Computing and Informatics, Vol. 36, 2017, 925–949, doi: 10.4149/cai_2017_4_925

OPTIMIZING MEMORY USAGE IN L4-BASED MICROKERNEL

Petre Eftime, Lucian Mogoșanu, Mihai Carabaș
Răzvan Deaconescu, Laura Gheorghe

Faculty of Automatic Control and Computers
University Politehnica of Bucharest, Splaiul Independentei nr. 313
Sector 6, București, 060042, Romania
e-mail: `petre.eftime@cti.pub.ro,`
`{lucian.mogosanu, mihai.carabas}@cs.pub.ro,`
`{razvan.deaconescu, laura.gheorghe}@cs.pub.ro`

Valentin Gabriel Voiculescu

VirtualMetrix, Inc.
16738 Zumaque Street, PO Box 569
Rancho Santa Fe, CA 92067, USA
e-mail: `gabi@virtualmetrix.com`

Abstract. Memory allocation is a critical aspect of any modern operating system kernel because it must run continuously for long periods of time, therefore memory leaks and inefficiency must be eliminated. This paper presents different memory management algorithms and their aplicability to an L4-based microkernel. We aim to reduce memory usage and increase the performance of allocation and deallocation of memory.

Keywords: Memory management, memory allocation, SLUB, SLAB, kernel, L4 microkernel

Mathematics Subject Classification 2010: 68N25, 68N99

1 INTRODUCTION

Memory management is a critical part of every operating system kernel and bad memory usage can negatively impact the usability of the system as a whole. Efficient and reliable memory management is a desired feature of any kernel.

The goal of this paper is to identify and fix shortcomings related to memory allocation and usage of an L4-based microkernel. Since a microkernel has limited resources at its disposal, it must allocate them efficiently and usefully. Failure to do so limits the system's usability and reliability, features which are desired in any kernel designed to run real life applications.

Allocation and freeing memory are some of the most frequent operations in an operating system kernel [1]. Good memory operation performance is required. Large overhead can slow down the whole system and negatively affect other aspects such as energy usage, which is important in all systems, from servers to cell phones.

In practice one cannot optimally allocate memory because the number, the order and the size of the objects cannot be predicted. We use state of the art approximation algorithms to ensure fast allocation, minimize the internal/external fragmentation, but we have taken into account cache alignment, memory locality at the expense of consuming more memory.

The paper is structured as follows. Section 2 introduces the reader to the basics of operating systems and memory management mechanisms. Section 3 lists previous work on memory allocation, using some of the most well known and widely used types of allocators available as examples. Sections 4, 5 and 7 explain the design, implementation and evaluation of a novel bitmap-based allocator and its benefits in comparison to previous solutions. Section 6 describes and measures improvements of memory-related updates added to the microkernel. In Section 8 we analyse how well the goal was accomplished and what more can be done to improve kernel memory utilisation.

2 BACKGROUND

This section describes the basic concepts of memory management and some of the attributes that will be extended to the L4 microkernel used as basis for our implementation.

An operating system (OS) is a collection of programs and services that act as intermediary between a computer and the user, such that the user does not need to know any specifics about how the computer works at a fundamental level. At the heart of every OS we find the kernel, with its main purpose of providing an abstraction layer for the hardware such that the userspace programs can be targeted at a broader range of devices, running the same – or similar, interface-wise – kernel, but on different hardware. The kernel does not (or should not) provide any functionality that the user needs directly; e.g., the kernel can provide a system call through which the user can write to an external storage, but the user would still need a program to do so, possibly provided with the OS. Therefore, from a user's perspective, the kernel should use the least amount of resources necessary, so that the user's own programs can benefit from them.

Kernels can be classified into multiple categories, based on their internal architecture: *monolithic kernels*, that have all the drivers running in kernel-mode – a privileged mode which lets them access the kernel's resources in a more direct way, *microkernels*, that have the drivers running in non-privileged (userspace) mode, or *hybrid systems*, where some drivers run in privileged mode, while some run in userspace.

