Verification of OS-level Cache Management

Renato Mancuso*, Sagar Chaki+
*Boston University, USA, rmancuso@bu.edu
+ Mentor Graphics, sagar.chaki@acm.org

***Abstract*—Recently, the complexity of safety-critical cyber-physical systems has spiked due to an increasing demand for performance, impacting both software and hardware layers. The timing behavior of complex systems, however, is harder to analyze. Real-time hardware resource management aims at mitigating this problem, but the proposed solutions often involve OS-level modifications. In this sense, software verification is key to build trust and allow such techniques to be broadly adopted. This paper specifically focuses on CPU cache management, demonstrating that OS-level hardware management logic can be verified at the source code level in a modular way, i.e., without verifying the entire OS.**

I. Introduction

In the last decade, there has been an uptrend in the complexity of safety-critical real-time systems. Such a trend is the result of an ever increasing demand for performance, features and efficiency. Multi-core platforms and heterogeneous hardware largely represents the industry's answer to such an increase in computational demand. As the hardware grows in complexity to match the demand for performance, it becomes increasingly hard to fully understand or to *predict* its timing behavior.

Unfortunately, the loss of timing predictability makes real-time analysis significantly harder, with two unwanted consequences. First, the inability to produce *tight* upper-bounds on workload worst-case execution time (WCET) leads to overprovision and waste of hardware resources. Nonetheless, the decreasing cost of hardware components partially mitigates this problem. Second, safety-critical systems are required to undergo a rigorous certification process in order to be considered for large-scale deployment. Difficulty in determining the logical and temporal correctness of a system heavily impacts certification costs. These costs easily surpass the sheer cost of hardware components by several orders of magnitude.

A number of works [9], [20], [12] have proposed OS-level mechanisms to explicitly manage those hardware components that, if unregulated, represent major sources of unpredictability: i.e. shared CPU caches, DRAM memory, and I/O subsystem. Management techniques proposed in the literature have been shown to achieve substantial real-time benefits. Yet, many industries are reluctant to widely adopt such solutions due to a fundamental lack of *confidence* about the correctness of their implementation. The fear is justified considering that hardware management mechanisms often operate at high-privilege level, and thus their misbehavior can lead to substantial failures.

This work represents a first step toward the verification of system-level components that implement hardware management techniques for real-time purposes. In fact, in this work we demonstrate that it is possible to verify the logic of a kernel-level component at the source code level in a modular way; i.e. without verifying the entire OS that can be assumed verified or trusted. Specifically, this paper presents the verification approach for Colored Lockdown [11]: a real-time last-level cache management scheme implemented in the Linux kernel. Colored Lockdown is part of a larger framework of hardware resources management techniques for multi-core platforms that goes under the name of Single Core Equivalence framework (SCE) [12], [13].

The rest of the paper is structured as follows. In Section II we provide an overview of the related work. Section III provides the required background knowledge for this work. A high-level description of our verification approach is discussed in Section IV, while additional implementation details are provided in Section V. Next, a brief evaluation is reported in Section VI. Finally, concluding remarks and possible future extensions are discussed in Section VII.

II. Related Work

As increasingly higher level of assurance is required from safety-critical systems, there has been an uptrend in the popularity of verification methodologies. A consistent body of works has used the "verified by design" approach. In this context, the SPARK language and toolkit [3] provide extensive capabilities to reason about the correctness of applications at a source code level. In the SPARK environment, verification is performed with a combination of static analysis and deductive verification. Deductive verification on the other hand, has been widely used on industrial use-cases [7], [10], [4]. Similarly, the level of assurance provided by formal static analysis based on abstract interpretation often represents a good trade-off in terms of scalability [16], [6].

Automated assertion checking is often used as an alternative to deductive verification. With this approach, it is typically possible to confine the explored state space to a manageable subset that is fundamental for the considered properties/assertions. Among the different techniques for assertion checking, bounded verification is often used for source code debugging. A number of consolidated tools implement assertion checking, e.g. SLAM [2], TASS [19], and CBMC [5] used in this paper.

Recent works have explored the use of verification techniques to validate application-level software in the domain of control systems [8], aerospace and avionic software [21], and railways systems [15]. In seL4 [14], the design and verification of an entire OS is proposed. While closely related to [14], we take a fundamentally different approach: we consider certified systems where new kernel-level functionality can be introduced to improve/optimize performance and demonstrate how modular verification of OS-level code can be performed. Finally, many works perform verification of the interaction between kernel modules and OS routines [17], [1]. Conversely, we focus on the verification of kernel-level logic that interacts with (i) kernel sub-routines, (ii) virtual memory, and (iii) CPU cache space.

