A case study on formal verification of the anaxagoros hypervisor paging system with frama-C

Allan Blanchard, N. Kosmatov, M. Lemerre, Frédéric Loulergue

A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C*

Allan Blanchard[1,3], Nikolai Kosmatov[1],
Matthieu Lemerre[1], and Frédéric Loulergue[2,3]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Inria πr^2, PPS, Univ Paris Diderot, CNRS, Paris, France
[3] Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
firstname.lastname@univ-orleans.fr

Abstract. Cloud hypervisors are critical software whose formal verification can increase our confidence in the reliability and security of the cloud. This work presents a case study on formal verification of the virtual memory system of the cloud hypervisor Anaxagoros, a microkernel designed for resource isolation and protection. The code under verification is specified and proven in the Frama-C software verification framework, mostly using automatic theorem proving. The remaining properties are interactively proven with the Coq proof assistant. We describe in detail selected aspects of the case study, including parallel execution and counting references to pages, and discuss some lessons learned, benefits and limitations of our approach.

Keywords: deductive verification, interactive proof, cloud hypervisor, Frama-C, specification, concurrency

1 Introduction

Recent years have seen a huge trend towards mobile and Internet applications. Well known applications are moving to the cloud to become "software as a service" offers. At the same time, more and more of our data is in the cloud. It is thus necessary to have reliable, safe and secure cloud environments.

Certification of programs in critical systems is an old concern, while a recent trend in this area is to *formally verify* the programs, the tools used to produce them [1, 2] (and even the tools used to analyze them), and the operating system kernel [3] used to execute them. This formal verification is mostly done using interactive theorem provers, and sometimes automated provers.

Anaxagoros [4] is a secure microkernel that is also capable of virtualizing pre-existing operating systems, for example Linux virtual machines, and can therefore be used as a hypervisor in a cloud environment. One distinctive feature of

* This work has been partially funded by the CEA project CyberSCADA and the EU FP7 project STANCE (grant 317753).

Anaxagoros is that it is capable of securely executing hard real-time tasks or operating systems, for instance the PharOS real-time system [5], simultaneously with non real-time tasks, on a single chip or on a multi-core processor. Our goal is to formally verify the prototype C implementation of Anaxagoros, starting with its most critical components. In this paper we focus on the virtual memory system of Anaxagoros and use the Frama-C toolset [6] for conducting the verification.

Frama-C is a platform for static analysis, deductive verification and testing of critical software written in C. It offers a collection of plugins for source code analysis. These plugins could be used in cooperation for a particular verification task. They interact through a common specification language for C programs: ACSL [6, 7]. In this work, the specifications are written in ACSL, and the weakest precondition calculus plugin Wp of Frama-C together with SMT solvers are used to provide automatic proof for most properties. Some remaining proof obligations (that were not proven automatically) are proven in the interactive proof assistant Coq [8].

The contributions of this paper include a case study on formal verification of a critical module of a Cloud hypervisor. Assuming a sequentially consistent memory model, we performed the verification for both sequential and concurrent execution for one of the key parts of the virtual memory module related to setting new page mappings. We show how a simulation-based approach allows us to take into account concurrent execution using the Frama-C plugin Wp that does not natively support parallel programs. One advantage of its usage is the possibility to perform the proof for most specified properties *automatically* with a very reasonable effort. Only a few lemmas in this case study have to be proven manually, and Wp allows the user to conveniently complete their proofs in the interactive proof assistant Coq, where the Coq statements to be proven are automatically extracted based on the specified code.

Moreover, the verification in this case study can be considered completely formal under the hypothesis that other functions do not interfere on the same variables (memory page mappings) with the function that we verify. That is realistic given that these mappings can be changed only by a couple of functions that can be included into the case study. On the other hand, we argue that, even seen as a partial formal verification, such a study of a critical module in isolation can still be quite efficient to avoid security issues. Finally, we argue that, even done under the assumption that the memory model is sequentially consistent, the presented case study remains valid for weak memory models.

Outline. The paper is organized as follows. Section 2 presents the Anaxagoros hypervisor and its virtual memory system. The verification of this system is described in Section 3, where we detail particular issues of the case study including simulation of parallel execution (Section 3.1), counting references to pages (Section 3.2), automatic proof with Frama-C (Section 3.3) and interactive proof with Coq (Section 3.4). Section 4 provides a discussion of the approach, some lessons learned and axes of improvement. Finally, Section 5 presents related work, while Section 6 gives a conclusion and future work.

2 The Anaxagoros Virtual Memory System

Anaxagoros [4, 9] is a secure microkernel and hypervisor developed at CEA LIST, that can virtualize preexisting operating systems, for example, Linux virtual machines. It puts a strong emphasis on security, notably resource security, so it is able to provide both quality-of-service guarantees and an exact accounting (billing) of CPU time and memory provided to virtual machines, thus satisfying requirements of cloud users.