All these architectures provide a similar interface to userspace applications in terms of usability, but they each have their respective strengths and weaknesses.

Monolithic kernels can be faster than microkernels and usually provide userspace programs with a better API, but they are less stable. Due to their large codebase a misbehaving driver can crash the whole system.

Drivers in microkernels are seen as servers. This means that driver failure only results in a few programs losing connection to the driver in question, and the driver my be restored. This provides improved reliability, but also increases communication overhead.

2.1 Kernel Memory Management

An integral and critical part of every modern kernel, regardless of kernel architecture, is the memory management. Overcommitted resources or slow performance during memory allocation or deallocation impacts the functionality and stability of the whole OS and the applications built upon it.

The kernel must allocate resources for the processes and threads running on top of it. These resources are usually allocated in a dynamic way, so that the OS can be used in any way the user wants. This means that most allocation and deallocation of resources is done at runtime, and can be a source of latency when launching a process or a thread, or when allocating memory in userspace. On mobile systems it can also mean reduced battery life.

The kernel usually possesses a limited amount of memory available. In the case of the microkernel, this memory can be very small (when compared to the total amount of memory available). Proper usage of available memory is necessary, as overcommitting memory can mean running out of the kernel memory for important tasks that a user wants to perform.

2.2 L4 Microkernel

In this paper we present the optimization of memory management mechanisms of a microkernel from the L4 family. A typical scenario is when the L4 microkernel acts as a host for Linux-based operating systems, allowing for sandboxing and sharing of resources between multiple instances of Linux or other native programs.

The original L4 microkernel was developed by Jochen Liedtke [2, 3] and was designed to address the performance problems that most microkernels suffered from, which prevented them being used in real-world applications.

3 KERNEL MEMORY ALLOCATION

Kernel memory management must be fast, (space) efficient and reliable. Over years many algorithms have been created to address these needs.

There are two factors which provide an insight into the efficiency of memory allocation algorithms, namely internal and external fragmentation. Internal fragmentation (Formula 1) is the measure of unused space inside an allocated region, while external fragmentation (Formula 2) is the measure of how well compacted the unallocated memory is.

$$\text{Internal Fragmentation} = \frac{\text{Used Memory}}{\text{Allocated Memory}} \tag{1}$$

$$\text{External Fragmentation} = 1 - \frac{\text{Largest Block of Free Memory}}{\text{Total Free Memory}} \tag{2}$$

A kernel memory allocator should be reliable: allocated areas should not overlap and the allocator should be able to allocate all the memory it manages. Bugs in the allocator almost always lead to corruption of data and the system instability.

Kernels usually do not have a large quantity of memory at their disposal. The memory they use is non-paged, which means that it can never be swapped out, and it has to fit inside the RAM available on the system at all times. If this memory is not properly managed, the system could be severely limited in terms of usability: only a small number of processes and threads could be created and drivers would not be able to reserve enough memory for their needs, making some hardware components unusable. Efficient management of memory should mean low internal and external fragmentation. Metadata memory usage should also be kept as light as possible.

The speed of memory management is usually secondary to reliability and efficiency, since a slow allocator that works well is better than a fast allocator that mismanages resources. However, this does not mean speed is not a desired feature, and a tradeoff must usually be made between the speed and efficiency. Low average allocation time translates to good performance of the system, while low maximum allocation time makes it worst for a real OS time.

In the rest of this section we present various algorithms that are commonly used in state of the art operating system kernels. Our focus lies mainly on Linux due to its popularity and its open source nature. Other kernels, e.g. iOS or Windows, are also very relevant to our topic, but they are difficult to analyze due to the lack of documentation.

3.1 Buddy Allocator

The buddy allocator is one of the oldest allocators available. It is first described by Kenneth C. Knowlton [4], but Donald Knuth attributes it to Harry Markowitz as early as 1963 [5]. It is simple but flexible, and a number of improvements have been made to it to better fit different scenarios or to provide better performance. It is normally used as an allocator for smaller areas of memory, since the algorithm is conducive for fragmentation.

