Physical Addressing on Real Hardware in Isabelle/HOL

Reto Achermann, Lukas Humbel, David Cock, and Timothy Roscoe

Department of Computer Science, ETH Zurich

Abstract. Modern computing platforms are inherently complex and diverse: a heterogeneous collection of cores, interconnects, programmable memory translation units, and devices means that there is no single physical address space, and each core or DMA device may see other devices at different physical addresses. This is a problem because correct operation of system software relies on correct configuration of these interconnects, and current operating systems (and associated formal specifications) make assumptions about global physical addresses which do not hold. We present a formal model in Isabelle/HOL to express this complex addressing hardware that captures the intricacies of different real platforms or Systems-on-Chip (SoCs), and demonstrate its expressivity by showing, as an example, the impossibility of correctly configuring a MIPS R4600 TLB as specified in its documentation. Such a model not only facilitates proofs about hardware, but is used to generate correct code at compile time and device configuration at runtime in the Barrelfish research OS.

1 Introduction

The underlying models of system hardware used by both widely-used operating systems like Linux and verified kernels like seL4 [15] or CertiKOS [12] are highly oversimplified. This leads to both sub-optimal design choices and flawed assumptions on which correctness proofs are then based. Both of these systems treat memory as a flat array of bytes, and model translation units (MMUs) in a limited fashion, if at all. This model of the machine dates to the earliest verified-systems projects (and earlier), and does not reflect the reality of modern hardware, in particular systems-on-chip (SoCs) and expansion busses such as PCI.

Early verified CPUs such as CLI's FM9001 [7] do not include anything beyond what would today be described as the CPU core. The later Verisoft VAMP [6] added a cache, but was still extremely simple, even compared to a mobile phone processor of the same era. None of these models attempted to capture the complexity of, for example, the PCI bus, or a multiprocessor NUMA interconnect: both already commonplace by that time. Modern instruction-set models, such as the HOL4 ARM model [10] or the ARM machine-readable specification [19] provide an excellent reference for reasoning about the behaviour of software, but say nothing about the complex interconnects in modern SoCs (which now include essentially all processor chips). No industrial projects [13] appear to claim to have tackled this area.

The weak memory modeling work of Sewell et. al. [4,9], goes deepest, defining the software-visible semantics of memory operations including the effects of pipelining and reordering (e.g. write buffers), but nevertheless only gets us as far as the last-level

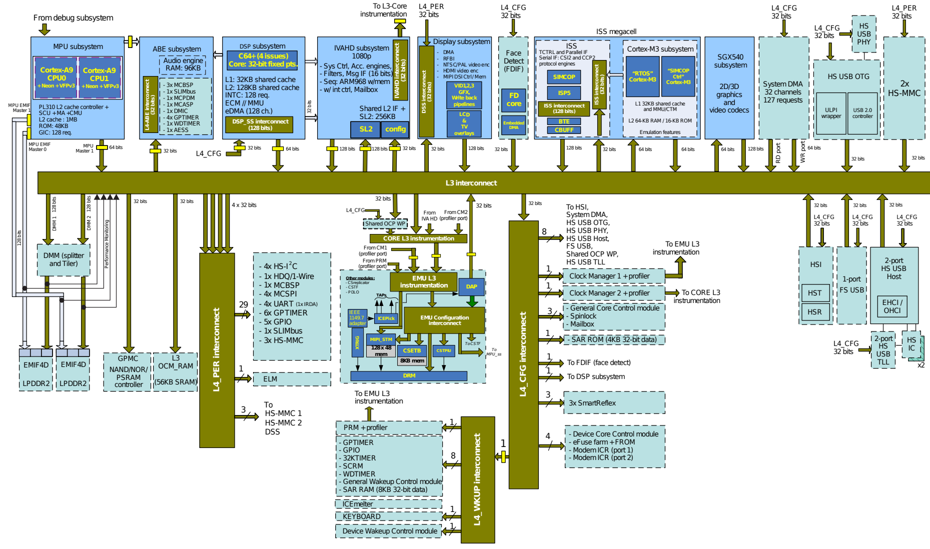

Fig. 1. The OMAP4460 — A 'Simple' SoC (OMAP4460 TRM [21])

cache: once we go beyond that, we're really in the Wild West and, as we demonstrate, the path an address takes from the CPU core to its final destination can be extremely complex (if it ever gets there at all)!

Addressing in a system is semantically far more complex than it first appears. Both Linux and seL4 assume a per-core virtual address space translated, at page granularity via a memory management unit (MMU), to a single global physical address space containing all the random access memory (RAM) and memory-mapped devices in the system. This model, found in many undergraduate textbooks, has been hopelessly inaccurate for some time.

Figure 1 shows the manufacturer's *simplified* block diagram for a 10-year-old mobile phone SoC, the Texas Instruments OMAP4460. Already on this chip we can identify at least 13 distinct interconnects, implementing complex address remapping, and at least 7 cores (not counting DMA-capable devices) each with a different view of physical addresses in the system. In addition, the SoC can be configured such that a memory access enters the same interconnect twice, effectively creating a loop.

Correct OS operation requires software to configure all the address translation and protection elements in this (or any other) platform correctly, and hence formal reasoning about the system requires a model which captures the complexity of addressing. Such a model does not fully exist, but the need is recognized even in the Linux community. The state of the art is DeviceTree [8], essentially a binary file format encoding how a booting OS can configure platform hardware in the absence of device discovery. However, DeviceTree's lack of semantics and narrow focus prevent both reasoning about correctness and runtime use beyond initialization.

As we have shown [1,3,11], systems of various architectures and sizes have no single physical address space, which may have been an illusion since early on. Thus, those

systems are better modeled as a network of address spaces. We therefore introduced a "decoding net" model and demonstrated how it captures a wide variety of complex modern hardware, from the OMAP SoC, to multi-socket Intel Xeon systems with peripheral component interconnect (PCI)-connected accelerators containing general-purpose cores (e.g. a Xeon Phi).

The contribution of this paper is our formal decoding-net model, mechanised in Isabelle/HOL and expanded relative to our previously-published descriptions, particularly in the treatment of possibly-non-terminating decoding loops. We show its utility in seL4-style refinement proofs by modeling the MIPS4600 TLB [14] and demonstrating that the imprecision of its specification prevents any proof of correct initialization.

2 Model

Our goals in formally specifying the addressing behavior of hardware include the highly practical aim of more easily engineering code for a real OS (Barrelfish [5]) which we are confident operates correctly on a diverse range of hardware platforms. Our model (accessible on Github [2]) is therefore a compromise between the simplicity required to provide meaningful abstractions of the system, and the detail needed to capture features of interest and make the model usable in the OS at compile time and run time.

