Prove your Colorings: Formal Verification of Cache Coloring of Bao Hypervisor

Axel Ferréol, Laurent Corbin, and Nikolai Kosmatov

Thales Research & Technology, Palaiseau, France
{axel.ferreol,laurent.corbin,nikolai.kosmatov}@thalesgroup.com

Abstract. Hypervisors allow sharing of computing resources between applications—possibly of various levels of criticality—that makes them increasingly relevant for modern embedded systems. In this context, memory isolation properties (including low-level cache isolation) are essential to guarantee. This paper presents a case study on formal verification of the cache coloring mechanism implemented in the Bao hypervisor. It proposes an original technique for coloring memory pages and assigning to each virtual machine only pages of certain colors, aimed to provide strong isolation guarantees. The implementation presents several challenges for formal verification, such as bit-level operations, complex arithmetic operations, multiple levels of nested loops, and linked lists. We identify two subtle bugs in the existing implementation breaking the expected guarantees, and propose bug fixes. We provide formal specification for the key functions of the mechanism and verify their (fixed) version in the Frama-C verification platform with a few lemmas proved in the Coq proof assistant. We present our specification choices, verification approach and obtained results. Finally, we outline possible optimizations of the current implementation.

Keywords: deductive verification, Frama-C, cache coloring, Bao hypervisor, memory pages, Coq.

1 Introduction

Hypervisors allow a host system to support multiple guest systems (*virtual machines*, or VMs) by virtually sharing its resources, such as memory and processing. Already intensively used in some domains (e.g. cloud infrastructures), hypervisors become highly relevant today for *critical embedded systems* due to an increasing number of necessary functions and features. Numerous functions have already been added to embedded systems, such as driver assistance or sensor management, and more functions need to be integrated today, for example, artificial intelligence (AI) solutions for mission-critical systems, or further entertainment and connectivity features. In many contexts, it is not possible to add more hardware because of size, weight and cost constraints. To enable this integration, it is necessary to share the same hardware between several functions (or systems), often with different levels of criticality. It can be achieved thanks to virtualization, when each system runs on a separate VM.

Hardware resources can be shared by the hypervisor in two ways: (i) *time sharing:* each VM has access to all resources in turn, i.e. VMs are scheduled; (ii) *partitioning:* each VM has access only to the part of resources dedicated to it. Time sharing requires a more complex and resource-hungry hypervisor, due to the scheduling function. That is why partitioning-based hypervisors (called *static hypervisors*) are more widely used in embedded systems. Static hypervisors allocate all hardware resources to VMs during the hypervisor start-up, so that each resource is allocated to only one VM. In addition, each VM has direct access to its resources, without interception by the hypervisor, which is particularly important for real-time systems. Thus, a static hypervisor seems to be an ideal solution for mixed-criticality systems.

However, some resources must be shared, such as *processor last-level cache* (LLC), which is by definition shared between several cores, each one possibly running a different VM. To tackle this problem, some static hypervisors implement *cache coloring*. The main idea is to split cache—without specific hardware—into several areas, each associated with a color. A color can then be associated with a VM, so that the data of memory pages used by this VM can be stored only in the cache area of the same color. The underlying implementation becomes more complex and highly critical, and its correctness is essential to guarantee.

The purpose of this work is formal verification of the cache coloring mechanism implemented in Bao [1,37], an open-source static hypervisor used in embedded systems. While it proposes an elegant optimized implementation, its code is also challenging for formal verification because it contains non-trivial logic, bit-level operations, complex arithmetic operations, multiple levels of nested loops, and linked lists. During this case study, we identified two subtle bugs[1] in the existing implementation breaking the expected guarantees, and proposed bug fixes[2]. We provide formal specification for the key functions of the mechanism and verify their (fixed) version in the Frama-C verification platform [31,10,33]. The proof requires carefully chosen predicates, ghost code, non-trivial loop invariants and lemmas. Some proof goals are not proved by automatic solvers: we prove them interactively (in Frama-C or in the Coq proof assistant [40,13]). We present our specification choices, verification approach and obtained results, and outline possible further optimizations of the current code.

Contributions. The contributions of this work include:

- a pedagogical presentation of the cache coloring mechanism of Bao;
- an identification of some subtle bugs in its implementation and proposals of bug fixes, as well as suggestions of possible further optimizations;
- formal specification and verification of a subset of (fixed) real-life code of this mechanism in Frama-C, publicly available via a companion artifact [27];
- an overview of key specification choices, verification solutions and results.

[1] present in the code since 2020 (commit d840da).

[2] Shortly before the final submission of this paper, the authors reported the bugs and the suggested fixes to Bao developers, who integrated the proposed fixes into the code (commit ee73f7e in the Bao repository [1] on January 6, 2025).

In a broader sense, this work promotes rigorous software engineering approaches, contributes to an empirical evaluation of modern verification tools, and enriches the record of successful formal verification case studies for critical real-life code in industrially relevant contexts.

Outline. Section 2 presents Frama-C. Bao and cache coloring are described in Sect. 3. The considered implementation is presented in Sect. 4. Section 5 presents the bugs, suggested fixes and optimizations. Section 6 describes key specification choices, verification solutions and results. Finally, Sect. 7 provides some related work and concludes the paper.

2 Frama-C Verification Platform

Frama-C [31,10,33] is an open-source verification platform for C code. It offers various plugins along with a kernel providing basic services for source-code parsing and analysis. The program under analysis can be annotated in ACSL (ANSI/ISO C Specification Language) [11,33], a formal specification language for C, that allows users to express functional properties of programs in the form of *annotations*, such as assertions or function contracts, written in special comments `/*@...*/` and `//@...` . A function contract includes pre- and postconditions (resp., **requires** and **ensures** clauses) expressing properties that must hold, resp., before and after a call to the function. It also includes an **assigns** clause listing (non-local) variables and memory locations that the function is allowed to modify. The **terminates \true** clause specifies that the function must terminate. Users can add *ghost code*, used only for verification purposes and written in annotations `/*@ ghost ... */`. Ghost code can also contain annotations, written in special comments `/@...@/` and `//@...` . ACSL offers built-in predicates and logic functions to express frequent properties such as pointer validity or memory separation, and provides different ways to define new predicates and logic functions. As it is often done, in this document some ACSL notation (e.g. `\forall`, `integer`, `==>`, `<=`, `!=`) is pretty-printed (resp., as $\forall$, $\mathbb{Z}$, $\Rightarrow$, $\leq$, $\neq$).

Frama-C offers a deductive verification plugin called Wp [33] . Given a C program annotated in ACSL, Wp generates the corresponding *proof obligations* (also called *proof goals* or *verification conditions*) that can be proved either by Wp itself, or (through the Why3 platform [28]) by SMT solvers [20,14,9] or an interactive proof assistant like Coq [40,13]. To ensure the absence of runtime errors (RTE), Wp can automatically add necessary assertions and try to prove them as well. In this work, we chose to use Frama-C/Wp due to its capacity to perform deductive verification of industrial C code with successful verification case studies [23] and the fact that it is currently the only tool for C source code verification recognized by ANSSI, the French Common Criteria certification body, as an acceptable formal verification technique for the highest certification levels EAL6–EAL7 [24].

3 The Bao Hypervisor and Cache Coloring

The cache issue. Caches in modern CPUs are organized in levels: each core has a first-level cache, and the data in these caches are replicated in (possibly several levels of) higher-level caches until last-level cache (LLC), shared by all cores. While data from a given page is always stored in the same cache area, the memory-to-cache mapping is not bijective: data from different memory addresses can end up being stored in the same cache area. This reduces isolation guarantees and can potentially increase the risk of (e.g. side-channel) attacks. Moreover, memory addresses mapped to the same cache set compete for space. If a VM running on one core frequently accesses a large amount of data, it can monopolize the shared cache, slowing down other VMs running on nearby cores. This is a serious issue for real-time applications. To prevent this, the hypervisor must ensure that memory pages assigned to different VMs do not overlap in cache.

Cache coloring. Cache coloring is a technique that assigns colors to memory pages such that pages of the same color compete for the same cache sets, while pages of different colors do not compete for the same cache sets. In essence, cache coloring segments the main memory based on cache segmentation. The minimum number of colors is one (i.e., no cache coloring), and the maximum is determined by the number of cache sets of the different caches.

When a hypervisor assigns a unique color to each virtual machine—meaning a VM is loaded exclusively on pages of its color that have not been allocated to other VMs—it ensures that: (i) VMs cannot access each other's data since they reside on separate pages in memory; (ii) VMs do not compete for the same cache sets because their data is stored in cache sets of different colors. Thereby, cache coloring is essential for memory isolation.

Bao. Bao [1,37] is a lightweight open-source static hypervisor specifically designed for embedded systems and real-time applications. It focuses on providing strong isolation between VMs and ensuring real-time guarantees, being thus particularly well-suited for environments where both performance and reliability are critical. Bao elegantly implements a general version of cache coloring where the uniqueness property can be relaxed, that is, each VM accepts a subset of colors. It is crucial to ensure correctness of this implementation, that makes it a highly relevant target for formal verification.

4 Implementation of Cache Coloring in Bao

This section presents an overview of the cache coloring mechanism in the Bao hypervisor (see the real-life code in [1]), and a simplified version of its key functions (given in Fig. 3). Several syntactical changes were realized to make the real-life code more compact and clearer for the paper. The only semantical change is the removal of lock and unlock instructions (in the beginning and the end of function `pp_next_clr`) used to prevent concurrent modifications of a page pool and page allocation statuses, which are classic and orthogonal to our main scope. The semantics of other instructions (with all real-life code optimizations and bit-level operations) was carefully preserved.

4.1 Overview of the Implementation

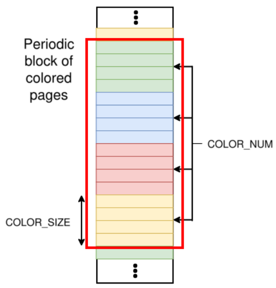

Fig. 1. The main memory layout in Bao with cache coloring, where the periodic block of colors is repeated to engender the coloring of pages for the whole memory.

When the option to use cache coloring is activated, Bao calculates during the boot the number of colors allowed by the hardware. Then, it attributes to every page a single color depending on the page number, so that its data is mapped to a cache area of the same color. `COLOR_SIZE` denotes the number of contiguous pages of the same color in memory, while `COLOR_NUM` represents the number of different colors in memory. In other words, pages are colored into the same color by consecutive groups of `COLOR_SIZE` pages, and the colors of the groups follow a constant sequence that loops every `COLOR_NUM` groups. Thus, the main memory is colored following a specific pattern—a periodic block of `COLOR_NUM*COLOR_SIZE` pages—as illustrated in Fig. 1 (where both constants are equal to 4, so the block contains 16 pages).

In a configuration file, for each VM, the user specifies a set of (possibly several) acceptable colors for pages where the VM will be loaded. When loading a VM, Bao maps the VM's address space into free pages of acceptable colors.

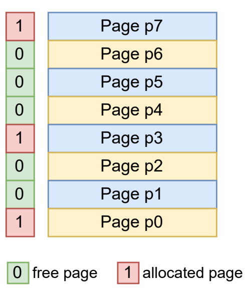

Fig. 2. Example of a pool of memory pages where `COLOR_SIZE` equals to 1 and `COLOR_NUM` equals to 2.

The allocation of suitably-colored pages is handled by function `pp_alloc_clr` (detailed below). It searches for a set $\{p_1, \dots, p_n\}$ of a required number n of free consecutive pages of acceptable colors $c_1, \dots, c_k$. To formalize these requirements for selected pages, it is convenient to introduce the notion of a *pset* (pronounced as *p-set*).

A set of pages $\{p_1, \dots, p_n\}$ is called a *pset* of n pages for acceptable colors $c_1, \dots, c_k$ if: (i) each page p_i in the set has one of the acceptable colors $c_1, \dots, c_k$; (ii) the pages of the set are (colorwise) consecutive, that is, there does not exist a non-selected page of an acceptable color between two selected ones (notice that there may exist a non-selected page between two selected pages if its color is not acceptable). We say that the pset is *free* if in addition: (iii) each page p_i in the set is free. We say that the pset is *in* (or *inside*) a pool of pages if: (iv) each page p_i belongs to the pool.

In this terminology, function `pp_alloc_clr` searches for *a free pset of a given size n for given acceptable colors $c_1, \dots, c_k$ inside a given pool of pages.* When the conditions are clear from the context, we may drop them and just say "a (free) pset".

For example, in the pool of 8 pages shown in Fig. 2, the set $\{p_2, p_4, p_6\}$ forms a free pset of size 3 for the yellow color; the set $\{p_1, p_3\}$ does not form a free pset of size 2 for the blue color (since p_3 is not free); while $\{p_1, p_5\}$ is not a pset of size 2 for the blue color, as (ii) fails (page p_3 is in-between).

Bao developers chose to search only for consecutive pages because it simplifies the process for other functions to access the newly allocated pages: from the starting page of a pset of size `n`, function `pp_next_clr` is iteratively called `n` times to obtain the first page of an acceptable color (that should return the starting page itself), then the second page of an acceptable color, and so forth (as it will be shown on lines 62–65 in Fig. 3).

Functions `bitmap_get` and `bitmap_set` are used, resp., to read and to write the allocation status of a page (allocated if nonzero or free if zero) from a bitmap, in which each bit represents the status of a page.

4.2 Basic Type Definitions and Constants

Lines 2–22 of Fig. 3 define basic types and constants used in the code. `P_SIZE` denotes the size of a memory page (in bytes). `COLOR_NUM` and `COLOR_SIZE` were presented above. `CELL_SIZE` defines the number of bits in an array cell of type `u32`, which will be used for a compact storage of bits in a bitmap (defined as an array of type `u32*`).

The `page_pool` structure (lines 5–11) represents a *pool of pages*, that is, a contiguous memory area, starting at the page address `base` and containing `size` pages. Each page is marked as free or allocated using the corresponding bit in `bitmap`. For heuristic purposes, the field `last` records the page that follows the last page of the last allocated pset. Field `node` is used (in higher-level functions) to link several pools into a linked list. Some other fields unrelated to the scope of this work were removed in this paper for simplicity (but this simplification does not impact the proof results).

A set of colors is encoded as a 64-bit unsigned integer, called a *vector* of colors, in which the i-th bit is set if the i-th color is authorized. The `ppages` structure (lines 12–16) is used to store a pset, described by the first page's address `base`, the number of pages `num_pages` and the vector of acceptable colors `colors`.