The algorithm [5] implements two operations: allocate(size) and free(address). The buddy allocator can only allocate sizes which are powers of two. It keeps a list of free spaces for each power of two it can allocate.

At allocation, the algorithm rounds the requested size up to the nearest power of two and searches in the list of free areas available for that size. If an area is available, it is allocated. If not, the algorithm allocates an area of size 2^{k+1} – repeating the process if no area of size 2^{k+1} exists – and splits it into two "buddy" areas, allocates one, and puts the other in the free list. This is equivalent to using the Best Fit Decreasing (BFD) policy.

At deallocation, the algorithm will try to locate the buddy area of the one being deallocated, and if it is in the free list, the areas are merged together and added to the list of blocks one size larger. The algorithm reiterates, if possible. It is easy to compute the starting address of a buddy block, since allocation preserves alignment and because of this only one bit needs to be flipped: for a block of size 2^k, it is the $(k+1)^{\text{st}}$ bit.

The main advantages of a buddy system allocator are simplicity and speed. The worst case for allocation and deallocation is $\Theta(\log n)$, where n is the size of the memory. However it can be improved to $O(1)$ at the cost of memory [6]. A comparison can be made with algorithms that are using the First Fit Decreasing (FFD) policy, which is much slower.

The biggest disadvantage of the buddy allocator is fragmentation, both internal and external. In fact, it is even possible that enough continuous memory is available, but it is spread in chunks of different sizes and cannot be allocated. The previously mentioned FFD policy has a much lower fragmentation compared to the Buddy Allocator and may be more desirable in some situations.

3.2 SLAB

The SLAB allocator was first introduced in the Solaris 2.4 kernel by Jeff Bonwick [1], and is currently used in many UNIX(-like) operating systems such as FreeBSD and Linux [7], although Linux has superseded it recently, with the introduction of SLUB.

The SLAB allocator was built around the observation that many times the costliest operation is not finding an empty memory area, but initializing the allocated object. The policy aims to reduce the cost of initialization by maintaining caches of common objects in an already-constructed state. SLAB can also be used by itself, above another allocator that does not know of the SLAB system, but it cannot free cache memory when it is needed, can complicate debugging and increase kernel code size and maintenance cost.

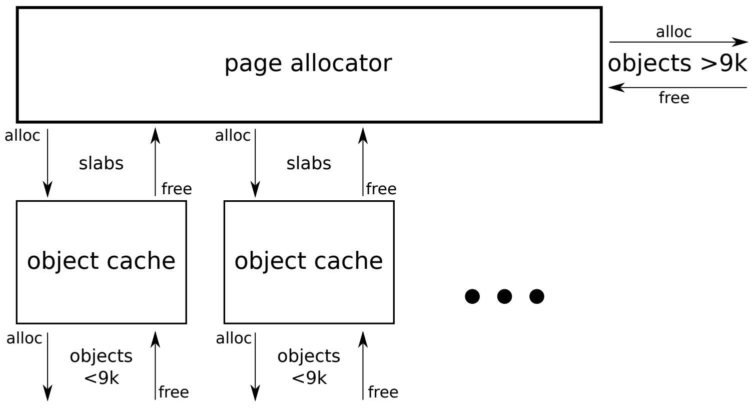


Figure 1. SLAB architecture

The SLAB allocator has two data structures at its base: caches and slabs. For each type of object, a cache must be created, and it needs to include other information besides size, namely the constructor, destructor and alignment of the objects.

Each cache has a front-end and a back-end, as shown in Figure 1. The front-end are the object caches and the back-end is the page allocator. At the front-end, the operations available are the allocation and deallocation of objects. These operations can be performed in constant time, only needing to prepend a list, or remove the head of the list. The back-end is hidden from the rest of the kernel and manages the available physical memory. The operations available to the back-end are growing or reaping the cache. The back-end is only accessed under stress conditions, usually when a cache runs out of objects.

The constructorless allocation operations are handled either through a series of internal caches of varying sizes, or directly from the back-end, for large sizes. Caches improve concurrency as well, as there can be a lock for each cache and one for the back-end, but back-end access is rare.