A critical component to ensure security in Anaxagoros is its *virtual memory system* [9]. The x86 processor (as many other high-end hardware architectures) provides a mechanism for *virtual memory translation*, that translates an address manipulated by a program into a real physical address. One of the goals of this mechanism is to help to organize the program address space, for instance, to allow a program to access big contiguous memory regions. The other goal is to control the memory that a program can access. The physical memory is split into equally sized regions, called *pages* or *frames*. Pages can be of several types: `data`, `pagetable`, `pagedirectory`. Basically, page directories contain mappings (i.e. references) to page tables, that in turn contain mappings to data pages. The page size is 4kB on standard x86 configurations.

Anaxagoros does not decide what is written to pages; rather, it allows tasks to perform any operations on pages, provided that this does not affect the security of the kernel itself, and of the other tasks in the system. To do that, it has to ensure only two simple properties. The first one ensures that a program can only change a page that it "owns". The second property states that pages are used according to their types.

Indeed, the hardware does not prevent a page table or a page directory from being also used as a data page. Thus, if no protection mechanism is present, a malicious task can change the mappings and, after realizing a certain sequence of modifications, it can finally access (and write to) any page, including those that it does not own.

The virtual memory module should prevent such unauthorized modifications. It relies on recording the type of each page and maintaining counters of mappings to each page (i.e. the number of times the page is referred to as a data page, page table, or page directory). The module ensures that pages can be used only according to their type. In addition, to allow dynamic reuse of memory, the module should make it possible to change the type of a page. To avoid possible attacks, changing the page type requires some complex additional properties. (Simplified) examples of properties include: page contents should be cleaned before any type change; still referred pages cannot be cleaned; the cleaning should be correctly resumed after an interruption; the counters of mappings (references) should be correctly maintained; cleaned pages are never referred to; etc.

For instance, in Anaxagoros, the function that sets a mapping to a page inside a page table (illustrated in Fig. 1 and described below) has to update the counters of mappings taking into account the ones it sets and removes. The counters are maintained by an array storing the state of every page, including the number of times it is mapped. The goal is to ensure that for every page,

the real number of mappings to it is at most equal to the value of the counter. Thus, checking if the counter is equal to zero allows us to ensure that the page is no longer referred to before it is cleaned and its type is changed. This prevents possible attacks.

The algorithm also has to take care of the memory management unit cache called the translation lookaside buffer (TLB), which has to be flushed before repurposing a page. Indeed, an entry left in this cache could allow a user program to change a page after it has been cleaned by the kernel. As TLB flushes are costly, the algorithm should avoid them whenever possible, i.e. when we can ensure that there are no entries left in the TLB for a page. We have currently excluded modeling of the TLB from the verification study.

This case study focuses on a simplified version of the virtual memory module that includes most of its key aspects such as data pages and page tables used with respect to the page type, setting new mappings to data pages, maintaining correct counters of mappings and concurrent execution. Simplifications include the replacement of bitfields used in page descriptors by a set of arrays of separate variables, and the fact that we do not take into account the multiple levels of hierarchy of pagetables in the considered properties. Another characteristic of the simplified version is that it splits some functions into smaller ones, and therefore allows to treat a more fine-grained concurrency than the original one.

3 Formal Verification

As any OS, Anaxagoros is inherently concurrent, so we have to deal with concurrency in this case study. Frama-C does not currently treat concurrency, and there are no concurrency primitives available in the considered version of C. Dealing with concurrency becomes even more difficult nowadays because of weak memory models. In this section, we assume a sequentially consistent memory model.

Since no concurrency primitives are available, we consider two classes of functions. The first one is the low-level functions that are atomic, so we verify them as sequential code. We specified all low-level functions of the virtual memory module in ACSL (15 functions, ≈500 lines of annotated C code) and successfully proved them in Frama-C, with the Wp plugin and the SMT solvers Z3, CVC3 and CVC4. This proof is automatic and takes about 90 seconds. This part of the case study was mostly standard and is not presented here in detail.

The second class is higher-order functions that are not atomic, so we decompose them as sequences of atomic instructions for which we *simulate* concurrency. We focus here on the most crucial function of the module that is in charge of setting mappings between pages. The rest of this section presents how we simulate parallelism by modeling the execution context of each thread and creating interleavings, introduces the main properties we want to verify, and describes their proof.

```
1  int set_entry(int fn, int idx, int new){
2    // Step 1 -> read_map_new
3    int c_n = mappings[new];
4    // Step 2 -> test_map_new
5    if(c_n >= MAX) return 1;
6    // Step 3 -> CAS_map_new
7    if(!compare_and_swap(&mappings[new], c_n, c_n+1))
8      return 1;
9    // Step 4 -> EXCH_entry
10   page_t p = get_frame(fn);
11   int old = atomic_exchange(&p[idx], new);
12   // Step 5 -> test_map_old
13   if(!old) return 0;
14   // Step 6 -> FAS_map_old
15   fetch_and_sub(&mappings[old], 1);
16   return 0;
17 }
```

Fig. 1. Function `set_entry` writes page reference `new` into page `fn` at index `idx`

3.1 Simulating Parallel Execution

To take into account parallel execution of code by several threads and to be able to verify it in Frama-C, we simulate parallel execution by sequential code. Let us illustrate it for the C function `set_entry` given at Fig. 1. It sets a mapping (i.e. a reference) to a data page of index `new` into the element of index `idx` of the page table of index `fn`, that can be seen as writing `new` into the corresponding page table element. It has to maintain a correct number of mappings to `new` in the counter `mappings[new]` to remain resistant to attacks. In addition, special care must be taken in case of parallel execution by several threads.

