Compositional Verification of a Baby Virtual Memory Manager

Alexander Vaynberg and Zhong Shao

Yale University

Abstract. A virtual memory manager (VMM) is a part of an operating system that provides the rest of the kernel with an abstract model of memory. Although small in size, it involves complicated and interdependent invariants that make monolithic verification of the VMM and the kernel running on top of it difficult. In this paper, we make the observation that a VMM is constructed in layers: physical page allocation, page table drivers, address space API, etc., each layer providing an abstraction that the next layer utilizes. We use this layering to simplify the verification of individual modules of VMM and then to link them together by composing a series of small refinements. The compositional verification also supports function calls from less abstract layers into more abstract ones, allowing us to simplify the verification of initialization functions as well. To facilitate such compositional verification, we develop a framework that assists in creation of verification systems for each layer and refinements between the layers. Using this framework, we have produced a certification of BabyVMM, a small VMM designed for simplified hardware. The same proof also shows that a certified kernel using BabyVMM's virtual memory abstraction can be refined following a similar sequence of refinements, and can then be safely linked with BabyVMM. Both the verification framework and the entire certification of BabyVMM have been mechanized in the Coq Proof Assistant.

1 Introduction

Software systems are complex feats of engineering. What makes them possible is the ability to isolate and abstract modules of the system. In this paper, we consider an operating system kernel that uses virtual memory. The majority of the kernel makes an assumption that the memory is a large space with virtual addresses and a specific interface that allows the kernel to request access to any particular page in this large space. In reality, this entire model of memory is in the imagination of the programmer, supported by a relatively small but important portion of the kernel called the virtual memory manager. The job of the virtual memory manager is to handle all the complexities of the real machine architecture to provide the primitives that the rest of the kernel can use. This is exactly how the programmer would reason about this software system.

However, when we consider verification of such code, current approaches are mostly monolithic in nature. Abstraction is generally limited to abstract data types, but such abstraction can not capture changes in the semantics of computation. For example, it is impossible to use abstract data types to make virtual memory appear to work like physical memory without changing operational semantics. To create such abstraction, a

change of computational model is required. In the Verisoft project[11, 18], the abstract virtual memory is defined by creating the CVM model from VAMP architecture. In AIM[7], multiple machines are used to define interrupts in the presence of a scheduler.

These transitions to more abstract models of computation tend to be quite rare, and when present tend to be complex. The previously mentioned VAMP-CVM jump in Verisoft abstracts most of kernel functionality in one step. In our opinion, it would be better to have more abstract computation models, with smaller jumps in abstraction. First, it is easier to verify code in the most abstract computational model possible. Second, smaller abstractions tend to be easier to prove and to maintain, while larger abstractions can be still achieved by composing the smaller ones. Third, more abstractions means more modularity; changes in the internals of one module will not have global effects.

However, we do not commonly see Hoare-logic verification that encourages multiple models. The likely reason is that creating abstract models and linking across them is seen as ad-hoc and tedious additional work. In this paper we show how to reduce the effort required to define models and linking, so that code verification using multiple abstractions becomes an effective approach. More precisely, our paper makes the following contributions:

- We present a framework for quickly defining multiple abstract computational models and their verification systems.
- We show how our framework can be used to define safe cross-abstraction linking.
- We show how to modularize a virtual memory manager and define abstract computational models for each layer of VMM.
- We show a complete verification of a small proof-of-concept virtual memory manager using the Coq Proof Assistant.

The rest of this paper is organized as follows. In Section 2, we give an informal overview of our work. In Section 3, we discuss the formal details of our verification and refinement framework. In Section 4, we specialize the framework for a simple C-like language. In Section 5, we certify BabyVMM, our small virtual memory manager. Section 6 discusses the Coq proof, and Section 7 presents related work and concludes.

2 Overview and Plan for Certification

We begin the overview by explaining the design of BabyVMM, our small virtual memory manager. First, consider the model of memory present in simplified hardware (left side of Figure 1). The memory is a storage system, which contains cells that can be read from or written to by the software. These cells are indexed by addresses. However, to facilitate indirection, the hardware includes a system called address translation (AT), which, when enabled, will cause all requests for specific addresses from the software to be translated. The AT system adds special registers to the memory system - one to enable or disable AT, and the other to point where the software-managed AT tables are located in memory. The fact that these tables are stored in memory is one of the sources of complexity in the AT system - updating AT tables requires updating in-memory tables, a process which goes through AT as well.

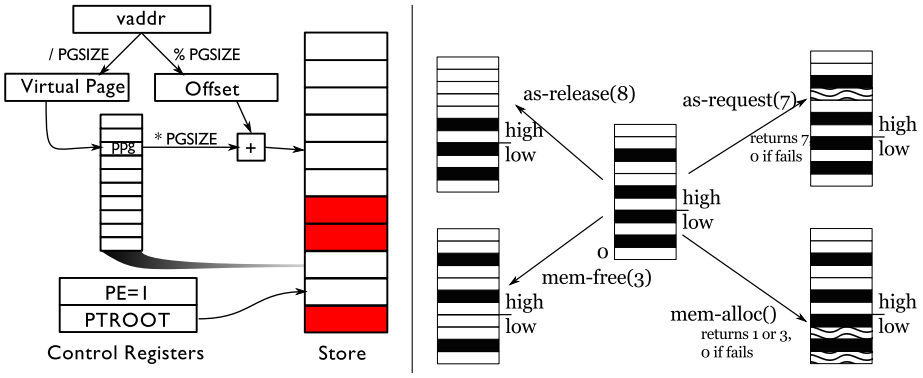

Fig. 1. Hardware (HW) and Address Space (AS) Models of Memory

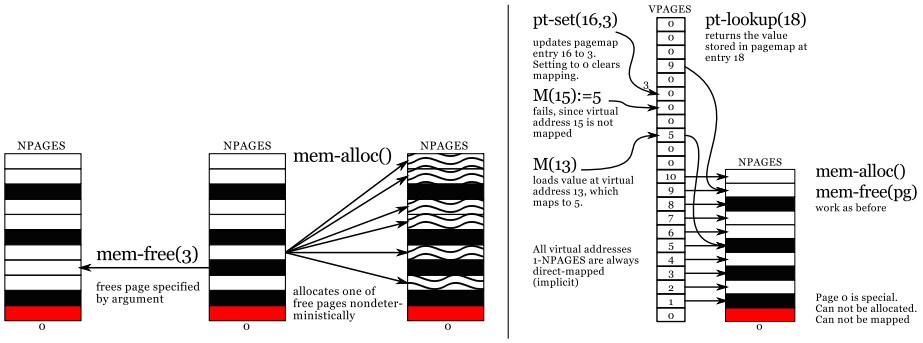

Fig. 2. Allocated (ALE) and Page Map (PMAP) Models of Memory

Because AT is such a complicated, machine-dependent, and general mechanism, BabyVMM creates an abstraction that defines specific restrictions on how AT will be used, and presents a simpler view of AT to the kernel. Although the abstract models of memory may differ depending on the features that the kernel may require, BabyVMM defines a very basic model, to which we refer as the address space (AS) model of memory (right side of Figure 1). The AS model replaces the small physical memory with a larger virtual address space with allocatable pages and no address translation. The space is divided into high and low areas, where the low area is actually a window into physical memory (a pattern common in many kernels). Because of this distinction, the memory model has two sets of allocation functions, one for the "high" memory area where the programmer requests a specific page for allocation, and one for the "low" memory area, where the programmer can not pick which page to allocate.