Slabs are areas of contiguous memory of one or more pages. They are the units used by the allocator: when growing or shrinking a cache, an entire slab is affected. Each slab has a reference count, and can be freed when that reference count is zero. The advantages of using slabs are numerous: reclaiming memory is trivial, allocating and freeing slabs is a constant time operation, internal fragmentation is minimal and external fragmentation is unlikely.

The SLAB allocator has good performance and minimizes fragmentation, but it requires extra data for each cache and slab, which can be a problem for systems with limited space. Another shortcoming of SLAB is that it can keep slabs busy with very few objects in each.

3.3 SLOB

SLOB is an allocator with a similar interface to SLAB [8], but which does not cache objects. Instead, it uses the constructor and destructor at every allocation and deallocation. This means it does not have any of the advantages SLAB has, but it does use less metadata, making it more useful for systems with limited memory.

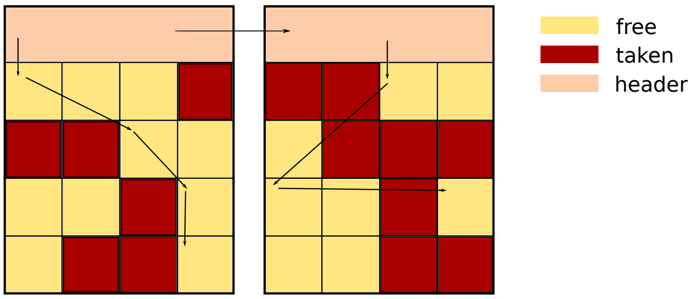


Figure 2. SLOB list overview

The SLOB allocator uses a few lists of pages, each allocating areas of a certain size. For areas bigger than one page, there is a page allocator. Areas smaller than a page will be allocated from one of the lists. The lists are linked lists of pages, each page having a header and a number of objects of a given size (shown in Figure 2). In Linux, by default there are 3 lists: one for objects under 256 B, one for objects under 1 kB, and one for objects under 4 kB (default page size) [9]. When no more areas can be allocated from any of the pages, a new page will be added.

Every page is divided into smaller chunks of a given size. Contiguous areas of chunks are linked in a list. At allocation, these lists inside pages will be traversed, and when a large enough area has been found, one or more chunks will be removed from it. If an area is completely allocated, it will be removed from the list. At deallocation, the page to which the area belongs is identified and either the space is coalesced into an already existing area or a new area is added to the list.

3.4 SLUB

SLUB is a Linux update [10] to SLAB meant to reduce memory overhead [11]. It simplifies slab structure and packs metadata into already existing structures. It also reduces the number of queues and introduces a thread that supervises which CPU uses each queue, in order to prevent cache line bouncing. SLAB uses per-CPU queues, which in systems with a large number of CPUs leads to large memory usage. SLUB uses queues per memory node, which allows multiple CPUs to use the same slabs.

SLUB is said to bring 5 % to 10 % better performance and reduce slab caches by 50 %, while also reducing slab fragmentation [11].

3.5 Ordered Linked List Allocator

An allocator using an ordered linked list (shown in Figure 3) to keep track of available free memory areas is an old and simple allocator, which like the buddy system, is easy to implement. Unlike the buddy system, it is not really fast except under certain conditions. It does not use extra memory, but it also cannot be improved easily, and does not relate well to common hardware architectures, as it accesses distant memory areas, which usually translates to cache misses. This was the default allocator under the L4 microkernel used for testing and is used in other microkernels from the L4 family such as OKL4 [12] and L4Ka::Pistachio [13].

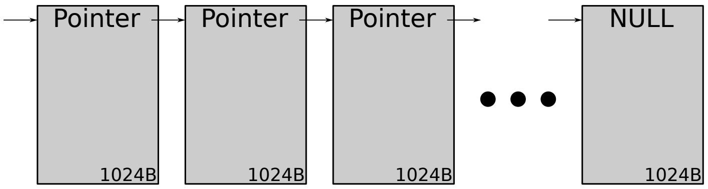


Figure 3. Ordered linked list

Allocation is done in chunks, the size of chunks being constant. This means that for allocations under the chunk size, the allocator would still allocate one chunk, which translates to internal fragmentation. As for external fragmentation, the allocator is a first-fit, which may lead to external fragmentation. Fortunately, if the chunk size is chosen to be a commonly allocated size, this diminishes the impact of first-fit in terms of fragmentation. Each free chunk in the list contains the address of the next free chunk.