III. Background

The philosophy behind SCE is that performance in a multi-core system can be analyzed and certified using a modular approach with respect to the rest of the system. In order to attain this goal, four main components are used in SCE to mitigate inter-core interference arising from a correspondent number of major sources [18], [12], [22], [23], [11]. Apart from other components used to manage DRAM and I/O, Colored Lockdown [11] is used to perform deterministic allocation of real-time task data and instruction in last-level shared cache. When Colored Lockdown is used, the portion

+This work was done while this author was an employee of Carnegie Mellon University.

of task memory allocated in cache will exhibit 100% hit rate. In this paper, we specifically focus on verifying the OS-level logic of Colored Lockdown. In this section, we provide an overview of the design of Colored Lockdown and briefly describes its internal components.

On multi-core systems, the timing of an application running on core A can be affected by a logically unrelated application running on core B if they share cache space. This timing inter-dependence goes under the name of "inter-core (performance) interference". The goal of Colored Lockdown [11] is to use *cache locking* to address inter-core interference while providing a trade-off between efficiency and flexibility. Colored Lockdown involves two main stages: an offline profiling stage; and an online cache allocation stage.

Profiling: during the offline stage, each real-time task is analyzed using a memory profiler [11]. When the task runs in the profiling environment, memory accesses are traced and per-page access statistics are maintained. Next, (i) pages of the task's addressing space are ranked by access frequency; and (ii) a profile is produced identifying frequently accessed (hot) pages by their relative position in the addressing space. The final profile can be used online to drive the cache allocation phase. Given the produced profile, two mechanisms are used to provide deterministic guarantees and a fine-grained cache allocation granularity, described below.

Page Coloring: last-level caches in modern multi-core platforms are typically set-associative physically indexed caches. As such, multiple main-memory pages can be mapped to a given *set* of shared cache pages. Pages in the same set are said to have the same "color". Pages with the same color can be allocated across cache *ways*, so that as many pages as the number of ways can be simultaneously allocated in last level cache. Any application page can be re-colored transparently to the application by only manipulating physical memory and page-table translations. Colored Lockdown relies on this mechanism to reposition task memory pages within the available colors, in order to exploit the entire cache space.

Lockdown: real-time applications are dominated by periodic execution flows. This characteristic allows for an optimized use of last level cache by locking hot pages first. Relying on profile data, Colored Lockdown first colors frequently accessed memory pages to remap them on available cache ways; next, it exploits hardware cache locking support to guarantee that such pages (once prefetched) will persist in the assigned location (locked), effectively overriding the default cache replacement policy.

IV. Verification Approach

This section provides an overview of the approach followed to verify the main properties of Colored Lockdown. We first establish the boundaries of the performed verification; next, we discuss what memory model is being considered; and finally we describe what components of the hardware/OS are abstracted.

Verification Strategy: we perform source-level verification via bounded model checking of the main block of code that is responsible for the allocation of memory pages in last-level cache within the Colored Lockdown module. The considered code is compiled as a Linux kernel module and runs at the highest level of privilege in the target platform. Verifying its correctness is therefore of great value.

In order to perform cache allocation, the Colored Lockdown module tightly interacts with the rest of the Linux kernel in two main ways: (i) it uses data from many descriptors used in the kernel; (ii) it invokes memory manipulation/translation procedures provided by the Linux kernel. The code base of the entire Linux kernel is too large and complex to be formally verified. For this reason, we restrict the verification to that portion of the cache allocation logic that is directly related to Colored Lockdown.

In order to focus the verification on the important components, we abstract the behavior of any invoked kernel routine, as detailed in Section V. For instance, a routine used to allocate a new generic memory page is abstracted as a function that returns an unsigned integer. The return value is non-deterministic, and such that: (i) it is aligned to the memory page size; and (ii) it is within the range defined by the bit-width of the considered memory layout.

Similarly, only sub-fields of kernel data structures that are relevant to verification are initialized by the verification routines. A portion of the initialization procedure is parameter-dependent, so that different cache allocation scenarios can be analyzed.

