Generating correct initial page tables from formal hardware descriptions

Reto Achermann
University of British Columbia
Vancouver, Canada

David Cock
ETH Zurich
Zurich, Switzerland

Roni Haecki
ETH Zurich
Zurich, Switzerland

Nora Hossle
ETH Zurich
Zurich, Switzerland

Lukas Humbel
ETH Zurich
Zurich, Switzerland

Timothy Roscoe
ETH Zurich
Zurich, Switzerland

Daniel Schwyn
ETH Zurich
Zurich, Switzerland

Abstract

Modern hardware platforms are increasingly complex and heterogeneous. System software use a hodgepodge of different mechanisms and representations to express the memory topology of the target platform. Considerable maintenance effort is required to keep them in sync while often sharing is impossible due to hard-coded values. Incorrect platform-specific values in the hardware initialization sequence can lead to security critical and hard-to-find bugs because of misconfigured translation hardware, inaccessible devices, or the use of bad pointers.

We present a better way for system software to express and initialize memory hardware. We adopt an existing, powerful hardware description language, and efficiently compile it to generate correct initial page tables and memory maps for OS kernels and firmware from a single system description.

We evaluate our system on multiple architectures and platforms, and demonstrate that we can use the generated data structures to successfully initialize translation hardware, devices, memory maps, and allocators enabling easy support of new hardware platforms.

1 Introduction

Hardware is becoming both more complex, and simultaneously more diverse: Even small SoCs now comprise a dozen dramatically different processors (application cores, DSPs, accelerators, etc.), bound together with a complex non-uniform interconnect with each agent having a unique view of system addresses. At the same time, the number of different platforms to which software must be ported is growing dramatically each year, beyond the rate at which high-quality initialization and management code can be written. One result is that, despite enormous investment by platform vendors, the state of the art for platform initialization is a hodgepodge of repurposed mechanisms, none quite fit-for-purpose, and the widespread reuse of canned initialization snippets, silently replicating and importing inaccurate assumptions about hardware.

In this paper, we show by example that there is a better way. We consider the particular case of page table generation and allocator initialization, which exposes much of the complexity of modern hardware (including non-uniform addressing and heterogeneous processing), while being critical to the correct and secure operation of the system. We adopt an existing description language for addressing architectures (Sockeye), which has been shown to be able to express real, extremely complex modern systems, and which has a rigorous formal interpretation (decoding nets). Finally, we formulate the problem of system initialization as one of compilation: can we (efficiently) generate correct initialization data (here, the initial page tables and allocator state) from a Sockeye description of the system?

In the remainder of this paper we show that the answer to the above question is a resounding yes. The decoding net formalization allows us to frame the problem as determining whether a page table exists which maps the desired CPU-visible ("virtual") address space to the projection of system resources (RAM, device registers, etc.) onto the CPU's "physical" addresses (as computed from the decoding net), within the constraints of the virtual memory system (e.g. granularity). As we will show, this requires simply the recursive projection of resources in reverse direction (from enclosed to enclosing AS) along the decoding net.

Lastly, we show that a straightforward expression of the decoding net rules in Prolog produces an efficient solution, which works in practice. We are able to generate correct initial page tables and allocator state for numerous real platforms, and boot and run an actual OS kernel using them. We further show that the solution is fully general, and not special-cased to any particular architecture, by generating

PLOS 2021, October 25, 2021, Online Workshop
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

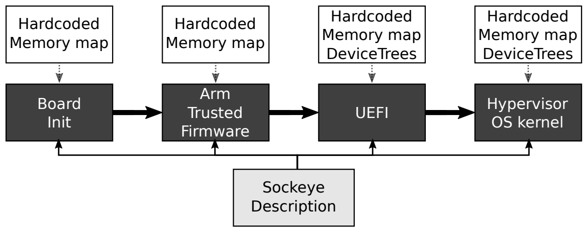

Figure 1. An illustration of the boot process. Individual platform descriptions above, and the proposed solution below.

both initialization data and configurations for the official Arm Fast Models simulator for a variety of pathologically-complex and nonuniform hypothetical platforms, and showing that the generated configuration suffices to boot the OS in all cases without any user intervention whatsoever.

2 Background

Consider the problem of simply booting a modern machine. A typical boot sequence for an ARMv8 platform is shown in Figure 1: a chain of different firmware and OS Components loaded one after the other, each requiring information about the hardware platform; indeed the OS is the last component to be loaded and executed.