The allocation is a simple process: a pointer moves through the list of free chunks, counting continuous memory areas. When it finds a continuous memory area of sufficient capacity, it removes it from the list and allocates it. This works especially well for the most common case, which is allocations of 1 kB, when it only has to remove the head of list. However, for larger areas of memory, this algorithm provides poor performance, especially when the memory becomes fragmented, since the cache lines are not big enough to fit more than the start of each chunk, which contains the address of the next chunk and nothing else.

Deallocation is also a simple process, but an expensive one. The algorithm must keep the list of free chunks ordered, so when the memory area is freed, it must find the original position from where the area came, which means iterating through the list. Again, this means memory accesses instead of cache accesses, and if the memory to be freed is towards the back of the list, this may take a long time.

This algorithm is advantageous for certain use cases (e.g., when the number of common case allocations is large and there is a small number of free operations). If kernel memory becomes fragmented, this algorithm may start performing badly. This also means that allocation and free time may vary in an unpredictable fashion.

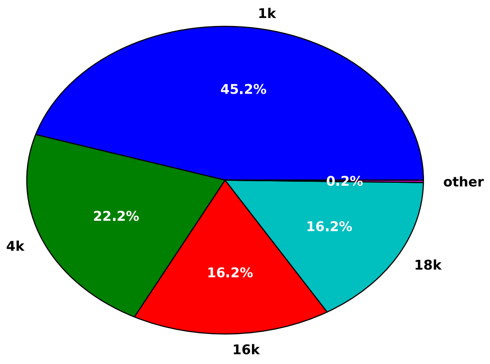
4 DESIGN OF A NEW L4 MICROKERNEL ALLOCATOR

A new allocator for the L4 microkernel has design requirements in terms of memory usage and performance requirements: stable time for allocation and deallocation (which is required for an OS to be considered real time), better mean allocation time, metadata usage kept to a minimum (so that more resources can be spent on useful data) and maintaining the same interface as the previous allocator (to avoid making large modifications to the kernel).

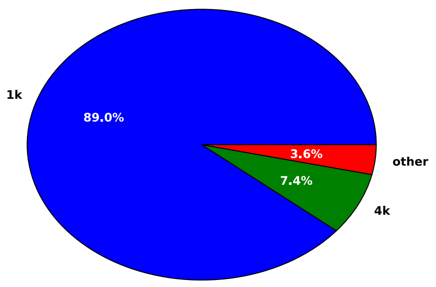
The allocators described in Section 3 are not necessarily good allocators for a microkernel: SLAB uses too much metadata, SLUB could be good, but hard to implement, as it relies on some other structures which the L4 microkernel does not have, the buddy system fragments the memory too much. SLOB seems well fitted for the task, but the microkernel already has a functional system for allocations under 1 kB, and SLOB uses extra metadata to deal with allocating different sizes in the same slab. Another point to be made about the SL*B family of allocators is that they work on top of a page allocator, with code and metadata of their own. It is not practical to have multiple allocators in a microkernel.

This design does not approach any optimizations to the scheduling algorithms. The L4 microkernel uses a simple round-robin scheduler in order to provide policy-free scheduling to processes on top of it. Our main concern is that the scheduling algorithm must be as generic as possible with no interest in optimality on processor allocation.

Figure 4 shows that an overwhelming majority of the objects allocated in the microkernel are of the sizes 1 kB and 4 kB; 16 kB and 18 kB are also numerous under our `ktest` test suite, but under real life conditions they end up at under 2 % each (see Section 7 for information about what data was collected). For objects under



a) ktest



b) Linux

Figure 4. Allocated sizes

1 kB there is a special allocation system called **smallpools** (see Section 6.1), which is available in the microkernel, so they do not show up on these charts.