At the same time, the characteristics of the underlying formalism (here Higher-Order Logic), and the kinds of reasoning efficiently supported by the existing tools and libraries (Isabelle/HOL) also influence the choice of model. Specifically, we make limited use of HOL's relatively simple type system (a formalization in Coq would look very different), but exploit Isabelle's extensive automation for reasoning with relations and flexible function definitions.

Our core abstraction is the *qualified name*: An address is a name, defined in the context of some *namespace*, identified by a natural number. As we have previously shown [3], this suffices to model a large number of interesting real-world examples.

In this view a processor's page tables, for example, define a namespace n by mapping names (addresses) *qualified* by the identifier n into another address space n', the "physical" address of the processor. In general, a name may be mapped to any name in any address space (even itself) or to no name at all. As addresses are discrete we also label them with natural numbers, and the translation behavior of an address space is a function:

$$\texttt{translate} : \mathbb{N} \times \mathbb{N} \rightarrow \{\mathbb{N} \times \mathbb{N}\}$$

mapping a fully-qualified name (n, a) (address a in address space n) to some set of names $\{(n', a')\}$ (address a' in space n'). That `translate` returns a *set*, not just an address, allows for nondeterminism and refinement e.g. the possible configurations of a translation unit can be modeled as "any input may map to any output", of which any particular configuration is a refinement. We do not yet use this feature of the model.

This process should end somewhere: any address should (hopefully) eventually refer to some device (e.g. RAM). To distinguish between this and the case where the translation of an address is simply undefined, we add a per-address space *accept set*:

$$\texttt{accept} : \mathbb{N} \rightarrow \{\mathbb{N}\}$$

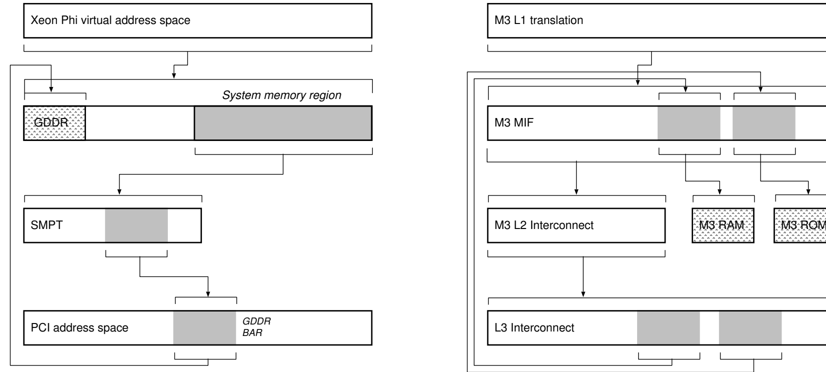

Fig. 2. Existing Loops in Hardware. Xeon Phi left and OMAP 4460 on the right.

Address $a \in \texttt{accept}\ n$ if a is accepted in address space n and thus address resolution *terminates* in address space n. We deliberately allow an address space to have both a non-empty accept set and non-empty translate sets to cover the behavior of e.g. a cache: some addresses may hit in the cache (and thus be accepted locally), while others miss and are passed through.

These two primitives define the entire semantics of the model: everything else is derived from these. The combination implicitly defines a directed graph on qualified names, where every well-defined translation path ends in an accepting set. We explicitly construct the associated (one-step) relation as follows:

$$\texttt{decodes_to} = \{((n, a), (n', a')).\ (n', a') \in \texttt{translate}\ (n, a)\}$$

Likewise the set of all fully-qualified names accepted anywhere in the network:

$$\texttt{accepted_names} = \{(n, a).\ a \in \texttt{accept}\ n\}$$

Finally we define the *net* to be a function from nodeid to node:

$$\texttt{net} : \mathbb{N} \rightarrow node$$

2.1 Views and Termination

One might think the model we have just described is overkill in generality for modeling address spaces. To motivate it, we show some examples of the complexity inherent in address resolution in modern hardware.

On the right of Figure 2 is a subgraph of the interconnect of the OMAP4460 from Figure 1, showing that it is not a tree. In fact, it is not even acyclic: For example, there is both an edge from the Cortex M3 cores (the ISS megacell) to the L3 interconnect, and another from the L3 back to the M3. The system can be configured so that an address issued by the M3 passes through its local address space *twice* before continuing to one of the L4 interconnects. There's no sensible reason to configure the system like this, but we must be able to express the possibility, to verify that initialisation code *doesn't*.

The left of Figure 2 shows a similar situation arising with a PCI-attached Intel Xeon Phi accelerator. Both examples are from our previous work [3], in which much more complex examples are modeled using the formalism presented here.

Thus the absence of true loops is a property we must formally prove from a description of the system, rather than an *a priori* assumption. Importantly, this proof obligation is not manually appended to the model, but (as we will see) arises naturally when attempting to define a functional representation of an address space.

The possibility of loops, and thus undefined translations, is captured by the general `decodes_to` relation above. While faithful, this relational model is not particularly usable: for practical purposes we are more interested in deriving the complete view of the system from a given processor: the eventual accepting set (if any) for each unqualified name in the processor's local name space.

This is expressible via the reflexive, transitive closure of the decoding relation[1] ($R \text{ ` } S$ is here the image of the set S under the relation R).

$$\lambda(n,a).\ \texttt{accepted_names} \cap (\texttt{decodes_to}^* \text{ ` } \{(n,a)\})$$

This is the set of names reachable in 0 or more steps from the root which are accepted by the network. The view from a particular node (the local address space) is then simply the curried function obtained by fixing a particular n.

The model so far is still not quite what we want: we'd like to express the resolution process as a function, preferably with an operational interpretation corresponding (hopefully) meaningfully to the actual hardware behavior. For this we exploit the flexibility of Isabelle's function definition mechanism to separate the simple operational definition of resolution from the more difficult proof of termination:

$$\texttt{resolve}\ (n,a) = \\ (\{(n,a)\} \cap \texttt{accepted_names}) \cup \bigcup \texttt{resolve} \text{ ` } (\texttt{decodes_to} \text{ ` } \{(n,a)\})$$

The resolution of a name is the set containing that name (if it's accepted here), together with the resolutions of all names reachable in one step via the decode relation. With this carefully-chosen definition, the correspondence with the relational model is trivial:

$$\textbf{assumes}\ \texttt{resolve_dom}\ (n,a) \\ \textbf{shows}\ \texttt{resolve}\ (n,a) = \texttt{accepted_names} \cap (\texttt{decodes_to}^* \text{ ` } (n,a))$$

The `resolve_dom` predicate is produced by the Isabelle function definition mechanism thanks to our incomplete definition of the `resolve` function. It asserts that the name n is in the *domain* of the function `resolve` i.e. that the function terminates for this argument (or equivalently that it lies in the reachable part of the recurrence relation). Establishing a sufficient, and significantly a *necessary*, condition for the domain predicate comprises the bulk of the proof effort.