However, creating an abstraction that makes the jump from the HW model directly to AS model is complex. As a result, we create two more intermediate models, which slowly build up the abstraction. The first model is ALE (left side of Figure 2), which incorporates allocation information into the hardware memory, requiring that programs only access memory locations that are marked allocated. The model adds primitives in the form of `mem_alloc` and `mem_free`, with semantics same as the ones in the AS

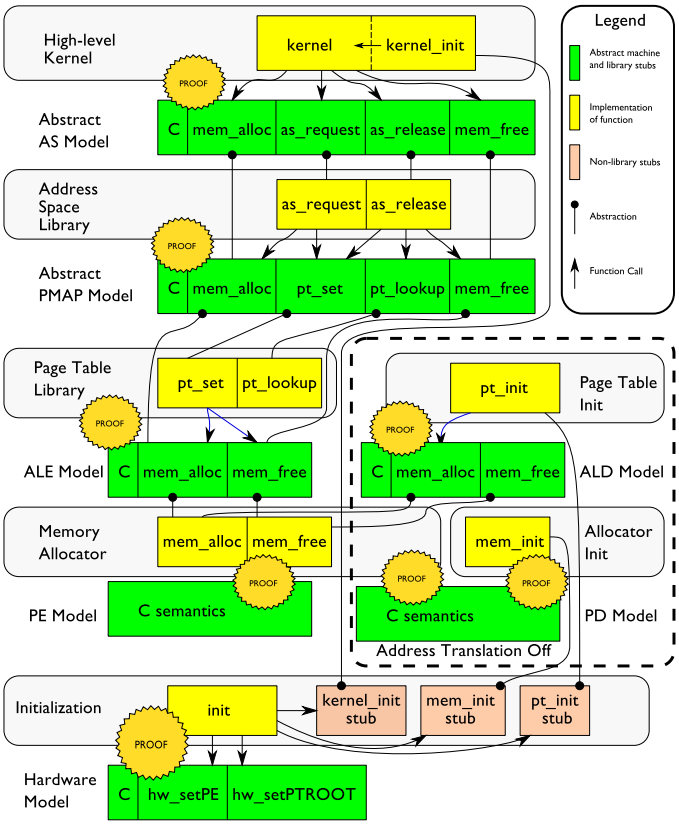

Fig. 3. Complete Plan for VMM Certification

model. Although this is not shown on the diagram, the ALE model still maintains the hardware's AT mechanism.

The second intermediate level, which we call PMAP (right side of Figure 2) is designed to replace the hardware's AT mechanism with an abstract one. The model features a page map that exists outside the normal memory space, unlike the lower level models. The page map maps virtual page numbers to physical page numbers, with a 0 value meaning invalid. In our particular design, the pagemap is always identity for the lower addresses, creating a window into physical memory from within the virtual space. The model still contains allocation primitives, and adds two more primitives, `pt_set` and `pt_lookup`, which update and lookup values in the pagemap.

Using these abstract memory models, we can construct the BabyVMM verification plan (Figure 3). The light-yellow boxes in the kernel represent the actual functions (actual code is given in Appendix A of TR[19]). The darker green boxes represent computational models with primitives labeled. The diagram shows how each module of BabyVMM will be certified in the model best suited for it. For example, the high-level kernel is certified in the AS model, meaning that it does not see underlying physical memory at all. The implementation of `as_request` and `as_release` are defined over

$$
\begin{array}{rlrl}
(\textit{State})\ \mathbb{S} & \in \Sigma & (\textit{State Predicate})\ \mathtt{p} & \in \Sigma \to \textit{Prop} \\
(\textit{Operation})\ \iota & \in \Delta & (\textit{State Relation})\ \mathtt{g} & \in \Sigma \to \Sigma \to \textit{Prop} \\
(\textit{Cond})\ b & \in \beta & (\textit{Operational Semantics})\ \mathtt{OS} & \in \{\iota \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* \\
(\textit{CondInterp})\ \Upsilon & \in \beta \to \Sigma \to \textit{Prop} & (\textit{Language / Machine})\ \mathcal{M} & \in (\Sigma,\Delta,\beta,\Upsilon,\mathtt{OS})
\end{array}
$$

where $\mathcal{M}(\iota) \triangleq \mathcal{M}.\mathtt{OS}(\iota)$ and $\mathcal{M}(b) \triangleq \mathcal{M}.\Upsilon(b)$

Fig. 4. Abstract State Machine

$$
\begin{array}{ll}
\textit{id} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True},\quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathbb{S}' = \mathbb{S}) \\
\textit{fail} & \triangleq (\lambda\mathbb{S}.\,\mathsf{False},\quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
\textit{loop} & \triangleq (\lambda\mathbb{S}.\,\mathsf{True},\quad \lambda\mathbb{S}.\lambda\mathbb{S}'.\,\mathsf{False}) \\
(\mathtt{p},\mathtt{g}) \circ (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,\mathtt{p}\ \mathbb{S} \wedge \forall\mathbb{S}'.\,\mathtt{g}\ \mathbb{S}\ \mathbb{S}' \to \mathtt{p}'\ \mathbb{S}',\quad \lambda\mathbb{S}.\lambda\mathbb{S}''.\,\exists\mathbb{S}'.\,\mathtt{g}\ \mathbb{S}\ \mathbb{S}' \wedge \mathtt{g}'\ \mathbb{S}'\ \mathbb{S}'') \\
(\mathtt{p},\mathtt{g}) \underset{c}{\oplus} (\mathtt{p}',\mathtt{g}') & \triangleq (\lambda\mathbb{S}.\,(\mathtt{p}\ \mathbb{S} \wedge c\ \mathbb{S}) \vee (\mathtt{p}'\ \mathbb{S} \wedge \neg c\ \mathbb{S}), \lambda\mathbb{S}.\lambda\mathbb{S}'.\,(c\ \mathbb{S} \wedge \mathtt{g}\ \mathbb{S}\ \mathbb{S}') \vee (\neg c\ \mathbb{S} \wedge \mathtt{g}'\ \mathbb{S}\ \mathbb{S}')) \\
(\mathtt{p},\mathtt{g}) \sqsupseteq (\mathtt{p}',\mathtt{g}') & \triangleq \forall\mathbb{S}.\,\mathtt{p}\ \mathbb{S} \to \mathtt{p}'\ \mathbb{S} \wedge \forall\mathbb{S},\mathbb{S}'.\,\mathtt{g}'\ \mathbb{S}\ \mathbb{S}' \to \mathtt{g}\ \mathbb{S}\ \mathbb{S}'
\end{array}
$$