During boot, all this system software needs to understand the complex memory topology of the platform and reflect this in the hardware initialization sequence. A key part of this challenge, and the one we focus on in this paper, is the creation of the page tables to configure the processor's MMU (and System MMU), constructing the memory map for populating the memory managers, and initializing devices at the right location and programming them with the correct memory addresses.

Today, each component in the boot sequence must initialize hardware, or make use of hardware configured in a prior step. The way this hardware is described today is typically hardcoded by programmers (even for "discoverable hardware"), and in a variety of ad-hoc and ill-defined formats. This results in a considerable engineering burden for each new device (as anyone who has done an OS bringup on a new piece of modern hardware can attest), and moreover has the potential to introduce catastrophic and hard-to-debug errors as a result of memory misconfiguration [11, 13, 22], or wrongly passed tables between components [12, 14, 15].

The memory topology of modern hardware platforms makes this problem even worse [7]. Memory accesses from processor cores and devices traverse multiple buses, memory controllers, and memory translation and protection units before reaching their destination e.g. memory cell or device register. For example, Figure 2 shows a typical modern SoC from NXP, where the meaning of a (physical) address is relative to the processor core (A35 or M4) or DMA-capable device (LPUART, PPB) leading to situations where the same

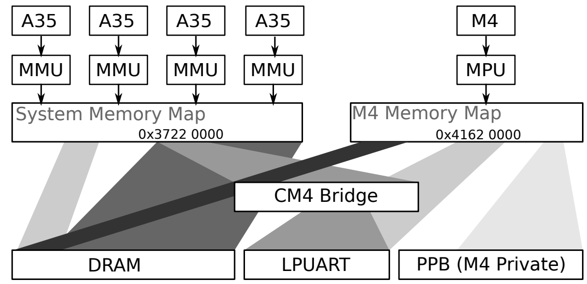

Figure 2. Subset of the NXP i.MX8 memory layout. The LPUART is accessed using a different addresses from the M4 cores, the >2GiB memory is only accessible from the A35 cores, the PPB is only reachable from the M4 cores.

resources appear at different addresses, or different resources appear at the same address, viewed from different cores.

Current approaches

Today, computers use a mix of different mechanisms and representations of system state (including memory maps) at different stages in the boot process: ACPI [16], UEFI [19], hardcoded values, DeviceTree[9], etc. Linux even builds initial page tables using hand-written assembly [18]. Arm Trusted Firmware [17] uses a C data structure to initialize system page tables [10].

Keeping these representations in sync is purely manual. Hard-coding various aspects of the platform makes it hard to share code between platforms and increases the maintenance effort needed to support a wide range of systems. OS developers try to separate initialization code from the *platform-specific values* it uses, but quickly run into problems: in practice, modern hardware cannot be described faithfully as a set of arguments to a C function, and the ideal of a single externalized platform description is never achieved.

The most complete description format for platforms today is DeviceTree [9], used by the Linux kernel to describe non-discoverable platform information, and also employed (with different device tree files) by some intermediate bootloaders.

DeviceTree files capture the platform's application processors, memory and caches, devices, interrupt sources, and clocks in a tree-like data structure with a single root. While sufficient for the Linux kernel, it fails to address the general problem for several reasons. As its name suggests, a DeviceTree is a tree. Modern machines are much more general (possibly cyclic!) graphs, even in memory addressing [7]. Multiple processors (as in the NXP example above) would require multiple, overlaid, *consistent*, trees. Moreover, DeviceTree files are not well specified: most DeviceTree nodes have addressing semantics that are defined by the C code of the corresponding compatible Linux kernel drivers, rather than a clean formal (or even semi-formal) description.

Consequently, DeviceTree files are of limited use in initializing a heterogeneous SoC, and cannot serve as basis for any assurance of correctness for firmware and kernels which rely on them for configuration information.

Discussion

We take the position that, rather initializing and booting a machine relying on replicated, hand-written, low-level code, interpreting a semantic-free and inherently incomplete description of the platform hardware, a better way is needed.

Instead, we start with a *formally-specified* way to describe platforms, which can *capture the full complexity* of the modern systems with different processors and interconnects, and then use this description to *generate* low-level system software and firmware components that is correct by construction. Such an approach is not merely motivated by reducing engineering cost, but is also an absolute prerequisite for formally verifying low-level system software for a given hardware platform.

