Unifying Timer and Interrupt Management for an ARM-RISC-V-Heterogeneous Multi-Core

Marcel Eckert
Helmut-Schmidt-University
Hamburg, Germany
marcel.eckert@hsu-hh.de

Jan Haase
Nordakademie
Elmshorn, Germany
jan.haase@nordakademie.de

Bernd Klauer
Helmut-Schmidt-University
Hamburg, Germany
bernd.klauer@hsu-hh.de

***Abstract*—A timer and interrupt management infrastructure is essential for a computer system executing a preemptive operating system. This holds for single- and also for multicore systems. When it comes to OS-capable ISA-heterogeneous multicore systems (systems that include cores of different ISAs, which all are capable of executing an operating system) the different timer and interrupt infrastructures of the different system ISAs are still applicable. However, if reconfigurability is added on top of the ISA heterogeneity, meaning that the composition (number of individual cores) is runtime adaptable, the timer and interrupt management infrastructure needs to be reconsidered. In this paper, a unified timer and interrupt management scheme is deduced for a runtime adaptable, OS-capable, ISA-heterogeneous multi-core system, consisting of ARM and RISC-V cores, that are capable of executing Linux. This is done by analyzing/surveying the timer and interrupt management schemes of RISC-V and ARM systems. Finally this paper will show that RISC-V timer and interrupt management devices are suitable to also manage timers and interrupts for ARM based systems.**

***Index Terms*—heterogeneous multicore, ARM, RISC-V, timer management, interrupt management**

I. Introduction

Heterogeneous Computing is a term to describe today's trend in computing architecture to mix different processing elements (therefore heterogeneous). In [1], Mittal gives a survey on heterogeneous architectures, including a terminology definition, as the term *heterogeneous* is somehow used ambiguously to express differences in the Instruction Set Architecture (ISA) being used and/or differences in the implemented micro-architecture of the cores of a multi-core processor. According to Mittal's terminology, *asymmetric single-ISA core multi-processors* categorizes architectures such as ARM's big.LITTLE architecture, where the cores differ not in the underlying ISA, but in the micro-architectural implementation.

Mittal's terminology also introduces the term *heterogeneous-ISA asymmetric multi-core processor*. However, this is still ambiguous. It is applicable to CPU-GPU systems (and also extends to CPU-FPGA systems), which are the most widely used architectures that use the term *heterogeneous*. But there is still another category of heterogeneous architectures, that differ fundamentally from CPU-GPU systems. These are called *OS-capable heterogeneous-ISA systems*. In a CPU-GPU system, only the CPU part is capable of executing an operating system, the GPU (or accelerators hosted inside FPGA areas) are usually not capable of executing an operating system on their own.

In a current project, we are going to construct such an *OS-capable heterogeneous-ISA system* by combining ARM and RISC-V cores into a joint architecture, as shown in Figure 1. On top of this architecture a full fledged operating system (Linux) shall be executed, which orchestrates the underlying (ISA-heterogeneous) cores. To carry this to the extremes, we further want to allow to change the composition of cores (number of ARM and RISC-V cores) by means of runtime reconfiguration. So in fact, a number of reconfigurable *core slots* is defined that can contain any core of the supported ISAs.

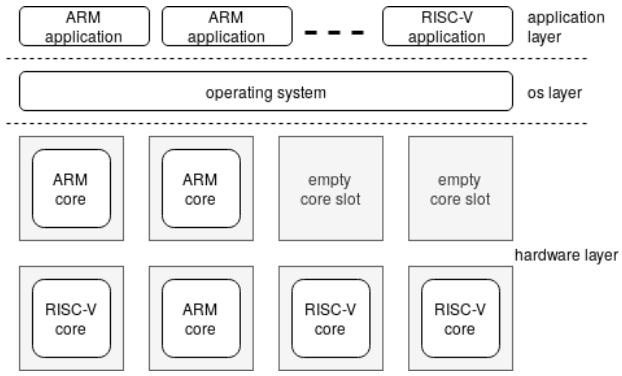

Fig. 1. Overall Project Idea.

This causes a lot of challenging problems to be solved and questions to be answered on the level of hardware architecture (e.g., how are the cores connected, as we need a coherent bus inbetween them) and operating system level (e.g., how to schedule on such heterogeneous architecture). To provide and discuss all of the resulting problems and questions is beyond the scope of this paper.

The focus of this paper is on harmonizing the interrupt and timer management schemes between the ISA-different cores. Timer and interrupt management is one of the fundamental aspects of enabling a core to execute an operating system like Linux.

The remainder of this paper is structured as follows: in Section II the state of the art in OS-capable heterogeneous systems is given. Section III gives a brief problem analysis

to point out the need for a interrupt and timer management for the proposed overall idea and also give a first discussion on possible solutions. As ARM and RISC-V cores are going to be used and need to be harmonized regarding the timer and interrupt management, the interrupt and timer management mechanisms used in pure ARM and RISC-V systems is presented in Sections IV and V. In Section VI, these information are used to deduce and discuss the chosen solution, which is proven to be usable in a prototype implementation in Section VII. Section VIII provides a discussion of the results. Finally, Section IX gives a conclusion.