4.3 Implementation of `pp_next_clr`

Function `pp_next_clr` (see Fig. 3, lines 23–29) looks for a first suitably-colored page starting from a given page. This function takes as arguments the address of a base page `base`, an offset `from` (in terms of page numbers with respect to the base page) of the starting page of the search, and a color vector `colors` indicating the acceptable colors. It returns the offset (again, in terms of page numbers with respect to the base page) of the first page whose color is one of the acceptable colors specified in the color vector. Notice that while the base page `base` is given by its address, the starting page and the returned page are identified by their page number offsets (with respect to the number of the base page) and not their address offsets. The page number of the base page with address `base` is

```c
#include <limits.h>
typedef unsigned char u8;
typedef unsigned int  u32;
typedef unsigned long u64;
typedef struct page_pool {
  struct page_pool *node;
  u64 base;
  u64 size;
  u64 last;
  u32 *bitmap;
} page_pool;
```

```c
typedef struct {
  u64 base;
  u64 num_pages;
  u64 colors;
} ppages;
#define P_SIZE (0x1000)
#define CELL_SIZE (sizeof(u32) * 8)
u64 COLOR_NUM;
u64 COLOR_SIZE;
#define P_NB(addr) ((addr)/P_SIZE)
#define P_NB_MAX (1UL << 52)
```

```c
u64 pp_next_clr(u64 base, u64 from, u64 colors){
  u64 clr_offset = (base / P_SIZE) % (COLOR_NUM * COLOR_SIZE);
  u64 index = from;
  while (!((colors >> ((clr_offset + index) / COLOR_SIZE % COLOR_NUM)) & 1))
    index++;
  return index;
}

u32 bitmap_get(u32 *map, u64 bit){
  return (map[bit / CELL_SIZE] & (1U << (bit % CELL_SIZE))) ? 1U : 0U;
}

void bitmap_set(u32 *map, u64 bit){
  map[bit / CELL_SIZE] |= 1U << (bit % CELL_SIZE);
}

u8 pp_alloc_clr(page_pool *pool, u64 n, u64 colors, ppages *ppages){
  u64 allocated = 0;
  u64 first_index = 0;
  u8  ok = 0;
  ppages->colors = colors;
  ppages->num_pages = 0;
  u64 index = pp_next_clr(pool->base, pool->last, colors);
  u64 top = pool->size;
  for (u64 i = 0; i < 2 ∧ !ok; i++){
    while ((allocated < n) ∧ (index < top)){
      allocated = 0;
      while ((index < top) ∧ bitmap_get(pool->bitmap, index))
        index = pp_next_clr(pool->base, ++index, colors);
      first_index = index;
      while ((index<top)∧(bitmap_get(pool->bitmap,index)==0)∧(allocated<n)){
        allocated++;
        index = pp_next_clr(pool->base, ++index, colors);
      }
      index++;                 // FIX: remove this line
    }
    if (allocated == n){
      ppages->num_pages = n;
      ppages->base = pool->base + (first_index * P_SIZE);
      for (u64 j = 0; j < n; j++){
        first_index = pp_next_clr(pool->base, first_index, colors);
        bitmap_set(pool->bitmap, first_index++);
      }
      pool->last = first_index;
      ok = 1;
      break;
    }
    else {
      index = 0;               // FIX: replace this line by the next one
      // index = pp_next_clr(pool->base, 0, colors);
    }
  }
  return ok;
}
```

Fig. 3. Simplified code of the cache coloring mechanism in Bao.

`base / P_SIZE`, while the starting page defined by the number offset `from` has page number `base / P_SIZE + from` and address `base + from * P_SIZE`.

Calculation of the color of a page. In Bao, the cache coloring mechanism defines the color of a page with page number `PNum` through the formula:

`PNum / COLOR_SIZE % COLOR_NUM`. (A)

The function keeps track of the offset of the current candidate page in the variable `index`, initialized to `from`, see line 25. The page number of the current page is `base / P_SIZE + index` and its color is naturally calculated as:

`(base / P_SIZE + index) / COLOR_SIZE % COLOR_NUM`. (B)

However, since the function frequently calculates this formula, it performs an optimization to calculate the color as[3]:

`(clr_offset + index) / COLOR_SIZE % COLOR_NUM`, (C)

where `clr_offset` is the (page number) offset of the base page with respect to the beginning of its periodic block of colored pages, defined on line 24.

Going through the pages. To find the next suitably-colored page, the function iterates over the pages (through the loop on lines 26–27), starting from the index `from` (line 25), each time checking if the color is acceptable using a bit shift of the color vector. The color `c` is acceptable if and only if `!((colors >>c) & 1)` (line 26). Once a page with an acceptable color is found, the loop condition fails, and the function returns the offset of the found page on line 28.

Callers always check that the color vector `colors` contains (hence, accepts) at least one existing color, which ensures the termination of the loop as it will eventually find a page of an acceptable color. Notice that the function does not guarantee that the returned page belongs to the valid range of page indices; this verification is supposed to be done in the upper-level functions.

4.4 Implementation of `bitmap_get` and `bitmap_set`

Function `bitmap_get` (see Fig. 3, lines 31–33) checks the allocation status of pages encoded by a bitmap. It takes two arguments: a bitmap `map` and a bit number `bit`, and returns 1 if the `bit`-th bit is set in `map`, and 0 otherwise. The `bit`-th bit is contained in the cell of index `bit / CELL_SIZE`, at offset `bit % CELL_SIZE`. This explains the calculation on line 32. Similarly, function `bitmap_set` (line 35–37) updates the allocation status of a page.

4.5 Implementation of `pp_alloc_clr`

Function `pp_alloc_clr` (see Fig. 3, lines 39–76) searches for a given number of free consecutive pages of acceptable colors in a given page pool, that is, a free pset. The function takes four arguments: a pointer to a pool structure `pool`, the number of pages `n`, a vector `colors` of acceptable colors, and a pointer to a physical page structure `ppages` to store the result of the search. In case of success, it returns (inside the structure) the number of pages `n` and the address of the first page; otherwise, it sets the number of pages to 0.

[3] We show below (in lemma `arith_1` in Sect. 6.4) that (B) and (C) are equal.

The variable `index` contains the number offset of the current candidate page (with respect to the base page of the pool). The search proceeds in two phases performed by two iterations of the loop on lines 47–74. During the first phase (`i==0`), it starts by searching the first page of an acceptable color from the page with number offset `last` (cf. line 45). Recall that `last` stores the page that follows the last page of the last pset found by the function. The intuition behind this heuristic is that starting from the `last` page is on average more efficient than starting always from the beginning of the pool, because after several allocations the pages in the beginning of the pool will be more likely to be already allocated. To be exhaustive, the second phase (`i==1`) starts from the beginning (line 71).

In each phase, the loop on lines 48–58 scans for free psets. It stops either when it finds a free pset of size `n` of acceptable colors or when the current candidate page runs outside the pool (cf. lines 46, 48).

To find such a pset, the function first searches for the first free page of an acceptable color, as shown in the loop on lines 50–51. If the candidate page has already been allocated (line 50), the function moves to the next candidate page of an acceptable color (line 51). The loop continues until it finds a free page of an acceptable color or the current candidate page runs outside the pool.

If the first page is within the pool and free, the loop on lines 53–56 verifies that the next `n-1` consecutive pages of acceptable colors are also within the pool and free. As long as `n` suitable pages are not yet found, the loop condition checks that the previously found page is free and belongs to the pool (line 53), and the loop body identifies the following page of an acceptable color (line 55). The number of already found pages is maintained in the counter `allocated` (cf. lines 40, 49, 54).

The iteration of the loop on lines 48–58 stops when it has found `n` pages (that is, a free pset is found) or when the candidate page is outside the pool or allocated. If the candidate page is allocated, the search for a new pset restarts just after the last candidate page, as shown on line 55. (The fixes for lines 57 and 71 are discussed in Sect. 5.)

If the function successfully finds `n` pages (line 59), it marks these pages as allocated (loop on lines 62–65), updates the number of allocated pages to `n` (line 60), sets the address of the first page in `ppages` (line 61), updates the address of the last allocated page of the pool (line 66), and, finally, returns.

If the function does not find `n` pages, it returns with `ppages` containing zero allocated pages, as set initially on line 44. The function always terminates and examines all pages in the pool (at least in the second phase).

5 Bugs, Corrections and Further Optimizations

Bugs and fixes. The current version of `pp_alloc_clr`, contrary to its intended behavior, does not guarantee that the returned set is indeed a free pset of `n` pages in the pool. In some intricate cases, depending on the status of the pages, the `n`-th page might be already in use or outside the pool. The first case may break memory isolation, while the second case may cause the VM to crash.

The bugs reside in the selection of the first page of a candidate pset: the function may choose a first page whose color is not acceptable. Indeed, the loop calculating the first page (lines 50–51) only checks that the page is free, without checking its color. This is sufficient for the very first execution or if the loop has already been executed at least once, as a call to `pp_next_clr` (resp., on line 45 or 51)—to select the new candidate page—guarantees it has an acceptable color.

However, if the function fails to find a pset of size `n`, it wrongly starts a new search from the page following the last candidate page (see line 57), whose color may be unacceptable. Additionally, during the second phase, the function starts searching from page index `0` (line 71), which might also have an unacceptable color. In these two faulty cases, if the candidate page is free, it will be selected as the first page of a tentative pset (lines 50–52).

To fix these bugs, we should ensure that the first page has an acceptable color before entering the loop. We propose two bug fixes: we remove line 57 and modify line 71 to `index=pp_next_clr(pool->base, 0, colors);`. These bugs were discovered during the formal specification step, and the fixed version was formally proved with Wp.

Counterexample. To illustrate the first bug, consider the mock pool of Fig. 2 on which `pp_alloc_clr` is called to find a free pset of size 2 for the blue color with `pool->last==p0`. The function will succeed and wrongly return (the address of page) `p4` in `ppages->base` as the first page of a pset. Recall (cf. Sect. 4.1) that in higher-level functions the pages are assigned to a VM via consecutive calls to `pp_next_clr` starting from the first page (like on lines 62–65). The VM will receive pages `p5` and `p7`, the latter being potentially already allocated to another VM! This counterexample (along with another one, due to the second bug) was formally confirmed in Frama-C with the static value analysis plugin Eva [33]. Eva was used to confirm the undesired situation (described with a few ACSL annotations that were proved by Eva) to avoid any risk of misinterpretation of the code. The counterexamples can be found in the companion artifact [27].

Suggestions of optimizations. It would be sufficient for the second phase in the outer loop on lines 47–74 (cf. Sect. 4.5) to perform the search of the first pset page *until* `pool->last`, instead of uselessly performing a full search until the end of the pool (and re-exploring the pages tried in the first phase). This can be done, for instance, by adding **if** `(i==1` $\wedge$ `index` $\geq$ `pool->last)` **return** `ok;` as a second instruction in the body of the loop on lines 50–51. Another optimization can be to perform direct jumps to the first page of the next color without enumerating all pages (as it is done on lines 26–27 in function `pp_next_clr`, *very frequently called*). This can be realized e.g. with a *precomputed array of jumps,* based on the number offset of the current page inside its periodic block of colors. We plan to submit these and some other suggestions to Bao developers before integrating them into the code under verification.

```
predicate ValidCacheCfg = 0 < COLOR_NUM ≤ 64 ∧ 0 < COLOR_SIZE < P_NB_MAX;
predicate IsValidPool(page_pool* pool) =
  \valid(pool) ∧ 0 ≤ P_NB(pool->base) < P_NB_MAX ∧
  0 ≤ pool->size < P_NB_MAX ∧ 0 ≤ pool->last ≤ pool->size ∧
  0 ≤ P_NB(pool->base) + pool->size ≤ P_NB_MAX ∧
  \valid(pool->bitmap + (0..pool->size/CELL_SIZE)) ∧
  \separated(pool,&(pool->bitmap[0..pool->size/CELL_SIZE]));
predicate flatPoolStatus(page_pool* pool) =
  \valid_read(pool) ∧ \valid_read(pool->bitmap + (0..pool->size/CELL_SIZE)) ∧
  \valid_read(&gPStatus[P_NB(pool->base)..(P_NB(pool->base)+pool->size-1)]) ∧
  ∀ ℤ idx; 0 ≤ idx < pool->size ⇒
    (((pool->bitmap[idx/CELL_SIZE] >> (idx%CELL_SIZE)) & 1)⟺
      gPStatus[P_NB(pool->base) + idx]);
predicate flatClrs(u64 colors) = ∀ ℤ clr; 0 ≤ clr < 64 ⇒
  (((colors >> clr) & 1) ⟺ gFlatClrs[clr]);
predicate IsInClrs(ℤ clr) = gFlatClrs[clr] ≠ 0;
predicate IsNotInClrs(ℤ clr) = gFlatClrs[clr] == 0;
predicate HasClrPages{L1,L2}(u64* PArr, ℤ p_base, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒  IsInClrs{L1}(P_CLR{L1}(p_base + \at(PArr[i],L2)));
predicate NoClrPBtw(ℤ p_base, ℤ start, ℤ end) =
  ∀ ℤ index; start ≤ index < end ⇒ IsNotInClrs(P_CLR(p_base + index));
predicate HasSeqPages{L1,L2}(u64* PArr, ℤ p_base, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i;  1 ≤ i < n ⇒ \at(PArr[i-1],L2) < \at(PArr[i],L2) ∧
    NoClrPBtw{L1}(p_base,\at(PArr[i-1],L2)+1,\at(PArr[i],L2));
predicate HasPagesInPool{L1,L2}(u64* PArr, page_pool* pool, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧ \at(\valid_read(pool),L1) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ 0 ≤ \at(PArr[i],L2) < \at(pool->size,L1);
predicate PSetInPool{L1,L2}(u64* PArr, page_pool* pool, u64 n, u64 colors) =
  \at(\valid_read(pool),L1) ∧ flatClrs{L1}(colors) ∧
  HasPagesInPool{L1,L2}(PArr,pool,n) ∧
  HasClrPages{L1,L2}(PArr,P_NB(\at(pool->base,L1)),n) ∧
  HasSeqPages{L1,L2}(PArr,P_NB(\at(pool->base,L1)),n);
predicate HasFreePages{L1,L2}(u64* PArr, page_pool* pool, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ \at(gPStatus[P_NB(pool->base)+\at(PArr[i],L2)],L1) == 0;
predicate HasAllocPages(u64* PArr, page_pool* pool, u64 n) =
  \valid_read(PArr + (0..n-1)) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ gPStatus[P_NB(pool->base) + PArr[i]] ≠ 0;
```

Fig. 4. Predicates used in the specification of the cache coloring mechanism of Bao.

