Verification of a Transactional Memory Manager under Hardware Failures and Restarts

Ognjen Marić and Christoph Sprenger

Institute of Information Security, Dept. of Computer Science, ETH Zurich
{omaric,sprenger}@inf.ethz.ch

Abstract. We present our formal verification of the persistent memory manager in IBM's 4765 secure coprocessor. Its task is to achieve a transactional semantics of memory updates in the face of restarts and hardware failures and to provide resilience against the latter. The inclusion of hardware failures is novel in this area and incurs a significant jump in system complexity. We tackle the resulting verification challenge by a combination of a monad-based model, an abstraction that reduces the system's non-determinism, and stepwise refinement. We propose novel proof rules for handling repeated restarts and nested metadata transactions. Our entire development is formalized in Isabelle/HOL.

1 Introduction

The IBM 4765 [1] cryptographic coprocessor resembles a general-purpose computer, encased in a tamper-proof housing and packed onto a PCIe card. Its security policies require that most access to the persistent storage be brokered through the built-in bootloader, and in particular its subsystem called the *Persistent Memory Manager* (PMM). Verification of the PMM is our driving case study, and this paper presents the main challenges, our techniques for overcoming them, and some of the lessons learned in the process.

The PMM's API offers a rudimentary persistent storage service. It abstracts the persistent memory into an arbitrary, but fixed number of storage slots of different capacities. The slots are called *regions*, and they are addressed by their indices. The API provides just two operations: update and fetch. The main requirement for this API are *atomic updates*: given new contents for a set of regions, an update operation updates either all of them, none of them, or fails.

The API does not support concurrency. Hence, designing and verifying such a system appears to be easy at first. Appearances can be deceiving, however, as we will require atomicity to hold even in the presence of:

(1) abrupt power-downs, possibly resulting in garbled writes. At power-up a startup procedure is called (that might itself be subject to abrupt restarts).
(2) failures of persistent storage, such as spontaneous corruption ("bit rot") or permanent hardware failures.

Algorithms that provide atomic updates in the presence of (1) have already been analyzed in the literature [2,3,4], but this is the first work we are aware of that also addresses (2). Moreover, our target system does not just detect such failures, but also aims for resilience against them, restoring corrupted data from spare copies when possible. This necessitates full redundancy in both user and metadata (i.e., administrative data used by the algorithm) stored in the persistent memory. It also complicates the details of the algorithm, requiring nested transactions in the metadata and permeating the implementation with special cases, integrity checks, and potential recovery actions. An example of the resulting implementation complexity is the seemingly innocuous fetch procedure, which simply retrieves the contents of a single region. Figure 1 shows its call graph. The complexity of implementation, and thus also reasoning, caused by the non-determinism of (2) is then further aggravated by (1), since longer implementations induce new restart points.

To tame this complexity and enable verification, we proceed using abstraction (or dually, refinement), building a stack of progressively more abstract models. These gradually remove redundancy, first in the metadata, then in the user data, and finally replace repeated restarts by a single one. Combining proofs of refinement between neighboring models with general property preservation results, we then transfer proofs of requirement compliance from the top of the model stack to the concrete model of the PMM on its bottom. We have formalized our entire development in the Isabelle/HOL theorem prover [5].[1]

Our contributions are twofold. First, we propose novel modeling and reasoning techniques for systems with restarts and failures. Our modeling limits the asynchrony of these effects and hence simplifies the verification by reducing non-determinism. Moreover, we model restarts at the language level, which allows us to derive structured refinement proof rules for repeated restarts and for eliminating the nested transactions that handle the metadata redundancy. Second, this combination of tools enabled the success of our case study, which is substantial, industrially relevant, and more complex than related ones published hitherto, due to the system's resilience to hardware failures. We believe that our approach is applicable in related areas such as smart cards and file systems.

We give an overview of the PMM API and describe its environment and the requirements we pose on it in the next section. Section 3 describes our modeling and reasoning framework, and Section 4 the models we create and the results we obtain. We review the related work in Section 5, and conclude in Section 6.

2 System Overview

The task of the PMM is to provide a simple API for transactional access to persistent memory, effectively resulting in an abstraction of the memory as a function $index \to contents$. The PMM (sub)system consists of three main procedures: `update`, `fetch`, and `startup`. The first two constitute the PMM API.

[1] Accessible at http://www.infsec.ethz.ch/research/software/pmm-verif.

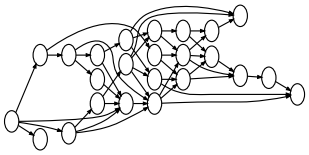


Fig. 1: Call graph for fetch

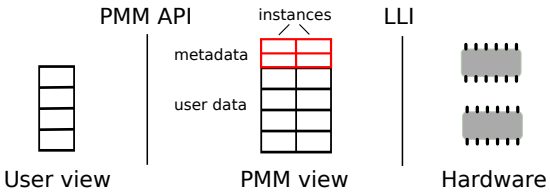


Fig. 2: Abstraction levels

The `fetch` procedure takes a single parameter, the index of the target region, and is supposed to return the corresponding contents. The `update` procedure also takes a single parameter, a map (partial function) $index \rightharpoonup contents$, and is supposed to override the memory abstraction with the given map, updating all the regions in the map's domain with the given contents. However, the behavior of the API is also conditioned on possible abrupt restarts and hardware failures.

The restarts cause the `startup` procedure to be run, which then performs cleanups and integrity checks. An *API call* to `update` or `fetch` may thus result in one or more (if `startup` is itself restarted) executions of `startup`. The same procedure is also executed in case a restart happens in between API calls.