Verification Boundaries and Assumptions: the hardware-level properties that are abstracted mostly concern the behavior of a typical cache controller that allows per-line cache locking. Hence, we make the following assumptions. First, we assume that the initial status of the cache is unknown. This reflects the status of a cold cache at the time of Colored Lockdown allocation. Second, we assume that the considered cache is physically indexed[1]. Third, we consider that the bits of the physical address that encode the index of a cache line correspond to the least significant bits following the cache offset bits. Hence, the structure of a physical address from the cache controller's perspective from most to least significant bit is: tag bits, index bits, offset bits. Since we consider cache controllers that support per-line locking, we assume that a special instruction is available to set a lock bit on a per-line basis. Once the lock bit has been set, the cache line cannot be evicted from cache. Finally, we assume that a cache look-up for a locked line will result in a cache hit.

We verify an implementation of Colored Lockdown as a Linux kernel module. The same logic, however, can be ported across different OS's, assuming that they provide kernel-level routines with similar semantics. In order to focus our attention on the target module, we assume that the descriptors belonging to the OS and used by Colored Lockdown have been correctly initialized (see Section V). Next, we assume that profile information about the process under consideration have been correctly passed from user-space to kernel-space. Finally, we assume that all the virtual memory pages of the process have a valid mapping in physical memory. The latter assumption is typically verified in RTOS's that do not perform demand-paging. Under Linux, this behavior can be achieved using the `mlockall` system call.

Memory Layout Specification: our verification is parametric with respect to the memory layout and cache controller configuration. Thus, it is possible to re-run the verification procedure on a specific memory/cache configuration and with a variable number of pages to be allocated, i.e. *profile pages*. The following five parameters suffice to fully define the considered memory subsystem as well as the address structure from the cache controller's perspective:

(1) P_s: Number of bits in a virtual address that encode the offset of a byte in a memory page, also known as *page shift*;
(2) B_w: Bit-width of a physical address in the considered platform, e.g. 32 for 32-bit architectures; 48 for 64-bit architectures[2].
(3) O: Number of bits in a physical address that encode the offset of a byte within a cache line;
(4) I: Number of bits in a physical address that encode the index in cache of a cache line;
(5) W: Associativity – i.e. number of ways of the cache.

[1]Last-level caches in multi-core platforms are typically physically tagged and indexed.

[2]Despite the bit-width of CPU registers is 64 bit, the memory subsystem typically works with 48 bit addresses. This results in 256 TB of addressable memory and a 4-level page tables layout is used.

Given the five parameters above, the rest of the parameters used to perform cache locking can be derived: size of a memory page; size of a single cache line; number of lines and pages (i.e., available colors) per way; number of cache sets; bit-width of the cache tag; and total size of the cache.

One more parameter controls the amount of memory that is allocated in cache for the process under consideration. This parameter defines a generic number of pages that is prefetched and locked in cache as a result of the coloring/locking logic. By default, all the pages are considered as process' heap pages. This however does not affect the generality of our approach since there is no difference in the way pages belonging to various regions are handled.

Verified Properties: a set of core properties of Colored Lockdown was successfully verified. The target of the verification is twofold: (i) that cache allocation is correctly performed when profiling data are correctly specified from user-space, and the amount of memory to be locked in cache is smaller than the cache size; and (ii) that the status of the system and cache is overall consistent. Note that using the current verification infrastructure, additional system/cache properties can be verified. The verified properties can be summarized as follows:

(1) If the number of pages to be allocated in cache is less or equal to the number of available cache space in pages, cache allocation for the considered process is entirely performed. Otherwise, no cache allocation is performed;
(2) If cache allocation is performed, then all the physical memory mapped to each virtual address within the range selected for allocation will be locked in cache;
(3) No more than the total number of locked pages are set as locked in cache at the end of the Colored Lockdown procedure;
(4) All the temporary kernel-level resources required by Colored Lockdown to execute are released at the end of the procedure.

Verification Challenges: we hereby summarize the challenges that had to be addressed to perform source-level verification of Colored Lockdown as a OS-level component. One of the first challenges we encountered in the attempt to verify a Linux kernel module was the large number of dependencies with the kernel source code that a module can exhibit. Three main type of dependencies exist: data type dependencies, procedural dependencies, and logic dependencies.

A Linux kernel module uses several types that are defined and exported by the kernel. Many of these types are complex C-language structures interconnected via pointers. Obviously, only a subset of the fields in such structures are required for focused verification. CBMC v. 5.2 [5], the source code verification tool we used, employs slicing to eliminate unused variables and reduce verification complexity. However, we found this to be inadequate for our target system. The first challenge was to manually prune the definitions of kernel-level structures to exclude all the irrelevant fields. In order to overcome this issue, we have incrementally transferred into the verification sandbox a number of kernel headers and systematically stripped them of unneeded data types and fields. For instance, one of the imported files was `sched.h` that in the Linux kernel defines constants and types relevant for process management. The file is about 2700 lines long in a typical Linux source tree. In the first pruning, we only maintained the process descriptor definition, reducing the file length to about 370 lines. Next, we identified the only two fields required for verification out of the 170+ fields included in a typical process descriptor.