[1] Note that this defines only the decoding *relation* i.e. the set of (name,address) pairs. We only need to show termination once we reformulate it as a recursive function: relations in Isabelle/HOL need only be well-founded if used in a recursive definition (or equivalent).

The general termination proof for `resolve` (i.e. establishing the size of `resolve_dom`) is roughly 500 lines of Isabelle, and consist of establishing a *variant*, or a well-formed ranking of addresses:

$$\begin{aligned}&\texttt{wf_rank}\ f\ (n,a) = \\ &\quad \forall x,y.\ (x,y) \in \texttt{decodes_to} \wedge ((n,a),x) \in \texttt{decodes_to}^* \longrightarrow f(y) < f(x)\end{aligned}$$

From the existence of a well-formed ranking it follows by a straightforward inductive argument that `resolve` terminates. A rather more complex argument shows that if each decoding step produces at most *finitely many* translations of a name (trivially true for any actual hardware), then the converse also holds i.e. we can find a well-formed ranking of names for any terminating resolution. This establishes a precise equivalence between relational and recursive-functional models:

$$\exists f.\ \texttt{wf_rank}\ f\ (n,a) \quad \longleftrightarrow \quad \texttt{resolve_dom}\ (n,a)$$

The argument proceeds by induction over the structure of the decode relation: For any leaf node, finding a well-formed ranking is trivial; if a well-formed ranking exists for all successors, then take the greatest rank assigned to any successor by any of these rank functions (here is where the finite branching condition is required), add one, and assign it to the current node.

2.2 Concrete Syntax, Prolog, and Sockeye

The goal of our work is to model real hardware and verify the algorithms used to configure it in the context of a real operating system. We therefore define a simple concrete syntax for expressing decoding nets:

$$\begin{aligned}
net_s &= \left\{ \mathbb{N}\ \textbf{is}\ node_s \;\middle|\; \mathbb{N}..\mathbb{N}\ \textbf{are}\ node_s \right\} \\
node_s &= \left[\textbf{accept [}\ \left\{block_s\right\}\ \textbf{]}\right]\ \left[\textbf{map [}\ \left\{map_s\right\}\ \textbf{]}\right]\ \left[\textbf{over}\ \mathbb{N}\right] \\
map_s &:= block_s\ \ \textbf{to}\ \mathbb{N}\ \left[\ \textbf{at}\ \mathbb{N}\right]\ \left\{\textbf{,}\ \mathbb{N}\ \left[\ \textbf{at}\ \mathbb{N}\right]\right\} \\
block_s &:= \mathbb{N}\text{–}\mathbb{N}
\end{aligned}$$

The interpretation of a decoding net expressed in this syntax is given by the `parse` function, in the accompanying theory files [2].

We use this syntax in the next section, in showing that important operations on the abstract model can be expressed as simply syntactic translations. It is also the basis for the (much more expressive) Sockeye language [18,20], now used for hardware description and configuration in Barrelfish. Programmers write descriptions of a hardware platform, and the Sockeye compiler generates both HOL and a first-order representation of the decode relation as Prolog assertions.

A set of Prolog inference rules is used to query or transform the model (for example implementing the flattening described above), both offline (for example, to preinitialize kernel page tables for bootstrap) and online (for device driver resource allocation) in the

Barrelfish OS. Representing the model in Prolog allows it to be dynamically populated at run time in response to core and device discovery, while retaining a formal representation. Establishing equivalence with the HOL model (i.e. verifying the Sockeye compiler) should be straightforward, and is an anticipated extension of this work.

2.3 View-Equivalence and Refinement

To use the model we need efficient algorithms to manipulate it. For example, to preinitialize kernel page tables (as now occurs in Barrelfish) we need to know where in the "physical" address space of a particular processor each device of interest appears. This information is implicit in the decoding net model, but not easily accessible.

To build this view, we transform the network in a way that preserves the processor's views while constructing an explicit physical address space for each. We first split every node such that it either accepts addresses (is a resource), or remaps them (is an address space), but not both. Next, we flatten the address space nodes by mapping each input address directly to all names at which it is accepted, i.e. construct the 1-step transitive closure of the decode relation. Eventually, we terminate with a single address space whose translate function maps directly to the resource of interest.

We say that two decoding networks are *view-equivalent*[2], written $(f, \mathit{net}) \sim_S (g, \mathit{net}')$ if all observers (nodes) in S have the same view (i.e. the results of resolve are the same), modulo renaming (f and g) of the accepting nodes. Given some c greater than the label of any extant node, define the `accept` and `translate` functions of the split net, ($\mathtt{accept}'_n$ and $\mathtt{translate}'_n$, for node n) as:

$$\begin{aligned} \mathtt{accept}'_n &= \emptyset \\ \mathtt{accept}'_{(n+c)} &= \mathtt{accept}_n \\ \mathtt{translate}'_n\ a &= \{(n+c, a) : a \in \mathtt{accept}_n\} \cup \mathtt{translate}_n\ a \\ \mathtt{translate}'_{(n+c)}\ a &= \emptyset \end{aligned}$$

This new net is view-equivalent to the original, with names that were accepted at n now accepted at $n + c$, and no node both accepting and translating addresses:

$$(n \mapsto n + c, \mathit{net}) \sim_S (\emptyset, \mathtt{split}(\mathit{net})) \tag{1}$$

Splitting on the concrete representation ($\mathtt{split}_C$) is a simple syntactic operation:

$$n\ \textbf{is accept}\ A\ \textbf{map}\ M \ \mapsto\ [n + c\ \textbf{is accept}\ A,\ n\ \textbf{is map}\ M(n \mapsto n + c)]$$

Refinement (in fact equivalence) is expressed as the commutativity of the operations (here `split` and $\mathtt{split}_C$) with the lifting function (`parse`):

$$\mathtt{split}\ (\mathtt{parse}\ s) = \mathtt{parse}\ (\mathtt{split}_C\ s) \tag{2}$$

Combining Equation 1 with Equation 2 we have the desired result, that the concrete implementation preserves the equivalence of the nets constructed by parsing:

$$(n \mapsto n + c, \mathtt{parse}\ s) \sim_S (\emptyset, \mathtt{parse}\ (\mathtt{split}_C\ s)) \tag{3}$$

[2] See definition `view_eq` in `Equivalence.thy` in the attached sources.