II. State of the Art

In the already mentioned survey [1], Mittal classifies and lists several heterogeneous systems. He also lists some publications that are related to OS-capable ISA-heterogeneous systems. These references can be summarized to two dedicated systems, that address OS-capable ISA-heterogeneous ISA systems.

The first is Popcorn Linux. According to [2], The Popcorn Linux project is exploring how to improve the programmability of emerging heterogeneous hardware, in particular, those with Instruction Set Architecture (ISA)-diverse cores, from node-scale (e.g., Xeon/Xeon-Phi, ARM/x86, CPU/GPU/FPGAs) to rack-scale (e.g., Scale-out processors, Firebox, The Machine), in both native and virtualized settings. Popcorn uses a middleware approach to manage cores of different ISAs and present user applications a common system view, that might use several different ISAs. ISA-heterogeneity is supported by a Xeon-ARM Xgene combination in [3], [4] and a combination of X86-64 Servers and ARM-boards, connected via an ethernet network in [5]. As the different ISAs reside on separate boards and are not runtime exchangeable on a per core basis, the problem of a unified timer and interrupt management does not arise.

Another system is Venkat's system. Venkat's PhD Thesis [6] summarizes several publications related to this system. Venkat's system targets several operating system related questions for OS-capable ISA-heterogeneous systems. One of Venkat's main conclusions is, that ISA-heterogeneous systems can reasonably improve the overall system performance compared to single-ISA systems [7]. Evaluation is done with *gem5* based simulation models of ISA-heterogeneous systems (e.g., a system composed of x86-64, Alpha and ARM-Thumb cores). However, a validation of Venkat's results on a "real" hardware platform (not only in simulation) is still missing. Furthermore, as with Popcorn Linux, the static arrangement of the cores imposes no need for a unified timer and interrupt management among the cores.

III. Problem Analysis

When thinking from the user level application perspective, it might seem unclear why to discuss a common timer and interrupt management scheme for ISA-different cores. When thinking from the system level (the perspective of the operating system) it will shortly become clear why it is necessary to discuss timer and interrupt management:

Modern multi-tasking operating systems implement preemptive scheduling mechanisms. A program in execution (a process inside the operating system) is only getting a limited amount of time to execute (the so called time slice). After this time, the application is forced to relinquish control to the operating system. This preemption mechanism implies some preconditions on the underlying hardware (even for a single-core system). First of all, a *timer* is required to measure the execution time. Secondly, *interrupts* are required to have a mechanism to inform the executing core (and thereby the operating system) that the available time slice has elapsed. As third and last part, a privilege mechanism inside the hardware is required, to hinder an application to just turn off interrupt handling inside the executing core and thereby trick the operating system. The first two requirements are fundamental for preemption, the last one is required for security reasons.

As processor cores differ in their ISA, they also differ in how they provide and handle interrupts. Therefore it is not possible to apply an interrupt management (sub)system of one ISA to another. This is a problem for our proposed architecture, as each processor core slot can potentially contain different types of cores over time. What solutions may arise is summarized in Figure 2.

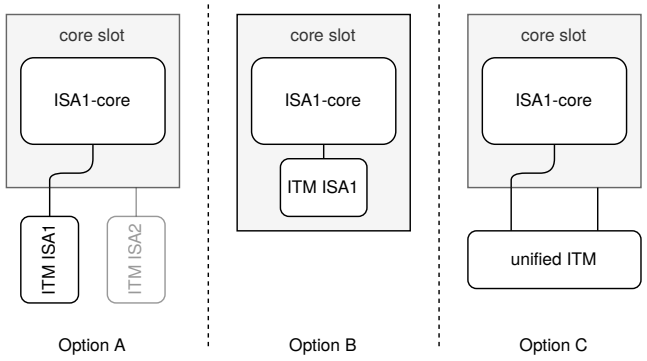

Fig. 2. Interrupt and Timer Management (ITM) Options.

A. Provide timer and interrupt management for each type of core for each core slot

This solution is the straightforward solution. The main disadvantage is the waste of area, as timer and interrupt management for one core slot is provided multiple times (in different forms) but only one is used at a distinct moment in time. This waste of area would be even worse, if more types of ISA different cores get involved. Therefore, this solution is not further considered useful.

B. Provide timer and interrupt management for each type of core inside the core slot

This solution would allow to further use the timer and interrupt management schemes used by the different ISAs, as the interrupt and timer management is tightly bound to the core

inside the reconfigurable core slot. This reduces the hardware overhead of Option A. The main challenge for this solution is the question if it is possible to scatter interrupt and timer management into the core slots.

C. Provide a unified timer and interrupt management

This solution requires an analysis whether or not a unified timer and interrupt management can be provided for all included core types. This unified management scheme can then be placed outside the core slots and be part of the static hardware.

As the latter two solutions seem feasible, a deeper analysis of the interrupt and timing management mechanisms of the intended core types (ARM and RISC-V) is necessary. This is done in the following Sections.