The second type of dependency is procedural dependency. The code that needs to be verified uses at top level a set of routines defined in the kernel code. To reduce the state space and the amount of code logic to be verified, one challenge consists in abstracting the semantics of the invoked procedures (if possible) and making a reasonable assumption on their output. In Section V we describe as an example the abstraction performed on the kernel procedure `get_user_pages`.

Finally, many logic dependencies exist between the state of the kernel and the verified module. This problem sets our verification approach apart from verification of standalone components. In fact, the Colored Lockdown module expects the status of a number of kernel-level descriptors to be initialized and valid. Some of these descriptors are created at boot-time, while others are constantly updated upon system events. Hence, it would be unfeasible to verify the code responsible for their initialization. To tackle this challenge, we have first identified all the logic dependencies. Next, we have introduced an initialization routine that either explicitly sets each referenced variable to its expected value or assumes its value to be within the expected range. A closer look at the initialization procedure is provided in Section V.

Overall, CBMC revealed a good maturity in handling C source code. However, when verifying kernel-level code, we have encountered a few glitches that need to be carefully addressed to avoid false negatives in the verification process. Relatively simple workarounds have been found for all the encountered glitches. Such problems, however, can represent a serious overhead in the verification process when reasoning over a large base of system-level code.

The first problem we encountered regards the way `void` pointers are handled in CBMC. The standard C semantics enforces that the increment of a `void *` data type is performed at the granularity of a single byte. Consider the following code:

```
int void_test(void)
{
    void * ptr = (void *)(1 << 12);
    ptr += 0x100;
    return (ptr == (void *)0x00001100UL);
}
```

The code compiles without warnings/errors under a standard GCC compiler. The expected return of the `test` function is always 1 under standard C-pointer arithmetic. However, a CBMC verification instance that relies on this behavior will fail. Running CBMC 5.2 on the considered procedure, produces the following output:

```
Counterexample:

State 21 file ./cbmc_test.c line 18 function void_test thread 0
----------------------------------------------------------
  ptr=NULL (00000000000000000000000000000000)

State 22 file ./cbmc_test.c line 18 function void_test thread 0
----------------------------------------------------------
  ptr=NULL + 4096 (00000000000000000001000000000000)

State 23 file ./cbmc_test.c line 19 function void_test thread 0
----------------------------------------------------------
  ptr=NULL + 3840 (00000000000000000000111100000000)

Violated property:
  file ./cbmc_test.c line 58 function main
  assertion return_value_void_test$1
  (_Bool)return_value_void_test$1

VERIFICATION FAILED
```

Clearly, State 23, which should reflect the pointer's status after the increment in the considered code extract, reports a wrong pointer value. This triggers a verification failure. A possible workaround consists in performing the pointer value increment after a conversion to unsigned long[3].

The second issue requires a longer explanation and due to space constraints we omit a detailed description. Briefly, CBMC seems to exhibit a glitch in the propagation of a variable value after it has been assigned using a bitwise operator. Consider the following snippet:

[3]The unsigned long type has typically the same width of a pointer.

```
int retval = 1;
for (...) {
    retval &= bool_function(...);
}
if (retval) ...
```

In this case, CBMC produced verification counterexamples that reported the execution of the `if` block even though the state value of the `retval` variable was 0 (false);

In our verification, we found that many subtle interactions in system-level code are hard to fully capture at a source level. Consider the following case. Colored Lockdown performs recoloring of a process page. For this purpose, a new physical page is allocated and its content copied from the original page, appropriately modifying the page tables of the process under consideration. This behavior is "correct" as far as Colored Lockdown is concerned. However, if no action is taken to de-allocate the original page correctly, Colored Lockdown can indirectly trigger a fault somewhere else in the system as the original page descriptor remains in an inconsistent state. Similar interplay problems can occur when a module accesses a data structure without acquiring the required lock. In a typical multi-threaded application, this problem would be easy to detect since all the execution flows are known. The problem is however significantly harder to solve without knowing where in the kernel potential data races can arise.