Together with the equivalent result for flattening, we can verify that physical address spaces read directly from the transformed model are exactly those that we would have found by traversing the original hardware-derived model for all addresses.

3 Refinement: Example of the MIPS R4600 TLB

Probably the most complex single translation element in a typical system is the processor's TLB, used to implement the abstraction of Virtual Memory. Translation hardware, such as an MMU, intercepts any load and store to a virtual address and maps it to a physical address, or triggers an exception (page fault). The translation implemented by the MMU is generally under the control of the operating system.

As a demonstration of our decoding net model's ability to capture real hardware, and to support reasoning about it, we present a model of the MIPS R4600 TLB [14]; a well-understood and clearly documented, but still comparatively simple such device.

In this section, we show that the behavior of the TLB can be captured by the decoding net model, that we can use refinement to abstract the behavior of a correctly-configured TLB, and prove that the manufacturer's specification is too vague to allow provably-correct initialization.

3.1 The TLB Model

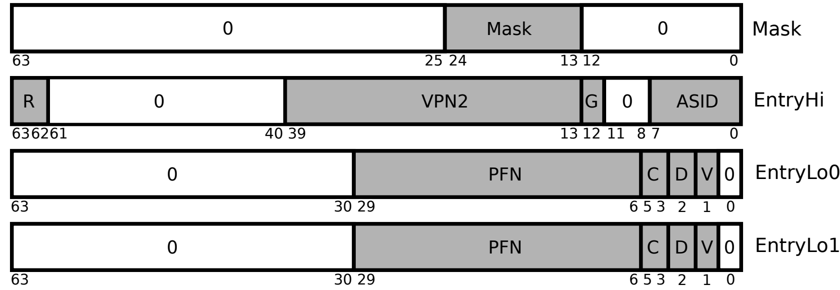

Fig. 3. A MIPS R4600 TLB entry with non-zero fields labelled.

The MIPS TLB is software-loaded: It does not walk page tables in memory for itself, but rather generates a fault whenever a virtual address lookup fails, and relies on the operating system to replace an existing entry with a mapping for the faulting address.

Figure 3 gives the layout of a TLB entry. There are 48 entries, each of which maps two adjacent virtual pages (identified by their virtual page number, or VPN) specified in `EntryHi`, to the physical frames (identified by their physical frame number, or PFN) specified by `EntryLo0` and `EntryLo1`. The TLB (and our model) supports up to seven pre-defined page sizes, but here we consider only the 4kiB case. Physical addresses are 36 bit, while virtual addresses are 40 bit. Addresses are matched against `EntryHi`

i.e. on the VPN and address-space identifier (ASID). An entry with the global bit set matches any ASID. We represent a TLB entry with the following Isabelle record type:

$$\begin{aligned}
\mathit{TLBEntryHi} &= (\text{region} : \mathbb{N},\ \text{vpn2} : \mathbb{N},\ \text{asid} : \mathbb{N}) \\
\mathit{TLBEntryLo} &= (\text{pfn} : \mathbb{N},\ \text{v} : \mathit{bool},\ \text{d} : \mathit{bool},\ \text{g} : \mathit{bool}) \\
\mathit{TLBEntry} &= (\text{hi} : \mathit{TLBEntryHi},\ \text{lo0} : \mathit{TLBEntryLo},\ \text{lo1} : \mathit{TLBEntryLo})
\end{aligned}$$

The TLB consists of an indexed set of entries, and two state terms (wired and random) which we will describe shortly. The capacity of the TLB is static, and is only included here to support our refinement proof.

$$\mathit{MIPSTLB} = (\text{wired} : \mathbb{N},\ \ \text{capacity} : \mathbb{N},\ \ \text{random} : \mathbb{N},\ \ \ \text{entries} : \mathbb{N} \rightarrow \mathit{TLBEntry})$$

All TLB state changes are made by the OS via 4 special instructions:

tlbp *TLB Probe* performs an associative lookup using the contents of the `EntryHi` register, and either returns the matching index or indicates a miss. The result of a multiple match is undefined (this is important).
tlbr *TLB Read* returns any requested TLB entry.
tlbwi *TLB Write Indexed* updates the entry at a user-specified index.
tlbwr *TLB Write Random* updates the entry at the index specified by the Random register, which is updated nondeterministically to some value in $[\text{wired}, \text{capacity})$.

In HOL these are state updates with the following types:

$$\begin{aligned}
&\texttt{tlbp} : \mathit{TLBENTRYHI} \rightarrow \mathit{MIPSTLB} \rightarrow \{\mathbb{N}\} \\
&\texttt{tlbr} : \mathbb{N} \rightarrow \mathit{MIPSTLB} \rightarrow \{\mathit{TLBENTRY}\} \\
&\texttt{tlbwi} : \mathbb{N} \rightarrow \mathit{TLBENTRY} \rightarrow \mathit{MIPSTLB} \rightarrow \{\mathit{MIPSTLB}\} \\
&\texttt{tlbwr} : \mathit{TLBENTRY} \rightarrow \mathit{MIPSTLB} \rightarrow \{\mathit{MIPSTLB}\}
\end{aligned}$$

The outcome of any of these operations may be undefined: Reading or writing an out-of-bounds index and probes that match more than one entry are unpredictable; moreover writing conflicting entries leaves the TLB in an unknown state. Both are modeled as nondeterminism: All operations return a *set* of possible outcomes (UNIV, the universal set being complete underspecification). For example, `tlbwi` returns UNIV for an out-of-bounds index ($i \geq \text{capacity } \mathit{tlb}$), and otherwise updates the specified entry[3], the singleton set indicating that the result is deterministic:

$$\begin{aligned}
\texttt{tlbwi}\ i\ e\ \mathit{tlb}\ =\ &\textbf{if}\ i < \text{capacity}\ \mathit{tlb} \\
&\textbf{then}\ \{\mathit{tlb}\,(\text{entries} := (\text{entries}\ \mathit{tlb})(i := e))\}\ \textbf{else}\ \text{UNIV}
\end{aligned}$$

A TLB random write is then the nondeterministic choice of some indexed write:

$$\texttt{tlbwr}\ e\ \mathit{tlb} = \bigcup_{i=\text{wired}\ \mathit{tlb}}^{(\text{capacity}\ \mathit{tlb})-1} \texttt{tlbwi}\ i\ e\ \mathit{tlb}$$

[3] $f(x := y)$ is Isabelle/HOL syntax for the function f updated at x with value y.