IV. Interrupt and Timer Management for ARM-cores

The first ARM-ISA processor was released by ARM (Advanced RISC Machines) in 1985. It was a 32-Bit ISA Single-Core processor. ARM cores starting with architecture variant armv3[1] defined the mechanisms interrupts are handled by all ARM cores till today [8]. This even holds for each of the cores inside ARM multicores.

A. ARM core Interrupt Interface

An ARM core has two interrupt lines, *FIQ* for fast interrupts and *IRQ* for general purpose interrupts. Furthermore, for both types of interrupts, an interrupt enable bit (IE-bit) exists inside the core status word that allows to prevent the core from reacting to an assertion of the corresponding interrupt line. Early ARM cores defined several *execution modes*, namely *user-*, *fiq-*, *irq-*, *abort-*, *undefined-*, *sytem-* and finally *supervisor-mode*. In effect this defines two privilege levels, the first is *unprivileged level* which corresponds to user-mode, the second is *privileged level* which summarizes all other modes. The different modes correspond to different exceptions that may arise inside the core. For interrupts, the interesting ones are fiq- and irq-mode. Both correspond to the according interrupt line (FIQ or IRQ). E.g., if the IRQ-enable bit is set and an interrupt occurs (IRQ-line is asserted), the core switches to irq-mode and disables the IRQ-enable bit in order to avoid responding to the same interrupt twice. The core then executes the instruction residing at the irq-exception address (the exact address depends on the architecture variant) which usually is a branch to the central interrupt handling routine of the operating system. When entering irq-mode, the FIQ-enable bit is still set, so FIQs have a higher priority than IRQs. If fiq-mode is entered, both interrupt enable bits for FIQ and IRQ are disabled.

As this paper is targeting Linux as operating system, side notes about ARM interrupts in Linux are relevant. Analysis of the ARM subtree (`arch/arm`) of the Linux kernel sources [9] reveals that most of the ARM systems running Linux are not using the FIQ mechanism of the cores. Furthermore, it can be observed that Linux mostly executes in supervisor-mode when running in privileged level. So all the other privileged modes (including irq-mode) are only used as shortly as possible.

For the ARM single cores, there was no definition of interrupt or timer management outside the core. This resulted in different timer and interrupt management device implementations by different vendors. This variety can still be found by investigating the corresponding timer- (subfolder `drivers/clocksource`) and interrupt- (subfolder `drivers/irqchip`) drivers inside the Linux kernel.

B. Interrupt and Timer Management

Starting with armv6 [8], ARM defined the *Vectored Interrupt Controller* [10], which is used in several ARM11 processors (single core processors). It provides the possibility to enter a device specific interrupt handler routine very fast, by setting up exception entry addresses on a per interrupt source line basis. However, this mechanism is not used by the corresponding Linux device driver (`drivers/irqchip/irq-vic.c`).

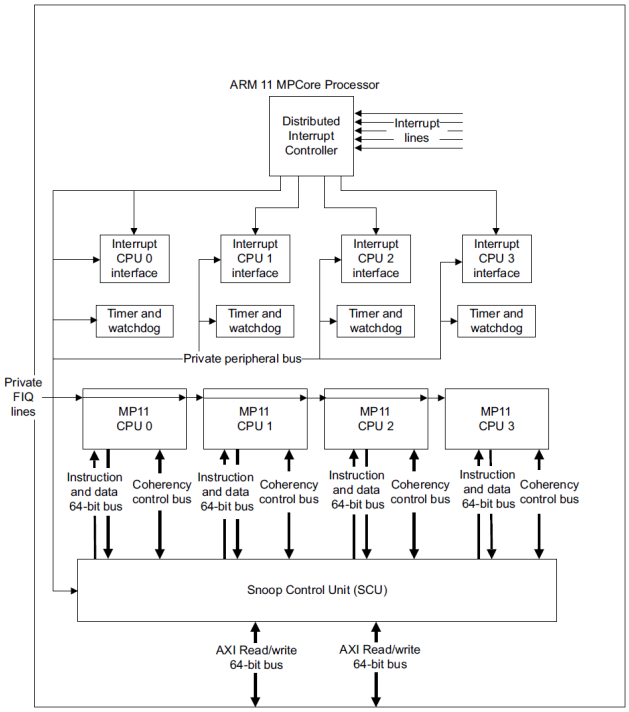

Fig. 3. ARM11 MPCore processor block diagram [11]

The first multi-core from ARM was the ARM11MPcore [11] implementing the armv6k architecture variant. For these (up to 4 cores) multi-core processors ARM introduced the *Distributed Interrupt Controller*. As can be seen in Figure 3, each of the cores has its own *interrupt interface* and a per core *timer* (and also a watchdog which is of no further interest here). These per core devices reside on a private peripheral bus and are accessible on a

[1]When discussing ARM processors be aware of the difference between the processor family and the underlying architecture variant (e.g., ARM7-processors are based on armv3 architecure, whereas ARM11-processors are based on armv6 architecture).

memory-mapped register scheme. Furthermore, the distributed interrupt controller only handles the IRQ lines for the cores, while the FIQ lines remain independent from it.