Finally, a challenge that affects source-level verification at large, is the quick increase in complexity as the state-space expands. In our verification attempt, we were able to overcome the vast majority of challenges described in this section. In spite of this, verification settings with realistic parameters required significant computational resources. We provide additional insights on the feasibility and limits of our verification approach in Section V.

V. Verification Details

In this section, we provide additional details about the performed verification. First, we discuss how the cache hardware is modeled; next, we discuss the initialization of kernel structures and OS state. A detailed overview about how cache and memory layout are initialized is also provided. Finally, we detail the structure and verification statements used to verify the core properties of Colored Lockdown.

Cache Model: traditional source-level verification tools, including CBMC, do not provide primitives to model platform hardware behavior. For this reason, we use a supporting data structure to maintain the cache state and to perform assertions on its state. Colored Lockdown allows deterministic allocation of memory pages in cache. Thanks to coloring, the mapping set is explicitly controlled. Conversely, the decision about the allocation way is left to the cache controller. The key insight, however, is that when the replacement policy attempts to allocate a line with a certain set, and the line for that set is marked as locked in a given cache way, the way cannot be selected for eviction. Thus, as long as a number of lines less or equal to the cache associativity is locked, each locking request can be satisfied. It follows that the logical view of a cache is a 2D structure (sets vs. ways). One index (set index) is derived from the physical address being allocated; while the other index (way index) is non-deterministically determined by the replacement policy.

Following this structure, the cache status is defined as:

```
1  typedef struct {
2      void * addr;
3      char locked;
4  } cache_line_t;
5
6  typedef cache_line_t cache_set_t [CACHE_ASSOC];
7  typedef cache_set_t cache_t [CACHE_NSETS];
8  cache_t cache;
```

In the listing above, `CACHE_ASSOC` and `CACHE_NSETS` refer to the number of sets and to the number of ways (associativity), respectively. Note that there is no need to record the *value* of the cached data, as we are only concerned with hit/miss behavior. Hence, only cached *address* and *locked status* are being tracked.

The assumption we make about the initial state of the cache is that no line is currently locked. As such, we initialize the locked state on all the cache elements as 0, and assign a non-deterministic value to the address field.

Profile Structure and Initialization: as stated in Section IV, we assume that profiling information has been passed from user-space to kernel-space before the lockdown procedure is invoked. Hence, for verification purposes, we explicitly initialize the kernel structures that hold kernel-side profile data. In Colored Lockdown, profiling data is provided via the Linux CGROUP virtual file-system interface. For a task for which a profile has been loaded via the CGROUP interface, a custom structure, namely `struct task_profile` is associated with the task descriptor. The most relevant fields of the structure are: (i) number of memory regions with pages to be locked; (ii) list of descriptors for memory regions with data to be locked; (iii) total number of pages to be allocated in cache; (iv) list of descriptors for pages to be locked.

Since the state of the `struct task_profile` object is assumed to be valid, an initialization routine was added. The routine allocates enough data to contain the full list of memory regions and memory pages. These parameters are set at profile loading time, hence they are known at the time Colored Lockdown is invoked. In the context of this paper, they constitute parameters for the creation of a verification instance. A default scheme is used to associate memory pages to areas. This choice however does not compromise the generality of the verification, as there is no difference in the way pages in different areas are handled.

Within each memory region's descriptor, only the index that the considered region has in the list of kernel-maintained virtual memory areas (VMA) is initialized. The logic that resolves such a (relative) index into an absolute range of virtual memory addresses is part of the Colored Lockdown logic. Hence, it is part of the verification.

Task Descriptor Setup: when Colored Lockdown is invoked as a system call by a task, it heavily relies on information contained within the kernel-maintained task descriptor `struct task_struct` to perform cache allocation. Whenever any system call is invoked in the kernel, a globally visible expression, namely `current`, expands to a pointer to the `struct task_struct` object for the calling process. For verification purposes, the object pointed by `current` needs to be initialized. The following is an extract of the task descriptor setup routine:

```
1   int pages;
2   struct vm_area_struct * prev_vma;
3   struct vm_area_struct * cur_vma;
4   /* ... */
5   prev_vma->vm_start = 0x08048000UL;
6   current->mm->mmap = prev_vma;
7   pages = nd_int();
8   __CPROVER_assume(pages >= AREA_MINPAGES && pages <= AREA_MAXPAGES);
9   prev_vma->vm_end = prev_vma->vm_start + (pages << PAGE_SHIFT);
10  /* Link VMAs */
11  cur_vma->vm_start = prev_vma->vm_end;
12  prev_vma->vm_next = cur_vma;
13  /* Use cur_vma to setup next VMA */
```

The first area in the list of VMAs is typically the `text` (i.e. the executable code) section of a process. The start of the first area is taken as the default address at which code is logically placed in compiled executables (line 5). The address of the first VMA descriptor is recorded inside the `current` object (line 6). Next, a non-deterministic number of pages between the established boundaries is generated in lines 7–8, and the end of the first VMA is set accordingly (line 9). As VMAs are initialized, they are placed in an unidirectional linked list (lines 11–12).