6 Verification of Cache Coloring

This section presents key specification and verification points and the results of the case study. Its full annotated code can be found in the companion artifact [27]. We mainly focus in the paper on the verification of the key functions presented in Fig. 3. The specified and verified code also includes a simplified version of two higher-level functions (`pp_alloc_ppages` and `mem_map`), which were verified to get confidence in consistency of the proposed contracts for the key functions with the expected behavior in the callers. For an easier navigation, unless otherwise stated, the line numbers in the figures and text below are kept as in the full annotated code.

6.1 Basic Predicates and Flattening Invariants

In this case study (cf. Fig. 3, lines 17, 22), we consider a 64-bit implementation with 2^{12}-byte pages and a maximum number of pages of 2^{52}, which aligns with the maximum number of pages supported by most 64-bit architectures. Additionally, we consider 64-bit long color vectors, which sets the maximal number of colors to 64 accordingly, and we do not impose any prior constraints on `COLOR_SIZE` to ensure compatibility with a wide range of hardware configurations, as specified in the definition of predicate `ValidCacheCfg` (Fig. 4, line 40).

Predicate `IsValidPool` (line 41) ensures that `pool` represents a valid segment of memory, and that its bitmap is sufficiently large to store the status of its pages and does not overlap with the pool structure. Macros `P_NB` and `P_NB_MAX` were defined in Fig. 3, lines 21–22.

Earlier verification efforts with Frama-C (e.g. [23]) demonstrated that reasoning on array cells instead of bits makes solvers *more efficient*. We introduce a global companion ghost array `u8 gPStatus[P_NB_MAX]` to store page allocation statuses, and express the equivalence between a bitmap and the companion array with predicate `flatPoolStatus` (Fig. 4, line 47). It guarantees that checking the i-th bit in the `bitmap` is equivalent to checking the i-th cell in `gPStatus`. A starting letter `g` (e.g. in `gPStatus`) indicates a ghost variable name in this work.

Similarly, we introduce a global companion ghost array `u8 gFlatClrs[64]` to flatten the color vector (unchanged in our scope[4]), and express the equivalence between a color vector and the companion array with predicate `flatClrs` (line 53), i.e. that checking the i-th bit in color vector `clrs` is equivalent to checking the i-th cell in `gFlatClrs`. Maintaining such *flattening invariants* in contracts enables expressing properties on array cells instead of bits.

Predicates `IsInClrs` and `IsNotInClrs` (lines 55, 56) state that color `clr` is, resp., acceptable and unacceptable w.r.t. the color vector encoded in `gFlatClrs`.

Predicate `HasClrPages` (line 57) states that array `PArr` of size `n` contains page number offsets (with respect to the base page `p_base`) of pages with acceptable colors. Labels `L1` and `L2` characterize, resp., the moment of calculation of the color and of reading the cell in `PArr`. Such a distinction of labels will often be used in predicates below. Logic function `P_CLR(PNum)` computes the color of a given page as in (A) in Sect. 4.2. Predicate `NoClrPBtw` (line 60) states that there is no page of an acceptable color with number offset between `start` and `end` (excluded) with respect to `p_base`.

Predicate `HasSeqPages` (line 62) ensures that page number offsets stored in array `PArr` of size `n` are in ascending order, and any other page between them does not have an acceptable color. Predicate `HasPagesInPool` (line 66) states that page number offsets stored in array `PArr` of size `n` are within the memory pool pointed to by `pool`.

Predicate `PSetInPool` (line 69) states that the page number offsets stored in array `PArr` of size `n` are within the memory pool pointed to by `pool`, consecutive and of an acceptable color. In other words, array `PArr` is a pset of size `n` for `colors` inside `pool`.

Predicate `HasFreePages` (line 74) states that pages with page number offsets stored in array `PArr` of size `n` are free according to `gPStatus`. Likewise, predicate `HasAllocPages` (line 77) states that those pages are allocated.

[4] this is not a limitation for larger scopes: such arrays can be ghost function arguments.

```
144  requires ValidCacheCfg;
145  requires 0 ≤ from < P_NB_MAX;
146  requires flatClrs(colors);
147  requires \valid_read(gFlatClrs + (0..63));
148  requires 0 ≤ gClrValid < COLOR_NUM ∧ IsInClrs(gClrValid);
149  terminates \true;
150  assigns \nothing;
151  ensures flatClrs(colors);
152  ensures clr: IsInClrs(P_CLR(P_NB(base) + \result));
153  ensures cons: NoClrPBtw(P_NB(base),from,\result);
154  ensures bnd: from ≤ \result < from + COLOR_NUM*COLOR_SIZE;
```

Fig. 5. Contract of `pp_next_clr`.

In order to verify the code with the deductive verification plugin Wp of Frama-C, we provide an ACSL specification for each of the considered functions. We overview here the contracts of `pp_next_clr` and `pp_alloc_clr`.

6.2 Specification of `pp_next_clr`

Preconditions. Given the scope of our verification, we bound the range of the page number offset `from` between 0 and 2^{52} (excluded), assuming there can be a single memory pool supporting up to 2^{52} pages (Fig. 5, line 145). We did not need to impose specific constraints on the address `base` since we are considering 12-bit wide pages and 52-bit wide page numbers, so the address is naturally bounded by its type. Predicate `ValidCacheCfg` (line 144) specifies the considered arithmetic constraints. The equivalence between the color vector and the companion array must hold before and after the call (lines 146, 151). Finally, recall that callers ensure that the color vector accepts at least one color (cf. Sect. 4.3). To guarantee termination, we express this constraint on line 148, where the existential property is replaced by a witness—a global ghost variable `gClrValid`—since this value will be used in ghost code inside the function. We preferred this (simple and sufficient) option for our scope to the alternative when a witness has to be found inside the function from an existential precondition.

Postconditions. We express that the function always terminates (line 149) and does not modify the memory (line 150). Finally, we express the functional properties. The returned page has an acceptable color (line 152) and is the closest page with this property to `from`, the starting page of the search (line 153); Line 154 gives an interval of values for the result, a maximum offset being the size of a color block (line 154). This upper bound is tight[5] and suffices to prove the absence of overflows during the update of `index`.

6.3 Specification of `pp_alloc_clr`

Preconditions. The preconditions (omitted in the paper) are relatively natural and mostly similar to those of `pp_next_clr`. An interval of values is specified for the number of allocated pages `n`, and the validity of `ppages` is required. The validity of `pool` and flattening invariants are present both in preconditions and postconditions (the latter on lines 259–261 in Fig. 6).

[5] this upper bound is reached for `COLOR_SIZE==1`.

```
258   ensures res: \result == 0 ∨ \result == 1;
259   ensures fltc: flatClrs(colors);
260   ensures vldp: IsValidPool(pool);
261   ensures fltp: flatPoolStatus(pool);
```

```
264   ensures suc: PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
265     HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ⇒ \result == 1;
266   ensures wit1: \result == 1 ⇒
267     PSetInPool{Pre,Post}((u64*)gFoundPSet,pool,n,colors);
268   ensures wit2: \result == 1 ⇒
269     HasFreePages{Pre,Post}((u64*)gFoundPSet,pool,n);
270   ensures fct1: \result == 1 ⇒
271     PSetInPool{Post,Post}((u64*)gFoundPSet,pool,n,colors);
272   ensures fct2: \result == 1 ⇒ HasAllocPages((u64*)gFoundPSet,pool,n);
273   ensures pps: \result == 1 ⇒  ppages->num_pages == n ∧
274     ppages->base == pool->base + (gFoundPSet[0]*P_SIZE);
```

```
279   ensures ppf: \result == 0 ⇒ ppages->num_pages == 0;
280   ensures ups: \result == 0 ⇒ ∀ ℤ i; 0 ≤ i < P_NB_MAX ⇒
281     \at(gPStatus[i],Pre) == \at(gPStatus[i],Post);
282   ensures upp: \result == 0 ⇒ \at(pool,Pre) == \at(pool,Post) ∧
283     \at(*pool,Pre) == \at(*pool,Post);
```

Fig. 6. Selected postconditions of the contract of `pp_alloc_clr`.

Additionally, we had to add a dozen of explicit *separation clauses* (omitted in the paper) between the arguments and the ghost variables. Some of these separation predicates are likely to become unnecessary in a future version of Frama-C/Wp that will be capable to deduce that the modification of ghost variables cannot impact non-ghost variables, and vice versa.

Postconditions. At the end of the function, there are two possible return values, 0 and 1 (line 258). Other notable postconditions fall into two categories: those for the success case and those for the failure case.

Predicates that hold on success (when the function returns `1`) must ensure that subsequent calls in the callers (cf. Sect. 4.1) to `pp_next_clr` starting from the first allocated page—the only page returned in `ppages`—will really return pages of a required pset in the pool (whose pages were *free before the call* and then *marked as allocated* by the function). Since ACSL does not allow using the C function `pp_next_clr` in the specification, we used predicates over the selected pages. We capture these pages by their number offsets (with respect to the starting page of the pool) in a global ghost array `u64 gFoundPSet[P_NB_MAX]`, by adding ghost code into the function. It is another illustration of an *advantageous usage of ghost code artifacts* for the specificaiton.

Precisely, we state that `n` pages in `gFoundPSet` were free before the call (lines 268–269) and are now allocated (line 272); and constitute a pset of size `n` for `colors` (line 270–271). Moreover the `ppages` structure must contain `n` pages and store the page address corresponding to the first cell of the `gFoundPSet` (line 273–274).

In case of a failure, the function returns `0` (line 279), it has not modified the page status array (lines 280–281) nor the pool (line 282–283).

Specification completeness. The completeness and disjointness of the two cases of the specification was non-trivial to ensure because of the complexity of the calling cases: either there exists a suitable subset of pages—free pset—on entry, or not. As the size of the subset depends on an argument of the function, the conditions involve an undetermined number of pages. Expressing such properties with an undetermined number of quantifiers is not directly allowed in ACSL and would only be possible indirectly (e.g. with a list, an array, or a set). However, since solvers often have issues with complex conditions involving multiple quantifiers, we decided to adopt another, more pragmatic approach.

We decided to represent the existence of a suitable subset of pages through the existence of a witness array containing the number offsets of the pages. Thus, we stated the existence case assumption by giving a witness pset in a global companion ghost array `u64 gExistPSet[P_NB_MAX]`, see lines 264–265. This establishes that the function returns `1` in this case. But this unique implication is not sufficient: the function could still return `0` while a suitable free pset existed *on entry*.

That is why we added another clause (lines 266–267) stating that the companion ghost array `gFoundPSet` mentioned above—*with its values on exit*—was a suitable pset (of size `n` with acceptable `colors` inside `pool`) *already on entry*. Along with lines 268–269, we deduce a condition similar to that on the left of the implication on lines 264–265.

Recall that the first label indicates when the property is evaluated while the second label indicates at which state the array values are read. Notice that, while the aforementioned clause on lines 270–271 looks similar, strictly speaking, it does not directly state the same property as on lines 266–267, since it considers the property at label **Post** instead of **Pre** (the values of `gFoundPSet` being, of course, considered at **Post** in both cases as it is computed during the function).

Therefore, we can deduce that the function returns `1` *if and only if* a suitable subset of pages existed *on entry, before the call*. As the function can only return `0` or `1` (line 258), our specification of both cases is complete and disjoint.

We did not use ACSL behaviors because Frama-C would not be capable to prove that behaviors are complete and disjoint for this version of specification. That is why we justify it here with an additional argument, external to Frama-C.

6.4 Selected Aspects and Difficulties of the Proof

The proof required *carefully chosen* predicates, ghost code and ghost variables, loop invariants, assertions and lemmas. The predicates, ghost variables and our approach to ensure the *completeness of the specification* of `pp_alloc_clr` were presented above. A companion ghost model and flattening invariants helped to efficiently deal with *bit-level operations*. This section presents some other selected aspects and verification choices.

Termination of `pp_next_clr`. To prove termination, we compute (in ghost code) an upper bound for the number offset `index` using the witness color `gClrValid`. We distinguish two cases of relative position of the starting page in its periodic

```
337     loop invariant I3_ex:
338       PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
339       HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
340       i == 1 ∧ allocated < n ⇒  index ≤ gExistPSet[0];
```

```
371       loop invariant I1_ex:
372         PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
373         HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
374         i == 1 ∧  allocated < n ∧  gExistPSet[0] ≤ index ⇒
375         (∃ ℤ i; 0 ≤ i ≤ allocated ∧ gExistPSet[i] == index);
```

Fig. 7. Loop invariants for the loops on lines 48–58 (above) and lines 53–56 (below) of `pp_alloc_clr` (in Fig. 3), used to prove the success when a pset exists on entry.

```
lemma arith_1: ∀ ℤ a,b,c,d; 0 ≤ a ∧ 0 ≤ b ∧ 0 < c ∧ 0 < d ⇒
  ((a+b)/c)%d == ((a+b%(c*d))/c)%d;
```

Fig. 8. One of the four arithmetic lemmas used in the proof of `pp_next_clr`.

color block: either its color lies before the existing acceptable color `gClrValid` or after it in (in the latter case, the upper bound is in the next color block).

Proof of `pp_alloc_clr`. With three levels of nested loops, a significant number of carefully chosen loop invariants was necessary. For instance, to prove that the function finds a pset in case there exists a suitable pset in the pool (lines 264–265), we have to ensure that if such a pset exists in the pool, then it is located in the part of the pool the function has not explored so far. Thus, if the function fails to find such a pset after going through the entire pool, then the existing pset must be located outside the memory, which is contradictory. Due to the structure of the function, we express this property in the main loop and then recursively in the nested loops to have it preserved. Figure 7 shows the invariants for two of them. We constrain only the second phase since it runs—in the current version—a full search from the beginning of the pool, that explains the condition `i==1`. The second invariant is relatively tricky. Indeed, the loop on lines 53–56 in Fig. 3—that attempts to complete a previously identified first page to a full pset—may possibly find the witness pset `pExistPSet` or another existing pset starting before it. To address this, we adjust the loop invariant by stating that *we did not miss the witness pset* `pExistPSet`: if the current candidate page is greater or equal to the first page of `pExistPSet`, then it lies inside it (line 375).

Arithmetic lemmas. As page colors are computed with modulo and division operations, reasoning about them involves such arithmetic operations. The solvers we used were unable to handle them directly. To address this issue, we introduced four arithmetic lemmas, and had to prove three of them in Coq (see the companion artifact [27] for Coq proof scripts). One lemma, proving the equivalence of (B) and (C) (see Sect. 4.3), is shown in Fig. 8.