At Step 1 (line 2–3 of Fig. 1), the current number of mappings to `new` is stored in `c_n`. It must be less than the maximal value to avoid an overflow, otherwise the operation is aborted (Step 2, line 4–5). At Step 3 (lines 6–8), the counter is incremented, but only after checking that its value is the same as the one previously read, using an atomic `compare_and_swap` (CAS) operation (note that it could have been modified several times, the only thing that matters is that it must be the same). Step 4 (lines 9–11) retrieves a pointer to the page table of index `fn` (using `get_frame` function), then atomically, again to avoid concurrent access issues, writes `new` into its element at index `idx` and stores the old value in `old`. Step 5 (line 12–13) checks if the old value was a mapping, that is, nonzero, and in that case Step 6 (line 14–15) atomically decrements the number of mappings to `old`, since one mapping has now been replaced by a new one. Notice that if `new` is equal to `old`, the same counter is first incremented and then decremented, as the mapping actually remains the same.

For the sake of verification with Frama-C, we simulate parallel execution of `set_entry` as shown in Fig. 2. Every single step is simulated by a separate simulating function (cf. comments in Fig. 1) that takes a thread number, performs the step for this thread and sets the number of the next step to be executed. Step 0 simply generates input values for the arguments being passed to `set_entry` function. When the execution reaches the end of the function, we assume it goes

```
1  #define NOF  2048 //nb of frames
2  #define THD  1024 //max nb of threads
3  #define MAX   256 //max nb of mappings
4  #define SIZE 1024 //size of a page
5  uint mappings[NOF];
6  uint new[THD], idx[THD], fn[THD];
7  uint old[THD], c_n[THD];
8  uint pct[THD];
9  //@ghost uint ref[THD];
10
11 page_t get_frame(uint fn);
12 void gen_args(uint th){     // Step 0
13   /* generate function args */
14   pct[th] = 1;
15 }
16 void read_map_new(uint th){ // Step 1
17   c_n[th] = mappings[new[th]];
18   pct[th] = 2;
19 }
20 void test_map_new(uint th){ // Step 2
21   pct[th] = (c_n[th] < MAX)? 3 : 0;
22 }
23 void CAS_map_new(uint th){  // Step 3
24   if(mappings[new[th]] == c_n[th]){
25     mappings[new[th]] = c_n[th]+1;
26     //@ghost ref[th] = new[th];
27     pct[th] = 4;
28   }
29   else pct[th] = 0;
30 }
31 void EXCH_entry(uint th){    // Step 4
32   page_t p = get_frame(fn[th]);
33   old[th] = p[idx[th]];
34   p[idx[th]] = new[th];
35   //@ghost ref[th] = old[th];
36   pct[th] = 5;
37 }
38 void test_map_old(uint th){  // Step 5
39   pct[th] = (!old[th])? 0 : 6;
40 }
41 void FAS_map_old(uint th){   // Step 6
42   mappings[old[th]]--;
43   //@ghost ref[th] = 0;
44   pct[th] = 0;
45 }
46 void interleave(){
47   while(true){
48     int th = choose_a_thread();
49
50     switch(pct[th]){
51     case 0 : gen_args(th); break;
52     case 1 : read_map_new(th); break;
53     case 2 : test_map_new(th); break;
54     case 3 : CAS_map_new(th); break;
55     case 4 : EXCH_entry(th); break;
56     case 5 : test_map_old(th); break;
57     case 6 : FAS_map_old(th); break;
58     }
59   }
60 }
```

Fig. 2. Simplified simulation of parallel execution for function `set_entry` of Fig. 1

to Step 0 and can start again with new arguments. Error cases are treated in the same way. Parallelism is simulated by an infinite loop (lines 47–59) that, at each iteration, randomly selects a thread and makes it execute one step.

Values of input and local variables of different threads are kept in arrays (`fn`, `idx`, `new`, `c_n`, `old`) that associate to each thread number the value of the corresponding variable for this thread. The array `pct` stores the current step (program counter) of each thread. Atomic instructions such as `compare_and_swap`, `atomic_exchange` and `fetch_and_sub` can be simulated by standard C instructions in the corresponding simulating functions (since each simulating function is already supposed to be an atomic step in our simulation approach).