Colored Lockdown Procedure: the Colored Lockdown module also performs a series of initialization routines as soon as it is loaded (once) into the kernel. The routines mostly initialize cache parameters and buffers required to perform page coloring. Due to space constraints, we omit the details about how initialization is performed inside the verification environment.

When Colored Lockdown core logic is invoked as a system call, the sequence of operations can be summarized as follows:

(1) Access to profile structure and validation of `current` object – to make sure Colored Lockdown is performed on the right task;
(2) Derivation of virtual addresses for each memory page in the profile to be allocated in cache;
(3) Resolution of virtual addresses into physical addresses and cache color calculation;
(4) Check of color availability in cache and assignment of first available color;
(5) If each page has been assigned a color, perform page re-coloring (as needed) and lockdown.

Hereby, we provide a few extracts of kernel logic that are relevant to understand the interaction with CBMC. The first point is trivially verified because we assume that profile data passing and Colored Lockdown invocation is performed correctly. The second step largely uses data in the `current` descriptor initialized as described in Section V. Next, in order to translate the virtual addresses of pages to be allocated, Colored Lockdown uses a kernel routine, namely `get_user_pages`. The `get_user_pages` routine represents an entry point for a number of page-wide kernel operations that can be selected via a `flag` parameter. When invoked with no flags, the function takes as input a range of (virtual) addresses and a task descriptor and returns an array of pointers to *page descriptors*. Each page descriptor corresponds to a page in the selected range. In Linux, the value of the pointer to a page descriptor is always a linear translation of the described page's physical address. Hence, knowing the pointer to the page descriptor for a page is equivalent to knowing its physical address. The `get_user_pages` logic is fairly complex, but since it is part of the kernel, it sits beyond our verification boundaries. As such, we have abstracted much of its functionality as follows.

```
1  long get_user_pages(struct task_struct *tsk, struct mm_struct *mm,
       unsigned long start,unsigned long nr_pages, int write, int
       force, struct page **pages, int *locked)
2  {
3      struct page * page_ptr;
4      assert(nr_pages == 1);
5      assert(write == 0);
6      assert(force == 0);
7      assert(tsk == current);
8      assert(mm == tsk->mm);
9      page_ptr = __CPROVER_uninterpreted_void_ptr(tsk, mm, start);
10     __CPROVER_assume(page_ptr >= mem_map && page_ptr <  (mem_map +
           MAX_PAGES));
11     __CPROVER_assume(((unsigned long)page_ptr & ((1 << sizeof(
           struct page)) - 1)) == 0);
12     *pages = page_ptr;
13     return 1;
14 }
```

First, a set of asserts on the passed parameters is performed (lines 4–8), to verify the expected value of a number of parameters when `get_user_pages` is called within Colored Lockdown. An uninterpreted function is used (line 9) to construct a valid return value for the routine. In general, the returned value can be any pointer to a `page_struct` object (line 11) with a value between `mem_map`[4] and the end of that portion of kernel memory where page descriptors are stored (line 10). For any specified parameter value of `tsk`, `mm` and `start`, the same page pointer should be returned by successive invocations of `get_user_pages`. Hence the use of an uninterpreted function at line 9. The derivation of physical addresses from page descriptor pointers follows a similar logic.

In the following step the availability of colors is checked. The check is performed using an internal structure that remembers the color associated to each page to be allocated. The step is performed with minimal kernel interaction. When a "conflict" page is encountered, i.e. a page with an unavailable color, the module selects the closest available color. It also marks the internal descriptor for the page to reflect the change. At this stage, no recoloring is performed, hence no final changes are carried out.