In this paper, we demonstrate that initial page tables can be constructed generically from formal specifications of the system at hand. We not only show how this can be done efficiently, but also demonstrate that it works for heterogeneous system with highly esoteric memory addressing. We start with the formal representation of addressing in modern SoCs that forms the foundation of our approach.

Decoding nets

We begin with the *decoding net* [2, 3], a formally specified model that has been shown to capture the memory topology of a broad variety of hardware platforms in a rigorous and well-defined way. Decoding nets express the addressing structure of a system as a directed graph: nodes represent (virtual or physical) address spaces or devices (including RAM), and edges the possible translation between them. The model distinguishes *address-space-local* names (*address*), and *global* names (*name*) that are qualified by their enclosing address space. Each node may **accept** a set of (local) addresses (e.g., RAM or device registers), and/or **translate** them to one or more global names (e.g., MMU or PCI bridges).

$$\begin{aligned} name &= \text{Name } nodeid\ address \\ node &= \text{Node } \textbf{accept} :: \{address\} \\ &\qquad \textbf{translate} :: address \rightarrow \{name\} \end{aligned}$$

An example decoding net for an x86 machine with a Xeon Phi [8] accelerator card attached over PCI is shown in Figure 3. The dark nodes are leaf nodes in the graph, they don't translate addresses but only accept them.

The Sockeye language [21] is a syntax to express the memory topology of a hardware platform as a decoding net. Sockeye bears some superficial similarities to the DeviceTree language, but in contrast has clear semantics that can express decoding nets formally (and, indeed, generate Isabelle/HOL representations of such nets). It provides syntactic elements such as regions and modules that help to express the system in a concise and understandable way. A small excerpt from the description for the system depicted in Figure 3 is shown in Figure 4.

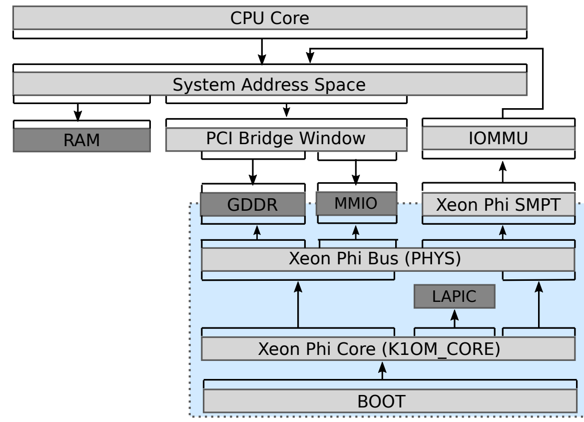

Figure 3. Example of a decoding net representing an x86 machine with a Xeon Phi accelerator card. Highlighted parts are described in Figure 4

```
module XEONPHI {
  memory (0 bits 40) GDDR
  GDDR accepts [(0x0 to 0x1ffffffff) (mem)]
  memory (0 bits 16) MMIO
  MMIO accepts [(0x0 to 0xfffff) (devreg)]
  memory (0 bits 40) PHYS
  PHYS maps [
    (0x0000000000 to 0x00fedfffff)
        to GDDR at (0x000000000 to 0x0fedfffff);
    (0x00fee01000 to 0x01ffffffff)
        to GDDR at (0x0fee01000 to 0x1ffffffff);
    (0x08007D0000 bits 16)
        to MMIO at (0 bits 16);
    (0x8000000000 to 0xffffffffff)
        to SMPT_IN at (0x0 to 0x7fffffffff)]
  // Description of one booting core
  memory (0 bits 40) LAPIC
  LAPIC accepts [(0 bits 12) (devreg)]
  memory (0 bits 40) K1OM_CORE
  K1OM_CORE maps [
    (0x00000000 to 0xfedfffff)
        to PHYS at (0x00000000 to 0xfedfffff);
    (0xfee00000 bits 12)
        to LAPIC at (0 bits 12);
    (0xfee01000 to 0xffffffffff)
        to PHYS at (0xfee01000 to 0xffffffffff)]
  // Initial pagetable for the boot process
  BOOT maps [
    (0x0 to 0xffffffffff)
        to K1OM_CORE at (0x0 to 0xffffffffff)]}
```

Figure 4. Simplified Sockeye description of a Xeon Phi coprocessor PCI card. Note the map to `SMPT_IN` providing a window to host resources.