The interrupt management system of armv6k distinguishes three types of interrupts. The first type of interrupts are *inter processor interrupts* (IPI). These are introduced due to the need for an operating system to orchestrate the multi-cores. The IPIs are software programmable flags (inside the interrupt interface) that can be set from the source core for the target core. ARM defines 16 of these IPIs with ID0 to ID15. The second type of interrupts are called *private interrupts* and are distinct for each core. The data (word) width on armv6k cores is 32 Bit and a core can query all IPIs and private interrupts with one access. Thus, 16 of these private interrupts are available. However, only three of these lines are defined: ID29 for the timer, ID30 for the watchdog and ID31 for the (legacy) external IRQ. The other unused private interrupts have no defined usage in armv6k. The third and last type of interrupts are called *hardware interrupts* which are connected to the interrupt lines of other (processor external) devices. In armv6k up to 224 of these hardware interrupts are supported starting with ID32. The distributed interrupt controller is managing the external interrupts and, in coordination with the per core interrupt interface, allows to implement an enable/disable- and prioritization scheme on a per core and per interrupt line base.

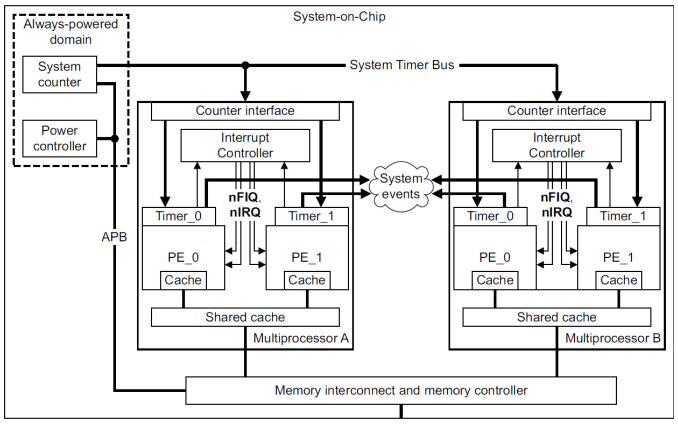

Fig. 4. Interrupt and Timer Management on armv7 and armv8 [12], [13]. PE is Processing Element and represents a core.

With the introduction of armv7 [12], ARM refined the interrupt and timer management to better scale for multi-cores with a higher core count. The overall architecture is shown in Figure 4. Instead of the distributed interrupt controller and the timer, ARM introduced the *generic interrupt controller (GIC)* and the *generic timer*. This refinement also includes a redefinition of terminology. The inter processor interrupts (IPI) are renamed to *software generated interrupts (SGI)*, private interrupts are renamed to *private peripheral interrupts (PPI)* and hardware interrupts are renamed to *shared peripheral interrupts (SPI)*. The number of available SGIs and PPIs is the same as before, 16 each. The number of SPIs was increased to 988 (with GICv2). The armv7 architecture also introduces another privilege level naming scheme on top of the still available processor modes. Privilege level 0 (PL0) is formed by user-mode, PL1 is formed by all of the remaining old modes and an additional new *secure mode*. Furthermore another PL2 is introduced to support the also new *hypervisor-mode* to support system virtualization. This new hypervisor mode also introduces a new type of interrupts: virtual interrupts (vFIQ and vIRQ). These are software programmable flags inside the GICs per core interfaces, only available to the corresponding core itself, to support the virtualization of operating systems. This introduces a complex set of rules how to switch between the processor modes on the occurrence of interrupts. These rules are far to complex to be discussed in this paper, but are mentioned for completeness, see the manuals for details [12].

The most recent architecture definition of ARM is armv8. With this architecture variant, a new 64-Bit ISA called AArch64 was introduced, which is available beside the old A32 and T32[2] ISAs. Current ARM processors are based on this architecture variant. However, the cores themselves still have two interrupt lines (FIQ and IRQ). ARM also rearranged the privilege levels again, which are now called *Execution Levels (EL)*. EL0 is for applications, EL1 is for supervisors (operating systems), EL2 is for hypervisors and EL3 for a security monitor. The old core-modes still exist, but it is harder to associate them with an EL. The EL- and mode-change rules got even more complex (compared to armv7) as it is now possible to switch between the EL, the mode, and the operating ISA (AArch64, A32 or T32) when an interrupt occurs. Even the generic interrupt controllers (GICv3 and GICv4) get more and more complex (e.g., so called *Locality-specific Peripheral Interrupts* are introduced) to handle the complexity and scalability of multi-processor constellations, where each processor is a multi-core. See [13] and [14] for details.

What still holds (even for armv8) is, that there are IRQ, FIQ, a dedicated timer and a dedicated interrupt control interface on a per core basis. The timer and the interrupt control interface are still memory mapped devices. An exception are hypervisor-related interrupt management mechanisms, which are accessed via a core's internal system register interface (which replaces the old system-coprocessor interface). Although ARM provides a generic timer and interrupt specification, one could still simply introduce another timer and interrupt management scheme targeting ARM compatible cores and by implementing device drivers for this, such a processor system can eventually execute an operating system like Linux.