4.1 Allocator Design

The observations made previously give some hints as to what could constitute a suitable allocator for our L4 microkernel: it must allocate memory in blocks of 1 kB, or multiples, so that internal fragmentation would be kept to a minimum; metadata usage should be kept to a minimum; the mechanisms for allocating or freeing data should be simple, but fast.

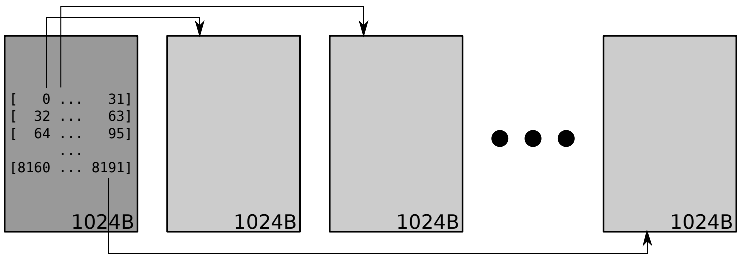


Figure 5. New allocator design

A page allocator satisfies these criteria, but there is no need to allocate pages, which are typically 4 kB and aligned to 4 kB. This is not required for the microkernel to function. The algorithms used by a page allocator could be modified and reused.

The previous allocator achieves this successfully, but with poor performance, because it keeps 1 kB chunks in a list of free chunks. Another design for a page allocator is bitmap-based, providing better performance.

A bitmap-based allocator (presented in Figure 5) uses little metadata – 1 bit for every chunk, with the value 0 for unallocated and 1 for allocated – which allows the allocator to verify more possible options at once instead of having to iterate through every item in a list. Allocation verifies a series of adjacent chunks at once if they are all free and deallocation simply means marking a series of adjacent chunks at once as free using a bit mask, thus making it faster than the ordered list iterator presented in Section 3.5.

For each chunk there is a bit in the bitmap to denote whether that chunk is allocated or free. Formula 3 transforms bit index into the starting address of the chunk it is referencing, needing the address of the first chunk and the size of each chunk to calculate a displacement. The reverse operation is true (Formula 4), allowing for immediate location of the bit indexing the chunk, needed during deallocation.

$$\text{Chunk Start Address} = \text{Kernel Memory Start Address} + \text{Bit Index} * \text{Chunk Size} \tag{3}$$

$$\text{Bit Index} = \frac{\text{Chunk Start Address} - \text{Kernel Memory Start Address}}{\text{Chunk Size}} \tag{4}$$

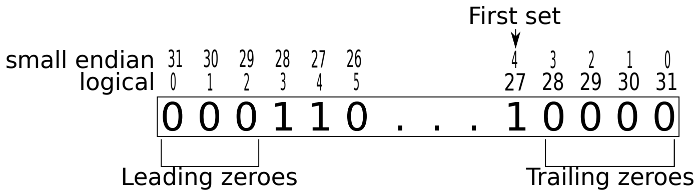

First set
small endian 31 30 29 28 27 26 4 3 2 1 0
logical 0 1 2 3 4 5 27 28 29 30 31
0 0 0 1 1 0 . . . 1 0 0 0 0
Leading zeroes
Trailing zeroes

Figure 6. Used instructions

This allocator can also be easily optimized for the architecture on which it runs, as most modern architectures provide instructions such as *find first zero*, *count leading zeroes*, *count trailing zeroes* (shown in Figure 6) or can implement such functionality using a small number of instructions. It preserves cache locality much better, because metadata is all in the same memory area. It is fairly easy to optimize to work well for the majority of allocations, knowing their size.

To reiterate, the theoretical advantages of the bitmap-based memory allocator over other allocators would be:

- Minimal metadata. Metadata is constant, and only 1 bit per chunk. This translates to one chunk of metadata for each 8192 chunks of data.
- Localized metadata. All allocator metadata is in the same place, making it easier to search through it and reducing the number of cache misses.