Separation issues and Frama-C's memory model. The need for additional separation clauses (in particular, between ghost and non-ghost variables) was already mentioned in Sect. 6.3. In many parts of the code, we also encountered difficulties in proving the preservation of seemingly trivial properties through assignments. These difficulties stem from the memory model used in Frama-C/Wp, where pointers are treated as indices within arrays, where cells correspond to the pointed values. Consequently, properties involving pointers in ACSL are translated into properties over arrays in Wp. When a pointed value is modified, the whole array is seen as possibly modified, making proofs non-trivial for solvers. To prove such properties, we often had to manually create proof scripts in Wp to demonstrate that the pointed values used in predicates remain unchanged through assignments. This process introduced a significant specification and verification overhead making the verification process more complex to maintain.

Semantic lemmas. To show the preservation of the `PSetInPool` predicate between two program points despite the modification of some variables, a preservation lemma was necessary (lemma `PSetInPool_preserved` in the companion artifact [27]). While the idea is well-known, a very careful formulation with four labels was necessary since each predicate has two labels. Moreover, its proof required a manually crafted proof script in Wp with carefully selected tactics.

Function `pp_next_clr` must ensure that the pages of the found free pset eventually become allocated. This task is handled in the loop on lines 62–65 in Fig. 3, which iterates through the found page indices and marks them as allocated. However, the function moves to the next page of the pset by calculating it through a call to `pp_next_clr`. To ensure that the function gets the same page indices as that of the free pset found earlier, another interesting lemma (lemma `unique_next_clr_page` in the companion artifact [27]) was necessary.

Linked lists. During the verification of higher-level function `pp_alloc_ppages`, which looks for a free pset of a given size in a set of pools *represented by a linked list of pools* (as mentioned in Sect. 4.2), an additional difficulty was related to linked lists. Indeed, contrary to simple linked lists, in our case list nodes contain several data fields and pointers to external arrays. Broadly inspired by previous work [15,36,16], this issue was solved using a companion ghost array containing the addresses of the nodes of the linked list and by defining and maintaining a suitable linking predicate, which establishes the link between them. Detailed specifications are available in the companion artifact [27].

Unstable proof scripts in Wp. During the last stages of the case study, we discovered an issue in Frama-C/Wp related to proof scripts. A created script, which leads to a successful proof at the time of its creation, fails with error messages during the proof replay. Presumably, this comes from a different degree of proof goal simplifications during the script creation and the proof replay, resulting in slight differences in the proof goal. This issue has been reported to the Wp team.

6.5 Proof Statistics

This verification case study took approximately three months of intensive work, including understanding the implementation, formal specification, verification, detecting and fixing the bugs, readability improvements and restructuring of the specification for the paper. Formal verification was, initially, carried out on the four key functions: `pp_next_clr`, `bitmap_get`, `bitmap_set` and `pp_alloc_clr`. To ensure the relevance of the proposed contracts, formal specification and verification for simplified versions of two upper-level functions, `mem_alloc_ppages` and `mem_map`, were realized as well, focusing on the mapping of colored pages (and excluding other behavior e.g. when cache coloring is deactivated). The former one searches for a pset in a linked list of pools, while the latter calls the former to assign a pset of pages to a VM. They are not detailed in the paper, but the annotated code is available in the companion artifact [27]. *The claim that formal verification is complete can be demonstrated with the artifact.*

The specified functions total, approximately, 100 lines of C code and 600 lines of ACSL. ACSL annotations include ghost code (20 lines), predicates (100 lines), contracts (455 lines), assertions (30 lines), and lemmas (25 lines).

The proof goals include function contracts, assertions, lemmas, the absence of run-time errors, smoke tests (to detect potential specification inconsistencies), and memory hypotheses made by Wp's typed memory model; they result in 463 proof goals and 60 extra goals for smoke tests; that is, 523 in total.

The proof was carried out with Frama-C v.29.0 and Why3 1.7.2, with the external solvers Alt-Ergo 2.5.4, CVC5 1.0.9 and Z3 4.8.12 (run in that order), and the proof assistant Coq 8.18.0. The proofs were run on a desktop computer running Ubuntu 22.04.5 LTS, with an Intel® Core™ i5-1145G7 CPU @ 2.60 GHz, featuring 4 cores 8 threads, with 32 GB RAM. We ran Frama-C/Wp with options `-wp-par=8` and `-wp-timeout=40`.

The full proof takes approx. 5 minutes. All smoke tests passed. Over the 523 goals[6], around 1% (6) were discharged by control-flow analysis; around 83% of the goals (433) were proved by automatic solvers: the internal simplifier engine Qed of Wp handled around 53% of the goals (277) in an average time of 146ms per goal, then Alt-Ergo discharged around 28% of the goals (147) in an average time of 110ms, CVC5 covered around 5% of the goals (26) in an average time of 725ms, Z3 proved 2% of the goals (9) in an average time of 1.4s. Around 11% of the goals (55) were achieved through proof scripts in Wp, while less than 1% of the goals (3) were proved in Coq. At the end of the case study, when the authors were used to proof contexts, the scripts in Wp required around 5 hours to be fully re-created manually, which was often necessary after code and specification updates. The proof scripts in Coq required a couple of hours to be created manually (and did not need to be re-created after the first attempt).

7 Conclusion and Future Work

Related work. A number of hypervisors are in use today. Some are used in IT infrastructures (e.g cloud) for their flexibility and dynamic resource management such as Xen [6], VMWare [4] or KVM [3]. Others are better suited to critical

[6] The per-solver results are given as an indication of a possible proof run, can vary and should not be used to compare solvers or draw any conclusions about their relative efficiency; our purpose was to reach a full proof and not to compare the solvers.

embedded systems such as Xen Dom0-less [5], Jailhouse [2] or Xtratum [7]. In this case, it is the *static resource sharing property* that is exploited. The use of hypervisors in critical embedded systems requires a high level of confidence in resource allocation, and particularly in maintaining *isolation between VMs.* Formal verification has been applied to provide high confidence in some resource allocation systems, such as ProvenCore [17] ans seL4 [32]. To the best of our knowledge, formal verification of cache coloring has never been addressed in previous work.

More generally, this work is related to other verification case studies on real-life code and empirical evaluations of verification tools [29]. Among other examples, the KeY tool was used for verification of several libraries and applications in Java [22,12]. Verification of a traffic tunnel control system [38] was realized with VerCors [8]. Verification for a real-world avionics example [25] and for security properties [23] provided useful feedback on using Frama-C. SPARK was used in the verification of a TCP Stack [19] and complex datastructures [26]. Verification of the Hyper-V hypervisor with VCC [34] highlighted some issues specific to hypervisor verification. Deductive verification of smart contracts [18] was realized with Dafny [35]. Several case studies [39] were performed using VeriFast [30]. Each new case study contributes to enhance verification tools by identifying their limitations and to push further the frontiers of what is achievable for formal verification.

Conclusion. This paper has presented a formal verification case study for an original, industrially relevant and security-critical target—the cache coloring in Bao. We have given its pedagogical presentation and emphasized main aspects of its verification with Frama-C. The target code is very elegant but challenging for deductive verification (containing bit-level operations, non-trivial logic, complex arithmetic operations, multiple nested loops, linked lists). This case study contributes to a better understanding of the capacities of modern deductive verifiers. It also allowed us to identify and fix two bugs in the target code, to suggest its further optimizations, and to discover a minor issue in the verification tool.

Future work. Future work includes the verification of optimized versions of cache coloring and a larger verification of critical parts of the Bao hypervisor, with a long-term goal to reach a highly optimized, provably correct static hypervisor ensuring strong isolation properties and suitable for modern embedded systems. Another work direction is to enhance automatic proof script generation [21].

Data availability statement. The companion artifact [27] contains the annotated code, counterexamples and a virtual machine (with all necessary tools installed), ready to reproduce the proof.

Acknowledgment. Part of this work was supported by ANR (grants ANR-22-CE39-0014, ANR-22-CE25-0018). We warmly thank Téo Bernier for his valuable advice and help, Allan Blanchard, Loïc Correnson and Frédéric Loulergue for fruitful discussions, the whole Frama-C team for their support, and the anonymous referees for helpful comments.

References

1. Bao project, https://github.com/bao-project/bao-hypervisor
2. Jailhouse hypervisor, https://github.com/siemens/jailhouse
3. KVM hypervisor, https://linux-kvm.org/page/Main_Page
4. VmWare hypervisor, https://www.vmware.com
5. Xen Dom0-less hypervisor, https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/
6. Xen project, https://xenproject.org
7. XtratuM hypervisor, https://www.fentiss.com/xtratum/
8. Armborst, L., Bos, P., van den Haak, L.B., Huisman, M., Rubbens, R., Sakar, Ö., Tasche, P.: The VerCors verifier: A progress report. In: Proc. of the 36th International Conference on Computer Aided Verification (CAV 2024). LNCS, vol. 14682, pp. 3–18. Springer (2024). https://doi.org/10.1007/978-3-031-65630-9_1
9. Barbosa, H., Barrett, C.W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Proc. of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2022). LNCS, vol. 13243, pp. 415–442. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_24
10. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
11. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html
12. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally verifying an efficient sorter. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024). LNCS, vol. 14570, pp. 268–287. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_15
13. Bertot, Y., Castéran, P. (eds.): Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
14. Bjørner, N.S.: Z3 and SMT in industrial R&D. In: Proc. of the 22nd International Symposium on Formal Methods (FM 2018). LNCS, vol. 10951, pp. 675–678. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_44
15. Blanchard, A., Kosmatov, N., Loulergue, F.: Ghosts for lists: A critical module of Contiki verified in Frama-C. In: Proc. of the 10th NASA Formal Methods Symposium (NFM 2018). LNCS, vol. 10811, pp. 37–53. Springer (2018)
16. Blanchard, A., Kosmatov, N., Loulergue, F.: Logic against ghosts: Comparison of two proof approaches for a list module. In: Proc. of the 34th Annual ACM/SIGAPP Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT 2019). pp. 2186–2195. ACM (2019). https://doi.org/10.1145/3297280.3297495
17. Bolignano, P.: Formal models and verification of memory management in a hypervisor. Ph.D. thesis, Université de Rennes ; Prove & Run (May 2017), https://theses.hal.science/tel-01637937

18. Cassez, F., Fuller, J., Quiles, H.M.A.: Deductive verification of smart contracts with Dafny. In: Proc. of the 27th International Conference on Formal Methods for Industrial Critical Systems (FMICS 2022). LNCS, vol. 13487, pp. 50–66. Springer (2022). https://doi.org/10.1007/978-3-031-15008-1_5
19. Cluzel, G., Georgiou, K., Moy, Y., Zeller, C.: Layered formal verification of a TCP stack. In: Proc. of the IEEE Secure Development Conference (SecDev 2021). pp. 86–93. IEEE (2021). https://doi.org/10.1109/SecDev51306.2021.00028
20. Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-Ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018), https://hal.inria.fr/hal-01960203
21. Correnson, L., Blanchard, A., Djoudi, A., Kosmatov, N.: Automate where automation fails: Proof strategies for Frama-C/WP. In: Proc. of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2024), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2024). LNCS, vol. 14570, pp. 331–339. Springer (Apr 2024). https://doi.org/10.1007/978-3-031-57246-3_18
22. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. Formal Aspects Comput. **35**(3), 18:1–18:26 (2023). https://doi.org/10.1145/3594729
23. Djoudi, A., Hána, M., Kosmatov, N.: Formal Verification of a JavaCard Virtual Machine with Frama-C. In: Proc. of the 24th International Symposium on Formal Methods (FM 2021). LNCS, vol. 13047, pp. 427–444. Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_23
24. Djoudi, A., Hána, M., Kosmatov, N., Kříženecký, M., Ohayon, F., Mouy, P., Fontaine, A., Féliot, D.: A bottom-up formal verification approach for common criteria certification: Application to JavaCard virtual machine. In: Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022) (Jun 2022)
25. Dordowsky, F.: An experimental study using ACSL and Frama-C to formulate and verify low-level requirements from a DO-178C compliant avionics project. Electronic Proceedings in Theoretical Computer Science **187**, 28–41 (2015). https://doi.org/10.4204/EPTCS.187.3
26. Dross, C., Moy, Y.: Auto-active proof of red-black trees in SPARK. In: Proc. of the 9th International Symposium on NASA Formal Methods (NFM 2017). LNCS, vol. 10227, pp. 68–83 (2017). https://doi.org/10.1007/978-3-319-57288-8_5
27. Ferréol, A., Corbin, L., Kosmatov, N.: Prove your colorings: Formal verification of cache coloring of Bao hypervisor. Companion artifact for the paper submitted to FASE 2025 (2025). https://doi.org/10.5281/zenodo.14616331
28. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Proc. of the 22nd European Symposium on Programming (ESOP 2013). LNCS, vol. 7792, pp. 125–128. Springer (2013)
29. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Computing and Software Science – State of the Art and Perspectives, LNCS, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
30. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Proc. of the Third International Symposium on NASA Formal Methods (NFM 2011). LNCS, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4

31. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Asp. Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
32. 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. of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009). pp. 207–220. ACM (2009). https://doi.org/10.1145/1629575.1629596
33. Kosmatov, N., Prevosto, V., Signoles, J. (eds.): Guide to Software Verification with Frama-C. Core Components, Usages, and Applications. Computer Science Foundations and Applied Logic Book Series, Springer (2024). https://doi.org/10.1007/978-3-031-55608-1
34. Leinenbach, D., Santen, T.: Verifying the microsoft Hyper-V hypervisor with VCC. In: Proc. of the Second World Congres on Formal Methods (FM 2009). LNCS, vol. 5850, pp. 806–809. Springer (2009). https://doi.org/10.1007/978-3-642-05089-3_51
35. Leino, K.R.M.: Program Proofs. The MIT Press (2023)
36. Loulergue, F., Blanchard, A., Kosmatov, N.: Ghosts for lists: from axiomatic to executable specifications. In: Proc. of the 12th International Conference on Tests and Proofs (TAP 2018). LNCS, vol. 10889, pp. 177–184. Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_11
37. Martins, J., Tavares, A., Solieri, M., Bertogna, M., Pinto, S.: Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In: Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020). Open Access Series in Informatics (OASIcs), vol. 77, pp. 3:1–3:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/OASIcs.NG-RES.2020.3
38. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Proc. of the 15th International Conference on Integrated Formal Methods (IFM 2019). LNCS, vol. 11918, pp. 418–436. Springer (2019). https://doi.org/10.1007/978-3-030-34968-4_23
39. Philippaerts, P., Mühlberg, J., Penninckx, W., Smans, J., Jacobs, B., Piessens, F.: Software verification with VeriFast: Industrial case studies. Sci. Comput. Program. **82**, 77–97 (2014). https://doi.org/10.1016/J.SCICO.2013.01.006
40. The Coq Development Team: The Coq proof assistant. http://coq.inria.fr,

A Appendix: Supplementary Material

The appendix includes two counterexamples—each one highlighting one cause of faulty page allocation, as well as the specified version of the target code (including the corrected version of the key functions presented in Fig. 3 and two simplified higher-level functions), and some additional explanations.