Sockeye is designed around reusable blocks of decoding net nodes called **modules**. Each module has a name, parameters, and a set of input and output nodes to be bound on instantiation. Figure 4 is an excerpt of a Xeon Phi PCI-based accelerator module, restricted to the view from its CPU (**K1OM_CORE**). This address space in turn has a window to its local APIC (**LAPIC**) and maps the rest to the core local space (**PHYS**). This in turn contains memory (**GDDR**), an MMIO region for the control registers (**MMIO**), and an aperture on the system address space (**SMPT_IN**).

Sockeye allows memory regions to be tagged with predicate logic terms. For example, memory regions are tagged with with *mem*, and device registers with *devreg*. We exploit this information in correctly mapping devices in our generated page tables.

3 Implementation

As discussed, we use generating initial kernel page tables as an example as it exercises the model (specifically in identifying device regions), without being dependent on the details of a particular operating system as all kernels use quite similar layouts (in contrast to, say, the operation of their memory allocators). We have also used the same techniques described here for both the static initialization and dynamic runtime state of the Barrelfish memory allocator and device manager, which we hope to present in followup work.

The initial structure of a kernel's virtual address space is quite simple, and generally consists of a 1–1 mapping of some portion of the system address space, including enough RAM for the kernel's internal needs, plus any devices (such as interrupt controllers) that the kernel itself relies upon. Additional device mappings are typically added at runtime, either within the kernel's own virtual address space (for a monolithic kernel), or into a user process's space (for a microkernel-like system).

The challenge in constructing the initial page table is thus not in constructing the page table itself. The virtual–physical map is unconstrained down to the translation granule and thus can trivially represent any desired mapping, and generally just consists of large 1–1 mappings in any case. The specific problem to be solved by querying the decoding net is rather to identify which regions are accessible to the processor (in particular the required devices), what their properties are (e.g. device registers must usually be mapped uncached), and at what address in the CPU's 'physical' address space they appear. The page-table generator needs to know, for example, whether a large mapping must be split to specify that some sub-range is to be mapped uncached for a device.

Complexity

As already described and as illustrated in Figure 3, the decoding network is a directed acyclic graph, with accepting regions (here RAM or devices) at the leaves, and CPU cores

```
% Datatypes
NodeId :: [String]
Block :: block(base :: Int, limit :: Int)
Region :: region(id :: NodeId, blocks :: [Block],
                 prop :: BooleanExp)
% Predicates
translate(in :: Region, out :: Region).
accept(r :: Region).
```

Figure 6. Prolog datatypes and dynamic predicates

(or other bus-mastering agents) at the roots. It is thus possible in principle to compute the regions visible at any node iteratively: beginning at the leaves, follow the edges in reverse to determine where this region appears in other spaces (noting that it may appear in many, only a sub-region may appear, it may appear twice, etc.), and repeating until all regions have been projected up as far as the target node of the CPU's page table mappings.

As Figure 5 illustrates, the number of regions (and hence complexity of any algorithm enumerating them) is exponential in the diameter of the decoding net. We here see one accepting region (Acc) mapped twice into the immediately preceding address space, which in turn is mapped twice into its predecessor. In this example we will generate at least 2^n distinct regions for a root address space at distance n from the resource. It is not necessarily possible to merge these regions (e.g. if not contiguous), and thus all 2^n regions might need to be represented.

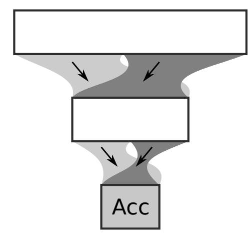

Figure 5. A worst-case decoding net for region enumeration.

As solving for a desired configuration will in general involve a search through this exponentially-large space (note that runtime algorithms such as allocation have stricter requirements than the initial page-table generation considered here), we cannot expect an efficient sub-exponential algorithm to exist for the general case. In practice, such pathological examples do not occur in actual hardware, and established heuristic search strategies perform well. Indeed, we take advantage of the fact that the experimental OS on which we evaluate (Barrelfish) incorporates the Eclipse/CLP solver for just such system configuration tasks (the so-called SKB or System Knowledge Base [20]), and encode the problem quite directly as a set of Prolog predicates which (as Section 4 shows) performs very well in practice.