3.2 The Validity Invariant

The MIPS TLB famously permits the programmer to configure the TLB in an unsafe manner, such that the future behavior of the processor is undefined. Indeed in early versions of the chip it was possible to permanently damage the hardware in this way. The source of the problem is in the virtual-address match process: this is implemented using a parallel comparison against all 48 entries. The hardware to implement such an associative lookup is very expensive, and moreover is on the critical path of any memory access. It is therefore highly optimized, taking advantage of the assumption that there will never be more than one match. Violating this assumption leads to two buffers attempting to drive the same wire to different voltages and, eventually, to smoke.

This assumption is exposed as a requirement that the programmer never configure the TLB such that two entries match the same virtual address. Note, the requirement is not just that two entries never *do* actually match a load or store, but that they never *can*[4]. Also note, that a match occurs independently of the value of the valid bit (`V`) and therefore even invalid entries must not overlap. This will shortly become important.

We model the above condition with the following invariant on TLB state:

$$\begin{aligned}\texttt{TLBValid}\ tlb = \texttt{wired}\ \mathit{tlb} \leq \texttt{capacity}\ \mathit{tlb} \wedge \\ \big(\forall i < \texttt{capacity}\ \mathit{tlb}.\ \texttt{TLBEntryWellFormed}\ (\mathit{tlb}\ i) \wedge \\ \texttt{TLBEntryConflictSet}\ (\texttt{entries}\ (\mathit{tlb}\ i))\ \mathit{tlb} \subseteq \{i\}\big)\end{aligned} \tag{4}$$

This predicate states that all entries of the TLB are well formed and do not conflict (overlap) with each other. An entry is well formed if its fields are within valid ranges. We further define the `TLBEntryConflictSet` function:

$$\texttt{TLBEntryConflictSet}\ ::\ \mathit{TLBEntry} \Rightarrow \mathit{MIPSTLB} \Rightarrow \{\mathbb{N}\}$$

This returns the indices of TLB entries that overlap the provided entry. The correctness invariant is thus that either this set is empty, or contains just the entry under consideration (e.g. the one being replaced). The TLB validity invariant is preserved by all 4 primitives e.g.

$$\begin{array}{rll}\textbf{assumes} & \texttt{TLBValid}\ tlb & \textbf{and}\ \texttt{TLBENTRYWellFormed}\ e \\ \textbf{and} & i < \texttt{capacity}\ \mathit{tlb} & \textbf{and}\ \texttt{TLBEntryConflictSet}\ e\ \mathit{tlb} \subseteq \{i\} \\ \textbf{shows} & \forall t \in \texttt{tlbwi}\ i\ e\ \mathit{tlb}.\ \texttt{TLBValid}\ t & \end{array}$$

3.3 Invariant Violation at Power On

After reset (e.g. after power on), it is software's responsibility to ensure that the TLB validity invariant is established. However, the specification of the power-on state of the TLB is sufficiently loose to render this impossible!

[4] We can only speculate as to the writer's intent here. One reason for such a restriction would be speculative execution: The CPU might *speculatively* cause a TLB lookup on an address that it never actually computes. The results would be discarded, but the damage would be done.

The MIPS R4600 manual [14] describes the reset state as follows: "*The Wired register is set to 0 upon system reset.*" The `random` register is set to $\texttt{capacity} - 1$. The state of the TLB entries is undefined: "*The TLB may be in a random state and must not be accessed or referenced until initialized*". As the MIPS TLB is always on (the kernel is provided with a special untranslated region of virtual addresses to solve the bootstrapping problem), and a strict reading of the invariant requires that there are never two matching entries for any address, even if invalid, the unpredictable initial state cannot be guaranteed to satisfy the invariant. We prove this formally by constructing a TLB state that satisfies the reset condition but not the invariant. A plausible initial state is one where all bits are zero (the `null_entry`):

$$(\text{wired} = 0,\ \text{random} = 47, \text{capacity} = 48, \text{entries} = \lambda_.\ \texttt{null_entry})$$

While this TLB does not actually translate anything as the valid bits of all entries are zero, addresses from the first page in memory will match all entries of the TLB. The straightforward reading of the manufacturer's specification requires that such a situation is impossible *even if it doesn't actually occur*.

Of course, in practice, operating systems demonstrably *do* successfully initialize MIPS processors. This indicates that the obvious solution is likely also the correct one: as long as two entries never actually match i.e. no translatable access is issued before the TLB is configured, there's no actual problem. In practice the operating system will execute in the non-translated physical window (KSEG0) until the TLB is configured.

This is an example of a *specification bug*, specifically an excessively cautious abstraction that inadvertently hides a correctness-critical detail. While this case is most likely harmless (and this hardware obsolete), recent experience (such as the Meltdown and Spectre attacks [16,17]) demonstrate that supposedly-invisible behavior hidden by an abstraction can unexpectedly become correctness- or security-critical. Indeed, the mechanism exploited by these attacks (speculative execution), could expose this invariant violation: even if no kernel code actually accesses a translatable address, the processor is free to *speculate* such an access, thus triggering the failure.

3.4 What Does a Fully-Wired TLB Do?

The entries of the MIPS TLB can be *wired* (see Figure 4). The lower w entries are protected from being overwritten by the random write operation (`tlbwr`). The manual states that *"Wired entries are non-replaceable entries, which cannot be overwritten by a TLB write random operation."* The number of wired entries w can be configured.

The random write operation uses the `random` register to select the entry to be overwritten. This register is initialized to $\texttt{capacity} - 1$ at reset and is decremented whenever the processor retires an instruction, skipping entries $w - 1$ down to 0. As its reset value is $\texttt{capacity} - 1$, the random write operation will always succeed regardless of the current value of value w. We can express the bounds as:

$$\texttt{RandomRange}\ tlb = \{x.\ \texttt{wired}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\}$$

This definition is problematic when we wire all entries of the TLB i.e. by setting w to `capacity`. Note, hardware does not prevent wiring more entries than the capacity.

