Towards Correct-by-Construction Interrupt Routing on Real Hardware

Lukas Humbel, Reto Achermann, David Cock, Timothy Roscoe
Systems Group, Dept. of Computer Science, ETH Zurich

Abstract

In this paper we address the problem of correctly configuring interrupts. The interrupt subsystem of a computer is increasingly complex: a zoo of different controllers with varying constraints and capabilities form a network with limited connectivity. An OS which aspires to provable correctness must manage a limited set of interrupt vectors, delegate interrupts to device drivers and configure the controllers correctly. No well-specified approach exists.

As a foundation for applying language-level techniques like program sketching and synthesis to this problem, we present a formal model for interrupt routing which can capture all the system topologies and interrupt controllers we have encountered in the wild, show applications of such a model not possible with informal, ad-hoc approaches like DeviceTrees, and finally discuss an implementation based on the model which forms the new interrupt subsystem of the Barrelfish OS.

CCS Concepts • **Software and its engineering → Operating systems**; • **Theory of computation →** *Constraint and logic programming*;

Keywords Hardware configuration, Hardware abstraction, Interrupt routing, Eclipse/CLP

1 Introduction

We report on work to solve the problem of correctly (and provably so) configuring interrupt routing across a range of increasing diverse and complex hardware platforms. We present a formal model which can represent the complete interrupt topology (sources, vectors, links, controllers and cores) of real computer systems from a PC to a phone System-on-Chip (SoC) and capture the constraints on interrupt routing imposed by real-world hardware components.

PLOS 2017, October 28, 2017, Shanghai, China
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5153-9/17/10.
https://doi.org/10.1145/3144555.3144557

We also show how to obtain properties of a given instance of the model, such as "Can every interrupt source be uniquely distinguished at its destination?", and to configure interrupt hardware to correctly route and deliver interrupts in the system based on operating system (OS) requirements. We also describe a concrete implementation based on the formalism which configures interrupt controllers on demand in the Barrelfish OS by realizing the model in Prolog.

The problem of correctly configuring the interrupt subsystem is surprisingly complex. As we show in Section 2, a modern computer includes many cores as potential destinations of interrupts and a complex network of interrupt controllers routing interrupt signals – it is not unusual for a signal to traverse more than 5 translation units before delivery to software. There also exists a wide variety of such controllers (we describe a representative set of 15 different ones), and support for virtualization adds further levels of complexity.

The problem is also important: a modern general-purpose OS has to handle the full complexity of a system like this, and the topology is generally not known when the OS is written. Correct operation of the system on a new piece of hardware depends on correct configuration of this network by software, and correct *reconfiguration* as device driver threads migrate and hardware is hot-plugged.

Moreover, the problem is not going away (and is therefore not amenable to a one-time hard-wired solution): new interrupt controllers are appearing all the time, and new SoC designs present new combinations of devices and heterogeneous cores with new constraints on which interrupts can be delivered where. Furthermore, formally *proving* the correctness on a system with interrupts must rest on a formal model of the underlying hardware.

Formally modeling interrupts also has value beyond system software design, since it can shed light on desirable properties of hardware designs (both complete platforms and individual controllers) as well.

This paper builds on our previous work, a formal model in Isabelle/HOL, on modeling memory and interrupt systems [1]. Our contributions over that paper are as follows: In section 3 we extend the model to capture constraints of real interrupt controllers, discuss how we have represented all the controllers we have seen to date in our systems, and state useful properties of any given system that can be determined from the model. In section 4 we describe a practical application: We have implemented the model in Prolog and C and we instantiate it online to configure interrupt routing in the Barrelfish OS. We show how it can be used online to derive valid configurations for interrupt hardware.

In the next section, we further motivate the problem and delve into the complexity of modern interrupt systems.

