x	`SEND (PSCI` x `) : Action`
PSCI	`:psci_state`
psci	`PSCI : psci_state`
PTW a x	`Walk` a x `: request`
$\mathbb{Q}$	`:reply`
$\mathbb{R}$	`:request`
R a d x	`Read` a d x `: request`
RcvPhys	`RCV (PHYS NONE c) : Action`
RcvPow x	`SEND (PSCI` x `) : Action`
RcvRpl q	`RCV (MRPL` q `) : Action`
rplR r v	`ReadValue` r v `: reply`
rplPTW r v	`WalkResult` r v `: reply`
rplW r	`WrittenValue` r `: reply`
RS	`idcore_req_sent : idcore -> request set`
$\mathbb{S}_g$	`:senderID` *(simplified, one sender ID type for all guests)*
$\mathrm{sgi}^q_{c,d}$	`SGI` q c d `: irqID`
sIGC.pend	`sIGC.flags : num -> bool`
sIGC.tgt	`sIGC.addressbook : num -> num option`
SndIGC s	`SEND (SIGC` s `) : Action`
SndPow l	`SEND (PSCIL` l `) : Action`
SndReq r	`SEND (MREQ` r `) : Action`
start c	`StartCore` c `: event`
StartUp g c	`RCV (STARTUP` g c `) : Action`
stop c	`StopCore` c `: event`
W a d v x	`Write` a d v x `: request`
$\mathcal{X}$	`:msginfo`