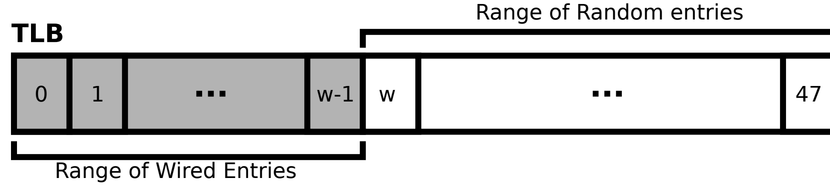

Fig. 4. Wired TLB Entries

The manual does not mention this case at all. Assuming that $w = \texttt{capacity}$ we obtain $\texttt{RandomRange}\ tlb = \{x.\ \texttt{capacity}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\} = \{\}$. This suggests, that no entries will be replaced randomly as intended. However, we know that upon reset the random register is set to $\texttt{capacity} - 1$ which is not in the (empty) RandomRange set. This contradicts the specification of either the semantics of the wired entries or the random write instruction. We therefore express the random range as follows:

$$\texttt{RandomRange}\ tlb = \{x.\ \texttt{wired}\ tlb \leq x \wedge x < \texttt{capacity}\ tlb\} \cup \{\texttt{capacity}\ tlb - 1\}$$

3.5 The TLB Refines a Decoding Net

The preceding specification bugs notwithstanding, we can nevertheless use the TLB model to do useful work. In the remainder of this section we first show that with an appropriate lifting function, our operational model of the TLB refines a decoding-net model of a translate-only node, such that the `tlbwi` operation corresponds to simply updating the appropriate translation. Finally, in section 3.6 through section 3.9 we model the action of TLB refill handler and show that, combined with a valid page table and the operational TLB model, that its action is indistinguishable (again under decoding-net semantics) from that of a TLB large enough to hold all translation entries at once (i.e. no TLB miss exceptions).

We lift a single TLB entry to a pair of address-range mappings as follows:

$$\begin{aligned}
&\texttt{EntryToMap} : \textit{nodeid} \Rightarrow \texttt{TLBENTRY} \Rightarrow \textit{addr} \Rightarrow \{\textit{name}\}\\
&\texttt{EntryToMap}\ n\ e\ va =\\
&\quad (\textbf{if}\ \texttt{EntryIsValid0}\ e \wedge va \in \texttt{EntryExtendedRange0}\ e\\
&\quad\quad \textbf{then}\ \{(n, \texttt{EntryPA0}\ e + (va \bmod \textit{VASize}) - \texttt{EntryMinVA0}\ e)\}\ \textbf{else}\ \{\}) \cup\\
&\quad (\textbf{if}\ \texttt{EntryIsValid1}\ e \wedge\ va \in \texttt{EntryExtendedRange1}\ e\\
&\quad\quad \textbf{then}\ \{(n, \texttt{EntryPA1}\ e + (va \bmod \textit{VASize}) - \texttt{EntryMinVA1}\ e)\}\ \textbf{else}\ \{\})
\end{aligned}$$

The `EntryExtendedRange(0,1)` functions consider the virtual address, the address-space identifier and the global bit, by extending the virtual address with the ASID bits such that the *extended virtual address* space contains all virtual addresses for ASID 0, followed by those for ASID 1, and so forth.

The TLB's representation is then the union of these translations:

$$\texttt{ConvertToNode}\ n\ tlb =$$
$$\Big(accept = \{\},\ translate = \lambda a. \bigcup \texttt{EntryToMap}\ n\ (\texttt{entries}\ tlb\ i)\ a\Big)$$

The abstract equivalent of `tlbwi` is the `replace_entry` function, which replaces entry $e1$ with $e2$ by updating the translation as follows:

$$\texttt{translate}\ n\ a \mapsto (\texttt{translate}\ n\ a - \texttt{EntryToMap}\ n\ e1\ a) \cup \texttt{EntryToMap}\ n\ e2\ a$$

The following lemma shows the equivalence of the `tlbwi` instruction and the TLB indexed write (`replace_entry`) function, i.e. that commuting with the lifting function maps one to the other:

assumes $i < \texttt{capacity}\ tlb$ **and** $\texttt{TLBValid}\ tlb$ **and** $\texttt{TLBEntryWriteable}\ i\ e\ tlb$
shows $(\texttt{ConvertToNode}\ n)'(\texttt{tlbwi}\ i\ e\ tlb) =$
$\texttt{replace_entry}\ n\ (\texttt{entries}\ tlb\ i)\ e\ (\texttt{ConvertToNode}\ n\ tlb)$

3.6 Modeling TLB Lookups and Exceptions

An MMU provides the illusion of a large virtual address space, using a small TLB, by loading entries on demand from a large in-memory translation table. On the MIPS, this is handled in software, according to the exception flowchart in Figure 4-19 of the MIPS manual [14]. The following three exceptions are defined:

TLB Refill No entry matched the given virtual address.
TLB Invalid An entry matched, but was invalid.
TLB Modified Access violation e.g. a write to a read-only page.

Match	Valid	Entry writable / Memory write	VPN even	Result
No	*	*	*	TLB Refill Exception
Yes	No	*	*	TLB Invalid Exception
Yes	No	No and memory write	*	TLB Modified Exception
Yes	Yes	Yes or memory read	Yes	Translate using EntryLo0
Yes	Yes	Yes or memory read	No	Translate using EntryLo1

Table 1. The outcome of the translate function.

The possible outcomes of a TLB lookup are summarized in Table 1, and modeled (for a particular entry) in our nondeterministic operational style as follows:

$$
\begin{aligned}
&\texttt{TLBENTRY_translate}\ e\ as\ vpn = \\
&\quad \textbf{if}\ \texttt{EntryMatchVPNASID}\ vpn\ as\ e\ \textbf{then} \\
&\quad\quad \textbf{if}\ \texttt{even}\ vpn\ \wedge \texttt{EntryIsValid0}\ e \\
&\quad\quad\quad \textbf{then}\ \{(\texttt{pfn}\ (lo0\ e)) + (vpn - \texttt{EntryMin4KVPN}\ e)\} \\
&\quad\quad \textbf{else}\ \textbf{if}\ \texttt{odd}\ vpn \wedge \texttt{EntryIsValid1}\ e \\
&\quad\quad\quad \textbf{then}\ \{(\texttt{pfn}\ (lo1\ e)) + (vpn - \texttt{EntryMin4KVPN1}\ e)\} \quad \textbf{else}\ \{\} \\
&\quad \textbf{else}\ \{\}
\end{aligned}
$$

Again exploiting nondeterminism, we define `MIPSTLB_translate` as follows:

$$
\begin{aligned}
&\texttt{MIPSTLB_translate}\ tlb\ vpn\ as = \\
&\qquad\qquad \bigcup_{i < \texttt{capacity}\ tlb} \texttt{TLBENTRY_translate}\ ((\texttt{entries}\ tlb)\ i)\ as\ vpn
\end{aligned}
$$

The TLB invariant (Equation 4) implies that at most one entry will match, and thus the union is trivial.