2 Background

Modern interrupt hardware is complex. Whereas in the distant past, an interrupt was a dedicated electrical signal to the processor, today a computer has a network of interrupt controllers which can be configured to deliver many distinct interrupts generated by a given device to different vectors on different cores.

Table 1 shows 15 different interrupt controllers used by machines in our server room. New interrupt controllers are introduced all the time, whether evolutions of existing designs or new, specialized functions for particular SoCs. Each has different capabilities and constraints on the number of interrupt signals they can source and sink, and how they can map between them. For instance, the venerable Intel 8259A PIC [9] has a fixed mapping of 8 input ports to a single output port and 8 bit vector. The Local APIC [11] maps interrupt messages on a bus to a corresponding local core vector. Intel IOAPIC controllers [10, 12] convert events directly from PCI functions or through PCI Link Devices [17] to APIC messages in a particular delivery mode, but behave differently when combined with an IOMMU [14], which can translate memory writes from devices to message-signaled interrupts [17]. The ARM GICv2 [3] supports 1024 different interrupts but not all can be delivered to all cores, and vectors cannot be changed. The compatible CoreLink GIC-400 [2] adds additional constraints on its reconfigurability, the GICv3 exists in two variants [4] with implementation-defined limits on vector size and GICv4 adds virtualization support [4]. Additionally, the ARM GIC are programmed using a memory mapped register and/or CPU interface.

These controllers are connected in a non-trivial platform-specific network. Figure 1 shows a simplified PC-based illustration. Interrupts may be delivered to a single core, a set (1-N) or broadcast. Virtualization allows interrupt delivery directly to a virtual machine [13, 14].

The OS must discover and correctly configure this network dynamically. Some topology data can be obtained from PCI discovery [17] and ACPI [23], but it is incomplete or may not exist at all. DeviceTree [8] files are used by many OSes to work around this, but the file format has no clear semantics, is error-prone [19], and despite containing controller information [18] fails to capture configuration constraints or

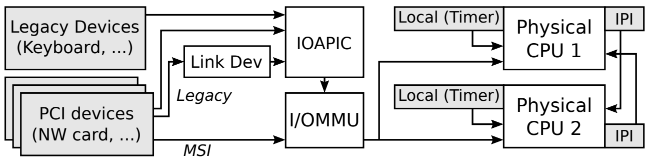

Figure 1. Simplified x86 interrupt network

cover inter-processor interrupts. Even so, the proliferation of DeviceTrees shows that configuration is a problem.

After discovery, correct configuration of a modern computer is essentially a network routing problem with highly constrained switches, but current OS designs reflect a legacy of much simpler hardware.

Linux, for example, defines a single namespace of "IRQ numbers" for all interrupts, and then attempts to map this to a strict hierarchy of interrupt controllers. "IRQ Domains" [15] map Linux IRQ numbers to hardware sources and implicitly hard-codes the topology. Device drivers are responsible for identifying the controllers they need to program (via a driver interface) to deliver interrupts correctly. The common case is to deliver an interrupt to all cores, and vector numbers are assumed to be the same across all cores. Constraints in interrupt routing are not well handled and generally special-cased in the code.

Chen *etal.* [7] verify an interruptible operating system kernel including a simple verified interrupt controller driver. The focus of our work is on the topology of the interrupt system, we are interested in properties of the configuration and ensure, for instance, that the correct controller is configured.

Stepping back, a better approach is to define a formal model which captures the complexity of modern interrupt subsystems and provides both a basis for verifying implementations and a template for engineering a correct solution which works across a wide variety of platforms. This paper describes early work in this direction: both a preliminary model and an implementation.

3 Model

We base our model on prior work [1] about formally specifying memory accesses and interrupts and extend it to enable interrupt controller configuration. Currently, the model is implemented informally in Prolog, described in section 4. We present the extensions necessary to provide a formal basis for that implementation.