Prolog Encoding

Figure 6 gives the syntax used to encode a decoding net as Prolog assertions. The translate and accept facts are generated by straightforward compilation from a Sockeye description of the system such as that of Figure 3. For efficient

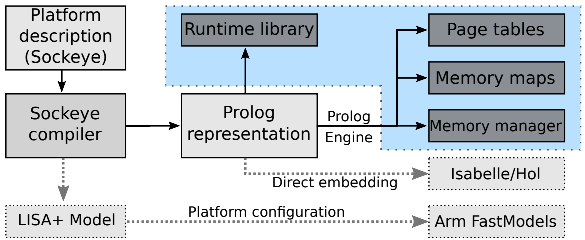

Figure 7. Integration of the query engine with the OS. Highlighted components are part of the boot image.

evaluation, individual addresses are not expressed directly, but only as part of larger blocks which in turn form (not-necessarily-contiguous) regions.

The one-step projection from destination region D up to some source region S in a predecessor address space is expressed by the following predicate:

```
decode_step_rev(D,SI,S) :-
  translate(SM,DM),
  reg_intersection(DM, D, SI),
  DM = region(_, [block(DMBase, _)],_),
  SI = region(_, [block(SIBase, SILimit)],_),
  SM = region(SNode,[block(SMBase,_)],_),
  SBase is SMBase + (SIBase - DMBase),
  SLimit is SBase + (SILimit - SIBase),
  S = region(SNode,[block(SBase,SLimit)],_).
```

This expresses the existence of a mapping from some region of a source address space SM to some region of a destination space DM, such that the intersection of DM and D is exactly SI, or the image of the source region under the mapping. The remaining conjuncts establish the position of S within the mapping source region SM as a function of the position of the image within DM.

Finding the location of all regions visible in some top-level address space A requires solving for S for every value of D for which accept(D) holds in the transitive closure of `decode_step_rev` i.e. where the source region *eventually* maps to the accepting region. Adding architecture-specific constraints on allowable page-table entries (e.g. alignment to 4kiB), and enumerating all solutions for S then gives the values for all page table entries. Properties (e.g. cacheability) are taken directly from the accepting region.

Output Generation and Integration

Figure 7 illustrates the integration of the decoding net query results with the rest of the system. The results of the reachable-region query are used in several locations, some at compile time, and some at runtime:

Firstly, the returned mapping entries are encoded into machine-specific page-table descriptors and output as static initializers for a C array comprising the initial page tables. As the page table is (on all these architectures) multilevel, we exploit the linker and loader to correctly finalize the descriptors. While the last-level descriptors directly refer to known CPU-physical addresses, higher level descriptors refer to lower-level tables, whose location is unknown at compile time. Instead of hard-coding these addresses, we emit carefully-generated ELF relocation records ensuring that the correct addresses will be filled in either by the linker (if building static, position-dependent code) or the (boot-)loader if the kernel is dynamically-loaded and position-independent.

Secondly, the query results are used to initialize various other OS data structures, including the initial state of the memory allocator (i.e. the location of all RAM regions), and the device manager (which is told the resources required by a driver for all statically-discoverable devices). These runtime structures are thus guaranteed consistent with the kernel's internal view of the memory system, and the offline model.

Finally, the decoding net itself (expressed in the syntax of Figure 6) is (on Barrelfish) seeded into the SKB. This data is queried dynamically at runtime (and indeed, extended as the result of online device discovery), and used to initialize additional devices requiring, for example, IOMMU page table configuration. This includes, among others, the Xeon Phi accelerator used as an example here, which incorporates numerous full CPU cores which run their own instance of the OS kernel. The online model is further used (with appropriate queries) to correctly allocate and map DMA-able memory accessible to devices with different views of the system from that of the CPU (again, including the Xeon Phi).

4 Evaluation

We evaluate our approach with two experiments: First, we show that it is feasible to generate initial kernel page tables and memory maps for an OS running on various real platforms (Section 4.1). Secondly, we demonstrate that it even works for constructed, intentionally hard to deal with memory topologies (Section 4.2).

4.1 Real Platforms

Previous work has shown that decoding nets can accurately capture memory topologies of real hardware [3]. Here we show that our Sockeye-generated page tables and memory maps suffice to boot an OS kernel on real hardware.