3.2 Counters of Mappings and Global Invariant

One of the key properties ensured by Anaxagoros states that the actual number of mappings to any valid page p is at most the value of the corresponding counter `mappings`$[p]$. Along with the property that this counter is under a certain limit, it ensures that the real number of mappings is also under this limit. Notice that we do not count mappings to the page 0 since, in this model, the value 0 in a page table stands for the absence of mapping.

Let Occ_a^v denote the number of occurrences of the value v in an array a (that can be also a page), and Occ^v the number of occurrences of v in all page tables

in memory. We can formalize the global invariant in the following form:

$$\forall e, \texttt{validpage}(e) \Rightarrow \mathrm{Occ}^e \leq \texttt{mappings}[e] \leq \texttt{MAX\_MAPPINGS}.$$

But, while this property is easily proven as maintained by the `set_entry` function after each instruction in monoprocess mode (as this function is not preemptible), it is not precise enough to be used in a multi-threaded context. Indeed, this invariant cannot easily ensure that before we decrement a counter (cf. Step 6 in Fig. 1) it is always greater than 0.

To keep track of values more precisely, we use an invariant in the following form:

$$\forall e, \texttt{validpage}(e) \Rightarrow \exists k, 0 \leq k \ \wedge \ \mathrm{Occ}^e + k = \texttt{mappings}[e] \leq \texttt{MAX\_MAPPINGS},$$

where k can be defined as the gap between the real number of mappings to (that is, occurrences of) e in page tables and the value indicated by its counter. This gap comes from the mappings already counted but not yet effectively set (between Steps 3 and 4 in Fig. 1), and from the valid mappings already removed whose counter is not yet decremented (between Steps 4 and 6 in Fig. 1). In other words, a thread executing `set_entry` creates a gap of 1 for the mappings to `new` at Step 3, then Step 4 removes this gap and creates one for the mappings to `old` (if `old` was a valid mapping, i.e. nonzero), and finally Step 6 removes the last gap (if `old` was not a valid mapping, Step 5 exits the execution before this last step). Therefore, any thread can only create a gap of at most 1 for at most one mapping at the same time.

To model the gap in our simulation approach, we add a ghost array `ref` that associates to each thread number the entry for which the thread creates a gap, and 0 if the thread provokes no gap at the moment. This ghost array is updated by ghost statements at lines 26, 35 and 43 in Fig. 2. This allows to ensure the desired property for `ref` formalized by the ACSL predicate of Fig. 5.

The precise definition for k is $\mathrm{Occ}^e_{\texttt{ref}}$, and the final global invariant is

$$\mathcal{I}: \ \forall e, \texttt{validpage}(e) \Rightarrow \ \mathrm{Occ}^e + \mathrm{Occ}^e_{\texttt{ref}} = \texttt{mappings}[e] \leq \texttt{MAX\_MAPPINGS}.$$

To express and prove assertions invoking the number of occurrences of a value `e` in memory pages, we define in ACSL two logic functions with related axioms to count occurrences of `e` over a range of indices `[from,to[` in one page referred by `t` (Fig. 3), and over a range of page tables `[from,to[` (Fig 4). The left bound of the range is included, while the upper bound is excluded. The label `L` defines the program point where the values are considered. For example, the value Occ^e at label `L` can be now expressed as `occ_m{L}(e,0,NOF-1)`, where `NOF` denotes the number of frames.

The axioms of Fig. 3 define the following cases: the range `[from,to[` is empty so there are no occurrences (axiom `end_occ_a`), or it is non-empty and there are two cases, the rightmost element contains `e`, so the result is one plus the number of occurrences over the reduced range `[from,to-1[` (axiom `iter_occ_a_true`), or it does not, and this is simply the number of occurrences on the reduced range

```
axiomatic OccArray{
logic integer occ_a{L}(integer e, uint* t,
                       integer from, integer to);

axiom end_occ_a{L}:
\forall integer e, uint* t, integer from, to;
  from >= to ==> occ_a{L}(e,t, from, to) == 0;
axiom iter_occ_a_true{L}:
\forall integer e, uint* t, integer from, to;
  (from < to && t[to-1] == e) ==>
    occ_a{L}(e,t,from,to) == occ_a{L}(e,t,from,to-1) + 1;
axiom iter_occ_a_false{L}:
\forall integer e, uint* t, integer from, to;
  (from < to && t[to-1] != e) ==>
    occ_a{L}(e,t,from,to) == occ_a{L}(e,t,from,to-1);
}
```

Fig. 3. Simplified logic function `occ_a` counting occurrences in a subarray

```
axiomatic OccMemory{
logic integer occ_m{L}(integer e,integer from,integer to);

axiom end_occ_m{L}:
\forall integer e, integer from, to;
  from >= to ==> occ_m{L}(e, from, to) == 0;
axiom iter_occ_m_true{L}:
\forall integer e, integer from, to;
  from < to && pagetable[to-1] == true ==>
    occ_m{L}(e,from,to) == occ_a{L}(e,frame(to-1),0,SIZE)
                         + occ_m{L}(e,from,to-1);
axiom iter_occ_m_false{L}:
\forall integer e, integer from, to;
  from < to && pagetable[to-1] != true ==>
    occ_m{L}(e,from,to) ==  occ_m{L}(e,from,to-1);
}
```

Fig. 4. Simplified logic function `occ_m` counting occurrences over a range of pages

(axiom `iter_occ_a_false`). Similarly, the axioms of Fig. 4 define how to count the number of occurrences of `e` in all page tables, hence we need an additional condition: we count occurrences in a page only if it is a page table.