3.7 Adding a Page Table

With a software-loaded TLB, the OS programmer is free to select any data structure for the page tables. The simplest, and a very common, choice is an array of `TLBEntryLo` values, indexed by address space and virtual page number:

$$
\mathit{MIPSPT} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathit{TLBENTRYLO}
$$

The replacement handler must then simply load the entry corresponding to the faulting address (if any) and restart the faulting process. In order to guarantee that the TLB invariant is maintained, we show that the following invariant holds of the page table, which simply applies the invariant to all elements of the in-memory representation:

$$
\begin{aligned}
&\textbf{assumes}\ \texttt{MIPSPT_valid}\ pt\ \textbf{and}\ \texttt{ASIDValid}\ as\ \textbf{and}\ vpn < \mathit{MIPSPT_EntriesMax} \\
&\quad \textbf{shows}\ \texttt{TLBENTRYWellFormed}\ (\texttt{MIPSPT_mk_tlbentry}\ pt\ as\ vpn)
\end{aligned}
$$

3.8 Modeling Replacement Handlers

Combining the page table representation above with the TLB, we can model a replacement handler that “caches” translations from the page table in the TLB:

$$
MipsTLBPT = \big(tlb : \mathit{MIPSTLB}, \quad pte : \mathit{MIPSPT}\big)
$$

The replacement handler writes entries constructed from the page table into the TLB. The TLB should thus always be an "instance" of (hold a subset of entries from) the page table:

$$\begin{aligned}\texttt{MipsTLBPT_is_instance}\ mt = \forall i < \texttt{capacity}\ (\texttt{tlb}\ mt).\\ \texttt{entries}\ (\texttt{tlb}\ mt)\ i = \texttt{MIPSPT_mk_tlbentry}\ (\texttt{pte}\ mt)\\ (\texttt{asid}\ (\texttt{hi}\ (\texttt{entries}\ (\texttt{tlb}\ mt)\ i)))\\ (\texttt{vpn2}\ (\texttt{hi}\ (\texttt{entries}\ (\texttt{tlb}\ mt)\ i)))\end{aligned}$$

This predicate ensures there are no other entries in the TLB than those constructed from the page table—a property we will use later when we show equivalence to a large TLB.

The replacement handler can either replace an entry deterministically by choosing the index as a function of the entry's VPN:

$$\texttt{MIPSTLBIndex}\ tlb\ entry\ =\ (\texttt{vpn2}\ (\texttt{hi}\ entry))\ mod\ (\texttt{capacity}\ tlb)$$

or make use of the nondeterministic random write function. The deterministic update function implements a direct mapped replacement policy i.e. an entry can only ever be present in a well defined slot which simplifies reasoning about the TLB invariant.

However, this placement policy is not applicable in general e.g. when the OS wants to divide the entries into wired and random (section 3.4) or in the presence of hardware table walkers and associative TLBs that non-deterministically replace an entry. Hence, the location of the entry in the TLB is no longer fixed.

In the non-deterministic model of the replacement handler, we make sure that we only ever change the state of the TLB when a translation attempt would trigger a *refill* exception:

$$\begin{aligned}&\texttt{MipsTLBPT_fault}\ mtlb\ as\ vpn =\\ &\quad \textbf{if}\ \texttt{MIPSTLB_try_translate}\ (\texttt{tlb}\ mtlb)\ as\ vpn\ =\ \mathit{EXNREFILL}\\ &\quad\ \textbf{then}\quad \texttt{MipsTLBPT_update_tlb}\ mtlb\ as\ vpn\quad \textbf{else}\quad \{mtlb\}\end{aligned}$$

Therefore, when we construct the new entry from the page table and update the TLB by replacing an existing entry with the new one, we are guaranteed not to cause and conflicts. Hence, we prove that `MipsTLBPT_fault` preserves the TLB invariance:

$$\begin{aligned}\textbf{assumes}\ &\texttt{MipsTLBPT_valid}\ mpt\quad \textbf{and}\quad \texttt{ASIDValid}\ as\\ \textbf{and}\ &vpn < \texttt{MIPSPT_EntriesMax}\\ \textbf{shows}\ &\forall m \in \texttt{MipsTLBPT_fault}\ mpt\ as\ vpn.\ \texttt{MipsTLBPT_valid}\ m\end{aligned}$$

Note, we use a stricter definition of validity in this case requiring also the *is_instance* predicate and that the page tables are well formed.

$$\begin{aligned}\texttt{MipsTLBPT_valid}\ mt =&\texttt{MIPSPT_valid}\ (\texttt{pte}\ mt) \wedge \texttt{TLBValid}\ (\texttt{tlb}\ mt)\\ &\wedge \texttt{MipsTLBPT_is_instance}\ mt\end{aligned}$$

3.9 Equivalence to Infinitely Large TLB

The final result regarding the TLB is to show that, together with the refill handler, it implements the expected abstraction: a single decoding-net node, that maps the virtual address space to the physical. We do this by showing that the TLB plus refill handler is equivalent, in the decoding-net semantics, to a hypothetical giant TLB, large enough to hold all mappings at once, and that therefore never faults.

We construct the large TLB by pre-loading all entries from the page table, according to our "extended virtual address" scheme, giving a unique, deterministic location for each entry:

$$\begin{aligned}
&\texttt{MipsTLBLarge_create}\ pt = \\
&\quad \big(capacity = \texttt{MaxEntries}, \quad wired = \texttt{MaxEntries}, \\
&\quad entries = \lambda n.\ \texttt{MIPSPT_mk_tlbentry}\ pt\ (\texttt{idx2asid}\ n)\ (\texttt{idx2vpn}\ n)\big)
\end{aligned}$$

We first define a translate function $TLB \Rightarrow ASID \Rightarrow VPN \Rightarrow PFN$ for both, the large TLB and the TLB with replacement handler, and we show that the two are equivalent for any sensible VPN and ASID:

$$\begin{aligned}
&\textbf{assumes}\ \ vpn < \textit{MIPSPT_EntriesMax}\ \textbf{and}\ as < \textit{ASIDMax} \\
&\quad \textbf{and}\ \texttt{capacity}\ (\texttt{tlb}\ mpt) > 0\ \textbf{and}\ \texttt{MipsTLBPT_valid}\ mpt \\
&\quad \textbf{shows}\ \texttt{MipsTLBPT_translate}\ mpt\ as\ vpn = \\
&\qquad \texttt{MIPSTLB_translate}\ (\texttt{MipsTLBLarge_create}(\texttt{pte}\ mpt))\ as\ vpn
\end{aligned} \tag{5}$$

We use the translate functions and the equivalence result above when lifting the large TLB and the TLB with replacement handler to the decoding net node. Here we show the variant for the real TLB:

$$\begin{aligned}
&\texttt{MipsTLBPT_to_node}\ nid\ mpt = \\
&\quad \big(accept = \{\}, translate = (\lambda a.(\textbf{if}\ \texttt{AddrValid}\ a\ \textbf{then} \\
&\qquad (\bigcup x \in (\texttt{MipsTLBPT_translate}\ mpt\ (\texttt{addr2asid}\ a)(\texttt{addr2vpn}\ a)). \\
&\qquad\quad \{(nid, \texttt{pfn2addr}\ x\ a)\})\ \textbf{else}\ \{\}))\big)
\end{aligned}$$

The accept set is empty. The node's translate function checks whether the address falls within defined range (AddrValid) and then either return an empty set or the result of the TLB's translate function. Note, we need to convert between addresses and VPN/ASID and PFN.