If the procedure has determined that there exist enough available space to perform cache allocation, the following actions are performed. First, the module performs re-coloring of all the conflict pages. Second, it executes a cache lockdown operation on each line of each profile page. In the considered architecture, the lockdown is performed using a dedicated assembly instruction, namely `DCBTLS`[5]. In order to perform verification, however, we also update the status of the structure used to model the cache. More in detail, we invoke the `lock_line` procedure on each address corresponding to every line in a page being allocated. The `lock_line` procedure is reported below.

```
1  void lock_line(void * addr)
2  {
3    unsigned int index = get_index(addr);
4    unsigned int way = nd_int();
5    __CPROVER_assume(way >= 0 && way < CACHE_ASSOC);
6    __CPROVER_assume(!cache[index][way].locked);
7    cache[index][way].addr = addr;
8    cache[index][way].locked = 1;
9  }
```

The procedure is invoked on physical addresses, hence it is easy to calculate the cache index of the line, i.e. the cache set where the line will map (line 3). Since no specific cache replacement policy is assumed, the way selected for the allocation is generated as a non-deterministic integer (`nd_int()`, line 4) between 0 and the number of available ways (line 5). The ways where a line has been previously locked in the same set are excluded (line 6), as per assumed hardware behavior. Finally, with selected set/way, line locking is carried out as in lines 7 and 8.

To complete the verification, after Colored Lockdown is invoked, we check that: (i) every physical address (at the granularity of single cache lines) in pages to be allocated, as per the profile, can be found in our cache structure; and that (ii) no more locked lines than what specified in the profile is marked as locked.

VI. Evaluation

In this section, we provide a brief evaluation of the time required to perform verification using the proposed approach. The evaluation has been performed under two memory/cache layout scenarios using CBMC version 5.2 on a workstation machine featuring a 28-core Intel Xeon E5-2658 CPU running at 2.10 GHz with 32 GB of RAM. Unfortunately, CBMC only uses only one core and it is not possible to parallelize the verification effort due to the large amount of memory required to acquire each sample.

In the first scenario, we consider a 32-bit system ($B_w = 32$) with the following memory layout: memory pages of size 256 bytes ($P_s = 8$); a cache line size of 64 byte ($O = 6$); and a way size of 512 bytes ($I = 3$), so that each cache way can entirely hold 2 memory pages. We study the length of the verification for an increasing number of profile pages and cache associativity. Moreover, we set the timeout for the verification to 2 hours. The results for this setup are reported in Figure 1.

[4]In a Linux kernel, this symbol represents the beginning of the array of page descriptors.

[5]This instruction is common to PowerPC-based platforms, such as Freescale MCPxxx and QorIQ P40xx platforms.

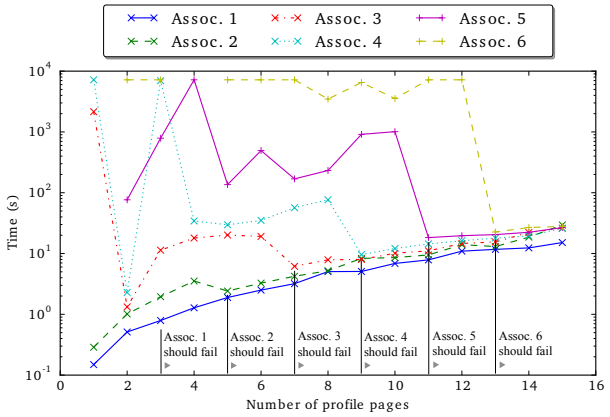

Fig. 1. Verification runtime for scenario: $P_s = 8, B_w = 32, O = 6, I = 3$ and associativity $W \in [1, 7]$.

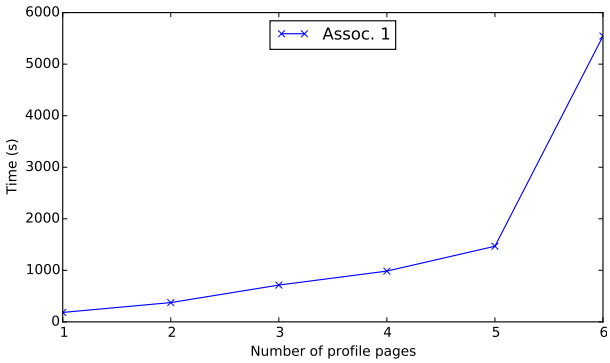

Fig. 2. Verification runtime for scenario with $P_s = 12, B_w = 32, O = 6, I = 10$ and $W = 1$.