We detail the hardware failures we consider below, however their global effect on the card is reflected in the three PMM *modes of operation*: `Normal`, `Degraded`, and `Fail`, corresponding to normal operation, read-only mode, and complete failure. To achieve resilience, the system stores all its data in two copies. This includes the data of the *user* regions, exposed by the API, but also the metadata stored in the extra *administrative* regions. Each copy of a region is called a region *instance*. Figure 2 gives an overview of the system's abstraction levels.

To signal irremediable hardware failures to the caller, the API calls use an exceptional *mode of termination*. Restarts lose the information about the original call and its return value. To facilitate modeling, we will also use the exceptional mode to signal the completion of the `startup` procedure. If an API call terminates in the normal mode, we expect it to behave as described at the beginning of the section. In the exceptional mode, however, we will have to loosen the requirements. We will make this more precise in Section 2.2.

2.1 The Environment

Next, we present the environment that the PMM interacts with, and our assumptions about it. The PMM controls the persistent storage, which consists of battery-backed RAM and flash memory. However, the PMM does not access the hardware directly, relying on lower-level firmware instead. The *lower-level interface* (LLI) abstracts the memory into logical *blocks* of varying sizes. It can read and write each block independently (regardless of the type of the underlying memory), by transferring data between the persistent memory and the DRAM (dynamic RAM). We assume both the DRAM and the CPU to be reliable.

The PMM maps each region instance to a unique memory block. The two blocks corresponding to the two instances of the same region have equal capacity. The LLI provides a convenient addressing scheme for mapping instances to blocks, but is otherwise oblivious of the connection between blocks and regions. Its task consists, first and foremost, of mapping the logical addresses onto the appropriate hardware ones, and performing blockwise read and write operations.

Additionally, the LLI tries to eliminate transient failures (e.g., bus interconnect problems) by repeating its reads, and checking the success of each write. It also detects and reports two kinds of permanent (irrecoverable) failures, namely:

- Read failures, where a block becomes completely unusable (e.g., due to a dead memory bank). We call such a block *dead*.
- Write failures, where a block can no longer be overwritten with new contents. We call such a block *degraded*.

A block without permanent failures is called *ok*. Other failures are undetectable by the LLI and the PMM must detect and try to correct them. An example is "bit rot", where some content can be retrieved from an instance, but it differs from the content that was last written to it. These failures are recoverable, as the block can still be overwritten with the correct contents, if they are available.

The environment can also trigger restarts, whereby control is transferred to the `startup` procedure. Restarts may interrupt write operations, causing another (recoverable) kind of write failure. We will further discuss restarts in Section 3.2.

2.2 The Requirements

We specify the requirements on the PMM in terms of the abstract *view* on the memory it provides to API users. We express this view as elements of the type $(\mathit{index} \to \mathit{contents})_\perp$, where $\tau_\perp = \tau + \{\perp\}$ and $\perp$ corresponds to a failure. The requirements concern entire API calls, including the possible runs of the `startup` procedure. We call a user region instance *active* if it matches the view's content.

(R1) *Atomic updates.* Given the current view v and an update map u, an update results in either the view $v \triangleleft u$ (successful update, where $\triangleleft$ overrides the function v with the map u), v (rollback), or $\perp$ (failure, also if $v = \perp$). A rollback may only be performed in the case of exceptional termination.

(R2) *Correctness of fetch.* Fetch returns the value of the view at the given index, or results in an exception in `Fail` mode or when interrupted by a restart.

(R3) *Unchanged view* during fetches, updates in non-`Normal` mode, and restarts in between API calls, except for when the mode is changed to `Fail`.

(R4) *Matching modes of operation and termination.* API calls can terminate exceptionally only in the case of restarts or non-`Normal` mode of operation.

(R5) *Correctness of the mode of operation.* In `Normal` mode, all region instances are active. In `Degraded` mode, each region has at least one active instance and there is at least one degraded block. In `Fail` mode, there exists a region with no reliable and up-to-date instances.

(R6) *Maximum redundancy.* In all but `Fail` mode, any 'ok' region instance is active (and hence all 'ok' instances of a region match).

3 The Framework

We embedded a framework for modeling and reasoning about imperative programs with restarts and failures in the theorem prover Isabelle/HOL [5]. Similar to Klein et al. [6], we build a series of models at different levels of abstraction, with each model having two-layers: an outer layer based on transition systems and a structured inner layer based on monads.

Most of the work is done in the inner layer, where we model the API and startup procedures in an imperative fashion. This layer also provides facilities for modeling restarts and hardware failures. Our treatment of both of these is possibilistic, since our requirements do not include probabilistic properties. The inner layer also provides constructs for modeling repeated restarts, allowing us to model entire API calls. The outer layer is a simple shell around the inner one, with the purpose of providing a trace semantics. Its transitions are derived directly from the definitions in the inner layer, as the union over all API calls and restarts in idle states. Given this trace semantics, we define a refinement infrastructure based on forward simulation akin to [7,8], allowing us to relate the different models. We transfer the refinement proof obligations from the outer to the inner layer, where we can prove them in a compositional manner. The refinements guarantee that the concrete models inherit the properties expressing our requirements, which we prove on the simpler abstract models.