We express the topology of a system as a *decoding net*, a directed graph consisting of nodes with two properties: *i)* a set of *accepted* addresses *ii)* a set of *translated* addresses that map onto another node, where addresses here represent interrupt ports. Address resolution starts at a particular node and address and terminates if a node accepts the input address or it is not in the set of translated addresses.

3.1 Model refinement

The nodes are a set of interrupt sources (e.g. devices), a set of destinations (e.g. an interrupt vector on a core) and a set of interrupt controllers. We refer to addresses on nodes in the decoding net as *ports*. We assign a globally unique identifier to all ports. We further extend the model in [1] with the refinements below and summarize the extensions in Figure 2.

Controller	In Port #	In Vector Size	Out Port #	Out Vector Size	Constraints
PIC	$n \cdot 8 + (8 - n)$	0 bit	1	8bit	fixed
I/OAPIC	24	0 bit	16	8 bit	None
I/OxAPIC	24	0 bit	256	8 bit	None
I/Ox2APIC	24	0 bit	2^{32}	8 bit	None
I/Ox2APIC + I/OMMU	24	0 bit	2^{16}	0 bit	None
LAPIC LVT	7	0 bit	1	8 bit	None
PCI Link Device	4	0 bit	4	0 bit	None
MSI Link Device	2^{0-4}	0 bit	2^{32}	32 bit	Same port, contiguous addresses
MSIx Link Device	64 − 2048	0 bit	2^{32}	32 bit	None
IRTE Mapper	2^{20}	32 bit	1	16bit	fixed
IPI RT	?	0 bit	16	8 bit	None
I/OMMU	1	16 bit	2^{32}	8 bit	None
ARM GICv2 Dist	987	0 bit	8	10 bit INTID	out port == in port
ARM GICv3 ITS	2^{32}	32 bit	1	10 bit INTID + 16 bit ICID	unique INTID outputs
ARM GICv3 CT	1	10 bit INTID + 16 bit ICID	2^{32}	10 bit INTID	INTID must match input INTID

Table 1. Characteristics of interrupt controllers showing input and output port numbers and vector sizes.

Definition (Port Set). $\mathbb{P} \subset \mathbb{N}$
Definition (Interrupt Format).

$$\mathbb{I} = \{Empty, Vector, Mem\}$$

Where
- $Empty = \{\}$ is an interrupt with no associated data.
- $Vector \subset \mathbb{N}$ is the set of interrupts that can be described using a single interrupt vector number.
- $Mem \subset \mathbb{N} \times \mathbb{N}$ the set of memory write operations represented as address-data word tuples.

Definition (Mapping Function). Partially defined function from an input to an output format-port tuple.

$$\mathbb{F} :: \mathbb{I} \times \mathbb{P} \rightharpoonup \mathbb{I} \times \mathbb{P}$$

Definition (Controller).

$$C = (inPorts, outPorts, mapValid) \in \mathbb{P} \times \mathbb{P} \times 2^{\mathbb{F}} = \mathbb{C}$$

Where $2^{\mathbb{F}}$ denotes all possible mapping functions and $mapValid \subseteq 2^{\mathbb{F}}$ are the valid mapping functions. $\mathbb{C}$ is the set of controllers.

Definition (Configuration).

$$Conf :: \mathbb{C} \rightharpoonup \mathbb{F}$$

A configuration is valid if $\forall C.Conf(C) \in C.mapValid$.

Definition (System).

$$\mathcal{S} = (inPorts, outPorts, ctrls) \in (\mathbb{P} \times \mathbb{P} \times \mathbb{C})$$

Where *inPorts* and *outPorts* are the sets of incoming and outgoing ports respectively *ctrls* is a set of controllers.

Figure 2. Interrupt Model Definition

Ports and Vectors: In a plain decoding net, a node only knows about addresses. Interrupt controllers, in contrast, can distinguish between source (i.e. *port*) and actual data (i.e. *vector*) transferred. Since ports and vectors are both of finite domain, we could define a mapping of (port, vector) pairs to a single numeric range through enumeration. However, since the separation into *ports* and *vectors* naturally reflects the exposed programming interface to interrupt controllers, we keep them separate in the refined model.