The specified code and counterexamples were proved with the following versions of tools and provers:

- Frama-C v29.0
- Why3 1.7.2
- Alt-Ergo 2.4.5
- CVC5 1.0.9
- Z3 4.8.12
- Coq 8.18.0

The proofs were run on a desktop computer running Ubuntu 22.04.5 LTS, with an Intel® Core™ i5-1145G7 CPU @ 2.60 GHz, featuring 4 cores 8 threads, with 32 GB RAM.

A.1 Counterexample 1, Confirmed with Eva

This section details the faulty execution of `pp_alloc_clr` caused by the buggy instruction `index = 0;` (on line 71 in Fig. 3) and provides a proof of its incorrect result. The proof relies on Frama-C/Eva, a value analysis plugin of Frama-C, which validates the provided ACSL assertions, as shown in Fig. 11. *The claim that the assertions of this example are proved with Eva can be demonstrated with the artifact [27].*

The pool layout is described in Fig. 9, where letters A,...,G refer to various pages or steps. The pool is made of `8` pages $\{p_0, \dots, p_7\}$, the field `last` is equal to `7`, it points to p_7, the highest page of the pool (see A). The memory pages are colored with alternate yellow and blue colors. The first page of the pool (p_0) is yellow. The first 2 pages (p_0, p_1) of the pool are free, the others are allocated. The function `pp_alloc_clr` is called to find a free pset of 2 blue pages in that pool.

During the execution, the function first sets `index` to `7` (corresponding to page p_7) on line 45 (see B), as p_7 is a blue page. Then, in the first phase of the search (`i==0` in the loop on line 47), the function starts searching for the free first page of the pset with `index` equal to `7`. As p_7 is not free, the loop (line 51) increments `index` and calls `pp_netx_clr` on it to update its value, leading to `index` being outside the pool. This causes the first phase to end. Prior to the second phase, `index` is set to `0` (corresponding to p_0) though the *buggy instruction* on line 71 (see C).

In the second phase (`i==1` in the loop on line 47), the function starts searching from `index` set to `0`. As p_0 is free, it satisfies the loop conditions (on line 50),

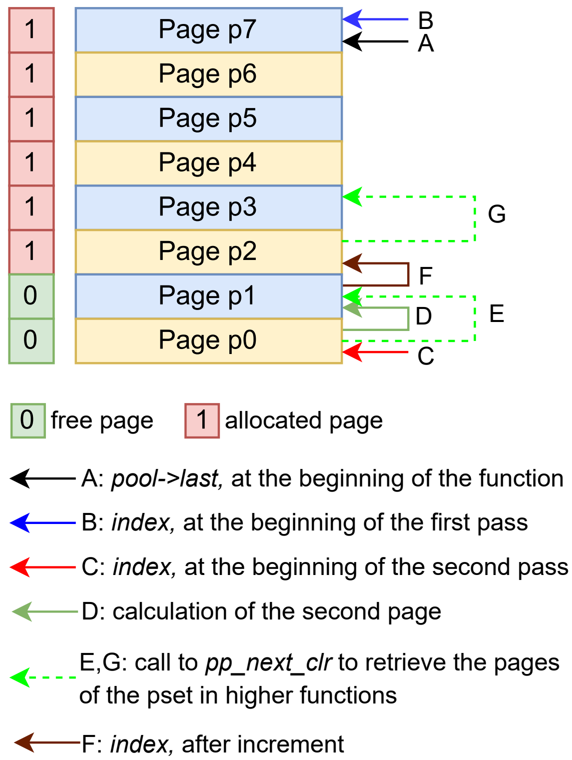

Fig. 9. Example of a pool of memory pages resulting in a wrong allocation of a free pset of **2 blue pages**. In that example, the pool is made of 8 pages $\{p_0, \ldots, p_7\}$, `pool->last` points to p_7, `COLOR_SIZE` is equal to 1 and `COLOR_NUM` is equal to 2.

so `index` is not updated and, p_0 is considered as the first page of the pset. Additionally, `index` also satisfies the loop conditions (on line 53). Consequently, `allocated` is incremented to `1` and `index` is incremented to `1` and then updated by a call to `pp_next_clr`; which sets `index` to `1` (see D), as the corresponding page p_1 is blue. Additionally, p_1 is free so it satisfies the loop conditions. In consequence, `allocated` is incremented to `2`, successfully completing the allocation.

The function, then, exits the loop on line 48, setting `ppages->base` to `pool->base + p0*P_SIZE`.

However, when upper-level functions attempt to retrieve the pages of the pset, they will, first, call `pp_next_clr` on page p_0, returning page p_1 (see E) and then on page p_2—after increment (see F)—returning page p_3 (see G). **At this point, page p_3 may have already been allocated, possibly to the same VM or another VM accepting the blue color!**

The concrete counterexample in Fig. 11 reproduces this scenario exactly. The blue color is encoded by color number `1`, and, the pool's `base` is set to `0` for simplicity. Accordingly, for all integer `i` within the pool's range, page p_i corresponds to page number `i`. We use `Eva` with option `-eva-slevel=4`, which

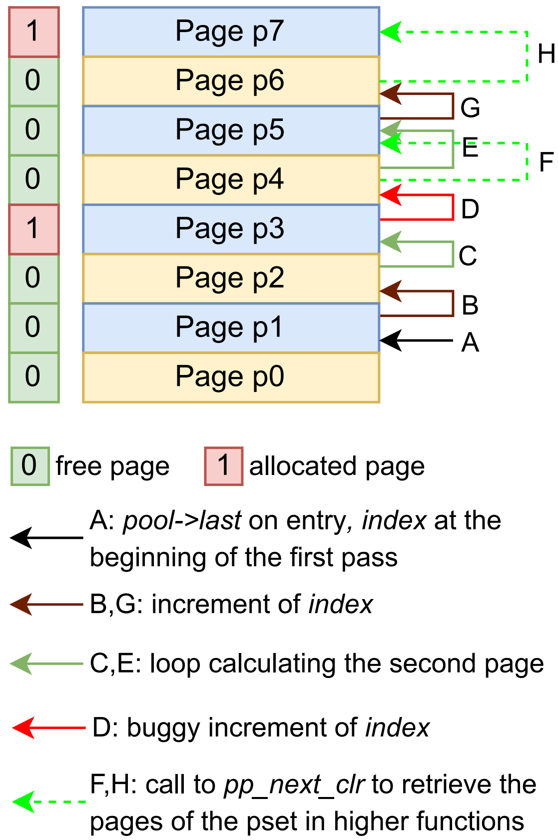

Fig. 10. Example of a pool of memory pages resulting in a wrong allocation of a free pset of **2 blue pages**. In that example, the pool is made of 8 pages $\{p_0, \ldots, p_7\}$, `pool->last` points to p_1, `COLOR_SIZE` is equal to `1` and `COLOR_NUM` is equal to `2`.

basically considers four additional abstract states in parallel and thus performs semantic loop unfolding for a better precision of the analysis, which allows Eva to realize a sound analysis for this example.

A.2 Counterexample 2, Confirmed with Eva

This section details the faulty execution of `pp_alloc_clr` mentioned in Sec. 5, caused by the buggy instruction `index++;` (on line 57 in Fig. 3) and provides a proof of its incorrect result. The proof relies on Frama-C/Eva, a value analysis plugin of Frama-C, which validates the provided ACSL assertions, as shown in Fig. 12. *The claim that the assertions of this example are proved with Eva can be demonstrated with the artifact [27].*

The pool layout described in Fig. 10, where letters A,. . . ,H refer to various pages or steps. The pool is made of 8 pages $\{p_0, \ldots, p_7\}$, the field `last` is equal to `1`, it points to p_1, the first page of the pool (see A). The memory pages are colored with alternate blue and yellow colors. The first page of the pool (p_0) is yellow.

Pages p_3 and p_7 are allocated, the others are free. The function `pp_alloc_clr` is called to find a free pset of 2 blue pages in that pool.

During the execution of `pp_alloc_clr`, the function first sets `index` equal to `1` (corresponding to page p_1) on line 45 (see A), as p_1 is a blue page. Then, in the first phase of the search (`i==0` in the loop on line 47), the function starts searching for the first free page of the pset with `index` equal to `1` (in loop line 50). As p_1 is free, it suits for the first page, and the loops stops.

Then the function searches for a second blue page in the loop on line 53. As `p1` satisfies the loop condition, it increments `index` to `2` (see B) and updates `index` with a call to `pp_next_clr`, which sets `index` to `3` (see C). Unfortunately, this page is not free, so the loop stops, and the function increments `index` to `4` through the *buggy instruction* on line 57 (see D).

At this point, the function resumes searching from `index` set to `4`, corresponding to page p_4. As this page is free, it satisfies the conditions of the loop (on line 50) leaving `index` unchanged, and page p_4 is considered as the first page of the candidate pset. Additionally, as `index` also satisfies the loop conditions (on line 53), `allocated` is incremented to `1` and `index` is incremented to `5` and updated by a call `pp_next_clr`, which sets `index` to `5` (see E), as page p_5 is blue. Again page p_5 satisfies the loop conditions and `allocated` can be incremented to `2`, completing the allocation successfully.

The function then exits the loop on line 48, setting `ppages->base` to `pool->base + p4*P_SIZE`.

However, when higher functions attempt to retrieve the pages of the pset, they will, first, call `pp_next_clr` on page p_4, returning page p_5 (see F) and then on page p_6—after increment (see G)—returning page p_7 (see H). **At this point, page p_7 may have already been allocated, possibly to the same VM or another VM accepting the blue color!!**

The concrete counterexample in Fig. 12 reproduces this scenario exactly. The blue color is encoded by color number `1`, and, the pool's `base` is set to `0` for simplicity. Accordingly, for all integer `i` within the pool's range, page p_i corresponds to page number `i`.

In a slightly modified version of this example, where `pool->size` equals to `6`, after the same steps, **the same page p_7 would lie outside the considered pool, and would be assigned to a VM, while this page can be non-existing, or belong to the hypervisor or be already allocated to the same or another VM.**

A.3 Complete corrected and verified code

Figures 13–23 give the complete version of the corrected, annotated and fully proved version of the Bao's cache coloring code, whose key functions are displayed in Fig. 3. The command used to run the proof is given at the end of the file.

Disclaimer: The proof results can vary depending on available resources (RAM, number of cores, timeout, etc.). This variation is expected. If the resources are insufficient and the proof is incomplete, the reader can update the command to increase the timeout (option `-wp-timeout`) and/or reduce the number of processes run in parallel (option `-wp-par`) and/or try on another machine with more RAM.

The next sections give additional explanations of the contracts for the two key functions presented in the paper, whose contracts were not detailed in the paper (as relatively straightforward and for lack of space).

A.4 Specification of `bitmap_get`

In the code, the function `bitmap_get` is always called with the `map` parameter equal to `pool->bitmap` for a certain pool. To enhance the scope of the specification and ensure that the function's behavior remains consistent within the broader context of memory, we added a pointer to the associated `pool` as a ghost argument, see line 211.

Preconditions. Before expressing properties with the `pool` ghost argument, we ensured that `pool` points to an existing and consistent pool of memory (line 201). Additionally, we ensure that the `bitmap` of `pool` is equivalent to its companion model (line 202). Finally, we bound the `map` parameter to the bitmap of `pool` (line 203) and constrain `bit` to ensure it fits within the pool's pages (line 204).

Postconditions. We express that the function always terminates (line 205) and does not modify memory locations (line 206). We ensure that the `pool` remains consistent after the function call (line 207) and maintains the equivalence with the companion model (line 208). This enables us to express that the function's result corresponds to the status of the page in the companion ghost array (line 209).

A.5 Specification of `bitmap_set`

Preconditions. The preconditions are similar to those of `bitmap_get`.

Postconditions. The preconditions are almost similar to those of `bitmap_get`. The difference is that, except for the page represented by `bit`, which must be marked as allocated (line 225), the status of all other pages remains unchanged (line 226).

```
#include "bao_cache_coloring.c"
#define NULL ((void *)0)
/*@
  logic ℤ P_CLR(ℤ page_num) = (page_num / COLOR_SIZE) % COLOR_NUM;
*/
void main() {
  COLOR_NUM = 2;        // Blue and yellow pages
  COLOR_SIZE = 1;
  u64 colors = 0b10;    // Search for blue pages
  u32 bit_map[1];
  page_pool pool = {
    .node = NULL,
    .base = 0,
    .size = 8,
    .last = 7,
    .bitmap = &bit_map
  };
  pool.bitmap[0] = 0b11111100;
  ppages pp;

  // Pool layout
  /*
  -----------------------------------
  | Page | Page Nb | Status | Color |
  -----------------------------------
  | p7   | 7       | Alloc. | 1     | <-- pool.last
  | p6   | 6       | Alloc. | 0     |
  | p5   | 5       | Alloc. | 1     |
  | p4   | 4       | Alloc. | 0     |
  | p3   | 3       | Alloc. | 1     |
  | p2   | 2       | Alloc. | 0     |
  | p1   | 1       | Free   | 1     |
  | p0   | 0       | Free   | 0     |
  -----------------------------------
  */

  // Status of page p1 before allocation: Free
  u32 p1_status = bitmap_get(pool.bitmap,1);
  //@ assert p1_status == 0;

  // Status of page p3 before allocation: Allocated
  u32 p3_status = bitmap_get(pool.bitmap,3);
  //@ assert p3_status == 1;

  // Allocate a pset of 2 pages of color 1
  // The first page is p0
  u8 res = pp_alloc_clr(&pool, 2, colors, &pp);
  //@ assert res == 1;
  //@ assert pp.num_pages == 2;
  //@ assert P_NB(pp.base) == 0;

  // First call to pp_next_clr over the pset
  // It retruns the offset corresponding to page p1
  u64 p_offset_call_1 = pp_next_clr(pp.base, 0, colors);
  //@assert P_NB(pp.base) + p_offset_call_1 == 1;

  // Second call to pp_next_clr over the pset
  // It retruns the offset corresponding to page p3
  // However p3 had already been allocated !
  u64 p_offset_call_2 = pp_next_clr(pp.base, ++p_offset_call_1, colors);
  //@assert P_NB(pp.base) + p_offset_call_2 == 3;

  return;
}

// To run:
// frama-c -eva -eva-slevel=4 counterexample_1.c
```