Fig. 5. Combinators and Properties of Actions

$$
\begin{array}{rlll}
(\textit{Meta-program}) & \mathbb{P} & ::= (\mathbb{C},\mathbb{I}) & [\![\mathbb{C},\mathtt{a}]\!]^0_{\mathcal{M}} := \textit{loop} \\
(\textit{Proc}) & \mathbb{I} & ::= \mathtt{nil} \mid \iota \mid [\mathtt{l}] \mid \mathbb{I}_1;\mathbb{I}_2 & [\![\mathbb{C},\mathtt{nil}]\!]^n_{\mathcal{M}} := \textit{id} \\
 & & \quad \mid (b?\ \mathbb{I}_1 + \mathbb{I}_2) & [\![\mathbb{C},\iota]\!]^n_{\mathcal{M}} := (\mathcal{M}(\iota)) \\
(\textit{Proc Heap}) & \mathbb{C} & ::= \{\mathtt{l} \rightsquigarrow \mathbb{I}\}^* & [\![\mathbb{C},[\mathtt{l}]]\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{C}(\mathtt{l})]\!]^{n-1}_{\mathcal{M}} \\
(\textit{Labels}) & \mathtt{l} & ::= n \text{ (nat numbers)} & [\![\mathbb{C},\mathbb{I};\mathbb{I}']\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \circ [\![\mathbb{C},\mathbb{I}]\!]^n_{\mathcal{M}} \\
(\textit{Spec Heap}) & \Psi,\mathcal{L} & ::= \{\mathtt{l} \rightsquigarrow (\mathtt{p},\mathtt{g})\}^* & [\![\mathbb{C},(b?\ \mathbb{I}_1+\mathbb{I}_2)]\!]^n_{\mathcal{M}} := [\![\mathbb{C},\mathbb{I}_1]\!]^n_{\mathcal{M}} \underset{\mathcal{M}(b)}{\oplus} [\![\mathbb{C},\mathbb{I}_2]\!]^n_{\mathcal{M}}
\end{array}
$$

Fig. 6. Syntax and Semantics of the Meta-Language

an abstract page map, and thus do not have to know how the hardware deals with page tables, and so on. The plan also indicates which primitives are implemented by which code (lines with circles). When we certify the code, these will be the cross-abstraction links we will have to prove. Lastly, the plan also indicates the stubs in the initialization, which are needed to certify calls from init to functions defined over higher abstraction. The PE and PD models are restrictions on HW model, where AT is always on, and always off respectively. ALD is an analogue of ALE, where AT is off.

On boot, the AT is off, and `init` is called. The `init` then calls `mem_init` to initialize the allocation table and `pt_init` to initialize the page tables. Then, `init` uses the HW primitives to enable AT, and jumps into the high-level kernel by calling `kernel_init`.

We will now focus on the technical details to put this plan in action.

3 Certifying with Refinement

Our framework for multi-machine certification is defined in two parts. First, we create a machine-independent verification framework that will allow us to define quickly and easily as many machines for verification as we need. Second, we will develop our notion of refinements which will allow us to link all the separate machines together.

$$\frac{\forall \mathtt{l} \in \mathtt{dom}(\mathbb{C}).\, \mathcal{M}, \Psi \cup \mathcal{L} \vdash \mathbb{C}(\mathtt{l}) : \Psi(\mathtt{l})}{\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi}\ (\textsc{code}) \qquad \frac{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}', \mathtt{g}') \quad (\mathtt{p}, \mathtt{g}) \supseteq (\mathtt{p}', \mathtt{g}')}{\mathcal{M}, \Psi \vdash \mathbb{I} : (\mathtt{p}, \mathtt{g})}\ (\textsc{weak})$$

$$\frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash (b?\, \mathbb{I}' + \mathbb{I}'') : (\mathtt{p}', \mathtt{g}') \underset{\mathcal{M}(b)}{\oplus} (\mathtt{p}'', \mathtt{g}'')}\ (\textsc{split}) \quad \frac{\mathcal{M}, \Psi \vdash \mathbb{I}' : (\mathtt{p}', \mathtt{g}') \quad \mathcal{M}, \Psi \vdash \mathbb{I}'' : (\mathtt{p}'', \mathtt{g}'')}{\mathcal{M}, \Psi \vdash \mathbb{I}'; \mathbb{I}'' : ((\mathtt{p}', \mathtt{g}') \circ (\mathtt{p}'', \mathtt{g}''))}\ (\textsc{seq})$$

$$\frac{}{\mathcal{M}, \Psi \vdash \iota : \mathcal{M}(\iota)}\ (\textsc{perf}) \qquad \frac{}{\mathcal{M}, \Psi \vdash [\mathtt{l}] : \Psi(\mathtt{l})}\ (\textsc{call}) \qquad \frac{}{\mathcal{M}, \Psi \vdash \mathtt{nil} : id}\ (\textsc{nil})$$

Fig. 7. Static Semantics of the Meta-Language

3.1 A Machine-Independent Certification Framework

Our Hoare-logic based framework is parametric over the definition of operational semantics of the machine, and is sound no matter what machine semantics it is parameterized with. To begin defining such a framework, we first need to understand what exactly is a machine on which we can certify code. The definition that we use is given in Figure 4. Our notion of the machine consists of the following parts:

- State type (Σ). Define the set of all possible states in a machine.
- Operations (Δ). This is a set of names of all operations that the machine supports. The set can be infinite, and defined parametrically.
- Conditionals (β). Defines a type of expressions that are used for branching.
- Conditional Interpreter (Υ). Converts conditionals into state predicates.
- The operational semantics `OS`. This is the main portion of the machine definition. It is a set of actions $(\mathtt{p}, \mathtt{g})$ named by all operations in the machine.

The most important bit of information in the machine are the semantics (`OS`). The semantics of operations are defined by a precondition (`p`), which shows when the operation is safe to execute, and by a state relation (`g`) that defines the set of possible states that the operation may result in. We will refer to the pair of $(\mathtt{p}, \mathtt{g})$ as an action of the operation. Later we will also use actions to define the specification of programs. Because the type of actions is somewhat complex, we define action combinators in Figure 5, including composition and branching. The same figure also shows the weaker than relation between actions.

Although, at this point we have defined our machines, it does not have any notion of computation. To make use of the machine, we will need to define a concept of programs, as well as what it means for the particular program to execute.

The definition of the program is given in Figure 6. The most important definition in that figure is that of the procedure, $\mathbb{I}$. The procedure is a bit of program logic that sequences together calls to the operations of a machine (ι), or to other procedures `[l]` (loops are implemented as recursive calls). Procedures also include a way to branch on a condition. The procedures can be given a name, and placed in the procedure heap $\mathbb{C}$, where they can be referenced from other procedures through the `[l]` constructor. The procedure heap together with a program rest (the currently executing procedure) makes up the program that can be executed.

The meaning of executing a program is given by the indexed denotational semantics shown on the right side of Figure 6. The meaning of the program is an action that is

constructed by sequencing operations. As programs can be infinite, the semantics are indexed by the depth of procedure inclusion.

We use the static semantics (Figure 7) to approximate the action of a procedure. These semantics are similar to the denotational semantics of the meta-language, except that the specifications of called procedure are looked up in the table (Ψ). This means that the static semantics works by the programmer approximating the actions of (specifying) the program, and then making sure that the actual action of the program is within the specifications. These well-formed procedures are then grouped into a well-formed module using the CODE rule, which forms the concept of a certified module $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, where every procedure in $\mathbb{C}$ is well-formed under specification in Ψ. The module also defines a library ($\mathcal{L}$) which is a set of specifications of stubs, i.e. procedures that are used by the module, but are not in the module. These stubs can then be eliminated by providing procedures that match the stubs (see Section 3.2). For a program to be completely certified, all stubs must either be considered valid primitives or eliminated.