This does lead to redundancy in the model and a potential choice in how to split the representation of a given controller between ports and vectors. We use the following rule of thumb: Multiple output ports should be used if the controller can direct interrupts to multiple destinations. Similarly, multiple input ports should be used if the controller can distinguish between different interrupt sources. Vectors should be used to distinguish between input events from the same source, or outputs to the same destination.

Interrupt formats: The controllers in Table 1 use three different interrupt formats: *i)* a *plain signal* asserting an occurred event (e.g. device interrupt), *ii) plain signal + vector* providing a word of information about the event (e.g. CPUs receive an interrupt vector) and *iii)* a *memory write* of a data word to a specific address (as in MSI-X). Therefore, the node's translate function must be able to differentiate and convert between different interrupt formats.

Configurability: One of our goals is to *correctly* configure interrupt controllers, thus we need to capture the set of possible configurations of a controller. We add a third property, the set of *valid translate functions*, to the decoding net nodes that expresses supported interrupt formats, data words, and transformations.

We further define an *interrupt system* as a tuple of incoming ports, outgoing ports and controllers, i.e. a complete decoding net. To express the current state of the system, each controller is assigned a *configuration*. We say that the configuration is *valid* if each controller is assigned a valid translation function.

Note the model allows two controllers that produce/consume different interrupt formats to be linked. For consistency, we interpret this as stating that the controllers cannot receive messages from each other.

3.2 Representing interrupt controllers

We have expressed all the interrupt controllers in Table 1. Note there is no unique representation of an interrupt system: identifiers of ports and vectors can be changed while

IOAPIC

$C_{IOAPIC} = (inPorts, toCPUs, apV)$ with $|inPorts| = 24, \quad f \in apV \leftrightarrow \forall port : f(\_, port).int \in \{32..255\}$

Intel IOMMU

$C_{IRTEMAP} = (inPorts, IRTE, irV)$ with $|inPorts| = 1, \quad irV = \{f(mem(addr, data), port) \rightarrow \{addr + data, \_\}\}$

$C_{IRTE} = (IRTE, toCPUs, 2^{\mathbb{F}})$ with $|IRTE| = 1$

MSI

$C_{MSI} = (inPorts, memWrite, msiV)$ with $inPorts = [0, 32], \ base_{addr} = \text{constant}, \ base_{data} = \text{constant}$
$f \in msiV \leftrightarrow f(in, port) \rightarrow \{mem(base_{addr}, base_{data} + port), \_\}$

Figure 3. Real interrupt controllers expressed in model

preserving the controller's semantics and even splitting and merging of controllers is possible – it is theoretically possible to express an entire interrupt system using a single controller and a complex predicate for valid mappings. Practical considerations of modularity, reuse, and readability determine a "good" representation; we give three examples (Figure 3).

On the **Intel IOAPIC** all of 24 input ports are directly connected to an interrupt source. The interrupt format is a plain signal. Each port can be configured independently, and interrupts are always delivered on the APIC bus with a vector in the range [32, 255]. We model it with 24 input ports because of the direct input connections and provide a valid mapping function to constrain the possible emitted vectors.

Devices supporting **Message Signaled Interrupts** (MSI) can trigger up to 32 different interrupts. We model such sources as interrupt controllers themselves since they determine the delivery destination of interrupts. As with the IOAPIC, we express the device as a controller with 32 plain-signal input ports that generate consecutive memory writes. These writes (the MSIs themselves) impose dependencies between different "ports" on the device, and so we need to carefully constrain the set of valid configurations for a MSI-capable device.