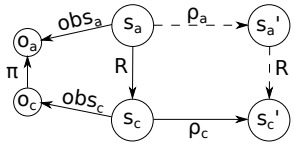
3.1 Specifications and Refinement (Outer Layer)

On the outer layer, we use transition systems of the form $T = (\Sigma, \Sigma_0, \rho)$, where Σ is the universe of states, $\Sigma_0 \subseteq \Sigma$ is the set of initial states, and $\rho \subseteq \Sigma \times \Sigma$ is the transition relation. A *behavior* of a transition system T is a finite sequence of states in which the first element belongs to Σ_0, and each pair of successive elements is related by the transition relation ρ. We denote the set of behaviors of T by $beh(T)$ and the set of states appearing in some behavior by $reach(T)$.

We extend transition systems to *specifications* of the form $S = (T, O, obs)$, where O is the universe of observations, and $obs : \Sigma \to O$ is an observation function. Observations abstract away the uninteresting details of the state. For example, we can project the state ("forgetting" some parts of it) or replace a list by a set (in case we do not care about the ordering). A specification S's *reachable observations* and *observable behaviors* are defined as $oreach(S) = obs(reach(T))$ and $obeh(S) = obs(beh(T))$, where obs is applied pointwise to reachable states (as set or behavior elements, respectively). An (internal) invariant of T (and S) is a set of states I such that $reach(T) \subseteq I$. An *external invariant* is a set $J \subseteq O$ such that $oreach(S) \subseteq J$. Given two specifications S_a and S_c, and a *mediator function* $\pi : O_c \to O_a$, we say that S_c *implements* S_a via π if $\pi(obeh(S_c)) \subseteq obeh(S_a)$. Mediator functions allow us to relate systems with different observations.

To prove that S_c implements S_a, we use refinement based on forward simulation. S_c *refines* S_a *under the simulation relation* $R \subseteq \Sigma_a \times \Sigma_c$ and the mediator function π if three conditions hold. (Ref1) $\Sigma_{0c} \subseteq R(\Sigma_{0a})$, i.e., each concrete initial state is related via R to some abstract one. (Ref2) R; $\rho_c \subseteq \rho_a$; R, where the

semicolon denotes forward relational composition, i.e., any concrete transition can be matched by an abstract one. We can visually represent this by requiring the existence of an s'_a that allows us to fill the dashed lines in the drawing below. (Ref3) $obs_a(s_a) = \pi(obs_c(s_c))$ whenever $(s_a, s_c) \in R$, i.e. the observations and the simulation relation R are *consistent* (the two paths from s_a to o_a in the drawing below commute).



3.2 Modeling Hardware Failures and Restarts (Inner Layer)

We now turn to the inner layer. The salient features of the system we wish to model are the imperative nature of the target algorithm and the non-determinism in the environment stemming from hardware failures and abrupt restarts. Our modeling of these features in HOL's functional language is based on a non-deterministic state monad [9], defined as $\texttt{nds\_monad}(\alpha, \sigma) = \sigma \to \mathcal{P}(\alpha \times \sigma)$. The parameters α and σ denote the types of return values and states. We call the monad's elements *computations*. We define the sequential composition (`bind`, written $\gg=$) and `return` monad operators as usual, and provide a non-deterministic choice construct (written $[+]$). We use a function *to_rel* : $\texttt{nds\_monad}(\alpha, \sigma) \to \mathcal{P}(\sigma \times \sigma)$ to derive outer-layer transition relations from given computations by simply forgetting the return values.

Hardware failures can, in reality, happen asynchronously, at any time. However, the PMM can only observe them through the LLI. We thus model them as happening synchronously (and non-deterministically), upon calls to the LLI. Restarts are also asynchronous in reality. They transfer control to the startup procedure. However, it is impossible to model the exact start and end times of this transfer as well as the precise system state handed to the startup procedure, without getting into electrical properties of circuits. All models of restarts are thus necessarily approximations - they must choose a granularity and approximate the effect on the state. Existing structured models (such as [10,4]) choose the granularity of a language statement, inserting non-deterministic restarts between statements. Fortunately, one observation allows us to enlarge this granularity and simplify our model: the persistent memory is accessed only during LLI calls. Hence, restarts outside of LLI calls can only affect the volatile memory, and their effects can be (over)approximated by inserting restarts only right before and after LLI calls, and allowing them to arbitrarily modify the volatile memory. The effect of restarts during an LLI call is call-specific (e.g., setting a block's contents to an arbitrary value during a write). We thus model all restarts as synchronous, by putting them in and around LLI calls.

Since restarts trigger a transfer of control from arbitrarily deep levels of the call stack, we chose to model them as exceptions. We also use exceptions for error

handling. For convenience, we model restarts with a distinguished exception. We thus transform the non-deterministic state monad into a PMM restart-exception monad, defined as $\texttt{pre\_monad}(\alpha, \epsilon, \sigma_v, \sigma_p) = \texttt{nds\_monad}(1 + \epsilon + \alpha, \sigma_v \times \sigma_p)$. Here, α represents (normal) return values, ϵ represents (regular) exceptions, 1 is the unit type representing the restart exception. Moreover, the state is partitioned into the volatile (σ_v) and persistent (σ_p) components. We lift `bind` and `return` as expected and define a `try/catch` construct for handling regular exceptions.