In the figure, we use logarithmic scale to visualize in a compact way the runtime of the considered scenarios. As can be seen, the verification runtime can require from few milliseconds to entire hours, depending on the complexity of the system. For a low number of pages and higher associativity, we consistently observe peaks in execution time. We believe that these peaks originate from the increased flexibility of in-cache placement, which negatively impacts the size of the state space. In general, as the number of pages is incremented with a fixed associativity, the increment in runtime follows a regular trend and is exponential in time. Intuitively, this arises from the exponential increase in state space size to be explored by CBMC. It can also be noted that the verification time sharply decreases in those instances of verification that are not supposed to succeed. These cases, highlighted in the figure, correspond to those setup where the cache space is insufficient to carry out allocation, and where verification fails as it should. In this cases, CBMC stops after encountering a verification counter-example, hence it does not perform a complete exploration of the state space. Unfortunately, cases beyond associativity 6 consistently timeout in our evaluation.

In a second scenario, we evaluate the verification time for a more complex memory/cache layout by fixing the associativity to 1 and varying the number of pages. We consider a 32-bit system with 4 KB memory pages ($P_s = 12$), 64 byte cache line size ($O = 6$), and a way size of 64 KB ($I = 10$). In this layout, a single cache way can contain up to 16 memory pages. The results are depicted in Figure 2.

As shown in the figure, a sharp increment in runtime is observed at 6 profile pages. Although not included in the graph, any verification attempt for pages beyond that boundary runs longer than the selected 2 hours timeout threshold. Nonetheless, even with the current approach, verification is feasible on a general-purpose machine for a limited number of profile pages.

VII. Conclusions and Future Work

In this work, we focused our attention on verification of kernel-level cache management logic. We have demonstrated that it is possible to perform verification by reasoning directly on the system-level C code of the target module. Key properties for advanced kernel-level features were verified in a modular way with respect to the rest of the OS logic. In our approach, we relied on bounded model checking via CBMC. The work opens many possibilities for improvement. As a part of our future work, we will investigate how to include elements of deductive verification to allow verification of more complex scenarios. Additionally, we will attempt verification of complementary real-time hardware kernel logic with the goal of establishing an industry-ready, verified real-time resource management framework.

References

[1] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static driver verification with under 4% false alarms. In *Formal Methods in Computer-Aided Design (FMCAD)*, Oct 2010.

[2] T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, 2001.

[3] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] S. Boldo and T. Nguyen. Hardware-independent proofs of numerical programs. In *NASA Formal Methods Symposium*, 2010.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, Lecture Notes in Computer Science. Springer-Verlag, March–April 2004.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7), July 2008.

[7] S. Duprat, P. Gaufillet, V. M. Lamiel, and F. Passarello. Formal verification of SAM state machine implementation. In *Embedded Real Time Software and Syst. (ERTSS)*, May 2012.

[8] R. A. B. e Silva, N. N. Arai, L. A. Burgareli, J. M. P. de Oliveira, and J. S. Pinto. Formal verification with frama-c: A case study in the space software domain. *IEEE Transactions on Reliability*, 65(3):1163–1179, Sept 2016.

[9] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. Rtos support for multicore mixed-criticality systems. In *IEEE Real Time and Embedded Technology and Applications Symposium*, April 2012.

[10] N. Kosmatov and J. Signoles. Frama-c, a collaborative framework for c code verification: Tutorial synopsis. In *International Conference on Runtime Verification*, Cham, 2016. Springer International Publishing.

[11] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, April 2013.

[12] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun. WCET(m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183, July 2015.

[13] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET derivation under single core equivalence with explicit memory budget assignment. In *Euromicro Conference on Real-Time Systems (ECRTS), Dubrovnik, Croatia*, 2017.

[14] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: From general purpose to a proof of information flow enforcement. In *Security and Privacy (SP), 2013 IEEE Symposium on*, May 2013.

[15] V. Prevosto, J. Burghardt, J. Gerlach, K. Hartig, H. Pohl, and K. Voellinger. Formal specification and automated verification of railway software with frama-c. In *IEEE International Conference on Industrial Informatics (INDIN)*, July 2013.

[16] A. Puccetti. Static analysis of the xen kernel using frama-c. *Journal of Universal Computer Science*, 16, 2010.

[17] V. V. Rubanov and E. A. Shatokhin. Runtime verification of linux kernel modules based on call interception. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 180–189, March 2011.

[18] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford. Real-time computing on multicore processors. *Computer*, 49(9):69–77, Sept 2016.

[19] S. F. Siegel and T. K. Zirkel. TASS: The toolkit for accurate scientific software. *Mathematics in Comp. Science*, 5(4), 2011.

[20] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. MERASA: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[21] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu. Formal verification of critical aerospace software. *AerospaceLab Journal*, May 2012.

[22] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[23] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64, April 2013.