As a final example, the **Intel IOMMU** translates all MSI memory writes into an index into a single table, using a non-injective function. While we could represent this constraint logically in the model, it is simpler and more elegant to split the IOMMU into two imaginary controllers: a fixed function called IRTEMAP which maps the MSI into an integer on a single output port, and the remapping table called IRTE with multiple output ports that captures the routing functionality. The IOMMU cannot perform a different routing decision based on the source, therefore we use one input port.

We have found this trick of splitting a controller into a fixed-mapping controller and a freely configurable one which as a pair preserve the original semantics to be useful and quite widely applicable: it pushes complexity out of the constraints and into the model, and as a side effect simplifies implementing the controller drivers themselves.

However, not all controllers can be split. Mapping constraints that depend on the configuration of other ports, such as in the case of the MSI controller, can not be split.

3.3 Useful properties

Once we have a model that can capture both the topology of a real machine's interrupt subsystem and the functionality of its programmable interrupt controllers, we can start to formulate useful properties of a given system that can be proved (or disproved) from the model representation.

A first case is **Reachability**. It is particularly the case with SoCs that a given interrupt cannot be delivered to any processor in the system. Using the model, we can derive the reachability matrix for interrupts in a given system. Furthermore, since our model also integrates configuration, we can also answer a slightly more challenging question: *Given a set of interrupt source-destination pairs, is it possible to find a configuration that connects all of them?*

For each source-destination pair, there exist multiple controller configurations that connect these two parts and devices can use different signaling mechanisms, which in turn may result in multiple distinct routes through the controller network. As long as controllers can distinguish incoming interrupts, the intermediate representation does not matter (e.g. IOMMU + MSI). Using the model, we can enumerate all possible configurations.

A second useful property of a system is **reliable delivery**: under what conditions can be guaranteed that every interrupt will arrive at the appropriate destination. This may be required to prove the liveness of the system, but even if not (where interrupts are a "hint" to improve the performance of a polling model), failure to deliver interrupts can create hard-to-diagnose performance degradation.

Note that this subsumes the problem of verifying whether a given configuration of the system is "correct" but is stricter: it includes the idea that at no point during a *reconfiguration* of the interrupt system will it enter a state where interrupts can be lost or misrouted. Given a system representation in our model we can verify the correct delivery of each interrupt for any given configuration of interrupt routers.

A less serious but dual problem is spurious interrupts. Our model can be used to constrain the set of possible causes of a spurious interrupt received at a given core, as long as our model of each controller is sufficiently faithful.

As a final example, **distinguishing interrupts** is an important requirement for an OS so that it can invoke the appropriate device driver. This should be trivial (each distinct interrupt should arrive on a different vector on a given core), actually *proving* that it is the case is not, and has some similarities with a network capacity problem. A controller is able to distinguish up to $N = \#ports \times \#vectors$ different interrupts where each ports-vector tuple identifies an entry in the controller's routing table. If there are more interrupt sources than the smallest interrupt controller can distinguish, interrupt *sharing* (two distinct sources trigger the same destination) may occur eventually – when using a sub-optimal configuration heuristic even before all port-vector tuples have been allocated. The model can identify which interrupts are shared and where. Thanks to the inclusion of configuration options we can pick a minimal sharing configuration. Currently used heuristics fail to do so on complex machines.

4 Implementation

We used our model to entirely replace the existing interrupt subsystem of the Barrelfish OS [5]. In Barrelfish, our work is made easier by the System Knowledge Base (SKB) [20] – a Prolog engine and constraint solver – which holds the state of the model as a set of Prolog facts and predicates, and implements the routing algorithm.

At time of writing, the implementation successfully configures device interrupts on demand on all real and virtual hardware used by the Barrelfish development team, including a variety of x86 and ARMv8-based server machines and ARMv7-A development boards.

While the current implementation is based on the formal model represented in logic programming, it does not provide the assurance of a fully-verified implementation, and the low-level hardware access for discovery and register programming is hand-coded in C and Barrelfish's (non-verified) Mackerel domain-specific language for hardware [22].