For a proof of partial correctness, please see the TR.

3.2 Linking

When we certify using modules, it will be very common that the module will require stubs for the procedures of another module. Linking two modules together should replace the stubs in both modules for the actual procedures that are now present in the linked code. The general way to accomplish this is by the following linking lemma:

Theorem 1 (Linking).

$$\frac{\mathcal{M}, \mathcal{L}_1 \vdash \mathbb{C}_1 : \Psi_1 \quad \mathcal{M}, \mathcal{L}_2 \vdash \mathbb{C}_2 : \Psi_2 \quad \mathbb{C}_1 \perp \mathbb{C}_2 \quad \mathcal{L}_1 \perp \Psi_2 \quad \mathcal{L}_2 \perp \Psi_1 \quad \mathcal{L}_1 \perp \mathcal{L}_2}{\mathcal{M}, ((\mathcal{L}_1 \cup \mathcal{L}_2) \setminus (\Psi_1 \cup \Psi_2)) \vdash \mathbb{C}_1 \cup \mathbb{C}_2 : \Psi_1 \cup \Psi_2} \text{ (LINK)}$$

where $\Psi_1 \perp \Psi_2 \triangleq \forall \mathtt{l} \in \mathtt{dom}(\Psi_1).\ (\mathtt{l} \notin \mathtt{dom}(\Psi_2) \vee \Psi_1(\mathtt{l}) = \Psi_2(\mathtt{l}))$.

However, the above rule does not always apply immediately. When the two modules are developed independently, it is possible that the stubs of one module are weaker than the specifications of the procedures that will replace the stubs, which breaks the linking lemma. To fix this, we strengthen the library.

Theorem 2 (Stub Strengthening).
If $\mathcal{M}, \mathcal{L} \vdash \mathbb{C} : \Psi$, then for any $\mathcal{L}'$ s.t. $\forall \mathtt{l} \in \mathtt{dom}(\mathcal{L}).\ \mathcal{L}(\mathtt{l}) \supseteq \mathcal{L}'(\mathtt{l})$ and $\mathtt{dom}(\mathcal{L}') \cap \mathtt{dom}(\Psi) = \emptyset$, the following holds: $\mathcal{M}, \mathcal{L}' \vdash \mathbb{C} : \Psi$.

This theorem allows us to strengthen the stubs to match the specs of procedures, enabling the linking lemma. Of course, if the specs of the real procedures are not stronger than the specs of the stubs, then the procedures do not correctly implement what the module expects, and linking is not possible.

3.3 The Refinement Framework

Up to this point, we have only considered what happens to the code that is certified over a single machine. However, the purpose of our framework is to facilitate multi-machine

verification. For this purpose, we construct the refinement framework that will allow us to refine certified modules in one machine to certified modules in another. The most general notion of refinement in our framework can be defined by the following:

Definition 1 (Certified Refinement).
A certified refinement from machine $\mathcal{M}_A$ to machine $\mathcal{M}_C$ is a pair of relations $(T_{\mathbb{C}}, T_\Psi)$ and a predicate over the abstract certified module Acc, such that for all $\mathbb{C}_A, {\Psi'}_A, \Psi_A$, the following holds

$$\frac{\mathcal{M}_A, {\Psi'}_A \vdash \mathbb{C}_A : \Psi_A \quad Acc(\mathcal{M}_A, {\Psi'}_A \vdash \mathbb{C}_A : \Psi_A)}{\mathcal{M}_C, T_\Psi({\Psi'}_A) \vdash T_{\mathbb{C}}(\mathbb{C}_A) : T_\Psi(\Psi_A)} \text{ REFINE}$$

This definition is not a rule, but a template for other definitions. To define a refinement, one has to provide the particular $T_{\mathbb{C}}$, T_Ψ, Acc together with the proof that the rule holds. However, instead of trying to define these translations directly, we will automatically generate them from the relations between the particular pairs of machines.

Representation Refinement The only automatic refinement we will discuss in this paper is the representation refinement. The representation refinement can be generated for an abstract ($\mathcal{M}_A$) and a concrete machine ($\mathcal{M}_C$), where both use the same operations and condtionals (e.g. $\mathcal{M}_A.\Delta = \mathcal{M}_C.\Delta$ and $\mathcal{M}_A.\beta = \mathcal{M}_C.\beta$) by defining a relation ($\texttt{repr} : \mathcal{M}_A.\Sigma \to \mathcal{M}_C.\Sigma \to \mathit{Prop}$) between the states of the two machines. Using `repr`, we can define our specification translation function:

$$T_{A-C}(\mathtt{p}, \mathtt{g}) \triangleq \begin{array}{l}(\lambda \mathbb{S}_C.\, \exists \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \wedge \mathtt{p}\ \mathbb{S}_A, \\ \quad \lambda \mathbb{S}_C.\, \lambda \mathbb{S}'_C.\, \forall \mathbb{S}_A.\, \mathtt{repr}\ \mathbb{S}_A\ \mathbb{S}_C \to \forall \mathbb{S}'_A.\, \mathtt{g}\ \mathbb{S}_A\ \mathbb{S}'_A \to \mathtt{repr}\ \mathbb{S}'_A\ \mathbb{S}'_C)\end{array}$$

This operation creates an concrete action from an abstract action. Informally it works as follows. There must be at least one abstract state related to the starting concrete state for which the abstract action applies. The action starting from state $\mathbb{S}_C$ results in set containing $\mathbb{S}'_C$, only if for all related abstract states for which the abstract action is valid result in sets of abstract states that contain a state related to $\mathbb{S}'_C$. Essentially, the resulting concrete action is an intersection of all abstract actions that do not fail.

To make this approach work, we require several properties over the machines and the `repr`. First, the refined semantics of abstraction operations have to be weaker than the semantics of their concrete counterparts, e.g. $\forall \iota_A \in \mathcal{M}_A.\, T_{A-C}(\mathcal{M}_A(\iota_A)) \supseteq \mathcal{M}_C(\iota_A)$.

Second, the refinement must preserve the branch choice, e.g. if the refined program chooses left branch, then abstract program had to choose the left branch in all states related by `repr` as well. This property is ensured by requiring the following:

$$\forall b.\, \forall \mathbb{S}, \mathbb{S}'.\, (\exists \mathbb{S}_C.\, \mathtt{repr}(\mathbb{S}, \mathbb{S}_C) \wedge \mathtt{repr}(\mathbb{S}', \mathbb{S}_C)) \to (\mathcal{M}(b)\ \mathbb{S} \leftrightarrow \mathcal{M}(b)\ \mathbb{S}')$$

With these properties, we can define a valid refinement by the following lemma:

Lemma 1 (`repr`-refinement valid).
Given `repr` with proofs of the two properties above, the following is valid:

$$\frac{\mathcal{M}_A, \mathcal{L}_A \vdash \mathbb{C} : \Psi_A}{\mathcal{M}_C, T_\Psi(\mathcal{L}_A) \vdash \mathbb{C} : T_\Psi(\Psi_A)}$$

where $T_\Psi(\Psi) := \{T_{A-C}(\Psi(\mathtt{l})) \mid \mathtt{l} \in \mathtt{dom}(\Psi)\}$

This refinement is interesting in that it preserves the code of the program, and performing point-wise refinement on specifications. Our actual work defines several other refinement generators. One of these, code-preserving refinement, is included in the TR, and is used as a stepping stone for proof of Lemma 1. Coq implementation features more general versions of refinements presented, as well as several others.