3.3 Proof with the Wp Plugin of Frama-C

Wp [6] is a weakest precondition calculus plugin integrated to Frama-C. Given a C program specified in ACSL, Wp generates proof obligations in the Why3 language that can be discharged with automatic or interactive provers.

To use Wp, we first write ACSL annotations to define the contract of each function as well as a few lemmas (detailed in Sec. 3.4) to help automatic provers. For the code of Fig. 2, our main goal is to ensure that for every simulating function, if the global invariant $\mathcal{I}$ holds before its execution, it is maintained after. Thus, $\mathcal{I}$ is formalized as an ACSL predicate that appears both in the precondition and the postcondition of the contract.

```
predicate pct_imply_for_thread(integer th) =
  (pct[th] <= 3 ==> ref[th] == 0      ) &&
  (pct[th] == 4 ==> ref[th] == new[th]) &&
  (pct[th] == 5 ==> ref[th] == old[th]) &&
  (pct[th] == 6 ==> ref[th] == old[th] && old[th] != 0);
```

Fig. 5. Predicate defining the link between the program counter and the array `ref`

```
lemma occ_a_separable{L}:
\forall integer e, uint* t, integer from, cut, to;
  from <= cut <= to ==>
    occ_a{L}(e,t,from,to) ==
      occ_a{L}(e,t,from,cut)+occ_a{L}(e,t,cut,to);
```

Fig. 6. Example of a lemma in ACSL for counting over two sub-ranges

Other clauses include some routine properties, for example, bounding local variables to the range of authorized values, or defining the relationship between `ref` and the thread's program counter illustrated by the predicate in Fig. 5.

The verified prototype simulating parallel execution of the `set_entry` function contains about 610 lines of code including 530 lines of ACSL annotations. 140 lines are needed for the axioms and lemmas related to occurrence counting. We also define some predicates to express the bounds of the different simulated local variables (about 50 lines). The remaining lines contain function contracts and some assertions necessary to guide the proof. In the function contracts, 200 lines are just duplicates of the actual invariant (about 10 lines), and could be auto-generated (cf. Section 4.2).

The specification of this function, the adaptation of the invariant for the model of concurrency, and the addition of the relation between the program counter and the ghost variable, together with the determination of the assertion needed to guide the proof took about a month for a junior verification engineer.

From the function contracts, WP generates about 320 proof goals, including 190 for the interleaving loop. Except the lemmas, all generated goals are successfully discharged by Z3 (v.4.3.1) or CVC4 (v.1.3) within about 65 sec. on a QuadCore Intel Core i7-4800QM @2.7GHz. We have also investigated if the constant values used for the size of a page (SIZE), the number of frames (NOF) or the maximal number of threads (THD), have an impact on the time needed to discharge the proof obligations. An experiment shows that this time does not depend on these values. Indeed, the axiomatic definition of logical functions prevents the provers from unrolling the recursion when properties involve the number of occurrences of values in arrays.

3.4 Proof of Lemmas in Coq

To facilitate the proof of formulas using the logic functions of Fig. 3 and 4, we state simple lemmas in ACSL that express useful properties of these logic

functions. For each function, we have three lemmas that express the same idea at the corresponding level: for a single page and for all page tables. The proof of these lemmas requires careful induction, paying attention to the right usage of the induction hypothesis and axioms, so they cannot be automatically proven by Z3 and CVC4. WP allows us to complete the proof of goals using COQ. So we first use WP to automatically extract the goals for the lemmas from ACSL into the COQ format, and then perform their proof interactively in COQ.

A good example of a lemma about counting occurrences in a single array is the property shown in Fig. 6. It states that we can split a range `[from,to[` of page elements on which we want to count into two subranges `[from,cut[` and `[cut,to[`, count separately on each of them, and then take the sum to obtain the number of occurrences over the complete range. This is a very useful property as it allows us to partition ranges in order to keep only smaller subranges that changed between two points, saying that "all other elements did not change". The proof of this lemma consists in an induction on `to` compared to `from` and a case analysis on `cut`, the most complex case being proven using the axioms `iter_occ_a_false` and `iter_occ_a_true`.

Another interesting lemma says that if in a range of array elements, none of them changed between two program points, then for any value, the number of its occurrences over the range did not change. The proof is done by a simple induction.