V. Interrupt and Timer Management for RISC-V-Cores

RISC-V is a new ISA that is under development for less than a decade. It was initially designed at UC Berkeley. Nowadays the RISC-V development is coordinated by the RISC-V foundation [15]. The RISC-V ISA was designed under

[2]T32 is the name for the Thumb ISA. It is a compressed subvariant of the conventional ARM ISA with an address and data width of 32 bit each, but only 16 Bit wide instructions.

consideration of the latest trends and lessons learned from the old ISAs like ARM, MIPS or x86. One of the big ideas of RISC-V is the definition of a fixed base ISA (32 Bit, 64 Bit or even 128 Bit, including some optional but standardized extensions), that is guaranteed to be stable in the future. Adaptations in the future can be done by so called *extensions*, which provide flexibility for designers and manufacturers for their targeted application space. The ratification of the base ISA and standard extension is yet under process. To discuss the entire RISC-V ISA is beyond the scope of this paper; the interested reader may have a look at [16] and [17] or the subsequent versions which are already available as drafts.

A major terminology issue when talking about RISC-V is the difference between a core and a *hart*. According to [16]: *A component is termed a core if it contains an independent instruction fetch unit. A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or harts, through multithreading.* In the following the term hart will be used to match the RISC-V specification. However, when talking about non-multithreaded cores, a core and a hart mean the same.

A. RISC-V hart Interrupt Interface

The RISC-V privileged architecture specification defines three privilege levels: L0, called *user/application mode* (U-mode); L1, called *supervisor mode* (S-mode) and finally L3, called *machine mode* (M-Mode)[3]. A RISC-V implementation supporting an operating system like Linux has to support all three modes.

Each of the three modes has three types of possibly pending interrupts: an external interrupt, a timer interrupt, and a software interrupt (for inter-hart communication). So in summary there are nine interrupt bits, that can be pending in a RISC-V hart. However, most of them are software programmable bits. Usually, USIP, UTIP, UEIP (U-mode software/timer/external interrupt pending) and also SSIP, STIP, SEIP (S-mode software/timer/external interrupt pending) are software programmable via the CSR (Control and Status Register) instructions of a hart. Only the corresponding bits of M-mode (MSIP, MTIP and MEIP) are connected to interrupt lines from outside of the hart. The pending bits will only cause the interrupt trap to be taken when the corresponding interrupt enable bit inside the hart is set. So there are nine of these interrupt enable bits.

If an interrupt trap needs to be taken, the hart will switch to M-mode and execute the instruction at the memory address given in the *mtvec* system register.[4] RISC-V further provides an interrupt delegation register, allowing a higher privilege level, to directly delegate specific interrupts to the interrupt handler of the delegated mode. This allows to speed up interrupt handling a little bit.

For the SEIP bit, the RISC-V privileged specification allows (marked as optional) to be connected to a fourth hart-external interrupt line. This is rather unconventional, because it is not strict, in contrast to the otherwise very strict RISC-V specification. As this is marked as optional, the designer might skip this in his RISC-V implementation. Unfortunately the current RISC-V subtree (`arch/riscv`) of Linux kernel (version 5.5 and earlier when writing this paper) expects this SEIP-bit functionality to be implemented.

B. Interrupt and Timer Management

The RISC-V ISA does not explicitly define timer and interrupt handling devices. However, RISC-V foundation provides specifications that propose a possible implementation of such devices. The current available software stack for RISC-V is based on these devices and only includes corresponding device drivers. Alternative hardware Implementations are possible, but would also require adaptations in the software stack.

RISC-V foundation provides two specifications, one is called *Platform Level Interrupt Controller* (PLIC) and the other one is called *Core-Local Interrupt Controller* (CLIC). The tools and simulators targeting RISC-V usually include another device called *Core Local Interrupter* (CLINT). CLIC is planned to be the successor of CLINT. However, as CLINT is already in place, the further disussion in this paper is based on the CLINT. CLINT (and in the future CLIC) is responsible for handling the software and timer interrupts of the harts. PLIC is responsible for handling the external interrupts for M- and S-mode of the harts. Both CLINT and PLIC are monolithic, memory-mapped devices.

CLINT's task is to provide the M-mode timer and software interrupt for a hart. The available CLINT implementation provides a 32-Bit wide register for software interrupts per hart. Furthermore it provides a single 64-bit timer counter (which is used as clock source for exact time measurement in the Linux kernel) as well as for each hart one 64-bit register as timer compare value; generation of a timer interrupt will be triggered if the timer counter value is greater or equal than the timer compare value. The available implementation of CLINT is able to support up to 4095 RISC-V harts[5].