We also define a tryR/catchR construct to handle the restart exception. Here, the "handler" is normally the `startup` procedure. However, this construct does not suffice to accurately model the possibility of `startup` being itself interrupted by a restart. Hence, we need a construct for repeated restarts. As a first step, we define the restarting (R) and non-restarting (N) *projections* of a computation m : $\texttt{pre\_monad}(\alpha, \epsilon, \sigma_v, \sigma_p)$, i.e., $m{\Downarrow_R}$ of type $\texttt{nds\_monad}(1, \sigma_v \times \sigma_p)$ and $m{\Downarrow_N}$ of type $\texttt{nds\_monad}(\epsilon + \alpha, \sigma_v \times \sigma_p)$. Now, we inductively define the desired repetition construct for a given handler h, written $\mathsf{rec\_tryR}(h)$, by preceding a single run of $h{\Downarrow_N}$ by zero or more runs of $h{\Downarrow_R}$ and lifting the resulting computation back to the `pre_monad`. We then define $\mathsf{tryR}\ m\ \mathsf{catchR}^*\ h$ by $\mathsf{tryR}\ m\ \mathsf{catchR}\ \mathsf{rec\_tryR}(h)$. These constructs allow us to adequately model our API calls. For instance, the (outer layer) transition corresponding to the `fetch` API call is defined as $\mathit{to\_rel}(\mathsf{tryR}\ \texttt{fetch}(\mathit{ind})\ \mathsf{catchR}^*\ \texttt{startup})$.

3.3 Compositional Reasoning

There are two kinds of properties we wish to prove of our monadic computations. First, we want to establish properties of individual computations, expressed as Hoare triples. These are denoted $\{\!|P|\!\}\ m\ \{\!|Q|\!\}$, where Q binds the return value of the computation m. When we care only about non-restarting results, we use the following variant: $\{\!|P|\!\}\ m\ \{\!|Q|\!\}_N = \{\!|P|\!\}\ m{\Downarrow_N}\ \{\!|Q|\!\}$. Second, we reduce the refinement condition (Ref2) from Section 3.1 to a monadic variant expressed as a *relational Hoare tuple* between pairs of monadic computations:

$$\{\!|R|\!\}\ m_a\ m_c\ \{\!|S|\!\} = R \subseteq \{(s_a, s_c) \mid \forall v_c\ s_c'.\ (v_c, s_c') \in m_c(s_c) \longrightarrow (\exists v_a\ s_a'.\ (v_a, s_a') \in m_a(s_a) \wedge (s_a', s_c') \in S(v_a, v_c))\}$$

Informally, given a pair $(s_a, s_c) \in R$, any value-state pair that can be obtained by running m_c on s_c, must be related via the post-relation S to some value-state pair obtained by running m_a on s_a. Unlike in the transition system setting, S is parametrized by return values and independent of R. Hence, this formulation is more general than the condition (Ref2). Defining $(s,t) \in \mathit{eq}(U)(v,w)$ iff $v = w$ and $(s,t) \in U$, we can recover (Ref2) (with the additional equality constraint on values) by $\{\!|R|\!\}\ m_a\ m_c\ \{\!|\mathit{eq}(R)|\!\}$. Same as for triples, we define $\{\!|R|\!\}\ m_a\ m_c\ \{\!|S|\!\}_N = \{\!|R|\!\}\ (m_a{\Downarrow_N})\ (m_c{\Downarrow_N})\ \{\!|S|\!\}$.

Starting from the work described in [11], we have embedded a *relational Hoare logic* [12] in Isabelle/HOL to reason compositionally about relational Hoare tuples. For instance, if both m_a and m_c are sequential compositions, a proof rule decomposes the Hoare tuple into two, one for each component computation.

Similar rules exist for other constructs such as try/catch. These decomposition rules are applicable if the two related implementations share the same structure. Usually, we apply them for as long as possible, until we are left with proving Hoare tuples between pairs of "small" monadic operations. At this point, the proof obligations usually become simple enough to discharge them by unfolding the relevant definitions and using Isabelle's proof automation. This decomposition strategy might fail, however, either because the two implementations have different structures, or because the rules yield unprovable goals. For these cases we have to derive two important novel rules, which we present next.

The first one relates a restart handler m with its repeated version realized as $\mathsf{rec\_tryR}(m)$. This rule is typically used with the `startup` procedure, which checks the system state and repairs inconsistencies; if it is itself restarted, we would intuitively expect it to pick up where it left off (at least when viewed abstractly enough). That is, a restarting run of the procedure, followed by a non-restarting run does not yield more results than just a single, non-restarting run. This property can be considered as a form of *idempotence* and is captured in the premise of the following inductively justified proof rule:

$$\frac{\{\!|Id|\!\}\ (m{\Downarrow_N})\ (m{\Downarrow_R};\ m{\Downarrow_N})\ \{\!|eq(Id)|\!\}}{\{\!|Id|\!\}\ m\ \ \mathsf{rec\_tryR}(m)\ \{\!|eq(Id)|\!\}_N}\ \textsc{Idem}$$

Here, $m_1; m_2 = (m_1 \gg\!= \lambda x.\, m_2)$ is the composition (`bind`) that ignores m_1's result and Id is the identity relation. The conclusion states that m itself retains all the possible non-restarting behaviors of $\mathsf{rec\_tryR}(m)$. At a high enough abstraction level, `startup` becomes simple enough to prove the rule's premise directly by unfolding the definitions.