The last lemma says that if only one array element changed to a different value between two labels, the number of occurrences decreases by 1 for the old value, increases by 1 for the new value, and all other values have the same number of occurrences. Its proof uses the two preceding lemmas. We use the first lemma to separate the subrange that changed from those that did not. Then we use the second lemma to prove that the number of occurrences did not change in the unmodified subrange, and finally prove that at the modified location, the number of occurrences respects the desired property.

For the level of all page tables (function `occ_m`), we define similar lemmas and use similar proof ideas. The complete proofs totalize about 300 lines of COQ code and took about 4 days to be written by a junior verification engineer.

4 Discussion

4.1 Weak Memory Model Compliance

The approach we applied to simulate concurrent execution of the function `set_entry` is based on the assumption that it respects an interleaving semantics. Actually, none of modern multi-processors respect this assumption, implementing *weak* (or *relaxed*) memory models that authorize memory access reordering [10]. It can lead to "strange" behavior, like shown in Fig. 7 where "`|`" stands for parallel composition of threads.

Indeed, we cannot find an interleaving that exhibits the `(*)` behavior. However, it can happen on weak memory for two possible reasons. First, as in the

```
1   R0 = R1 = [x] = [y] = 0                Authorized behaviors :
2                                            R0 = 1 /\ R1 = 1
3 // Thread 1:        Thread 2:              R0 = 0 /\ R1 = 1
4   [x] <- 1     |   [y] <- 1               R0 = 1 /\ R1 = 0
5   R0  <- [y]   |   R1  <- [x]             R0 = 0 /\ R1 = 0 (*)
```

Fig. 7. Example of a two-thread program and its possible weak memory behavior

first thread there is no dependency between the write of `x` (line 4) and the read of `y` (line 5), these instructions could be reordered by the compiler or the processor itself. A similar reordering can occur for the second thread. So the reads would be performed before the writes, setting 0 to both `R0` and `R1`. The second reason is that memory writes are added into a store buffer before accessing the real shared memory. So each thread could register its write in its buffer and then read the global shared memory before the write of the other thread hits it, thus reading 0 instead of 1.

For a weak memory model, what is called the "Fundamental Property" by Saraswat et al. [11], is the fact that any program whose sequentially consistent executions do not have any *data race* must only have executions that are sequentially consistent. Any reasonable memory model should have this property. It allows to reason about programs in a weak memory model using sequential consistency. Of course this property should have been proved for the weak memory model, and indeed it has been done for most weak memory models (e.g. [12]).

A data race is a pair of conflicting operations, i.e. two accesses to the same memory address, one of them being a write, that are concurrent, i.e. without any temporal dependencies between them. There are several ways to formalize what it means for two events to be concurrent. One of them is to use a happens-before relation, which is a transitive, irreflexive partial order: one event happens-before another one if they belong to the same thread, and synchronizations introduce pairs in this relation for events in different threads. Concurrent events are events that are not related with a happens-before relation. Thus, if we want to analyze concurrent programs by generating interleavings, we first need to justify that these programs are race-free.

There are several methods to ensure data-race freedom. For example by automatic static analysis [13], or by respecting a programming discipline that adds to a program the guarantee that its execution will respect sequentially consistent behavior by construction [14]. One way to enforce such a programming discipline is to prove the correctness of the program with a program logic such as concurrent separation logic [15].

Actually, for the function `set_entry` of Fig. 1 such justification is trivial as every shared memory access is performed by an atomic routine that flushes write caches, thus introduces a synchronization: we use `compare_and_swap` to increment the mapping counter, `atomic_exchange` to swap the page entry, and `fetch_and_sub` to decrement the mapping counter. Thus, this program does not contain data-races.

We can also justify an (almost) total ordering on the instructions. The function call with argument passing (simulated by Step 0) comes necessarily first.

Then, the next three steps (read, test and CAS) are ordered by their control or data dependencies. In the x86 model, the fence between the CAS (Step 3) and the atomic exchange (Step 4) is implicit, while in a model that does not place this fence (e.g. Power or ARM) we would need to add it explicitly. The test on `old` (Step 5) is in data-dependency with the atomic exchange (Step 4). Finally, the counter decrementation at Step 6 is control-dependent on the test at Step 5.

The read `page_t p = get_frame(fn)` is the only instruction that could be reordered everywhere between the function call (Step 0) and the atomic exchange (Step 4). Since it actually only depends on a static array (used in the implementation of `get_frame`) and the parameter `fn` which are never assigned after the function call, possible reorderings of this read do not change anything in the execution , so we chose to place it near the atomic exchange (cf. Step 4 in Fig. 2).

Consequently, in this case, this simulation-based approach is sound and remains valid for weak memory models. We aimed to know what can be done for concurrent programs with Frama-C provided that they are correctly and fully synchronized. Currently, ensuring that the programming discipline is respected is not done by a dedicated tool, a future work would be to automate this verification.