4 Certifying C Code

Since BabyVMM is written in C, we define a formal specification of a tiny subset of the C language using our framework. This C machine will be parameterized by the specific semantics of the memory model, as our plan required. We will also utilize the C machine to further speed up the creation of refinements.

4.1 The Semantics of C

To define our C machine in terms of our verification framework, we need to give it a state type, a list of operations, and the semantics of those operations expressed as actions. All of these are given in Figure 8.

The state of the C machine includes two components, the stack and the memory. The stack is an abstract C stack that consists of a list of frames, which include call, data, and return frames. In the current version, the stack is independent from memory (one can think of it existing within a statically defined part of the loaded kernel). The memory model is a parameter in the C machine, meaning that it can make use of any memory model as long as it defines load and store operations. The syntax of the C machine is different from the usual definition, in that it relies on the meta-machine for its control flow by using the meta-machines call and branch. Our definition of C adds atomic operations that perform state updates. Thus the operations include two types assignments - one to stack and one to memory, and 4 operations to manipulate stack for call and return, which push and pop the frames.

Because control flow is provided by a standard machine, the code has to be modified slightly. For example, a function call of the form $r = f(x)$ will split into a sequence of three operations: $fcall([x]); [f]; readret([r])$, the first setting up a call frame, the second making the call, and the third doing the cleanup. Similarly, the body of the function $f(x)\{body; return(0);\}$ will become $args([x]); body; ret(0)$, as the function must first move the arguments from the call frame into a data frame. Loops have to be desugared into recursive procedures with branches. These modifications are entirely mechanical, and hence we can claim that our machine supports desugared linearized C code.

4.2 Refinement in C machines

C machines at different abstraction layers differ only in their memory models, with the stack being the same. We can use this fact to generate refinements between the C machines using only the representation relation between memory models. This relation ($M_1 \preceq M_2$) can be completely arbitrary as long as these conditions hold:

$$\forall l, v.\, load(M_1, l) = v \rightarrow load(M_2, l) = v$$
$$\forall l, v, M_1'.\, \left(M_1' = (store(M_1, l, v))\right) \rightarrow \left(M_1' \preceq (store(M_2, l, v))\right)$$

$$
\begin{array}{rll}
(\textit{State}) & \mathbb{S} & ::= (M, S) \\
(\textit{Memory}) & M & ::= \textit{(any type over which load}(M, \mathtt{l}) \textit{ and store}(M, \mathtt{l}, w) \textit{ are defined)} \\
(\textit{Stack}) & S & ::= \textit{nil} \mid \textit{Call}(\textit{list } w) :: S \mid \textit{Data}(\{\mathtt{v} \rightsquigarrow w\} :: S) \mid \textit{Ret}(w) :: S \\
(\textit{Expressions}) & \mathtt{e} & ::= \mathrm{se} \mid *(\mathtt{e}) \\
(\textit{StackExpr, Cond}) & \mathrm{se}, b & ::= w \mid \mathtt{v} \mid \textit{binop}(\textit{bop}, \mathtt{e}_1, \mathtt{e}_2) \\
(\textit{Binary Operators}) & \textit{bop} & ::= + \mid - \mid * \mid / \mid \% \mid == \mid < \mid <= \mid >= \mid > \mid != \mid \&\& \mid \mid\mid \\
(\textit{Variables}) & \mathtt{v} & ::= \textit{(a decidable set of names)} \\
(\textit{Words}) & w & ::= n \text{ (integers)} \\
(\textit{Operation}) & \iota & ::= \mathtt{v} := \mathtt{e} \mid *(\mathtt{e}_{loc}) := \mathtt{e} \mid \mathrm{fcall}(\mathrm{list}\ \mathtt{e}) \mid \mathrm{ret}(\mathtt{e}) \mid \mathrm{args}(\mathrm{list}\ \mathtt{v}) \mid \mathrm{readret}(\mathtt{v})
\end{array}
$$

Operation (ι) =	Action ($\mathcal{M}(\iota)$) =
$\mathtt{v} := \mathtt{e}$	$(\lambda\mathbb{S}.\, \exists S', F, w.\, \mathbb{S}.S = \textit{Data}(F) :: S' \wedge \textit{eval}(\mathtt{e}, \mathbb{S}) = w,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', F, w.\, \mathbb{S}.S = \textit{Data}(F) :: S' \wedge \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(F\{v \rightsquigarrow w\}) :: S')$
$*(\mathtt{e}_{loc}) := \mathtt{e}$	$(\lambda\mathbb{S}.\, \exists l, w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge \textit{eval}(\mathtt{e}_{loc}, \mathbb{S}) = l \wedge \exists M'.\, M' = \textit{store}(M, l, w),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists l, w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w \wedge \textit{eval}(\mathtt{e}_{loc}, \mathbb{S}) = l \wedge$ $\quad \mathbb{S}'.M = \textit{store}(\mathbb{S}.M, l, w) \wedge \mathbb{S}'.S = \mathbb{S}.S)$
$\mathrm{fcall}([e_1, \ldots, e_n])$	$(\lambda\mathbb{S}.\, \exists v_1, \ldots, v_n.\, \textit{eval}(e_1, \mathbb{S}) = v_1 \wedge \ldots \wedge \textit{eval}(e_n, \mathbb{S}) = v_n,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists v_1, \ldots, v_n.\, \textit{eval}(e_1, \mathbb{S}) = v_1 \wedge \ldots \wedge \textit{eval}(e_n, \mathbb{S}) = v_n \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Call}([v_1, \ldots, v_n]) :: \mathbb{S}.S)$
$\mathrm{args}([\mathtt{v}_1, \ldots, \mathtt{v}_n])$	$(\lambda\mathbb{S}.\, \exists w_1, \ldots, w_n, S'.\, \mathbb{S}.S = \textit{Call}([w_1, \ldots, w_n]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists w_1, \ldots, w_n, S'.\, \mathbb{S}.S = \textit{Call}([w_1, \ldots, w_n]) :: S' \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(\{\mathtt{v}_1 \rightsquigarrow w_1, \ldots, \mathtt{v}_n \rightsquigarrow w_n\}) :: S')$
$\mathrm{readret}(\mathtt{v})$	$(\lambda\mathbb{S}.\, \exists S', w.\, \mathbb{S}.S = \textit{Ret}(w) :: \textit{Data}(D) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', w.\, \mathbb{S}.S = \textit{Ret}(w) :: \textit{Data}(D) :: S' \wedge$ $\quad \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Data}(D\{\mathtt{v} \rightsquigarrow w\}) :: S')$
$\mathrm{ret}(\mathtt{e})$	$(\lambda\mathbb{S}.\, \exists w.\, \textit{eval}(\mathtt{e}, \mathbb{S}) = w,\ \lambda\mathbb{S}, \mathbb{S}'.\, \mathbb{S}'.M = \mathbb{S}.M \wedge \mathbb{S}'.S = \textit{Ret}(\textit{eval}(\mathtt{e}, \mathbb{S})) :: \mathbb{S}.S)$