- Bitwise operations. CPUs have instructions for bitwise operations, making the code easy to optimize by the compiler.
- Simplicity. While it is more complicated than the buddy allocator and the sorted list allocator, it simpler than SLAB/SLOB/SLUB, which in a microkernel is an advantage.
- Flexibility. It allows for the system to have heuristics and can be changed between first fit and best fit easily.
- Efficiency. It leads to less internal and external fragmentation than the buddy allocator, and is on par with any other first fit allocators.
- Performance. It uses operations easily translated into assembly and does not need to access the memory as often as other algorithms to read or write metadata.

In contrast, the disadvantages of a Bitmap-Based Chunk Allocator are:

- Complexity. Optimizations for special cases lead to longer and more complex code that is more difficult to debug.
- Metadata. Metadata is minimal, but existent.
- Architecture specific code. Some of the operations work differently on little endian systems than on big endian system. They are few, but require extra testing and work.
- Performance. It is a general purpose cacheless allocator, so it will perform worse than SLAB and SLUB.
- Fragmentation. Being a first fit allocator, external fragmentation is probably larger than it is with a best fit or cache-based allocator. It allocates multiple chunks of 1 kB, so unless it is used with another allocator for areas under 1 kB it can also lead to internal fragmentation.

5 IMPLEMENTATION OF THE BITMAP-BASED ALLOCATOR IN THE MICROKERNEL

The allocator uses chunks as units of allocation, and allocates areas which have sizes equal to an integer multiple of the **chunk size**, by default 1 kB. When the microkernel is initialized it reserves at least one memory area. This memory area will be used for allocating kernel objects dynamically. When a memory area is being initialized for use, it checks the size of the area and allocates the first few chunks to use as a bitmap. The number of chunks depends on how large the memory actually is. It needs to allocate one chunk of memory as a bitmap for each $8 *$ **chunk size** chunks of memory it manages. For 1 kB sized chunks, this means 1 kB for 8 MB of memory.

For brevity, the term "word" will be used to refer to an unsigned integer of 32 or 64 bits, depending on the architecture.

The bitmap is organized into words. For a significant speedup, the index of the first word that contains a zero bit is memorized, so that the search is limited, and more likely to find an empty area to allocate in the first few tries. Allocation moves this index forward if necessary, while deallocation moves it backwards.

There are two types of allocation: aligned and unaligned. There is a public method for each of them, and they hide the size of specific functions implemented by the allocator.

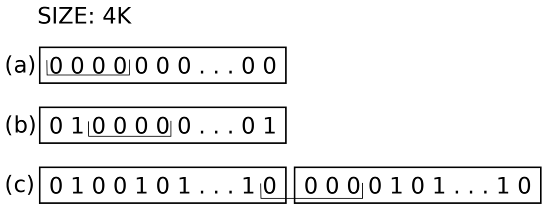

SIZE: 4K
(a) 0 0 0 0 0 0 0 . . . 0 0
(b) 0 1 0 0 0 0 0 . . . 0 1
(c) 0 1 0 0 1 0 1 . . . 1 0 | 0 0 0 0 1 0 1 . . . 1 0

Figure 7. Word-sized allocation

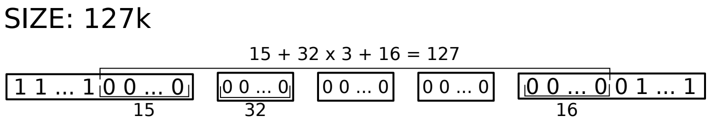


Figure 8. Large allocation

5.1 Unaligned Allocation

Unaligned allocation is done through one of three functions: one for one chunk, one for allocations that could be fully completed with bits from one word, and one for sizes larger than that.

The function for one chunk uses *find first set* to locate an empty chunk. However, *find first set* identifies the first active bit, so the word must be negated first. Because the index of the first word to contain an empty chunk is known, this algorithm does not need to iterate through the bitmap. Because this is the most allocated size, speed is very important and lack of loops helps achieve this performance. After locating the index of the first empty chunk, the algorithm marks the bit as occupied by using a bitmask.