Nevertheless, the implementation is functional, demonstrates the viability of the approach, and has greatly simplified and unified device programming across diverse platforms in Barrelfish. Moreover, it is clear that a different (perhaps more complex) implementation is possible in C for monolithic kernel systems like Linux.

Our implementation consists of 179 lines of Prolog for the generic model. As an example, 185 additional lines of Prolog implement the x86 specific part; the bulk of the latter dealing with populating the model with topology information discovered from ACPI and the (user-space) PCIe driver. For a given interrupt controller, the constraints on routing it imposes can usually be expressed in a single line. The high-level architecture is shown in Figure 4.

4.1 The routing service

The *Interrupt Routing Service* (IRS) is implemented inside the SKB as a set of inference rules (analogous in this case to stored procedures in a relational database) and executes the routing algorithm incrementally over the SKB's representation of the interrupt topology and current configuration.

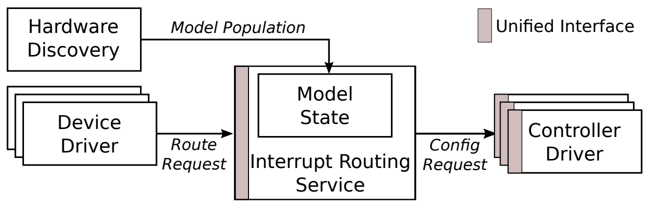

Figure 4. System Architecture

The output of the algorithm is a set of (re)configurations for specific interrupt controllers, which are then programmed by their respective driver processes.

Barrelfish's Multikernel [6] architecture, as in a microkernel, implements most drivers in user space, including most of the interrupt controller drivers. The IRS model encodes the routing constraints specific to particular interrupt controller types, but the interface to an interrupt controller used by the IRS can be entirely generic, simplifying implementation.

The model contains all information necessary to route interrupts. A routing request consists of an interrupt source and an interrupt destination. For simplicity, we assume the interrupt destination is provided by the requester. Often, the interrupt destination has some freedom: It is important on which CPU the interrupt ends up, but the exact vector triggered is not important.

The routing algorithm determines a valid configuration for each controller such that all the existing routes plus the new request are satisfied. A natural, though inefficient, approach in Prolog is a back-tracking, depth-first search. The entering interrupt to be routed is followed to its first controller, the first configuration that does not discard the interrupt is considered, tracing the interrupt to the next controller and repeating it until a destination is found. If the interrupt destination does not match, or it discards one of the existing routes, we backtrack. In practice, we can improve this by only picking output ports that get us closer to the desired destination, and failing (or resorting to interrupt sharing) as soon as we encounter a link where the new interrupt doesn't "fit" in the identifier space. This latter event is rare, and only occurs in highly resource-constrained systems.

So far, solving time has had a negligible impact on system performance, even on complex multisocket platforms.

4.2 Topology discovery

Various sources of hardware discovery populate the model for a given machine. Existing drivers for ACPI and PCIe in Barrelfish were simple to modify for this purpose, since they already entered discovered information in the SKB, indeed, in some cases a single Prolog rule provided an appropriate "view" over existing information.

Ideally, PCIe, ACPI, etc. would allow the OS to discover the entire interrupt topology online. However, many platforms (in particular, ARMv7-A SoCs), completely lack a discovery mechanism for devices and interrupts. In other cases, discovery is incomplete for one or more reasons (such as devices which are not on a discoverable bus). For non-discoverable interrupt information we fall back to static information in compiled Prolog files provided in the startup RAM disk. The OS loads at boot the relevant predicates based on what system (and core) it is booting on.