$$
\textit{eval}(\mathtt{e}, \mathbb{S}) ::= \begin{cases} w & \text{if } \mathtt{e} = w \\ \mathbb{S}.S(\mathtt{v}) & \text{if } \mathtt{e} = \mathtt{v} \\ \textit{load}(\mathbb{S}.M, \textit{eval}(\mathtt{e}_1, \mathbb{S})) & \text{if } \mathtt{e} = (*\mathtt{e}_1) \\ b(\textit{eval}(\mathtt{e}_1, \mathbb{S}), \textit{eval}(\mathtt{e}_2, \mathbb{S})) & \text{if } \mathtt{e} = \textit{binop}(b, \mathtt{e}_1, \mathtt{e}_2) \end{cases}
$$

$$
\Upsilon(b) ::= \lambda\mathbb{S}.\, \textit{eval}(b, \mathbb{S}) \neq 0
$$

Fig. 8. Primitive C-like machine

The above properties make sure that the load and store operations of memory behave in a similar way. We construct the `repr` between C machine as follows:

$$
\mathtt{repr} := \lambda\mathbb{S}_A, \mathbb{S}_C.\ (\mathbb{S}_A.S = \mathbb{S}_C.S)\ \wedge\ (\mathbb{S}_A.M \preceq \mathbb{S}_C.M)
$$

Using the properties of load and store, we show properties needed for `repr`-refinement to work: that for every operation ι in the C machine $T_{M1-M2}(\mathcal{M}_{M1}(\iota)) \supseteq \mathcal{M}_{M2}(\iota)$, and that `repr` preserves branching. For details, please see the TR. Now we can define the actual refinement rule for C machines:

Corollary 1 (C Refinement).
For any two memory models $M1$ and $M2$, s.t. $M1 \preceq M2$, the following refinement works for C

Definition	Value	Description
PGSIZE	4096	Number of bytes per page
NPAGES	unspecified	Number of phys. pages in memory
VPAGES	unspecified	Maximum page number of a virtual address
Pg(*addr*)	*addr*/PGSIZE	gets page of address
Off(*addr*)	*addr*%PGSIZE	offset into page of address
LowPg(*pg*)	$0 \leq pg <$ NPAGES	valid page in low memory area
HighPg(*pg*)	NPAGES $\leq pg <$ VPAGES	valid page in high memory area

Fig. 9. Page Definitions

machines instantiated with $M1$ and $M2$.

$$\frac{\mathcal{M}_{M1}, \mathcal{L} \vdash \mathbb{C} : \Psi}{\mathcal{M}_{M2}, T_{M1-M2}(\mathcal{L}) \vdash \mathbb{C} : T_{M1-M2}(\Psi)} \; M1 - M2$$

Thus we know that if we have two C-machines that have related memory models, then we have a working refinement between the two machines. Our next step is the to show the relations between all the memory models shown in our plan (in Figure 3).

5 Virtual Memory Manager

At this point, we have all the machinery necessary to start building our certified memory manager according to the plan. The first step is to formally define and give relations between the memory models that we will use in our certification. Then we will certify the code of the modules that make up the VMM. These modules will then be refined and linked together, resulting in the conclusion that the entire BabyVMM is certified.

5.1 The Memory Models

Because of the space limit, we will only formally present the PMAP memory model (Figures 9 and 10). For the definitions of others, please see the TR.

The state of the PMAP memory has three components, the actual memory store D, the allocation table A, and the first-class pagemap PM. The memory store contains the actual data in memory, indexed by physical addresses. The allocation table A, keeps track of which pages are allocated and which are not. This allocation information is abstract - it does not have to correspond to the actual allocation table used within the VMM. For example, the hardware page tables, which this model abstracts, are still in memory, but are hidden by the allocation table. The page map is the abstract mapping of virtual pages to physical pages, which purposefully skips all addresses mappable to physical memory. This mapping is used in loads and stores of the memory model, which use the *trans* predicate to translate addresses by looking up mappings in the PM.

The PMAP model relies on the stub library ($\mathcal{L}_{PMAP}$) for updating auxiliary data structures. There are two stubs for memory allocation, mem_alloc and mem_free. Their specs show how they modify the allocation table, and how allocating a page is non-deterministic and may potentially return any free page. The other two stubs, pt_set and pt_lookup update and look up page map entries; their specs are straightforward.

$$
\begin{array}{rll}
\text{(Global Storage System)} & M & ::= (D, A, PM) \\
\text{(Allocatable Memory)} & D & ::= \{addr \rightsquigarrow w \mid \text{LowPg}(Pg(addr)) \wedge addr\%8 = 0\}^* \\
\text{(Page Allocation Table)} & A & ::= \{pg \rightsquigarrow \texttt{bool} \mid \text{LowPg}(pg)\}^* \\
\text{(Page Map)} & PM & ::= \{pg \rightsquigarrow pg' \mid \text{HighPg}(pg)\}^*
\end{array}
$$

Notation	Definition	
$load(M, va)$	$M.D(trans(M, va))$	if $M.A(Pg(trans(M, va))) = true$
$store(M, va, w)$	$(M.D\{trans(M, va) \rightsquigarrow w\}, M.A, M.PM)$	if $M.A(Pg(trans(M, va))) = true$

$$
trans(M, va) := \begin{cases} M.PM(Pg(va)) * \texttt{PGSIZE} + \text{Off}(va) & \text{if HighPg}(Pg(va)) \\ va & \text{otherwise} \end{cases}
$$

Label	Specification
mem_alloc	$(\lambda\mathbb{S}.\, \exists S'.\, \mathbb{S}.S = Call([]) :: S',$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S'.\, (\mathbb{S}.S = Call([]) :: S') \wedge ((\mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M) \vee$ $(\exists pg.\, \mathbb{S}'.S = Ret(pg) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow true\} \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\wedge \mathbb{S}.M.A(pg) = false \wedge \forall l.\, \mathbb{S}.M.A(Pg(l)) = true \rightarrow (\mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))))$
mem_free	$(\lambda\mathbb{S}.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}.M.A(pg) = true,$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', pg.\, \mathbb{S}.S = Call([pg]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.PM = \mathbb{S}.M.PM \wedge$ $\mathbb{S}'.M.A = \mathbb{S}.M.A\{pg \rightsquigarrow false\} \wedge \forall l.\, \mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_set	$(\lambda\mathbb{S}..\exists S', vp, pp.\, \mathbb{S}.S = Call([vp, pp]) :: S' \wedge \text{HighPg}(vp) \wedge \text{LowPg}(pp)$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', vp, pp.\, \mathbb{S}.S = Call([vp, pp]) :: S' \wedge \mathbb{S}'.S = Ret(0) :: S' \wedge \mathbb{S}'.M.A = \mathbb{S}.M.A \wedge$ $\mathbb{S}'.M.PM = \mathbb{S}.M.PM\{vp \rightsquigarrow pp\} \wedge \forall l.\, \mathbb{S}'.M.A(Pg(l)) = true \rightarrow \mathbb{S}'.M.D(l) = \mathbb{S}.M.D(l))$
pt_lookup	$(\lambda\mathbb{S}.\, \exists S', vp.\, \mathbb{S}.S = Call([vp]) :: S' \wedge \text{HighPg}(vp),$ $\lambda\mathbb{S}, \mathbb{S}'.\, \exists S', vp.\, \mathbb{S}.S = Call([vp]) :: S' \wedge \mathbb{S}'.S = Ret(\mathbb{S}.M.PM(vp)) :: S' \wedge \mathbb{S}'.M = \mathbb{S}.M)$

Fig. 10. PMAP Memory Model (M_{PMAP}) and Library ($\mathcal{L}_{PMAP}$)

5.2 Relation between Memory Models

Our plan calls for creation of the refinements between the memory models. In Section 4.2, we have shown that we can generate a valid refinement by creating a relation between the memory states, and then showing that abstract loads and stores are preserved by this relation. These relations and proofs of preserving the memory operations are fairly lengthy and quite technical, and thus we leave the mathematical detail to our Coq implementation, opting for a visual description shown in Figure 11.