4.2 Lessons Learned, Benefits and Limitations of the Approach

This case study confirms that an obvious benefit of deductive verification based on automatic theorem provers, combined when necessary with interactive proof, is its cost efficiency. Indeed, most specified properties are proven automatically by modern SMT solvers. The possibility to easily complete unsuccessful proofs afterwards in the interactive proof assistant Coq offered by the Wp plugin appears to be very convenient and allows the verification engineer to focus on really difficult properties, leaving routine proofs to automated tools. The time needed to complete interactive proofs in this study appeared to be much less than the overall effort of code specification.

Another lesson learned in this work is the ability of this approach to treat concurrent code in Frama-C/Wp that originally does not offer this possibility. Moreover, the effort needed to model concurrent context remains reasonable against the specification effort, at least for short functions.

One could argue that this verification study remains valid only if this function is the only one able to access and modify page tables and their properties. Actually, another function, responsible for cleaning pages before changing their type, can also modify them. Its algorithm is however very simple: “for any entry, replace its value by null (we do not count references to the null page) and decrement the counter for the old value”, so we can perform simulation for this part as we did for the `set_entry` function. The proof can be performed in a similar way.

This work also suggests a generic verification approach that can be summed up as follows. Given a concurrent program that respects an interleaving semantic, and a shared region of data that needs to respect a particular invariant, we

analyze in isolation the group of functions that might access it. We model every local variable by an array associating to each thread the corresponding value, while the position of each thread in its execution is modeled by an array of program counters. Every single atomic action should be modeled by a separate simulating function. The interleavings are modeled by a loop that randomly executes a step of a thread. Finally, the global invariant is attached both to the loop and the functions in their contracts.

Since writing the specified simulating program by hand is error-prone, the next step is to make this approach automatic. The program transformation described above is quite simple. Its automation would require to extend ACSL in order to allow more precise specification of concurrent properties (e.g. when some part of the invariant depends on the position of some thread in its execution, cf. the argument leading to the definition of $\mathcal{I}$ in Sec. 3.2) that could be then translated into simulating function contracts and interleaving loop invariant in the simulating program.

We expect this verification approach to have a limited scalability on complete real-sized programs. Indeed, the interleaving loop is very short in our case. Treating numerous functions can require to track a great number of local variables globally, that can make the automatic proof more difficult, typically for the contract associated to the interleaving loop that would become much bigger. Nevertheless, thanks to the automation perspective of the program transformation and the cost-efficiency of deductive verification, conducting in-depth verification of critical algorithms by extracting the interesting part and analyzing it in isolation can still be a practical approach to identify potential problems.

5 Related Work

Klein et al. [3] present formal verification for seL4, a microkernel allowing devices running it to achieve the EAL7 level of the Common Criteria. Another formal verification of a microkernel is described in [16]. Both projects take into account concurrency between the processor and the devices (represented by their drivers), whereas our aim here is to treat the multi-processor concurrency of a particular function. Their verification uses interactive, machine-assisted and machine-checked proof with the theorem prover Isabelle/HOL.

Another recent work on verification of a virtual memory manager [17] relies on the fact that virtual memory managers are constructed in layers, and uses this to structure the proof by successive small refinements, making it easier to achieve and to maintain. A framework is provided to lighten the work needed for refinement and layers definition. The proof is also done interactively, with the Coq proof assistant.

[18] presents a verification of a model of virtualization. Both implementation and verification are done in Coq. Being relatively far from a real implementation, it allows reasoning about isolation between guests on an axiomatic basis modeling hypervisor behavior including caches and TLB. In contrast, our work is interested in low-level details of the real implementation.

Unlike the aforementioned projects, we aim to maximize the amount of automatic proof in our work.

The formal verification of a simple hypervisor [19] uses VCC, an automatic first-order logic based verifier for C. The underlying architecture is precisely modeled and represented in VCC, where the mixed-language system software is then proven correct. Unlike [3] and [16], this technique is based on automated methods. The verification consists in verifying that the invariant of the system is respected by an infinite loop of steps. While VCC is intrinsically concurrent, Frama-C is not. Our goal is to investigate what has to be done to achieve concurrent program proof with Frama-C/Wp, in particular, in order to benefit from the multiple analysis plugins available in the toolset.

In [20], Alkassar et al. report on verification of the translation lookaside buffer (TLB) virtualization, a core component of modern hypervisors. As devices, like memory management units (MMUs), run in parallel with software, they require concurrent program reasoning even for single-threaded software. Their work gives a general methodology for verifying virtual device implementations, and demonstrates the verification of TLB virtualization code in VCC.

As we mentioned previously, [14] presents a programming discipline to write concurrent programs that allow only sequentially consistent behaviors. [21] points out that this method is not sufficient to deal with programs that edit their own page tables and proposes an extension to complete the programming discipline. Instead of considering a precise model of the x86 memory management unit (MMU) [20], it proposes an abstract MMU model that allows to verify that the MMU of a thread will not access page tables of another one. As we explained in Section 2, our analysis does not yet consider the MMUs nor the TLB, and could be extended with a similar approach.