The other important proof rule allows us to gradually enlarge the granularity of persistent data access in our abstractions. Consider an abstract computation m_a, which uses some atomic persistent memory operation that is realized as a series of persistent memory accesses in m_c. Due to atomicity, m_a has fewer restart points, which causes the standard decomposition rule for tryR/catchR to fail to prove goals as in the conclusion of the following proof rule:

$$\frac{\begin{array}{c}\{\!|R|\!\}\ m_a\ m_c\ \{\!|eq(S)|\!\}_N \\ \{\!|R|\!\}\ (m_a{\Downarrow_R};\ \mathtt{rec\_tryR}(h_a^1))\ (m_c{\Downarrow_R};\ \mathtt{rec\_tryR}(h_c^1))\ \{\!|eq(T)|\!\}_N \\ \{\!|T|\!\}\ (\mathsf{tryR}\ h_a^2\ \mathsf{catchR}^*\ (h_a^1;\ h_a^2))\ (\mathsf{tryR}\ h_c^2\ \mathsf{catchR}^*\ (h_c^1;\ h_c^2))\ \{\!|eq(S)|\!\}_N\end{array}}{\{\!|R|\!\}\ (\mathsf{tryR}\ m_a\ \mathsf{catchR}^*\ (h_a^1;\ h_a^2))\ (\mathsf{tryR}\ m_c\ \mathsf{catchR}^*\ (h_c^1;\ h_c^2))\ \{\!|eq(S)|\!\}_N}\ \textsc{Gran}$$

The first premise requires a refinement between non-restarting computations of m_a and m_c. The second premise is the key to our rule. Intuitively, it states that, in case of a restart, h_c^1 completes the (non-atomic) operation of m_c and matches the behavior of the abstract counterpart. The third premise is similar to the conclusion, but concerns h_a^2 and h_c^2. We can prove it either using the standard proof rule for tryR/catchR, or by reapplying the rule Gran on this premise if h_c^2 uses the same non-atomic operation as m_c. The last two premises are connected via an intermediate relation T.

This rule works in synergy with the IDEM rule: if we prove the idempotence of h_c^1, we can simplify the second premise of GRAN by dropping the `rec_tryR` constructs. The restarting behavior of h_a^1 is then no longer constrained by the premises of GRAN, allowing us to remove restarts from h_a^1 completely. This simplifies the abstract model and facilitates the idempotence proof for the entire restart handler. We will employ the rule GRAN to eliminate the nested transactions managing the metadata redundancy (see Section 4.3).

3.4 Properties and their Preservation

We formalize the system requirements either as external invariants or as observation-based Hoare triples, i.e., triples of the form $\{\!|obs^{-1}(P)|\!\}\ m\ \{\!|\lambda v.\, obs^{-1}(Q(v))|\!\}$ for sets of observations P and $Q(v)$. Preservation of external invariants holds trivially: if S_c implements S_a via π, then $oreach(S_a) \subseteq J$ implies $\pi(oreach(S_c)) \subseteq J$ (or, equivalently, $oreach(S_c) \subseteq \pi^{-1}(J)$). We also show that observation-based Hoare triples are preserved under implementation. Hence, we can prove the requirements on the most abstract (and thus simplest) model possible and transfer them onto the concrete model. We give an example in Section 4.4.

Two implicit system requirements are termination and deadlock freedom, where the latter means that no branch of a (non-deterministic) computation yields an empty set of results. We informally argue that these properties hold. The primitives used in our concrete model neither deadlock nor use nonterminating constructs, and the results compose since HOL is a logic of total functions.

4 The Models

This section gives an overview of our development. We first present the abstract model, followed by an overview of the concrete implementation. Then we describe the series of models and refinements connecting these two. Finally, we sketch our formalizations and proofs of the requirements from Section 2.2.

4.1 The Abstract Model

This model directly represents the abstract memory view exposed by the API, as introduced in Section 2. Its persistent state component is realized by the following record type.

```
record abs_mem =
  memory : index → contents
  reg_health : index → log_health
  global_health : log_health
```

The `memory` field models the abstract memory. For each region (i.e., index), the `reg_health` field tracks its health, which is either 'ok', 'degraded', or 'dead', depending on whether any permanent failures have happened to it. Similarly, `global_health` records failures that are not directly related to individual regions.

It serves to capture those behaviors of concrete models, where card failure or degradation occur during the handling of metadata.

The volatile state component consists of the card mode. The observation function $\texttt{obs}_6$ is the identity on all the fields except for `memory`, which it maps to an observation `view` : $(\mathit{index} \to \mathit{contents})_\perp$, thus formalizing the abstract memory view. The `view` is $\perp$ if there is a 'dead' region, in which case the card goes into `Fail` mode and the memory contents become inaccessible to the user.

Even at this level of abstraction, the procedures are not entirely trivial, since they need to capture the variety of possibilities present in the concrete models. To get a flavor of what they look like, consider the definition of `fetch`:

```
fetch6(ind) ≡ do {
    fail_if_fail_mode; do {
       cnt ← read_success(ind);
       degrade [+] skip;
       throw_mode_error(EC(cnt));
       return(cnt)
    } [+] fail [+] restart_mangle_sp
  }
```

where we use Haskell-like `do` notation for `bind`. Here, `fail_if_fail_mode` checks that we are not in `Fail` mode; `read_success` reads the contents of the selected region if possible; `degrade` and `fail` respectively degrade and fail one or more regions and set the card mode appropriately; `throw_mode_error` checks the card mode and potentially throws the appropriate (possibly value-carrying) exception; and `restart_mangle_sp` restarts, lowering the health of zero or more regions.

Still, the definitions are simple enough to keep the proofs performed on this model reasonably easy. This includes the idempotence of the `startup` procedure, which, by the restart reduction rule IDEM from Section 3.3, allows us to remove the `rec_tryR` construct from the (outer-layer) transitions.