Fig. 11. Counterexample illustrating a faulty allocation due to the execution of the buggy instruction on line 71 in Fig. 3.

```c
#include "bao_cache_coloring.c"
#define NULL ((void *)0)
/*@
  logic ℤ P_CLR(ℤ page_num) = (page_num / COLOR_SIZE) % COLOR_NUM;
*/
void main() {
  COLOR_NUM = 2;        // Blue and yellow pages
  COLOR_SIZE = 1;
  u64 colors = 0b10;    // Search for blue pages
  u32 bit_map[1];
  page_pool pool = {
    .node = NULL,
    .base = 0,
    .size = 8,
    .last = 1,
    .bitmap = &bit_map
  };
  pool.bitmap[0] = 0b10001001;
  ppages pp;

  // Pool layout
  /*
  -----------------------------------
  | Page | Page Nb | Status | Color |
  -----------------------------------
  | p7   | 7       | Alloc. | 1     |
  | p6   | 6       | Free   | 0     |
  | p5   | 5       | Free   | 1     |
  | p4   | 4       | Free   | 0     |
  | p3   | 3       | Alloc. | 1     |
  | p2   | 2       | Free   | 0     |
  | p1   | 1       | Free   | 1     | <-- pool.last
  | p0   | 0       | Alloc. | 0     |
  -----------------------------------
  */

  // Status of page p5 before allocation: Free
  u32 p5_status = bitmap_get(pool.bitmap,5);
  //@ assert p5_status == 0;

  // Status of page p7 before allocation: Allocated
  u32 p7_status = bitmap_get(pool.bitmap,7);
  //@ assert p7_status == 1;

  // Allocate a pset of 2 pages of color 1
  // The first page is p4
  u8 res = pp_alloc_clr(&pool, 2, colors, &pp);
  //@ assert res == 1;
  //@ assert pp.num_pages == 2;
  //@ assert P_NB(pp.base) == 4;

  // First call to pp_next_clr over the pset
  // It retruns the offset corresponding to page p5
  u64 p_offset_call_1 = pp_next_clr(pp.base, 0, colors);
  //@assert P_NB(pp.base) + p_offset_call_1 == 5;

  // Second call to pp_next_clr over the pset
  // It retruns the offset corresponding to page p7
  // However p7 had already been allocated !
  u64 p_offset_call_2 = pp_next_clr(pp.base, ++p_offset_call_1, colors);
  //@assert P_NB(pp.base) + p_offset_call_2 == 7;

  return;
}

// To run:
// frama-c -eva -eva-slevel=4 counterexample_2.c
```

Fig. 12. Counterexample illustrating a faulty allocation due to the execution of the buggy instruction on line 57 in Fig. 3.

```
#include <limits.h>
typedef unsigned char u8;
typedef unsigned int  u32;
typedef unsigned long u64;
typedef struct page_pool {
  struct page_pool *node;
  u64 base;
  u64 size;
  u64 last;
  u32 *bitmap;
} page_pool;
typedef struct {
  u64 base;
  u64 num_pages;
  u64 colors;
} ppages;
#define P_SIZE (0x1000)
#define CELL_SIZE (sizeof(u32) * 8)
```

```
u64 COLOR_NUM;
u64 COLOR_SIZE;
#define P_NB(addr) ((addr)/P_SIZE)
#define P_NB_MAX (1UL << 52)
#define PL_NB_MAX (100)
#define NULL ((void *)0)
page_pool *page_pool_list;
/*@ ghost
  u64 gClrValid;
  u64 gExistPool;
  u64 gFoundPool;
  u8  gFlatClrs[64];
  u8  gPStatus[P_NB_MAX];
  u64 gFoundPSet[P_NB_MAX];
  u64 gExistPSet[P_NB_MAX];
  u64 gPageTable[P_NB_MAX];
  page_pool* gPools[PL_NB_MAX];
*/
```

```
/*@
logic ℤ P_CLR(ℤ page_num) = (page_num / COLOR_SIZE) % COLOR_NUM;

predicate ValidCacheCfg = 0 < COLOR_NUM ≤ 64 ∧ 0 < COLOR_SIZE < P_NB_MAX;
predicate IsValidPool(page_pool* pool) =
  \valid(pool) ∧ 0 ≤ P_NB(pool->base) < P_NB_MAX ∧
  0 ≤ pool->size < P_NB_MAX ∧ 0 ≤ pool->last ≤ pool->size ∧
  0 ≤ P_NB(pool->base) + pool->size ≤ P_NB_MAX ∧
  \valid(pool->bitmap + (0..pool->size/CELL_SIZE)) ∧
  \separated(pool,&(pool->bitmap[0..pool->size/CELL_SIZE]));
predicate flatPoolStatus(page_pool* pool) =
  \valid_read(pool) ∧ \valid_read(pool->bitmap + (0..pool->size/CELL_SIZE)) ∧
  \valid_read(&gPStatus[P_NB(pool->base)..(P_NB(pool->base)+pool->size-1)]) ∧
  ∀ ℤ idx; 0 ≤ idx < pool->size ⇒
    (((pool->bitmap[idx/CELL_SIZE] >> (idx%CELL_SIZE)) & 1)⟺
      gPStatus[P_NB(pool->base) + idx]);
predicate flatClrs(u64 colors) = ∀ ℤ clr; 0 ≤ clr < 64 ⇒
  (((colors >> clr) & 1) ⟺ gFlatClrs[clr]);
predicate IsInClrs(ℤ clr) = gFlatClrs[clr] ≠ 0;
predicate IsNotInClrs(ℤ clr) = gFlatClrs[clr] == 0;
predicate HasClrPages{L1,L2}(u64* PArr, ℤ p_base, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒  IsInClrs{L1}(P_CLR{L1}(p_base + \at(PArr[i],L2)));
predicate NoClrPBtw(ℤ p_base, ℤ start, ℤ end) =
  ∀ ℤ index; start ≤ index < end ⇒ IsNotInClrs(P_CLR(p_base + index));
predicate HasSeqPages{L1,L2}(u64* PArr, ℤ p_base, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i;  1 ≤ i < n ⇒ \at(PArr[i-1],L2) < \at(PArr[i],L2) ∧
    NoClrPBtw{L1}(p_base,\at(PArr[i-1],L2)+1,\at(PArr[i],L2));
predicate HasPagesInPool{L1,L2}(u64* PArr, page_pool* pool, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧ \at(\valid_read(pool),L1) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ 0 ≤ \at(PArr[i],L2) < \at(pool->size,L1);
predicate PSetInPool{L1,L2}(u64* PArr, page_pool* pool, u64 n, u64 colors) =
  \at(\valid_read(pool),L1) ∧ flatClrs{L1}(colors) ∧
  HasPagesInPool{L1,L2}(PArr,pool,n) ∧
  HasClrPages{L1,L2}(PArr,P_NB(\at(pool->base,L1)),n) ∧
  HasSeqPages{L1,L2}(PArr,P_NB(\at(pool->base,L1)),n);
predicate HasFreePages{L1,L2}(u64* PArr, page_pool* pool, u64 n) =
  \at(\valid_read(PArr + (0..n-1)),L2) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ \at(gPStatus[P_NB(pool->base)+\at(PArr[i],L2)],L1) == 0;
predicate HasAllocPages(u64* PArr, page_pool* pool, u64 n) =
  \valid_read(PArr + (0..n-1)) ∧
  ∀ ℤ i; 0 ≤ i < n ⇒ gPStatus[P_NB(pool->base) + PArr[i]] ≠ 0;
```

Fig. 13. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 1/11.

```
predicate IsHeadOfPoolList(page_pool * pool) =
  gPools[0] == pool ∧ gPools[PL_NB_MAX-1] == NULL ∧
  (∀ ℤ i; 0≤i<PL_NB_MAX-1 ⇒\valid(gPools[i]) ∧ gPools[i]->node==gPools[i+1])∧
  (∀ ℤ i, j; 0 ≤ i < PL_NB_MAX ∧ 0 ≤ j < PL_NB_MAX ∧ i ≠ j ⇒
    \separated(gPools[i],gPools[j])) ∧
  (∀ ℤ i,j; 0 ≤ i < PL_NB_MAX -1 ∧ 0 ≤ j < PL_NB_MAX -1 ∧ i ≠ j ⇒
      \separated(&(gPools[i]->bitmap)[0..gPools[i]->size/CELL_SIZE],
           &(gPools[j]->bitmap)[0..gPools[j]->size/CELL_SIZE]) ∧
      (P_NB(gPools[i]->base) + gPools[i]->size ≤ P_NB(gPools[j]->base) ∨
       P_NB(gPools[j]->base) + gPools[j]->size ≤ P_NB(gPools[i]->base))) ∧
  (∀ ℤ i,j; 0 ≤ i < PL_NB_MAX -1 ∧ 0 ≤ j < PL_NB_MAX -1 ⇒
    \separated(&(gPools[i]->bitmap)[0..gPools[i]->size/CELL_SIZE],gPools[j]))∧
  (∀ ℤ i; 0 ≤ i < PL_NB_MAX -1 ⇒
    \separated(gPools[i], &gExistPSet[0..(P_NB_MAX -1)]) ∧
    \separated(gPools[i], &gFoundPSet[0..(P_NB_MAX -1)]) ∧
    \separated(gPools[i], &gPageTable[0..(P_NB_MAX -1)])) ∧
  (∀ ℤ i; 0 ≤ i < PL_NB_MAX -1 ⇒
    \separated(&(gPools[i]->bitmap)[0..gPools[i]->size/CELL_SIZE],
        &gExistPSet[0..(P_NB_MAX -1)]) ∧
    \separated(&(gPools[i]->bitmap)[0..gPools[i]->size/CELL_SIZE],
        &gFoundPSet[0..(P_NB_MAX -1)]) ∧
    \separated(&(gPools[i]->bitmap)[0..gPools[i]->size/CELL_SIZE],
        &gPageTable[0..(P_NB_MAX -1)])) ∧
  (∀ ℤ i; 0 ≤ i < PL_NB_MAX -1 ⇒
    IsValidPool(gPools[i]) ∧ flatPoolStatus(gPools[i]));
predicate uPools{L1,L2} =
  \at(gPools,L1) == \at(gPools,L2) ∧
  \at(gPools[PL_NB_MAX-1 ],L1) == \at(gPools[PL_NB_MAX-1 ],L2) ∧
  ∀ ℤ i; 0 ≤ i < PL_NB_MAX-1 ⇒ \at(gPools[i],L1) == \at(gPools[i],L2) ∧
    (∀ ℤ bit; 0 ≤ bit < \at(gPools[i]->size/CELL_SIZE,L1) ⇒
      \at(gPools[i]->bitmap[bit],L1) == \at(gPools[i]->bitmap[bit],L2));
predicate uPStatus{L1,L2} =
  \at(gPStatus,L1) == \at(gPStatus,L2) ∧
  (∀ ℤ i; 0 ≤ i < P_NB_MAX ⇒ \at(gPStatus[i],L1) == \at(gPStatus[i],L2));
predicate uExistPSet{L1,L2} =
  \at(gExistPSet,L1) == \at(gExistPSet,L2) ∧
  (∀ ℤ i; 0 ≤ i < P_NB_MAX ⇒ \at(gExistPSet[i],L1) == \at(gExistPSet[i],L2));
predicate uPageTable{L1,L2} =
  \at(gPageTable,L1) == \at(gPageTable,L2) ∧
  (∀ ℤ i; 0 ≤ i < P_NB_MAX ⇒ \at(gPageTable[i],L1) == \at(gPageTable[i],L2));
predicate IsMappedTo(u64 vp, u64* PArr, page_pool* pool, u64 n) =
  ∀ ℤ i; 0 ≤ i < n ⇒ gPageTable[vp+i] == P_NB(pool->base) + PArr[i];
*/
/*@
lemma arith_1: ∀ ℤ a,b,c,d; 0 ≤ a ∧ 0 ≤ b ∧ 0 < c ∧ 0 < d ⇒
  ((a+b)/c)%d == ((a+b%(c*d))/c)%d;
lemma arith_2: ∀ ℤ a,b,c,d; 0 ≤ a ∧ 0 ≤ b ∧ 0 < c ∧ 0 < d ∧ (a/c)%d ≤ b ⇒
  ((a+c*(b-(a/c)%d))/c)%d == b%d;
lemma arith_3: ∀ ℤ a,b,c,d; 0 ≤ a ∧ 0 ≤ b ∧ 0 < c ∧ 0 < d ⇒
  ((a+c*(d+b-(a/c)%d))/c)%d == b%d;
lemma arith_4: ∀ ℤ a, b, c; 0 ≤ a ∧ 0 < b ∧ 0 < c ⇒  0 ≤ (a/b)%c < c;
lemma unique_next_clr_page: ∀ ℤ p_base, from, r1, r2;
  NoClrPBtw(p_base,from,r1) ∧ from≤r1 ∧ NoClrPBtw(p_base,from,r2) ∧ from≤r2 ∧
  IsInClrs(P_CLR(p_base+r1)) ∧ IsInClrs(P_CLR(p_base+r2)) ⇒ r1 == r2;
```