PLIC's task is to manage the M-mode and S-mode external interrupts for RISC-V harts, as shown in Figure 5. The current definition allows to handle up to 1024 external interrupt sources and mange interrupt distribution for up to 7936 RISC-V harts, where each hart requires two so-called *contexts* (one for M-mode and one for S-mode). PLIC further allows to define a priority for each interrupt line, an enable/disable of interrupt generation on a per context per interrupt source basis and finally a per context base threshold mechanism (an interrupt source might only generate an interrupt for a context, when

[3]L2 is marked as reserved in that latest specifications. In earlier drafts L2 was called hypervisor-mode. However, according to [17]: *H-mode has been removed, as we are focusing on recursive virtualization support in S-mode. The encoding space has been reserved and may be repurposed at a later date.*

[4]RISC-V specification also defines the usage of vectored interrupts, meaning that, depending on the interrupt cause, different trap handler entry address can be specified. However, the current RISC-V subtree (`arch/riscv`) of the Linux kernel does not use these vectored interrupts, similar to the ARM case in Section IV-A.

[5]4095 is not a typo, one hart is "lost" due to the memory mapped global timer counter register.

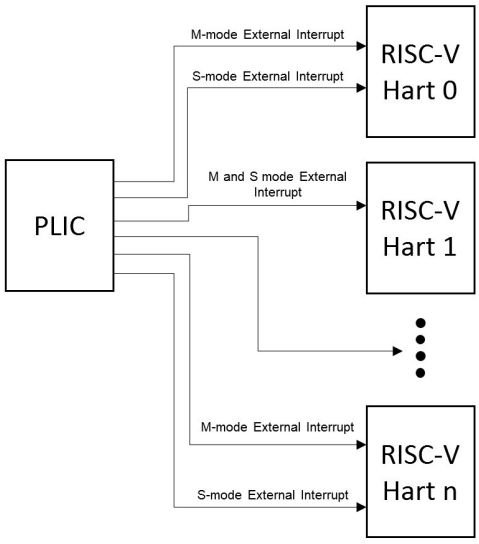

Fig. 5. RISC-V PLIC Interrupt Architecture Block Diagram [18]

the interrupt source priority is higher than the threshold value of the context). As a side note, the corresponding Linux kernel driver (`drivers/irqchip/irq-sifive-plic.c`) is not taking advantage of the priority and threshold mechanism. The driver sets all interrupt sources to the same priority (1 if enabled, 0 if disabled) and all context thresholds to 1.

VI. A Unifying Approach

To achieve a reconfigurable heterogeneous multi-core-system, several possibilities have been presented in Section III. Two solutions that seemed feasible remained. The first includes timer and interrupt management into the core slot. The second proposed to identify a unified interrupt and timer management which thereafter only connects the relevant interrupt lines to the core slots.

For RISC-V, both timer and interrupt management are handled by monolithic devices. For ARM, the timer is core related, but an interrupt management device (GIC) is managing several cores. This observation makes the first proposed solution (include timer and interrupt management into the core-slot) infeasible, as this would mean to rip the aforementioned devices into pieces. From the hardware perspective this might be possible, from the driver perspective of the operating system this is hard to manage, due to memory management issues. So in fact, only the proposed solution to identify a unified interrupt and timer management scheme remains feasible.

As is explained in the previous Section, each RISC-V hart has four interrupt lines, two external (device) interrupt lines MEIP and SEIP, an inter-processor interrupt line (software interrupt, MSIP) and the timer interrupt MTIP. ARM cores use two interrupts FIQ and IRQ. So it comes to the question if it is possible to map one scheme to the other. Clearly, MEIP, SEIP, MSIP, and MTIP cannot be reconstructed from IRQ and FIQ. On the other hand, it is possible to calculate IRQ and FIQ, based on MEIP, SEIP, MSIP and MTIP by simply or-ing the latter four. To make this more manageable, eight enable bits are introduced ($en_{irq,meip}$, $en_{irq,seip}$, $en_{irq,msip}$, $en_{irq,mtip}$, $en_{fiq,meip}$, $en_{fiq,seip}$, $en_{fiq,msip}$ and $en_{fiq,mtip}$). These now allow to calculate IRQ and FIQ:

$$\begin{aligned}\text{IRQ} = \;& \text{MEIP}\wedge en_{irq,meip} \vee \text{SEIP}\wedge en_{irq,seip} \vee \\ & \text{MSIP}\wedge en_{irq,msip} \vee \text{MTIP}\wedge en_{irq,mtip} \\ \text{FIQ} = \;& \text{MEIP}\wedge en_{fiq,meip} \vee \text{SEIP}\wedge en_{fiq,seip} \vee \\ & \text{MSIP}\wedge en_{fiq,msip} \vee \text{MTIP}\wedge en_{fiq,mtip}\end{aligned}$$

The management of the enable bits can be handled by introducing a new device called *interrupt proxy controller*. This device also performs the above calculations. In doing so, it is possible to use all of the RISC-V interrupt and timer infrastructure to generate interrupts for ARM cores, as shown in Figure 6. This requires very little hardware overhead