We use the following existing Sockeye specifications:

- x86_64: Normal x86-64 PC, and QEMU emulator.
- k1om: Intel Xeon Phi co-processor (Knights Landing)
- Armv7: Pandaboard, a TI OMAP44xx based board
- Armv8: Arm Cortex-A57 FVP, a simulated dual core reference platform, and QEMU emulator.

We generate page tables and other initialization data for each platform as described in Section 3, and use them to boot Barrelfish [6]. The bootdriver uses the generated page tables to initialize the kernel AS by simply setting the translation base register to the start of the page table binary. The bootdriver

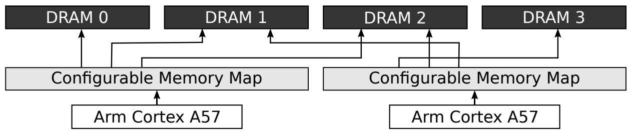

Figure 8. Memory topology of the *Swapped + Private* simulator platform with shared regions DRAM 1/2 and private regions DRAM 0/3

then loads the kernel, passing the locations of page tables and device mappings as an argument. All kernel memory and device accesses are via the generated tables.

In all cases, the kernel boots and configures all devices correctly using the generated page tables installed.

Despite the simplicity of the Prolog implementation, the generation process is robust enough to handle a wide variety of real hardware platforms. While we use Barrelfish for demonstration, nothing about the generated tables or data structures is system specific, and could be used just as well in other OSs such as Linux or seL4.

4.2 Simulated Platforms

Here we show that the generator handles not only conventional platforms but also ones with pathologically-hard memory topologies, where different agents (cores) in the system have completely different views on memory.

As before, we specify the memory topology of the platforms using the Sockeye language. Additionally, via another backend we also generate a configuration for the Arm FastModels simulator [4] which corresponds to the specified topology. We use this to generate a range of unusual platforms for evaluation as follows:

The base topology is that of the A57 FVP of the previous experiment. From this, we generate 3 additional topologies, whose basic structure is shown in Figure 8. There are a total of four DRAM regions each one GiB in size. Each core has its own configurable memory map in addition to its MMU, mapping DRAM into its address space.

In the base case, both cores have the same view: `[0,1,2,3]` the first GiB maps to DRAM0, the second to DRAM1, etc. The remainder are configured as follows:

1. *Swapped*: DRAM is split in two and the address ranges where the cores see the halves are swapped relative to each other. One sees DRAM as `[0,1,2,3]` and the other as `[2,3,0,1]`.
2. *Private*: DRAM `[1,2]` are shared, and each core has a private region of DRAM. We have the mappings `[1,2,0]` and `[1,2,3]`.
3. *Swapped + Private*: This combines the others: the shared regions of the previous topology are swapped. The resulting mappings are `[1,2,0]` and `[2,1,3]`.

The description used to generate the simulator is also used to generate page tables and memory maps. We supply the configuration to Arm FastModels, generated directly from the Sockeye file. As before, we boot Barrelfish on the simulated systems.

In each case Barrelfish boots successfully into userspace on both cores. Both cores use the same code with the exception of the parts generated from the topology information. In addition, processes on the two cores can communicate over shared memory.

We see that the generation approach is robust enough not only for real hardware, but in adversarial scenarios with exceptionally peculiar memory topologies. That processes can establish shared memory channels between cores in all cases shows that the generated maps for the different cores are consistent because they have been generated from a single Sockeye platform description.

5 Future Work

Generating page tables is a first step towards OS configuration based on the decoding net model. We plan to apply the approach outlined in the paper to the full boot process. If we can precisely specify the starting state for each boot stage, then we can not only eliminate unsafe memory accesses due to wrongly configured translation tables, but also precisely specify the contract between two stages.

Similarly, we can use the same approach to express additional protection mechanisms (e.g., Arm TrustZone [5]) in Sockeye and generate configurations to divide resources in secure and non-secure worlds.

Moreover, we plan to use the runtime representation and algorithms presented in this paper in memory allocators to find memory regions that can be shared between the driver software and accelerators/devices, and to guide configuration using the recently proposed `mmapx` interface [1].

Finally, our deep embedding of Prolog in Isabelle/HOL provides a framework to link the algorithms and facts produces by the Sockeye compiler back to the decoding net model and enables proofs about its correctness.