Fig. 14. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 2/11.

```
lemma PSetInPool_preserved{L1,L2,L3,L4}:
  ∀ page_pool* pool, u64 allocated, u64 colors, u64* PArr;
    PSetInPool{L1,L3}(PArr,pool,allocated,colors) ∧
    \at(\valid_read(pool),L2) ∧ \at(pool,L1) == \at(pool,L2) ∧
    \at(pool->base,L1) == \at(pool->base,L2) ∧
    \at(pool->size,L1) == \at(pool->size,L2) ∧
    flatClrs{L2}(colors) ∧ \at(COLOR_SIZE,L1) == \at(COLOR_SIZE,L2) ∧
    \at(COLOR_NUM,L1) == \at(COLOR_NUM,L2) ∧
    0 < \at(COLOR_SIZE,L1) ∧ 0 < \at(COLOR_NUM,L1) ≤ 64 ∧
    \at(\valid_read(PArr + (0..allocated-1)),L4) ∧
    (∀ ℤ i;  0 ≤ i < allocated ⇒ \at(PArr[i],L3) == \at(PArr[i],L4)) ⇒
    PSetInPool{L2,L4}(PArr,pool,allocated,colors);
*/
/*@
  requires ValidCacheCfg;
  requires 0 ≤ from < P_NB_MAX;
  requires flatClrs(colors);
  requires \valid_read(gFlatClrs + (0..63));
  requires 0 ≤ gClrValid < COLOR_NUM ∧ IsInClrs(gClrValid);
  terminates \true;
  assigns \nothing;
  ensures flatClrs(colors);
  ensures clr: IsInClrs(P_CLR(P_NB(base) + \result));
  ensures cons: NoClrPBtw(P_NB(base),from,\result);
  ensures bnd: from ≤ \result < from + COLOR_NUM*COLOR_SIZE;
*/
u64 pp_next_clr(u64 base, u64 from, u64 colors){
  u64 clr_offset = (base / P_SIZE) % (COLOR_NUM * COLOR_SIZE);
  u64 index = from;
  /*@ ghost
  u64 gBasePageNum = (base / P_SIZE);
  u64 gFromPageNum = gBasePageNum + from;
  u64 gFromClr = (gFromPageNum / COLOR_SIZE) % COLOR_NUM;
  u64 gIndexMax;
  if(gFromClr ≤ gClrValid){
    gIndexMax = index + (gClrValid - gFromClr) * COLOR_SIZE;
  //@ assert aif: 0≤(gClrValid-gFromClr)*COLOR_SIZE<COLOR_NUM*COLOR_SIZE;
  //@ assert aif: index ≤ gIndexMax < index + COLOR_NUM * COLOR_SIZE;
  /@  assert aif: ∀ ℤ x,c,d,e; 0 ≤ x ∧  0 < d ∧ 0 < e ∧ 0 ≤ c ∧ (x/d)%e ≤ c ⇒
        ((x+d*(c-(x/d)%e))/d)%e == c%e;
  @/
  //@ assert aif: P_CLR(gBasePageNum + gIndexMax) == gClrValid;
  }
  else {
    gIndexMax = index + (COLOR_NUM + gClrValid - gFromClr) * COLOR_SIZE;
    /@ assert aelse:
      0 ≤ (COLOR_NUM-(gFromClr-gClrValid))*COLOR_SIZE < COLOR_NUM*COLOR_SIZE;
    @/
    //@ assert aelse: index ≤ gIndexMax < index + COLOR_NUM * COLOR_SIZE;
    /@  assert aelse: ∀ ℤ x,c,d,e; 0 ≤ x ∧ 0 < d ∧ 0 < e ∧ 0 ≤ c ⇒
          ((x+d*(e+ c-(x/d)%e))/d)%e == c%e;
    @/
    //@ assert aelse: P_CLR(gBasePageNum + gIndexMax) == gClrValid;
  }
  */
  //@ assert clrim1: P_CLR(gBasePageNum + gIndexMax) == gClrValid;
  //@ assert clrim2: P_CLR(clr_offset + gIndexMax) == gClrValid;
  //@ assert frame: index ≤ gIndexMax < index + COLOR_NUM * COLOR_SIZE;
  /*@
  loop invariant Icons:
    ∀ ℤ i; from ≤ i < index ⇒ IsNotInClrs(P_CLR(P_NB(base) + i));
  loop invariant Ibnd: from ≤ index ≤ gIndexMax;
```

Fig. 15. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 3/11.

```
  loop assigns A: index;
  loop variant V: gIndexMax - index;
  */
  while (!((colors >> ((index + clr_offset) / COLOR_SIZE % COLOR_NUM)) & 1))
    index++;
  return index;
}

/*@
  requires IsValidPool(pool);
  requires flatPoolStatus(pool);
  requires map == pool->bitmap;
  requires 0 ≤ bit < pool->size;
  terminates \true;
  assigns \nothing;
  ensures IsValidPool(pool);
  ensures flatPoolStatus(pool);
  ensures \result ⟺ gPStatus[P_NB(pool->base) + bit];
*/
u32 bitmap_get(u32*map, u64 bit) /*@ ghost (page_pool * pool) */{
  return (map[bit / CELL_SIZE] & (1U << (bit % CELL_SIZE))) ? 1U : 0U;
}

/*@
  requires IsValidPool(pool);
  requires flatPoolStatus(pool);
  requires map == pool->bitmap;
  requires 0 ≤ bit < pool->size;
  terminates \true;
  assigns map[bit/ CELL_SIZE];
  assigns gPStatus[P_NB(pool->base) + bit];
  ensures IsValidPool(pool);
  ensures flatPoolStatus(pool);
  ensures gPStatus[P_NB(pool->base) + bit] ≠ 0;
  ensures cps: ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ∧ pNb ≠ P_NB(pool->base) + bit ⇒
    \at(gPStatus[pNb],Pre) == \at(gPStatus[pNb],Post);
*/
void bitmap_set(u32*map, u64 bit) /*@ ghost (page_pool * pool) */{
  map[bit / CELL_SIZE] |= 1U << (bit % CELL_SIZE);
  //@ ghost gPStatus[P_NB(pool->base) + bit] = 1;
}

/*@
  requires ValidCacheCfg;
  requires \valid_read(gFlatClrs + (0..63)) ∧ flatClrs(colors);
  requires 0 ≤ gClrValid < COLOR_NUM ∧ IsInClrs(gClrValid);
  requires IsValidPool(pool) ∧ flatPoolStatus(pool);
  requires 0 < n < P_NB_MAX;
  requires \separated(pool, ppages);
  requires \separated(pool, &gFlatClrs[0..63]);
  requires \separated(pool, &gFoundPSet[0..(P_NB_MAX-1)]);
  requires \separated(pool, &gExistPSet[0..(P_NB_MAX-1)]);
  requires \separated(pool, &gPStatus[0..(P_NB_MAX-1)]);
  requires \separated(ppages, &gFlatClrs[0..63]);
  requires \separated(ppages, &gFoundPSet[0..(P_NB_MAX-1)]);
  requires \separated(ppages, &gExistPSet[0..(P_NB_MAX-1)]);
  requires \separated(ppages, &gPStatus[0..(P_NB_MAX-1)]);
  requires \separated(&gFoundPSet[0..(P_NB_MAX-1)], &gFlatClrs[0..63]);
  requires \separated(&(pool->bitmap[0..pool->size/CELL_SIZE]),
    &gFoundPSet[0..(P_NB_MAX-1)]);
  requires \valid(ppages);
  terminates \true;
  assigns A: *ppages, gFoundPSet[0..(n-1)];
  assigns A: pool->last, pool->bitmap[0..pool->size/CELL_SIZE];
  assigns A: gPStatus[P_NB(pool->base)..(P_NB(pool->base) + pool->size - 1)];
```

Fig. 16. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 4/11.

```
  // ALWAYS
  ensures res: \result == 0 ∨ \result == 1;
  ensures fltc: flatClrs(colors);
  ensures vldp: IsValidPool(pool);
  ensures fltp: flatPoolStatus(pool);
  ensures ppclr: ppages->colors == colors;
  // ON SUCCESS
  ensures suc: PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
    HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ⇒ \result == 1;
  ensures wit1: \result == 1 ⇒
    PSetInPool{Pre,Post}((u64*)gFoundPSet,pool,n,colors);
  ensures wit2: \result == 1 ⇒
    HasFreePages{Pre,Post}((u64*)gFoundPSet,pool,n);
  ensures fct1: \result == 1 ⇒
    PSetInPool{Post,Post}((u64*)gFoundPSet,pool,n,colors);
  ensures fct2: \result == 1 ⇒ HasAllocPages((u64*)gFoundPSet,pool,n);
  ensures pps: \result == 1 ⇒  ppages->num_pages == n ∧
    ppages->base == pool->base + (gFoundPSet[0]*P_SIZE);
  ensures pss: \result == 1 ⇒ ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ⇒
    (∀ ℤ i; 0 ≤ i < n ⇒  pNb ≠ P_NB(pool->base) + gFoundPSet[i]) ⇒
    \at(gPStatus[pNb],Pre) == gPStatus[pNb];
  // ON FAILURE
  ensures ppf: \result == 0 ⇒ ppages->num_pages == 0;
  ensures ups: \result == 0 ⇒ ∀ ℤ i; 0 ≤ i < P_NB_MAX ⇒
    \at(gPStatus[i],Pre) == \at(gPStatus[i],Post);
  ensures upp: \result == 0 ⇒ \at(pool,Pre) == \at(pool,Post) ∧
    \at(*pool,Pre) == \at(*pool,Post);
  ensures ubm: \result == 0 ⇒ ∀ ℤ i; 0 ≤ i ≤ pool->size/CELL_SIZE ⇒
    \at(pool->bitmap[i],Pre) == pool->bitmap[i];
*/
u8 pp_alloc_clr(page_pool *pool, u64 n, u64 colors, ppages *ppages){
  u64 allocated = 0;
  u64 first_index = 0;
  u8  ok = 0;
  ppages->colors = colors;
  ppages->num_pages = 0;
  //@ ghost u64 gIndex;
  u64 index = pp_next_clr(pool->base, pool->last, colors);
  u64 top = pool->size;
  /*@
  loop invariant I4_idx_clr: IsInClrs(P_CLR(P_NB(pool->base) + index));
  loop invariant I4_PSetInPool:
    PSetInPool{Pre,Here}((u64*)gFoundPSet,pool,allocated,colors);
  loop invariant I4_FP:
    HasFreePages{Pre,Here}((u64*)gFoundPSet,pool,allocated);
  loop invariant I4_flatPS: flatPoolStatus(pool);
  loop invariant I4_flatClrs: flatClrs(colors);
  loop invariant I4_VP: IsValidPool(pool);
  loop invariant I4_pp: 0 < allocated ⇒ gFoundPSet[0] == first_index;
  loop invariant I4_pp_num_pages: ppages->num_pages == 0;
  loop invariant I4_pp_clr: ppages->colors == colors;
  loop invariant I4_ok: ok == 0;
  loop invariant I4_ex_i1:
    PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
    HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
    i == 1 ⇒  index ≤ gExistPSet[0];
  loop invariant I4_ex_i2:
    PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
    HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
    i == 2 ⇒  gIndex ≤ gExistPSet[0];
  loop invariant I4_i: 0 ≤ i ≤ 2;
  loop invariant I4_top : i == 2 ⇒ top ≤ gIndex;
  loop invariant I4_allocated: 0 ≤ allocated ≤ n;
```

Fig. 17. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 5/11.

```
  loop assigns A4: i, allocated, index, first_index;
  loop assigns A4: gIndex, gFoundPSet[0..(n-1)];
  loop variant V4: 2 - i;
  */
  for (u64 i = 0; i < 2 ∧ !ok; i++){
    /*@
    loop invariant I3_idx_clr: IsInClrs(P_CLR(P_NB(pool->base) + index));
    loop invariant I3_PSetInPool:
      PSetInPool{Pre,Here}((u64*)gFoundPSet,pool,allocated,colors);
    loop invariant I3_FP:
      HasFreePages{Pre,Here}((u64*)gFoundPSet,pool,allocated);
    loop invariant I3_flatPS: flatPoolStatus(pool);
    loop invariant I3_pp: 0 < allocated ⇒ gFoundPSet[0] == first_index;
    loop invariant I3_allocated: 0 ≤ allocated ≤ n;
    loop invariant I3_ex:
      PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
      HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
      i == 1 ∧ allocated < n ⇒  index ≤ gExistPSet[0];
    loop assigns A3: allocated, index, first_index, gFoundPSet[0..(n-1)];
    loop variant V3: top - index;
    */
    while ((allocated < n) ∧ (index < top)){
      allocated = 0;
      /*@
      loop invariant I2_idx_clr: IsInClrs(P_CLR(P_NB(pool->base) + index));
      loop invariant I2_VP: IsValidPool(pool);
      loop invariant I2_ex:
        PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
        HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
        i == 1 ⇒ index ≤ gExistPSet[0];
      loop invariant I2_dec: \at(index,LoopEntry) ≤ index ∧ allocated < n;
      loop assigns A2: index;
      loop variant V2: top - index;
      */
      while ((index<top)∧(bitmap_get(pool->bitmap,index)/*@ ghost (pool)*/))
        index = pp_next_clr(pool->base, ++index, colors);
      first_index = index;
      /*@
      loop invariant I1_flatPS: flatPoolStatus(pool);
      loop invariant I1_PSetInPool:
        PSetInPool{Pre,Here}((u64*)gFoundPSet,pool,allocated,colors);
      loop invariant I1_FP:
        HasFreePages{Pre,Here}((u64*)gFoundPSet,pool,allocated);
      loop invariant I1_idx_clr: IsInClrs(P_CLR(P_NB(pool->base) + index));
      loop invariant I1_VP: IsValidPool(pool);
      loop invariant I1_flatClrs: flatClrs(colors);
      loop invariant I1_idx_NoClrPagesBtw: 0 < allocated ⇒
        NoClrPBtw(P_NB(pool->base),gFoundPSet[allocated-1]+1,index) ∧
        gFoundPSet[allocated-1] < index;
      loop invariant I1_idx_0: 0 == allocated ⇒ index == first_index;
      loop invariant I1_fst_idx: 0 < allocated ⇒ gFoundPSet[0]==first_index;
      loop invariant I1_ex:
        PSetInPool{Pre,Pre}((u64*)gExistPSet,pool,n,colors) ∧
        HasFreePages{Pre,Pre}((u64*)gExistPSet,pool,n) ∧
        i == 1 ∧  allocated < n ∧  gExistPSet[0] ≤ index ⇒
        (∃ ℤ i; 0 ≤ i ≤ allocated ∧ gExistPSet[i] == index);
      loop invariant I1_allocated: 0 ≤ allocated ≤ n;
      loop invariant I1_idx_inc: \at(index,LoopEntry) ≤ index;
      loop invariant I1_idx_inc_s: 0<allocated ⇒ \at(index,LoopEntry)<index;
      loop assigns A1: allocated, index, gFoundPSet[0..(n-1)];
      loop variant V1: top - index;
      */
      while ((index<top)∧(bitmap_get(pool->bitmap,index)/*@ghost (pool)*/==0)
        ∧ (allocated < n)){
```