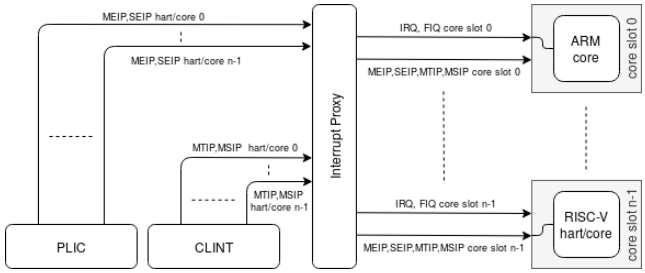

Fig. 6. Architecture including Interrupt Proxy.

compared to an entire core. A major positive aspect of this approach is the fact that the entire RISC-V software stack does not require any modification! For the ARM software stack, CLINT and PLIC have to be made available as a driver in the Linux kernel. Also the *interrupt proxy controller* needs to be supported by a (chained) interrupt chip driver in the Linux kernel for ARM. For RISC-V there is no need to have an interrupt proxy related device driver as the interrupt proxy only pass through the interrupt pending bits of RISC-V harts, which requires no further management capabilities.

ARM provides 16 inter-processor interrupt lines which corresponds to the software interrupt lines in RISC-V. As RISC-V CLINT provides 32 of these lines, the ARM inter-processor interrupts can also be handled by it.

The RISC-V timer is fixed to be 64 bits wide, which is also the number of timer compare bits, defined by the ARM generic timer manual [13]. Therefore, CLINT is also applicable to manage ARM cores' timer interrupt generation.

VII. Proof of Concept

To prove the applicability of the unified interrupt and timer management for RISC-V and ARM cores/harts a proof of concept demonstrator was implemented, based on the Partial Reconfigurable Heterogeneous System (PRHS) Framework [19], [20].

The PRHS consists of hardware and software components. It is suitable to easily construct entire computer systems inside an FPGA. PRHS supports a variety of different evaluation boards designed around Xilinx FPGAs (from 7-Series to UltraScale+ Series). The hardware part of PRHS consists of self-implemented VHDL-Modules. This includes a lot of

peripheral devices and also two kinds of processors. The first kind of processor is compatible to the armv6k architecture (the first ARM architecture supporting multi-cores). The second kind of processors is compatible to RISC-V ISA (rv32ima is already available, rv64, F- and D- extension are currently under development). For ARM compatible cores, a 32-core multi-core is already available; for RISC-V only a single core implementation is available yet (as the cache coherent interconnect of the ARM multi-core is not suitable to construct RISC-V multi-cores). The software part of PRHS includes an adapted *u-boot* bootloader, an adapted Linux kernel (currently supported version is 5.1) and targets *busybox* as minimal distribution. For the RISC-V cores an adopted *openSBI* is used for the machine-mode execution environment.

Figure 7 displays the overall architecture of the proof of concept demonstrator. As can be easily seen, it is a simplified version of Figure 6. To show the feasibility of the interrupt proxy, a single core system is sufficient, as the interrupt proxy straightforward scales with the core count. The restriction to one core arises from not (yet) having a fully functional cache coherent core-interconnect, supporting the RISC-V and ARM cores of PRHS in a multi-core setup.

As explained in Section VI the proof of concept demonstrator works out-of-the box, if the core slot is filled with a RISC-V core/hart. For the version using an ARM core inside the core slot, several adaptations to software elements were required. First of all the bootloader needs to be adapted. The adaptation is related to timing measurement and external interrupts. Hence, u-boot device drivers were implemented to allow the usage of PLIC and CLINT for ARM-cores. For Linux kernel, two new device drivers were implemented, another one required adaptation. The driver related to PLIC required adaptations to be used by ARM-cores, as the original driver (`drivers/irqchip/irq-sifive-plic.c`) contains macros that result in RISC-V inline assembly which cannot be executed by ARM cores. CLINT itself needed to be split into two device drivers. Fortunately the address mapping of CLINT allows to present it as two devices to Linux kernel. The first one is a timer-driver, that handles the *event timer* management and also the *clock source* management. The second CLINT related device driver is responsible to manage the inter-processor (software) interrupts for the ARM cores. The challenge here was the fact that CLINT is not available as Linux kernel device driver, since the low-level details of timer control and inter-processor communication is hidden behind the SBI (Supervisor Binary Interface) presented by the M-mode execution environment (in PRHS, openSBI is used as M-mode execution environment) in the RISC-V software stack. Such an environment is not required for ARM systems.

After all drivers were implemented, PRHS is now able to boot ARM and RISC-V based systems, that use the same (hardware) devices for handling timers and interrupts. This shows the feasibility of the suggested unified timer and interrupt management approach.

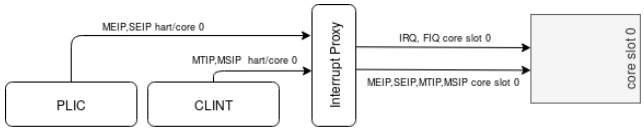

Fig. 7. Proof of Concept Demonstrator including one slot. This slot can either be filled with an ARM or RISC-V core that connects to the appropriate incoming interrupt lines of the slot.

VIII. Discussion

One final question remains: What is the overhead (in time) introduced by the interrupt proxy for ARM systems?