Lastly, we prove the equivalence of the lifting functions and their result when applied to the large TLB and the TLB with replacement handler respectively:

$$\begin{aligned}
&\textbf{assumes}\ \texttt{capacity}\ (\texttt{tlb}\ mpt) > 0 \quad \textbf{and} \quad \texttt{MipsTLBPT_valid}\ mpt \\
&\quad \textbf{shows}\ \texttt{MipsTLBPT_to_node}\ nid\ mpt = \\
&\qquad \texttt{MIPSLARGE_to_node}\ nid\ (\texttt{MipsTLBLarge_create}\ (\texttt{pte}\ mpt))
\end{aligned}$$

Recall, we have shown that the translate function of the two TLB's have identical behavior (Equation 5) if the large TLB was pre-populated with the same page tables. Therefore, applying the lifting functions produces equivalent nodes in decoding net semantics.

4 Conclusion

In this paper, we have demonstrated the use of Isabelle/HOL to formally model and reason about the increasingly complex process of address resolution and mapping in modern processors. The traditional model of a single virtual address space, mapped onto a global physical address space has been a gross oversimplification for a long time, and this is becoming more and more visible.

Our decoding-net model, and the Sockeye language that we have developed from it, present an semantically rigorous and formally-verified alternative to Device Trees. In our prior work, we have demonstrated that this model can be applied to a wide range of very complex real hardware, and in this paper we further demonstrate its application to modelling the MIPS TLB. That we were able to prove that this correctly implements the virtual-memory abstraction shows not just that this particular hardware is indeed correct, but that our model is tractable for such proofs. That we discovered a number of specification bugs demonstrates clearly the benefit of a rigorous formal proof, and is further evidence in favor of the formal specification of the semantics of hardware.

The Sockeye language is already in use in the Barrelfish operating system, and we anticipate verifying the Sockeye compiler, in particular that the Prolog assertions generated are equivalent to the HOL model. Further, the model as it stands is principally a static one: expressing the configuration space of the system in a more systematic manner than simply allowing general functions (for example to model region-based remapping units, as used in PCI), and reasoning about the dynamic behavior of requests as translations are updated (for example that a series of updates to different translation units never leaves the system in an intermediate state that violates invariants) is an exciting future direction that we intend to explore. Likewise modelling request properties (read, write, cacheable, etc.), and their interaction with existing weak memory models, presents a challenge.

The ultimate prize is to model the memory system in sufficient detail to be able to specify the behavior of a system including partly-coherent caches (such as ARM) and table-walking MMUs that themselves load page table entries via the cache and/or second-level translations (as in two-level paging for virtualization). This goal is still a long way off, but the increasing quality and availability of formal hardware models leaves us hope that it is indeed attainable.

References

1. R. Achermann. Message Passing and Bulk Transport on Heterogenous Multiprocessors. Master's thesis, Department of Computer Science, ETH Zurich, Switzerland., 2017.
2. R. Achermann, D. Cock, and L. Humebl. Hardware Models in Isabelle/HOL. Online. `https://github.com/BarrelfishOS/Isabelle-hardware-models`, January 2018.
3. R. Achermann, L. Humbel, D. Cock, and T. Roscoe. Formalizing Memory Accesses and Interrupts. In *Proceedings of the 2nd Workshop on Models for Formal Analysis of Real Systems*, MARS 2017, pages 66–116, 2017.
4. J. Alglave. A Formal Hierarchy of Weak Memory Models. *Form. Methods Syst. Des.*, 41(2):178–210, Oct. 2012.
5. The Barrelfish Operating System. Online. `https://www.barrelfish.org`.

6. S. Beyer, , C. Jacobi, D. Kröning, D. Leinenbach, and W. J. Paul. Putting it all together — Formal verification of the VAMP. *Int. J. Softw. Tools Technol. Transfer*, 8(4):411–430, Aug 2006.
7. M. K. Bishop C. Brock, Warren A. Hunt. The FM9001 Microprocessor Proof. Technical Report 86, Computational Logic, Inc., 1994.
8. devicetree.org. *Devicetree Specification*, May 2016. Release 0.1, Online. `http://www.devicetree.org/specifications-pdf`.
9. S. Flur, K. E. Gray, C. Pulte, S. Sarkar, A. Sezgin, L. Maranget, W. Deacon, and P. Sewell. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'16, pages 608–621, St. Petersburg, FL, USA, 2016. ACM.
10. A. Fox and M. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In M. Kaufmann and L. Paulson, editors, *ITP*, pages 243–258, Berlin, Heidelberg, 2010. Springer.
11. S. Gerber, G. Zellweger, R. Achermann, K. Kourtis, T. Roscoe, and D. Milojicic. Not Your Parents' Physical Address Space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 16–16, 2015.
12. R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Savannah, GA, USA, 2016. USENIX Association.
13. W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova. Industrial hardware and software verification with ACL2. *Phil. Trans. R. Soc. A*, 375(2104), 2017.
14. Integrated Device Technology, Inc. *IDT79R4600 TM and IDT79R4700 TM RISC Processor Hardware User's Manual*, revision 2.0 edition, April 1995.
15. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, Big Sky, Montana, USA, 2009. ACM.
16. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, Jan. 2018.
17. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.
18. T. B. Project. *Sockeye in Barrelfish*.
19. A. Reid. Trustworthy Specifications of ARM V8-A and V8-M System Level Architecture. In *FMCAD'16*, pages 161–168, Austin, TX, 2016. FMCAD Inc.
20. D. Schwyn. Hardware Configuration With Dynamically-Queried Formal Models. Master's thesis, Department of Computer Science, ETH Zurich, Switzerland., 2017.
21. Texas Instruments. *OMAP44xx Multimedia Device Technical Reference Manual*, April 2014. Version AB, `www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf`.