Formal verification nowadays remains very expensive. [22] estimates that the verification of the seL4 microkernel took around 25 person-years, and required highly qualified experts. seL4 contains only about 10,000 lines of C code, and verification cost is about $700 per line of code.

Our present work continues the previous efforts and presents a case study on formal verification of a critical module of a hypervisor in Frama-C. To minimize the verification cost, we use automatic theorem proving as much as possible, complete it by interactive proof when necessary and apply a sound simulation-based approach compliant with weak memory models to deal with parallelism.

The only previous work [23] on verification of Anaxagoros presented partial formal verification, completed by test generation for unproven functions, did not consider parallel execution and did not use interactive proof.

6 Conclusion and Future Work

One of the most critical modules in the Anaxagoros hypervisor is its virtual memory mechanism. We present here the formal verification of a slightly simplified version of it for a sequentially consistent memory model. In this component,

the low-level functions are atomic and we verified them as sequential functions. The ACSL specifications were automatically proven in FRAMA-C using its weakest precondition calculus plugin WP, and the proof obligations discharged by Z3, CVC3 and CVC4.

Higher level functions are no longer atomic. To deal with concurrency we simulated parallelism: the execution context of each thread and interleavings. The verification of its key part, the function that sets mappings between pages, has been performed using this technique. Again, the specifications were written in ACSL and the proofs conducted by Z3 and CVC4. However, in order to write the specifications, we introduced axiomatized functions. Basic results about these functions were needed to allow the SMT solver to conclude, but these lemmas themselves cannot be proven by automatic provers. We used the proof assistant COQ to prove them.

This case study illustrates formal verification of a critical module in isolation, that can be still quite efficient to detect various functionality and security issues such as the recent Heartbleed bug[4] in OpenSSL. The main benefits of our approach include the possibility to conduct most proofs automatically, to reduce interactive proof to a minimum, and to take into account parallel execution.

In order to prove the actual code of Anaxagoros, we should deal with bit vectors. To avoid the need for a lot of interactive proofs, it would be interesting to design a library of basic results for bit vectors that could be then used automatically by automated provers. While the simulation approach was sufficient to deal with this case study, we do not expect it to scale to the whole hypervisor. Therefore, it would be interesting to be able to deal directly with parallelism in FRAMA-C, in particular in the case of weak memory models.

Acknowledgment. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defence. The authors thank the FRAMA-C team for providing the tools and support. Special thanks to François Bobot and Loïc Correnson, the main author of WP, for many fruitful discussions, suggestions and advice. Many thanks to the anonymous referees for their helpful comments.

References

1. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4) (2009) 363–446
2. Leroy, X.: Verified squared: does critical software deserve verified tools? In: POPL 2011, ACM
3. Klein, G., Andronick, J., Elphinstone, K., Murray, T.C., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1) (2014)
4. Lemerre, M., David, V., Vidal-Naquet, G.: A communication mechanism for resource isolation. In: IIES 2009
5. Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., Jacques, M.B.: Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In: AMICS 2011

[4] http://blog.regehr.org/archives/1125

6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A program analysis perspective. In: SEFM 2012
7. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. `frama-c.cea.fr/acsl.html`.
8. The Coq Development Team: The Coq Proof Assistant. `http://coq.inria.fr`
9. Lemerre, M., David, V., Vidal-Naquet, G.: A dependable kernel design for resource isolation and protection. In: IIDS 2010
10. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29**(12) (1996) 66–76
11. Saraswat, V.A., Jagadeesan, R., Michael, M.M., von Praun, C.: A theory of memory models. In: PPoPP, ACM (2007) 161–172
12. Boudol, G., Petri, G.: Relaxed memory models: an operational approach. In: POPL 2009
13. Dabrowski, F., Pichardie, D.: A Certified Data Race Analysis for a Java-like Language. In: TPHOLs 2009
14. Cohen, E., Schirmer, B.: From total store order to sequential consistency: A practical reduction theorem. In: ITP 2010
15. Brookes, S.D.: A semantics for concurrent separation logic. In: CONCUR 2004
16. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel. In: VSTTE 2010
17. Vaynberg, A., Shao, Z.: Compositional verification of a baby virtual memory manager. In: CPP 2012
18. Barthe, G., Betarte, G., Campo, J.D., Chimento, J.M., Luna, C.: Formally verified implementation of an idealized model of virtualization. In: TYPES 2013
19. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: VSTTE 2010
20. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.J.: Verification of TLB virtualization implemented in C. In: VSTTE 2012
21. Chen, G., Cohen, E., Kovalev, M.: Store buffer reduction with MMUs: Complete paper-and-pencil proof. Technical report, Saarland University, Saarbrücken (2013)
22. Klein, G.: From a verified kernel towards verified systems. In: APLAS 2010
23. Kosmatov, N., Lemerre, M., Alec, C.: A case study on verification of a cloud hypervisor by proof and structural testing. In: TAP 2014