Finally, even the information gained from a discovery mechanism is usually insufficient to instantiate the model, even if all the controllers are discovered. The topology itself is often represented only implicitly, such as through the hierarchy of the PCI bus. Often, certain links or translations are missing (for example, in ACPI, it is not discoverable how MSI interrupts are translated to CPU vectors). For this information, we also fall back knowledge the system programmer has extracted from datasheets (or, conceivably, DeviceTree files) and coded into the configuration algorithm or a supplemental Prolog file. This information is crucial for any operating system. Our approach explicitly exposes all translation units, while in commodity systems, this knowledge is implicitly contained in program code. Note that the topology and set of controllers in the system can be entirely dynamic.

4.3 Clients

Clients of the IRS are device drivers which wish to receive interrupts from the devices they manage. A full description of the Barrelfish driver protocol (including the authorization framework for interrupts) is beyond the scope of this paper, but can be briefly summarized as follows.

When a device driver is started by the Barrelfish device manager, it receives *capabilities* for resources it needs to access the device. This includes memory-mapped I/O register areas but also capabilities granting the right to receive interrupt notifications from a specific interrupt source. The driver creates a communication endpoint (also represented by a capability) and hands this together with the interrupt source capability to the IRS. Capabilities are also used to grant access to specific vectors in interrupt controllers (up to and including interrupt delivery vectors on destination cores). This allows more decentralized implementation in the future, but crucially isolates the interrupt resources of a device and its driver from others in the system.

4.4 Discussion

Our implementation was driven both by the formal model and the particular architecture and facilities of Barrelfish. However, the separation of mechanism from policy (routing) that results in, we claim, an elegant solution and we see no strong reason why the techniques are not equally applicable to monolithic systems like Linux or microkernels like seL4.

The approach allows high-level language techniques (like inference and constraint solving) to be applied to low-level concerns (interrupts), with a consequent simplification both of individual drivers for peripheral device and interrupt controllers, and also the core of the OS as a whole. A further benefit is that generic interfaces to interrupt controllers and IRS do not end up in contradiction with the behavioral quirks of specific components.

5 Ongoing work and conclusion

Two major aspects of interrupts are not yet fully captured by our model. The first is that interrupts are currently unicast: we configure interrupt controllers to forward an interrupt to *one* destination, rather than multicasting (or broadcasting) the interrupt to many destinations. This is well-suited to the Barrelfish architecture, but less so for a monolithic system like Linux, and in any system is valuable for, e.g., optimizing TLB shootdowns within a shared physical address space. Extending the model to support multicast is straightforward.

Secondly, we do not address dynamic aspects of interrupt delivery, such as how interrupts are acknowledged, and the distinction between edge- and level-triggered interrupts. While rare these days, level-triggered interrupts have important use-cases, and how to capture the distinction is a topic of ongoing work.

Nevertheless, we have devised a model of interrupt delivery and formulated it, together which descriptions real hardware platforms and components in Prolog and demonstrated its practicality in a real OS. In previous work we have shown how a similar model can be formalized in Isabelle/HOL [16], and we plan on fully formalizing the propsed model.

To avoid manual translation between the dual Prolog and formal representation, we plan to extend a concrete syntax and compiler written to express complex memory subsystems [21] to include interrupt topologies and controller configurations. Then have this language generate both Prolog facts for runtime use and a formal representation for offline reasoning.

A practical extension would be to compile a DeviceTree file into this syntax widen our device support. However, we have found that the lack of clear DeviceTree semantics still requires a manual (human) step in the translation process to a formal specification.

Our programming code works on static snapshots of the interrupt subsystem, but does not address how to get from one configuration to a new one. We are exploring *program synthesis* techniques to generate a series of atomic reconfiguration operations that can, by construction, reconfigure the interrupt subsystem so that at no point does it pass through a "bad" state (e.g. where interrupts are lost or misdelivered).

We are also exploring program synthesis for programming individual interrupt controllers. In particular, by expressing the hardware registers and their meanings in the form of a *program sketch*, we can use synthesis techniques to generate correct register operations on each device.