On the right is a state of the hardware memory, whose operational semantics gives little protection from accessing data. Some areas of memory are dangerous, some are empty, others contain data, including the allocation tables and page tables. This memory relates to the ALE memory model by abstracting out the memory allocation table. This allocation table now offers protection for accessing both the unallocated space, and the space that seems unallocated, but dangerous to use (marked by wavy lines). An example of such area is the allocation table itself - the ALE model hides the table, making it appear to be unusable. The ALE mem_alloc primitive will never allocate pages from these wavy areas, protecting them without complicating the memory model.

The relation between the PMAP and ALE models shows that the abstract pagemap of PMAP model is actually contained within the specific area of the ALE model. The relation makes sure that the mappings contained in the PMAP's pagemap are the same as the translation results of the ALE's page table structures. To protect the in memory

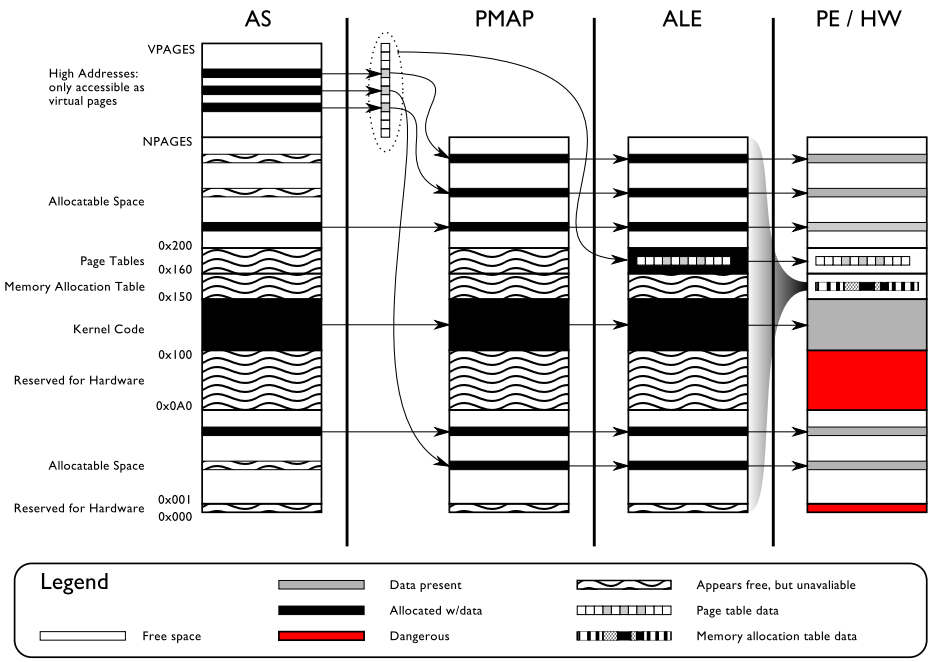

Fig. 11. Relation between Memory Models

page tables, the relation hides the page table memory area from the PMAP model, using the same trick as the one used to protect the allocation tables in the ALE model.

The relation between the AS and PMAP models collapses PMAP's memory and the page maps into a single memory like structure in the AS model. This is mostly accomplished by chaining the translation mechanism with the storage mechanism. However, to make this work, it is imperative that the relation ensures that no two pages of the AS model ever map to the same physical page in the PMAP model. This means that all physical pages that are mapped from the high-addresses become hidden in the AS model. We will not go into detail about the preservation of load and stores, as these proofs are mostly straightforward, given the relations.

5.3 Certification and Linking of BabyVMM

We have verified all the functions of the virtual memory on the appropriate memory models. This means that we have defined appropriate specifications for our functions, and certified our code. We also make an assumption that a kernel is certified in the AS model. The result is the following certified modules:

$$\mathcal{M}_{PE}, \mathcal{L}_{PE} \vdash \mathbb{C}^{mem} : \Psi^{mem}_{PE} \quad \mathcal{M}_{ALE}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{as} : \Psi^{as}_{PMAP} \quad \mathcal{M}_{PD}, \mathcal{L}_{PD} \vdash \mathbb{C}^{meminit} : \Psi^{meminit}_{PD}$$
$$\mathcal{M}_{ALE}, \mathcal{L}_{ALE} \vdash \mathbb{C}^{pt} : \Psi^{pt}_{ALE} \quad \mathcal{M}_{AS}, \mathcal{L}_{AS} \vdash \mathbb{C}^{kernel} : \Psi^{kernel}_{AS} \quad \mathcal{M}_{ALD}, \mathcal{L}_{ALD} \vdash \mathbb{C}^{ptinit} : \Psi^{ptinit}_{ALD}$$

However, the `init` function makes calls to other procedures that are certified in more abstract machines. Thus to certify `init` over the $\mathcal{M}_{HW}$ machine, we will need to

create stubs for these procedures, which have to be carefully crafted to be valid for the refined specifications of the actual procedures. Thus, the specification of `init` results in the following:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup \left\{\text{kernel\_init} \rightsquigarrow \mathtt{a}_{HW}^{kernel-init}, \text{mem\_init} \rightsquigarrow \mathtt{a}_{HW}^{meminit}, \text{pt\_init} \rightsquigarrow \mathtt{a}_{HW}^{ptinit}\right\} \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

With all the modules verified, we proceed to link them together. The first step is to refine the kernel. We use our AS-PMAP refinement rule to get the refined module:

$$\mathcal{M}_{PMAP}, T_{AS-PMAP}(\mathcal{L}_{AS}) \vdash \mathbb{C}^{kernel} : T_{AS-PMAP}(\Psi_{AS}^{kernel})$$

Then we show that the specs of functions and the primitives of the PMAP machine are proper implementation of the refined specs of $\mathcal{L}_{AS}$, more formally, $T_{AS-PMAP}(\mathcal{L}_{AS}) \supseteq \mathcal{L}_{PMAP} \cup \Psi_{PMAP}^{as}$. Using library strengthening and the linking lemma, we produce a certified module that is the union of the refined kernel and address space library:

$$\mathcal{M}_{PMAP}, \mathcal{L}_{PMAP} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} : T_{AS-PMAP}(\Psi_{AS}^{kernel}) \cup \Psi_{PMAP}^{as}$$

Applying this process to all the modules over all refinements, we link all parts of the code, except `init` certified over $\mathcal{M}_{HW}$. For readability, we hide chains of refinements. For example, T_{AS-HW} is actually $T_{AS-PMAP} \circ T_{PMAP-ALE} \circ T_{ALE-PE} \circ T_{PE-HW}$.

$$\begin{aligned}\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} :& \\ T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup& \\ T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit})&\end{aligned}$$