4.2 The Concrete Model

The concrete model contains our implementation of the PMM algorithms. It is based on informal descriptions provided by IBM researchers, and discussions with their PMM developers. Currently, it exists only in terms of Isabelle/HOL definitions. That is, it is neither extracted from a program executable on the coprocessor, nor do we synthesize code for it. We thus verify the PMM algorithms rather than a concrete system. However, our implementation is roughly at the same level of abstraction as what one might see in, e.g., C++ code, in that almost all the statements could be mapped 1-1 to C++ statements (save for unfolding monadic maps and folds and equality checks on lists). It consists of about 700 lines of (Isabelle) code. While we had some freedom in the implementation, the algorithm itself was fixed (and already deployed in the IBM 4765). Its verification is therefore essentially post-hoc, making our task more challenging. We now give a brief overview of the algorithm and its data structures.

The concrete model uses two administrative regions to store metadata. The first is used for checking data integrity. It stores the checksums of all logical

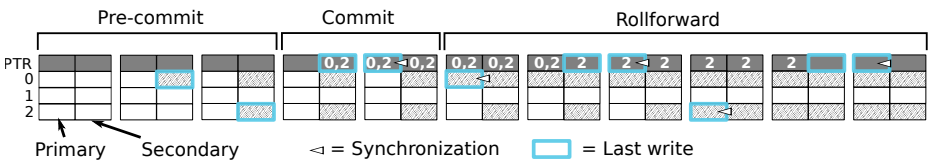


Fig. 3: A sample PMM update operation (regions 0 and 2)

blocks, including its own instances. These checksums are realized with hash functions, and the region is thus named the *hash region*. In our model, we assume the hash function to be perfect, that is, injective. The second is the *pending-transactions register* (PTR), used by the `startup` procedure to "break ties" between instances of user regions, as will be explained shortly.

The centerpiece of the system is the update algorithm. Figure 3 sketches a sample execution, where we write new contents to the regions 0 and 2. Each image shows the state of the memory at a different update step. We divide the process into three stages: pre-commit, commit, and roll-forward.

The two instances of each region are referred to as *primary* and *secondary*. In the pre-commit stage, the new content is sequentially written to the secondary instances of each target region. Then, during the commit stage, we record the set of updated regions' indices (the domain of our update map) in the PTR. This will give precedence to the corresponding secondary instances in the `startup` procedure, in case the system is abruptly restarted. The final stage is the roll-forward stage, where we progressively synchronize the two instances of every freshly-written region, by successively overwriting the contents of the primary instances with the contents of the secondary ones. In each iteration, once the instances are synchronized, we remove the region index from the PTR.

Missing from the diagram are updates to the hash region. Every single step shown actually entails three block writes. First, we write the new content to the target instance, and then update the two corresponding hashes, first in the secondary hash region instance, and then in the primary one.

Also missing is the treatment of restarts and hardware failures. These complicate matters greatly, as a number of special cases arise, especially if the failures occur in instances of the administrative regions. For reasons of space, we will only look at restarts here, giving a short account of the `startup` procedure.

This procedure brings the system into a maximally redundant state. It first synchronizes the instances of the hash region. Both directions of synchronization are possible, depending on the exact scenario. They correspond to a "mini" roll-back or roll-forward, and result in either a failure or a single value in both instances. We can thus view writes to the hash region as implicit *nested transactions* within our system. Synchronization is then also performed in the PTR, forming another layer of transactions on top (since writes to the PTR also involve writes to the hash region). At this point we have unambiguous metadata. The procedure next iterates through all of the user regions, again performing checks and synchronizations as necessary. To determine the direction of the syn-

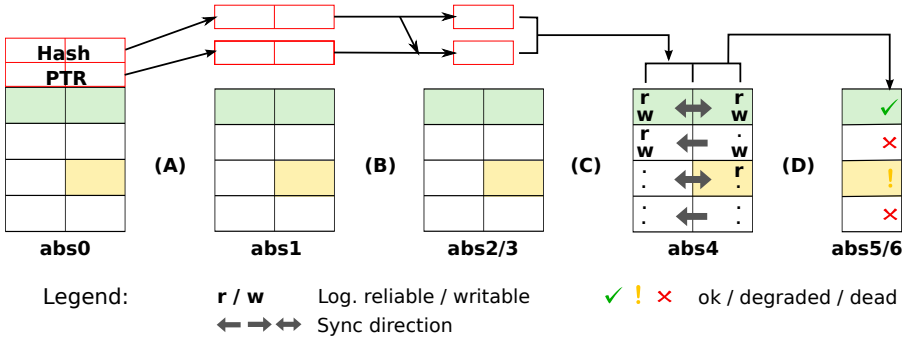


Fig. 4: Model abstractions

chronization, it needs to figure out which of the two instances is *current*. The criterion is as follows: if the hashes of both instances match, both are current. Otherwise, we examine the PTR. If it contains the region's index, only the secondary instance is current, otherwise only the primary one is current. If the synchronization completes successfully, the index is removed from the PTR.

The global effect of a restart on an update thus depends on the stage where it occurs: during pre-commit, the state is rolled back; during roll-forward, the update is applied; and during commit, either is possible, as the nested transactions (to the PTR and the hash region) can still be rolled either back or forward.

4.3 Abstractions

We now sketch the refinement between the abstract and the concrete models, explaining the intermediate models and their relations. We build five such models, in a bottom-up fashion. At first, we tried the more conventional top-down approach; but this made finding the right abstractions of our fixed target hard, as the many different failure and restart behaviors would often only creep up low in the stack, breaking the models higher up. Going bottom-up exposed them more quickly, and allowed us to gradually build usable abstractions.