Our work is at an early stage, but our experience both with the formal and implementation aspects suggests that we have a solid foundation for our ongoing work, with the long-term goal of generating correct and efficient OS code for an increasingly complex hardware landscape.

References

[1] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2017. Formalizing Memory Accesses and Interrupts. In *2nd Workshop on Models for Formal Analysis of Real Systems (MARS 2017)*. Electronic Proceedings in Theoretical Computer Science, Uppsala, Sweden, 66–117. https://doi.org/10.4204/EPTCS.244.4

[2] ARM Ltd. 2011. *CoreLink GIC-400 Generic Interrupt Controller - Technical Reference Manual* (revision r0p0 ed.). ARM.

[3] ARM Ltd. 2016. *ARM Generic Interrupt Controller - Architecture version 2.0* (issue b ed.). ARM.

[4] ARM Ltd. 2016. *ARM Generic Interrupt Controller Architecture Specification - GIC architecture version 3.0 and version 4.0* (issue c ed.). ARM.

[5] Barrelfish team. 2017. The Barrelfish Research Operating System. (August 2017). www.barrelfish.org.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, Big Sky, Montana, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[7] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. 2016. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, Santa Barbara, CA, USA, 431–447. https://doi.org/10.1145/2908080.2908101

[8] Devicetree.org. 2016. *Devicetree Specification* (release 0.1 ed.). Linaro, Ltd. http://www.devicetree.org/specifications-pdf.

[9] Intel Corporation. 1988. *8259A - Programmable Interrupt Controller*. Intel Corporation. Order Number: 231468-003.

[10] Intel Corporation. 1996. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC)*. Intel Corporation. Order Number: 290566-001.

[11] Intel Corporation. 1997. *MultiProcessor Specification* (revision 006 ed.). Intel Corporation.

[12] Intel Corporation. 2014. *Intel 64 Architecture x2APIC Specification*. Intel Corporation. Reference Number: 318148-004.

[13] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual* (volume 3, systems programming guide ed.). Intel Corporation.

[14] Intel Corporation. 2016. *Intel Virtualization Technology for Directed I/O - Architecture Specification* (revision 2.4 ed.). Intel Corporation.

[15] Grant Likely, Linus Walleij, Jiang Liu, Jianyu Zhan, Marc Zyngier, Kevin Cernekee, Xishi Qiu, and Mark Brown. 2016. *irq_domain interrupt number mapping library*. The Linux Foundation. https://www.kernel.org/doc/Documentation/IRQ-domain.txt.

[16] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. 2017. Isabelle / HOL Proof Assistant. (August 2017). http://isabelle.in.tum.de.

[17] PCI Special Interest Group. 2004. *PCI Local Bus Specification Revision 3.0* (revision 2.3 ed.). PCI Special Interest Group.

[18] Thierry Reding, Rob Herring, Grant Likely, and Bjorn Helgaas. 2014. *Specifying interrupt information for devices*. Kernel.org. https://www.kernel.org/doc/Documentation/devicetree/bindings/interrupt-controller/interrupts.txt.

[19] Mark Rutland. 2013. Device Tree - The Disaster So Far. Online. (2013). ELC Europe. http://elinux.org/images/8/8e/Rutland-presentation_3.pdf.

[20] Adrian Schüpbach, Andrew Baumann, Timothy Roscoe, and Simon Peter. 2011. A Declarative Language Approach to Device Configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, Newport Beach, California, USA, 119–132. https://doi.org/10.1145/1950365.1950382

[21] Daniel Schwyn. 2017. *Hardware Configuration with Dynamically-Queried Formal Models*. Master's thesis. Systems Group, ETH Zurich.

[22] Timothy Roscoe. 2013. *Barrelfish Technical Note 2 - Mackerel User Guide* (version 1.5 ed.). Barrelfish Project.

[23] UEFI Forum. 2017. *Advanced Configuration and Power Interface Specification* (version 6.2 ed.). UEFI Forum.