Fig. 18. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 6/11.

```
        //@ ghost gFoundPSet[allocated] = index;
        allocated++;
        index = pp_next_clr(pool->base, ++index, colors);
      }
      // index++; // FIXED: removed this line
    }
    if (allocated == n){
      ppages->num_pages = n;
      ppages->base = pool->base + (first_index * P_SIZE);
      //@ ghost u64 gFirst_index = first_index;
      /*@
      loop invariant I0_pp:
        ppages->base == \at(pool->base,Pre) + (gFoundPSet[0] * P_SIZE);
      loop invariant I0_variant: 0 ≤ j ≤ n;
      loop invariant I0_PSetInPool:
        PSetInPool{Pre,Here}((u64*)gFoundPSet,pool,n,colors);
      loop invariant I0_AP: HasAllocPages((u64*)gFoundPSet,pool,j);
      loop invariant I0_size: \at(pool->size,Pre) == pool->size;
      loop invariant I0_FP:
        HasFreePages{Pre,Here}((u64*)gFoundPSet,pool,n);
      loop invariant I0_flatPS: flatPoolStatus(pool);
      loop invariant I0_flatClrs: flatClrs(colors);
      loop invariant I0_VP: IsValidPool(pool);
      loop invariant I0_pp_clr: ppages->colors == colors;
      loop invariant I0_gFirst_index:
        (0 == j ⇒ gFirst_index == gFoundPSet[0]) ∧
        (0 < j ⇒ gFirst_index == gFoundPSet[j-1]);
      loop invariant I0_fst_idx:
        (0 == j ⇒ first_index == gFirst_index) ∧
        (0 < j ⇒ first_index == gFirst_index + 1);
      loop invariant I0_ps_mod: ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ⇒
        (∀ ℤ i;  0 ≤ i < n ⇒ pNb ≠ P_NB(pool->base) + gFoundPSet[i]) ⇒
        \at(gPStatus[pNb],Pre) == \at(gPStatus[pNb],Here);
      loop assigns A0: j, first_index;
      loop assigns A0: {pool->bitmap[gFoundPSet[i]/CELL_SIZE] | ℤ i; 0≤i<n};
      loop assigns A0: {gPStatus[P_NB(pool->base)+gFoundPSet[i]]|ℤ i;0≤i<n};
      loop assigns A0: gFirst_index;
      loop variant V0: n - j;
      */
      for (u64 j = 0; j < n; j++){
        first_index = pp_next_clr(pool->base, first_index, colors);
        //@ ghost gFirst_index = first_index;
      /*@ assert S0: NoClrPBtw(P_NB(pool->base),\at(first_index,LoopCurrent),
            gFoundPSet[j]);*/
      //@ assert S0: IsInClrs(P_CLR(P_NB(pool->base)+gFoundPSet[j]));
      //@ assert S0: \at(first_index,LoopCurrent) ≤ gFoundPSet[j];
      /*@ assert S0: NoClrPBtw(P_NB(pool->base),\at(first_index,LoopCurrent),
            gFirst_index);*/
      //@ assert S0: IsInClrs(P_CLR(P_NB(pool->base) + gFirst_index));
      //@ assert S0: \at(first_index,LoopCurrent) ≤ gFirst_index;
      //@ assert S0 : gFirst_index == gFoundPSet[j];
        bitmap_set(pool->bitmap, first_index++) /*@ ghost (pool)*/;
      }
      pool->last = first_index;
      ok = 1;
      //@ assert B_wit1:
           PSetInPool{Pre,Here}((u64*)gFoundPSet,pool,n,colors);
      //@ assert B_fct1:
           PSetInPool{Here,Here}((u64*)gFoundPSet,pool,n,colors);
      //@ assert B_VP: IsValidPool(pool);
      //@ assert B_flatPS: flatPoolStatus(pool);
      //@ assert B_flatClrs: flatClrs(colors);
      //@ assert B_pp_clr: ppages->colors == colors;
      break;
    }
```

Fig. 19. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 7/11.

```
    else {
      /*@ ghost
        if (i == 1){ gIndex = index; }
      */
      // index = 0; // FIXED: replaced this line by the next one
      index = pp_next_clr(pool->base, 0, colors);
    }
  }
  return ok;
}

/*@
  requires ValidCacheCfg;
  requires \valid_read(gFlatClrs + (0..63)) ∧ flatClrs(colors);
  requires 0 ≤ gClrValid < COLOR_NUM ∧ IsInClrs(gClrValid);
  requires 0 < num_pages < P_NB_MAX;
  requires 0 ≤ gExistPool < PL_NB_MAX-1;
  requires IsHeadOfPoolList(page_pool_list);
  assigns A: gFoundPSet[0..(num_pages-1)];
  assigns A: { gPools[i]->last | ℤ i; 0 ≤ i < PL_NB_MAX-1 };
  assigns A: { gPools[i]->bitmap[j] | ℤ i, j; 0 ≤ i < PL_NB_MAX-1 ∧
    0 ≤ j ≤ (gPools[i]->size/CELL_SIZE)};
  assigns A: gPStatus[0..(P_NB_MAX -1)];
  assigns A: gFoundPool;
  // ALWAYS
  ensures res: \result.num_pages == 0 ∨ \result.num_pages == num_pages;
  ensures Epl: IsHeadOfPoolList(page_pool_list);
  ensures fltc: flatClrs(colors);
  // ON SUCCESS
  ensures suc:
    PSetInPool{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages,colors)∧
    HasFreePages{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages) ⇒
      \result.num_pages == num_pages;
  ensures witres: \result.num_pages == num_pages ⇒ 0≤gFoundPool<PL_NB_MAX-1;
  ensures wit1: \result.num_pages == num_pages ⇒
    PSetInPool{Pre,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors);
  ensures wit2: \result.num_pages == num_pages ⇒
    HasFreePages{Pre,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  ensures fct1: \result.num_pages == num_pages ⇒
    PSetInPool{Post,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors);
  ensures fct2: \result.num_pages == num_pages ⇒
    HasAllocPages((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  ensures pps: \result.num_pages == num_pages ⇒
    \result.base == gPools[gFoundPool]->base + (gFoundPSet[0] * P_SIZE) ∧
    \result.colors == colors;
  ensures cps: \result.num_pages == num_pages ⇒
    ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ∧ (∀ ℤ i; 0 ≤ i < num_pages ⇒
      pNb ≠ P_NB(gPools[gFoundPool]->base) + gFoundPSet[i]) ⇒
      \at(gPStatus[pNb],Pre) == gPStatus[pNb];
  ensures cpl: \result.num_pages == num_pages ⇒
    ∀ ℤ i; 0≤i<PL_NB_MAX-1 ∧ i≠gFoundPool ⇒ \at(gPools[i],Pre)==gPools[i] ∧
      (∀ ℤ bit; 0 ≤ bit < \at(gPools[i]->size/CELL_SIZE,Pre) ⇒
        \at(gPools[i]->bitmap[bit],Pre) == gPools[i]->bitmap[bit]);
  // ON FAILURE
  ensures ups: \result.num_pages == 0 ⇒ uPStatus{Pre,Post};
  ensures upl: \result.num_pages == 0 ⇒ uPools{Pre,Post};
*/
ppages mem_alloc_ppages(u64 colors, u64 num_pages){
  ppages pages = {.num_pages = 0};
  //@ghost u64 i = 0;
  /*@
    loop invariant Ires: pages.num_pages == 0;
    loop invariant i: 0 ≤ i < PL_NB_MAX;
    loop invariant ll: pool == gPools[i];
    loop invariant pl: IsHeadOfPoolList(page_pool_list);
    loop invariant I_PS: uPStatus{Pre,Here};
```

Fig. 20. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 8/11.

```
    loop invariant I_PL: uPools{Pre,Here};
    loop invariant I_EPS: uExistPSet{Pre,Here};
    loop invariant I_clr: flatClrs(colors);
    loop invariant I_sc:
PSetInPool{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages,colors) ∧
HasFreePages{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages) ⇒
  PSetInPool{Here,Here}((u64*)gExistPSet,gPools[gExistPool],num_pages,colors)∧
  HasFreePages{Here,Here}((u64*)gExistPSet,gPools[gExistPool],num_pages);
    loop invariant I_succ:
    PSetInPool{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages,colors)∧
    HasFreePages{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages) ⇒
      i ≤ gExistPool;
    loop assigns A0: pages, pool, i, gFoundPool;
    loop assigns A0: gFoundPSet[0..(num_pages-1)];
    loop assigns A0: { gPools[i]->last | ℤ i; 0 ≤ i < PL_NB_MAX-1 };
    loop assigns A0: { gPools[i]->bitmap[j] | ℤ i, j; 0 ≤ i < PL_NB_MAX -1 ∧
        0 ≤ j ≤ (gPools[i]->size /CELL_SIZE)};
    loop assigns A0: gPStatus[0..(P_NB_MAX-1)];
    loop variant v0: PL_NB_MAX - i;
  */
  for (page_pool *pool = page_pool_list; pool ≠ (page_pool *)0; pool =
      pool->node){
    u8 ok = pp_alloc_clr(pool, num_pages, colors, &pages);
    if (ok)
      //@ghost gFoundPool = i;
      //@ assert succ_pl: IsHeadOfPoolList(page_pool_list);
      break;
    //@ghost i++;
  }
  return pages;
}

/*@
  requires 0 ≤ vp < P_NB_MAX;
  requires 0 ≤ pp < P_NB_MAX;
  assigns gPageTable[vp];
  ensures gPageTable[vp] == pp;
*/
void pte_set(u64 vp, u64 pp);

/*@
  requires ValidCacheCfg;
  requires \valid_read(gFlatClrs + (0..63)) ∧ flatClrs(colors);
  requires 0 ≤ gClrValid < COLOR_NUM ∧ IsInClrs(gClrValid);
  requires 0 < num_pages < P_NB_MAX;
  requires 0 ≤ gExistPool < PL_NB_MAX-1;
  requires IsHeadOfPoolList(page_pool_list);
  requires 0 ≤ vp;
  requires vp + num_pages ≤ P_NB_MAX;
  assigns A: gFoundPSet[0..(num_pages-1)];
  assigns A: { gPools[i]->last | ℤ i; 0 ≤ i < PL_NB_MAX-1 };
  assigns A: { gPools[i]->bitmap[j] | ℤ i, j; 0 ≤ i < PL_NB_MAX-1 ∧
    0 ≤ j ≤ (gPools[i]->size/CELL_SIZE)};
  assigns A: gFoundPool;
  assigns A: gPStatus[0..(P_NB_MAX-1)], gPageTable[0..(P_NB_MAX-1)];
  // ALWAYS
  ensures res: \result == 0 ∨ \result == 1;
  ensures Epl: IsHeadOfPoolList(page_pool_list);
  ensures fltc: flatClrs(colors);
  // ON SUCCESS
  ensures suc:
  PSetInPool{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages,colors) ∧
  HasFreePages{Pre,Pre}((u64*)gExistPSet,gPools[gExistPool],num_pages) ⇒
```

Fig. 21. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 9/11.

```
  \result == 1;
  ensures witres: \result == 1 ⇒ 0 ≤ gFoundPool < PL_NB_MAX-1;
  ensures wit1: \result == 1 ⇒
    PSetInPool{Pre,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors);
  ensures wit2: \result == 1 ⇒
    HasFreePages{Pre,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  ensures fct1: \result == 1 ⇒
    PSetInPool{Post,Post}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors);
  ensures fct2: \result == 1 ⇒
    HasAllocPages((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  ensures fct3: \result == 1 ⇒
    IsMappedTo(\at(vp,Pre),(u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  ensures cps: \result == 1 ⇒
    ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ∧ (∀ ℤ i; 0 ≤ i < num_pages ⇒
      pNb ≠ P_NB(gPools[gFoundPool]->base) + gFoundPSet[i]) ⇒
      \at(gPStatus[pNb],Pre) == gPStatus[pNb];
  ensures cpl: \result == 1 ⇒
    ∀ ℤ i; 0 ≤i<PL_NB_MAX-1 ∧ i≠gFoundPool ⇒ \at(gPools[i],Pre)==gPools[i] ∧
      (∀ ℤ bit; 0 ≤ bit < \at(gPools[i]->size,Pre)/CELL_SIZE ⇒
        \at(gPools[i]->bitmap[bit],Pre) == gPools[i]->bitmap[bit]);
  ensures cpt: \result == 1 ⇒
    ∀ ℤ pNb; 0 ≤pNb<P_NB_MAX ∧ (∀ ℤ i; 0≤i<num_pages ⇒ pNb ≠ \at(vp,Pre)+i) ⇒
      \at(gPageTable[pNb],Pre) == gPageTable[pNb];
  // ON FAILURE
  ensures ups: \result == 0 ⇒ uPStatus{Pre,Post};
  ensures upl: \result == 0 ⇒ uPools{Pre,Post};
  ensures upl: \result == 0 ⇒ uPageTable{Pre,Post};
*/
u8 mem_map(u64 colors, u64 vp, u64 num_pages){
  ppages temp_ppages = mem_alloc_ppages(colors, num_pages);
  if (temp_ppages.num_pages < num_pages)
    return 0;
  u64 index = 0;
  /*@
  loop invariant I_i: 0 ≤ i ≤ num_pages;
  loop invariant I_idx0: i == 0 ⇒ index == 0;
  loop invariant I_idxi: 0 < i ⇒ index == gFoundPSet[i-1]-gFoundPSet[0]+1;
  loop invariant I_base: temp_ppages.base == gPools[gFoundPool]->base +
       (gFoundPSet[0] * P_SIZE);
  loop invariant I_vp: vp == \at(vp,Pre) + i;
  loop invariant I_pt:
    IsMappedTo(\at(vp,Pre),(u64*)gFoundPSet,gPools[gFoundPool],i);
  loop invariant I_pl: IsHeadOfPoolList(page_pool_list);
  loop invariant I_fltc: flatClrs(colors);
  loop invariant I_wit:
  PSetInPool{Pre,Here}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors)∧
  HasFreePages{Pre,Here}((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  loop invariant I_fct:
  PSetInPool{Here,Here}((u64*)gFoundPSet,gPools[gFoundPool],num_pages,colors)∧
  HasAllocPages((u64*)gFoundPSet,gPools[gFoundPool],num_pages);
  loop invariant I_PS: uPStatus{LoopEntry,Here};
  loop invariant I_PL: uPools{LoopEntry,Here};
  loop invariant I_upt:
    ∀ ℤ pNb; 0 ≤ pNb < P_NB_MAX ∧
      (∀ ℤ i; 0 ≤ i < num_pages ⇒ pNb ≠ \at(vp,Pre) + i) ⇒
      \at(gPageTable[pNb],Pre) == gPageTable[pNb];
  loop assigns A0: i, index, vp, gPageTable[0..(P_NB_MAX-1)];
  loop variant temp_ppages.num_pages - i;
  */
  for (u64 i = 0; i < temp_ppages.num_pages; i++){
    index = pp_next_clr(temp_ppages.base, index, temp_ppages.colors);
    //@ assert A_idx: index == gFoundPSet[i] - gFoundPSet[0];
    u64 pp = P_NB(temp_ppages.base) + index;
    //@ assert A_pp: pp == P_NB(gPools[gFoundPool]->base) + gFoundPSet[i];
```

Fig. 22. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 10/11.

```
    pte_set(vp, pp);
    vp += 1;
    index++;
  }
  return 1;
}


// To run:
// frama-c-gui -wp -wp-rte -wp-smoke-tests -wp-par=4 -wp-timeout=30
     -wp-prover=script,alt-ergo,cvc5,z3,coq specified_cache_coloring.c
```

Fig. 23. Corrected and specified version of the key functions ensuring the allocation and mapping of pages using the cache coloring mechanism of Bao, part 11/11.