Figure 4 gives a schematic view of our abstractions in four main steps. We (A) extract the metadata from the memory in a preparatory step; (B) successively remove the redundancy it contains by merging the different instances, giving us unambiguous metadata; (C) interpret the metadata in a more abstract way; (D) merge the pairs of user regions' instances. Here are some additional details of this process. To simplify the presentation we elide most details about the blocks' health status from the figure and the description below.

abs0: Concrete model. As described in Section 4.2.

abs1: Extract hash and PTR regions. An auxiliary step. Hash and PTR instances are pulled out of the memory, leaving only the user regions there.

abs2/3: Eliminate metadata redundancy. We make the nested metadata transactions described in Section 4.2 atomic, using the proof rule Gran introduced in Section 3.3. We achieve this by successively collapsing the pairs

of hash and PTR region instances into a single instance. This eliminates the complexity of keeping the metadata copies in sync and provides us with unambiguous metadata. The simulation relation states that an abstract administrative (hash or PTR) region coincides with a concrete instance whenever that instance is not 'dead' and its integrity is intact, i.e., its computed hash matches the one stored in the hash region.

abs4: Abstract the administrative and status information. We abstract the hash and PTR regions into a combination of per-instance reliability and writability flags, and a per-region *arrow* field. A region instance is *reliable* if neither it nor the hash region is 'dead' and its integrity is intact (i.e., its computed hash equals its stored hash). It is *writable* if both it and the hash region are 'ok'. The arrow indicates the possible directions of instance synchronization for each region. A region instance is current (as defined in Section 4.2 in terms of the administrative regions) exactly if the arrow is bi-directional or points away from the instance.

abs5/6: Eliminate user regions' redundancy. Abstract model. The persistent state becomes the one described in Section 4.1. It is obtained by collapsing the two user region instances into one. Each `abs5` region matches all of its reliable and current `abs4` region instances. If no such instance exists, the region's contents are arbitrary and its health status is 'dead'. Otherwise, the status is either 'ok' (if both instances are writable), or 'degraded' (if at least one is unwritable). In `abs5`, update operations are still performed sequentially and region-wise. We turn these into one-shot atomic updates in `abs6` and replace repeated by single restarts as sketched in Section 4.1.

Our models' observation functions are identities except that, in those models obtained by collapsing instances, the observation of the resulting collapsed field becomes $\perp$ when no reliable and current instances are available. Thus, the concrete observation function $\texttt{obs}_0$ is the identity and the abstract observation function $\texttt{obs}_6$ is as described in Section 4.1. To relate our specifications, we also need mediator functions that are consistent with the simulation relations. Since the inverses of the simulation relations sketched above are functional or almost functional, each mediator function is basically a facsimile of its associated (inverse) simulation relation, again up to the possible mapping to $\perp$. Their composition π maps the concrete observations to the abstract ones.

4.4 Establishing the Requirements

Next, we give a brief overview of how we have formalized the requirements from Section 2.2 and verified in Isabelle/HOL that the concrete model satisfies them.

Requirements (R1-R4) describe properties of individual API calls, which we express and prove as observation-based Hoare triples on `abs6`. We state and prove requirements (R5) and (R6) as external invariants of `abs4`, since these refer to individual region instances. Our refinement proofs and property preservation theorems then enable us to transfer these properties onto the concrete model. We will sketch this on the example of our main requirement, (R1).

On the abstract model, we can state this property using the following two sets of observations, where `Inl` and `Inr` are the left and right constructors of the sum type, corresponding to exceptional and normal termination respectively.

```
view_in(S) = {(sv, sp) | view(sp) ∈ S}
view_post_upd(v, u, r) = case r of
    Inl _ ⇒ view_in({v ◁ u, v, ⊥})
  | Inr _ ⇒ view_in({v ◁ u})
```

The following two Hoare triples then express (R1) on the abstract and concrete models respectively. We prove the first one directly and use our preservation theorems to derive the second one from the refinement results (Section 4.3).

$$\{\!|obs_6^{-1}(view\_in(\{v\}))|\!\}$$
$$\quad tryR\ update_6(u)\ catchR\ startup_6$$
$$\{\!|\lambda r.\ obs_6^{-1}(view\_post\_upd(v, u, r))|\!\}_N$$

$$\{\!|obs_0^{-1}(\pi^{-1}(view\_in(\{v\}))) \cap reach(S_c)|\!\}$$
$$\quad tryR\ update_0(u)\ catchR*\ startup_0$$
$$\{\!|\lambda r.\ obs_0^{-1}(\pi^{-1}(view\_post\_upd(v, u, r)))|\!\}_N$$

The latter triple constrains the behavior of the update API call of the concrete model. It states that, if the call is performed in any reachable concrete state which maps (via $\pi \circ \mathtt{obs}_0$) to an abstract memory view `v`, the resulting state will map to a view in $\mathtt{upd\_post}(\mathtt{v}, \mathtt{u}, \mathtt{r})$. Notice that the property on the concrete model encompasses an arbitrary number of restarts and calls to `startup`.

5 Related Work

Two transaction mechanisms similar to the one described here have been studied before in the literature, both of them targeting smart cards. One is due to Sabatier and Lartigue [2], who use the B method for development and verification. As usual in B, the system is modeled as an (unstructured) transition system, which makes modeling restarts easy. Their main proof technique is refinement. From the final model, they derive a C implementation by hand, without a formal link to the B development. Our first attempt also followed a similar modeling approach, using an Event-B inspired framework in Isabelle/HOL. However, the considerations of hardware failures render our system more complex than theirs. Since restarts force a small event granularity, the models quickly became unmanageable, due to the large number of events and their unstructured nature.