To get the initialization to link with the refined module, we must make sure that the stubs that we have developed for init are compatible with the refined specifications of the actual functions. This means that we prove the following:

$$\mathtt{a}_{HW}^{kernel-init} \supseteq T_{AS-HW}(\Psi_{AS}^{kernel})(\texttt{kernel-init})$$

$$\mathtt{a}_{HW}^{meminit} \supseteq T_{PD-HW}(\Psi_{PD}^{meminit})(\texttt{mem-init}) \qquad \mathtt{a}_{HW}^{ptinit} \supseteq T_{ALD-HW}(\Psi_{ALD}^{ptinit})(\texttt{pt-init})$$

Using these properties, we apply stub strengthening to the init module:

$$\mathcal{M}_{HW}, \mathcal{L}_{HW} \cup T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \vdash \mathbb{C}^{init} : \Psi_{HW}^{init}$$

This certification is now linkable to the rest of the VMM and kernel, to produce the final result that we need:

$$\begin{aligned}\mathcal{M}_{HW}, \mathcal{L}_{HW} \vdash \mathbb{C}^{kernel} \cup \mathbb{C}^{as} \cup \mathbb{C}^{pt} \cup \mathbb{C}^{mem} \cup \mathbb{C}^{meminit} \cup \mathbb{C}^{ptinit} \cup \mathbb{C}^{init} :& \\ T_{AS-HW}(\Psi_{AS}^{kernel}) \cup T_{PMAP-HW}(\Psi_{PMAP}^{as}) \cup T_{ALE-HW}(\Psi_{ALE}^{pt}) \cup& \\ T_{PE-HW}(\Psi_{PE}^{mem}) \cup T_{PD-HW}(\Psi_{PD}^{meminit}) \cup T_{ALD-HW}(\Psi_{ALD}^{ptinit}) \cup \Psi_{HW}^{init}&\end{aligned}$$

This result means that given a certified kernel in the AS model, we can refine it to the HW model of memory by linking it with VMM implementation. Furthermore, it is safe to start this kernel by calling the `init` function, which will perform the setup, and then call the `kernel-init` function, the entry point of the high-level kernel.

6 Coq Implementation

All portions of this system have been implemented in the Coq Proof Assistant[5]. The portions of the implementation directly related to the BabyVMM verification, including C machines, refinements, specs, and related proofs (excluding frameworks) took about 3 person-months to verify. The approximate line counts for unoptimized proof are:

- Verification and refinement framework - 3000 lines
- Memory models - 200-400 lines each
- `repr` and compatibility between models - 200-400 lines each
- Compatibility of stubs and implementation - 200-400 lines per procedure
- Code verification - less than 200 lines per procedure (half of it boilerplate).

7 Related Work and Conclusion

The work presented here is a continuation of the work on Hoare-logic frameworks for verification of system software. The verification framework evolved from SCAP[8] and GCAP[3]. Although our framework does not mention separation logic[17], information hiding[16], and local action[4] explicitly, these methods had great influence on the design of the meta-language and the refinements. The definition of `repr` generalizes the work on certified garbage collector[15] to fit our concept of refinement. The project's motivation is the modular and reusable certification of the CertiKOS kernel[10].

The well-known work in OS verification is L4.verified[12, 6], which has shown a complete verification of an OS kernel. Their methodology is different, but they have considered verification of virtual memory[13, 14]. However, their current kernel verification does not abstract virtual memory, maintaining only the invariant that allows the kernel to function, and leaving the details to the user level.

The Verisoft project [9, 2, 1, 11, 18] is the work that is closest to ours. We both aim for pervasive verification of OS by doing foundational verification of all components. Both works utilize multiple machines, and require linking. As both projects aim for certification of a kernel, both have to handle virtual memory. Although Verisoft uses multiple machine models, they use them sparingly. For example, the entire microkernel, excluding assembly code, is specified in a single layer, with correctness shown as a single simulation theorem between the concurrent user thread model (CVM) and the instruction set. The authors mention that the proof of correctness is a more complex part of Verisoft. Such monolithic approach is susceptible to local modifications, where a small change in one part of microkernel may require changes to the entire proof.

Our method for verification defines many more layers, with smaller refinement proofs between them, and composes them to produce larger abstractions, ensuring that the verification is more reusable and modular. Our new framework enables us to create abstraction layers with less overhead, reducing the biggest obstacle to our approach. We have demonstrated the practicality of our approach by certifying BabyVMM, a small virtual memory manager running on simplified hardware, using a new layer for every non-trivial abstraction we could find.

Acknowledgements. We thank anonymous referees for suggestions and comments on an earlier version of this paper. This research is based on work supported in part by DARPA grants FA8750-10-2-0254 and FA8750-12-2-0293, ONR grant N000141210478, and NSF grants 0910670 and 1065451. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. E. Alkassar, M. A. Hillebrand, D. C. Leinenbach, N. W. Schirmer, A. Starostin, and A. Tsyban. Balancing the load: Leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: OS Verification*, 42:389–454, 2009.
2. E. Alkassar, N. Schirmer, and A. Starostin. Formal pervasive verification of a paging mechanism. *Proc. TACAS'08*, pages 109–123, 2008.
3. H. Cai, Z. Shao, and A. Vaynberg. Certified self-modifying code. In *Proc. PLDI'07*, pages 66–77, New York, NY, USA, 2007. ACM.
4. C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. LICS'07*, pages 366–378, July 2007.
5. Coq Development Team. The Coq proof assistant reference manual. The Coq release v8.0, Oct. 2005.
6. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. HoTOS'07*, San Diego, CA, USA, May 2007.
7. X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proc. PLDI'08*, pages 170–182. ACM, 2008.
8. X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni. Modular verification of assembly code with stack-based control abstractions. In *PLDI'06*, pages 401–414, June 2006.
9. M. Gargano, M. A. Hillebrand, D. Leinenbach, and W. J. Paul. On the correctness of operating system kernels. In *TPHOLs'05*, 2005.
10. L. Gu, A. Vaynberg, B. Ford, Z. Shao, and D. Costanzo. Certikos: A certified kernel for secure cloud computing. In *Proc. APSys'11*. ACM, 2011.
11. T. In der Rieden. *Verified Linking for Modular Kernel Verification*. PhD thesis, Saarland University, Computer Science Department, Nov. 2009.
12. 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 *Proc. SOSP'09*, pages 207–220, 2009.
13. G. Klein and H. Tuch. Towards verified virtual memory in l4. In *TPHOLs Emerging Trends '04*, Park City, Utah, USA, Sept. 2004.
14. R. Kolanski and G. Klein. Mapped separation logic. In *Proc. VSTTE'08*, pages 15–29, 2008.
15. A. McCreight, Z. Shao, C. Lin, and L. Li. A general framework for certifying garbage collectors and their mutators. In *Proc. PLDI'07*, pages 468–479, 2007.
16. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL'04*, pages 268–280, Jan. 2004.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. LICS'02*, pages 55–74, July 2002.
18. A. Starostin. *Formal Verification of Demand Paging*. PhD thesis, Saarland University, Computer Science Department, Mar. 2010.
19. A. Vaynberg and Z. Shao. Compositional verification of BabyVMM (extended version and Coq proof). Technical Report YALEU/DCS/TR-1463, Yale University, Oct. 2012. `http://flint.cs.yale.edu/publications/babyvmm.html`.