For the proof of concept demonstrator it is possible to determine the exact number of instructions that are added by the interrupt chaining from the interrupt handler of the *interrupt proxy* to the interrupt handlers of the CLINT based timers and / the interrupt handler of PLIC. This number of instructions can be determined by performing a RTL simulation of PRHS running Linux. The measured number of overhead machine instructions is: 111. For an operating system like Linux, this number seems affordable, when we compare it with the number of instructions that are executed in a time slice of the operating system[6]. In Real-Time contexts this might be too high, though. But even then the suggested approach is still feasible: The instruction overhead is introduced by providing separate drivers for CLINT timer management, PLIC and the interrupt proxy. This driver separation is related to Linux kernels philosophy of clean code separation regarding functionality; timer management is different to interrupt chip management. For real-time operating systems it might be possible to implement a unifying device driver for all of the required functionality. The instruction overhead related to the interrupt proxy then shrinks to a device register query, what kind of interrupt (timer, external or inter processor) caused the ARM core to trap, a switch-case block to differ between the reasons and finally a jump/branch to the appropriate handler.

IX. Conclusion

Based on the initial idea of constructing a runtime adaptable, OS-capable ISA-heterogeneous multi-core, this paper discusses the need for a unified timer and interrupt management. Furthermore it gives an overview on interrupt and timer management mechanisms for ARM and RISC-V based systems. This results in the deduction that the RISC-V timer and interrupt management infrastructure can also be used for ARM systems by introducing a device called *interrupt proxy* that translates RISC-V interrupts to ARM interrupts. The feasibility of the suggestion is shown with a proof of concept demonstrator that includes VHDL based modules and the appropriate drivers to allow for the execution of Linux on top of RISC-V and ARM cores, both using the same, unified interrupt and timer infrastructure. The next step in the overall

[6]For the proof of concept demonstrator, a time slice has a length of 10ms; if we assume a clock frequency of 100MHz (due to FPGA realization) and assume a pessimistic IPC (Instructions per Cycle) of 0.5, 500000 instructions are executed in one time slice. In comparison, 111 is rather small.

project is to identify and implement a suitable cache coherent interconnect, that allows to tightly couple a mixed, runtime-adaptable composition of RISC-V and ARM cores. This will also allow to proof the idea of a unifying interrupt proxy in a multi-core setup.

REFERENCES

[1] S. Mittal, "A survey of techniques for architecting and managing asymmetric multicore processors," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 45:1–45:38, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/2856125

[2] [Online]. Available: http://www.popcornlinux.org/

[3] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-isa datacenters," *SIGPLAN Not.*, vol. 52, no. 4, pp. 645–659, Apr. 2017. [Online]. Available: http://doi.acm.org/10.1145/3093336.3037738

[4] P. Olivier, S.-H. Kim, and B. Ravindran, "Os support for thread migration and distribution in the fully heterogeneous datacenter," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: ACM, 2017, pp. 174–179. [Online]. Available: http://doi.acm.org/10.1145/3102980.3103009

[5] P. Olivier, A. K. M. F. Mehrab, S. Lankes, M. L. Karaoui, R. Lyerly, and B. Ravindran, "Hexo: Offloading hpc compute-intensive workloads on low-cost, low-power embedded systems," in *HPDC '19*, 2019.

[6] A. Venkat, *Breaking the ISA Barrier in Modern Computing*. University of California, San Diego, 2018. [Online]. Available: https://books.google.de/books?id=71PevQEACAAJ

[7] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 121–132, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665692

[8] *ARM Architecture Reference Manual*, ARM Limited, ARM DDI 0100I.

[9] L. Torvalds, "Linux kernel." [Online]. Available: https://github.com/torvalds/linux

[10] *ARM PrimeCell Vectored Interrupt Controller (PL192) Technical Reference Manual*, ARM Limited, ARM DDI 0273A.

[11] *ARM11 MPCore Processor Technical Reference Manual*, ARM Limited, ARM DDI 0360F.

[12] *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*, ARM Limited, ARM DDI 0406C.b.

[13] *ARM Architecture Reference Manual - ARMv8, for ARMv8-A architecture profile*, ARM Limited, ARM DDI 0487A.j.

[14] *ARM Generic Interrupt ControllerArchitecture Specification - GIC architecture version 3.0 and version 4.0*, ARM Limited, ARM IHI 0069C.

[15] RISC-V Foundation. [Online]. Available: https://riscv.org/

[16] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*, December 2019. [Online]. Available: https://riscv.org/specifications/isa-spec-pdf/

[17] ——, *The RISC-V Instruction Set Manual - Volume II: Privileged Architecture*, June 2019. [Online]. Available: https://riscv.org/specifications/privileged-isa/

[18] *RISC-V Platform-Level Interrupt Controller Specification*. [Online]. Available: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

[19] M. Eckert, "Fpga-based system virtual machines," Ph.D. dissertation, Helmut-Schmidt-Universität, Holstenhofweg 85, 22043 Hamburg, 2014.

[20] D. M. Marcel Eckert and B. Klauer, "Context save and restore of partial reconfiguration regions for xilinx fpgas," in *14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2019)*, 2019.