6 Conclusion

In this paper, we have presented a system that leverages the sound foundation provided by the decoding net model, and the Sockeye language to generate platform-specific data structures such as page tables and memory maps. We have outlined the required algorithms, their implementation in Prolog, and the integration into the build system to obtain a page table binary image that is then used by the operating system to configure the translation hardware.

Our evaluation qualitatively shows the application and integration of the address space model into the OS toolchain to *generate* low-level, platform-specific OS code and data structures. Our approach and implementation thereof is functional even when run on simulated platforms with unusual address space topologies not supported by other systems.

References

[1] Reto Achermann, David Cock, Roni Haecki, Nora Hossle, Lukas Humbel, Timothy Roscoe, and Daniel Schwyn. 2021. Mmapx: Uniform Memory Protection in a Heterogeneous World. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 159–166. https://doi.org/10.1145/3458336.3465273

[2] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2017. Formalizing Memory Accesses and Interrupts. *Electronic Proceedings in Theoretical Computer Science* 244 (Mar 2017), 66–116. https://doi.org/10.4204/eptcs.244.4

[3] Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe. 2018. Physical Addressing on Real Hardware in Isabelle/HOL. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 1–19. https://doi.org/10.1007/978-3-319-94821-8_1

[4] ARM Ltd. 2019. Development Tools and Software: Fast Models. https://www.arm.com/products/development-tools/simulation/fast-models

[5] ARM Ltd. 2021. Arm TrustZone Technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[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* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[7] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. 2015. Not Your Parents' Physical Address Space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Switzerland) *(HOTOS'15)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=2831090.2831106

[8] Intel Corporation. 2014. *Intel Xeon Phi Coprocessor System Software Developers Guide.*

[9] Linux Kernel Documentation. 2019. *Device Tree Source Format (version 1)*. Linux. Retrieved 06 August 2021 from https://git.kernel.org/pub/scm/utils/dtc/dtc.git/plain/Documentation/dts-format.txt

[10] Arm Ltd. 2021. *ATF - Translation (XLAT) Tables Library.* https://trustedfirmware-a.readthedocs.io/en/latest/components/xlat-tables-lib-v2-design.html

[11] Red Hat Bugzilla 2010. *Bug 654665 - EFI/UEFI page table initialization is incorrect for x86_64 in physical mode.* Red Hat Bugzilla. https://bugzilla.redhat.com/show_bug.cgi?id=654665

[12] sunxi 2012. *Unable to pass memory configuration from u-boot to kernel.* sunxi. https://github.com/linux-sunxi/u-boot-sunxi/issues/11

[13] Kernel.org Bugzilla 2013. *Bug 56461 - Memory corruption on PAE x86 systems.* Kernel.org Bugzilla . https://bugzilla.kernel.org/show_bug.cgi?id=56461

[14] Linux Kernel Mailing List 2017. *efi/x86: Prune invalid memory map entries and fix boot regression.* Linux Kernel Mailing List. https://lore.kernel.org/patchwork/patch/752197/

[15] launchpad 2019. *Full RAM on Pi4 isn't accessible when using u-boot.* launchpad. https://bugs.launchpad.net/ubuntu/+source/u-boot/+bug/1847500

[16] UEFI Forum, Inc. 2021. *Advanced Configuration and Power Interface (ACPI) Specification.* UEFI Forum, Inc. https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/

[17] Arm Ltd. 2021. *Arm Trusted Firmware.* Arm Ltd. https://github.com/ARM-software/arm-trusted-firmware

[18] Linux 2021. *Linux x86 boot code.* Linux. https://github.com/torvalds/linux/blob/master/arch/x86/boot/compressed/head_64.S

[19] UEFI Forum, Inc. 2021. *Unified Extensible Firmware Interface (UEFI) Specification.* UEFI Forum, Inc. https://uefi.org/sites/default/files/resources/UEFI_Spec_2_9_2021_03_18.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* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 119–132. https://doi.org/10.1145/1950365.1950382

[21] Daniel Schwyn. 2017. *Hardware Configuration With Dynamically-Queried Formal Models.* Master's Thesis. Department of Computer Science, ETH Zurich, Switzerland. https://doi.org/10.3929/ethz-b-000203075

[22] Ulf Frisk. 2018. *Total Meltdown?* http://blog.frizk.net/2018/03/total-meltdown.html