Another transaction mechanism was proposed by Hartel et al. [3,4]. They combine Z notation and SPIN [3] (resp. JML in [4]) to analyze a C implementation, but the unclear relationship between the different formalisms and the lack of machine-checked proofs obscure the resulting guarantees.

Andronick [10] discusses a general verification methodology for reasoning about C programs under restarts, but aimed at transaction mechanisms. Her approach is the one most similar to ours, in that restarts are modeled as exceptions in a structured input language, while allowing for an arbitrary number of

successive restarts to be analyzed. Verification is performed directly on C source code, by leveraging the Why/Caduceus tool. However, her model of restarts does not include any effects on the state and the paper describes only a toy case study. It also mentions a larger one, but without providing any details.

The PMM could also be viewed as a highly primitive file system. In response to Hoare's Grand Verification Challenge, Joshi and Holzman [13] propose verifying a file system as a "mini challenge", identifying restarts and hardware failures as major hurdles in overcoming it. Despite some progress, the challenge still stands open. While the PMM is a far cry from a full-blown POSIX file system, we may claim to have completed a micro challenge with its verification.

6 Conclusion

We have presented our verification of an industrially deployed persistent memory manager. The main challenges to the PMM's correctness (and thus its verification) stem from the rampant non-determinism caused by the combination of possible restarts and hardware failures. The latter have not been considered in the relevant literature before, and they greatly increase the system complexity, forcing us to develop a verification approach which could scale appropriately.

Its key points are as follows. We use a structured (rather than event-based) model. This helped us keep the models understandable, eased discussions with IBM researchers, and enabled compositional reasoning. Modeling restarts synchronously significantly reduced the number of cases we had to consider in the proofs. We identify the concepts of idempotence and nested transactions, and provide two related proof rules, allowing us to tackle the system complexity piece by piece. We believe that our approach is applicable to a class of related systems such as smart cards and file systems (e.g., the 'mini' challenge from [13]).

All our Isabelle/HOL theories amount to around 39,000 lines. These are composed of the modeling and reasoning infrastructure (~12,000 lines), the models (ranging from ~700 for the concrete to ~200 lines for the abstract model), the refinement proofs (~11,000 lines), and the invariant proofs (~9,000 lines). We approximate our development effort at somewhere between 1 and 1.5 person years. The choice of Isabelle/HOL was a mixed bag. HOL's expressiveness was crucial for representing our system's unorthodox features. While Isabelle's connection to external provers helped a great deal, we still had to implement several custom tactics in order to obtain a sufficient degree of automation.

Our development lacks an executable implementation. However, we believe that deriving one from our concrete model would only require a modest effort, leveraging modern Isabelle tools for C code. Unfortunately, non-technical barriers would likely prevent a deployment of an implementation on actual devices, thus disincentivizing us from pursuing this further.

Our work leaves open some interesting research questions. Error resilience is partly reflected in our requirements, but our formalization does not quantify it, offering no way to compare it in two systems. One possibility to address this would be to switch to a probabilistic model. Furthermore, how should one scale

the verification to a full-blown file system? We believe that currently the only feasible method would be a development from scratch and with verification in mind, as in [6]. Even so, this would still require further advances in modeling and reasoning techniques. In particular, it would be interesting to see how to facilitate proofs of idempotence, as well as proving and composing (nested) transactions.

Acknowledgements This work was supported by the Zurich Information Security Center and IBM Open Collaborative Research funding. We thank T. Visegrady of IBM Research Zurich for our collaboration, and D. Basin, A. Lochbihler, B. T. Nguyen, and G. Petric Maretić for their careful proof-reading.

References

1. Arnold, T.W., Buscaglia, C., Chan, F., Condorelli, V., Dayka, J., Santiago-Fernandez, W., Hadzic, N., Hocker, M.D., Jordan, M., Morris, T., Werner, K.: IBM 4765 cryptographic coprocessor. IBM Journal of Research and Development **56**(1.2) (2012) 10:1–10:13
2. Sabatier, D., Lartigue, P.: The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. Formal Methods in System Design (2000) 245–272
3. Hartel, P., Butler, M., de Jong, E., Longley, M.: Transacted memory for smart cards. In: FME 2001: Formal Methods for Increasing Software Productivity, Springer (2001) 478–499
4. Poll, E., Hartel, P., de Jong, E.: A Java reference model of transacted memory for smart cards. In: Proceedings of the 5th conference on Smart Card Research and Advanced Application Conference (CARDIS'02). (2002) 1–14
5. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer-Verlag (2002)
6. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP). (2009) 207–220
7. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2) (1991) 253–284
8. Sprenger, C., Basin, D.: Refining key establishment. In: Proceedings of Computer Security Foundations Symposium (CSF), IEEE. (2012) 230–246
9. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (July 1991) 55–92
10. Andronick, J.: Formally proved anti-tearing properties of embedded C code. In: 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006). (November 2006) 129–136
11. Sprenger, C., Basin, D.: A monad-based modeling and verification toolbox with application to security protocols. In: Theorem Proving in Higher Order Logics. LNCS. Springer Berlin Heidelberg (2007) 302–318
12. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Proc. Principles of Programming Languages (POPL), ACM (2004) 14–25
13. Joshi, R., Holzmann, G.J.: A mini challenge: build a verifiable filesystem. Formal Aspects of Computing **19**(2) (June 2007) 269–272