The function for sizes under 32 or 64 chunks is the second most commonly used and it has an advantage over a more general function: it only needs to check one word or the border of two adjacent words. There is no need for it to search more than two words at once since an allocation does not stretch over more than one word. This reduces the number of possible branches, making branch prediction more efficient. There are three possibilities for allocating an area: the current word is equal to zero (see (a) in Figure 7), which means that allocation can be done starting with the first bit; the word is not zero, but has a number of consecutive zero bits equal to or larger than the number of chunks required (see (b) in Figure 7); or there are enough zeroes at the end of the current word and the beginning of the next word to fit the required size (see (c) in Figure 7). After these cases are tried, if an area large enough can be allocated, the corresponding bits are set using a mask, and the starting address is returned. If not, the algorithm continues, testing the next available word, if there is one.

The function for sizes exceeding the number of bits in a word is rarely used. It has only one possible case (shown in Figure 8): it tries to acquire the trailing bits of the word currently tested, and then all the bits from the following words, until it either reaches a word that is not equal to zero, it acquired enough space or it reaches the end of the memory reserved for allocation. If it reaches a word that is not zero without acquiring enough memory for allocation, it acquires the leading zeroes of this word as well. If it is still not enough space, it restarts the algorithm using this last word as the start word.

5.2 Aligned Allocation

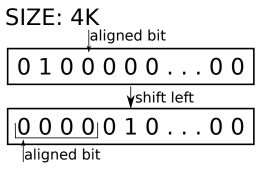


Figure 9. Aligned allocation case

Aligned allocations have an extra requirement besides size: the starting address must be aligned to a certain value, i.e., the starting address must be divisible by the alignment. This is required usually because of hardware limitations, as certain registers might not have enough bits to store the whole address and need to consider the last few bits as 0.

Aligned allocations test if memory areas starting at bit indexes that provide the required alignment are empty. The first bit index to test is generated from the first address following the start of the empty memory that is properly aligned. The subsequent bit indexes to test are generated by adding the number of chunks of alignment to the bit index to be tested.

The allocations that would fit entirely inside one word must test two cases: if the tested bit is 0 and there are no 1 bits between it and the end of the word, then the number of zero bits following it plus the number of leading zero bits of next word must be greater or equal than the number of chunks that needs to be allocated (similar to (c) in Figure 7); otherwise, if it is 0, but there is at least a 1 bit between it and the end of the word, the 0 bits between the tested bit and the first 1 bit get counted and must be greater or equal than the number of chunks that need to be allocated (shown in Figure 9). Otherwise, the next aligned area is checked.

The large area allocation only checks the case where the tested bit is followed only by zero bits. It tests the same bit indexes as the previous algorithm, but it counts bits from more words, the same way the function for large unaligned allocation does (Figure 8), but starting at the bit index corresponding to an aligned location.

5.3 Free

The free function has two parameters, address and size, which can be easily transformed to bit index and number of bits. These two pieces of information transform the problem of freeing an allocated area into a trivial problem, with $O(n)$ complexity, where n is the number of words that must be modified. In practice, n is at most 2, since the microkernel only allocates one area large enough to stretch over more than two words and never frees it.

6 OTHER MEMORY USAGE OPTIMIZATIONS

The allocator was not the only problem regarding memory usage that the microkernel had. The new allocator did not fix internal fragmentation for small objects or reduce number of structures allocated. However, these problems were addressed separately, leading to less memory usage overall.

6.1 Smallpools

The previous and current kernel allocators use 1 kB sized chunks by default as allocation units and all areas allocated will be 1 kB, or a multiple of 1 kB. However, not all objects allocated in the kernel are at least 1 kB, which means that unless another mechanism is used, some memory will go to waste, resulting in internal fragmentation.

The kernel has a mechanism named smallpools to employ in these cases, but it was not used for all the objects smaller than 1 kB. After identifying these objects, they were moved to the smallpools mechanism, which provides other benefits besides better usage of memory.

Smallpools are caches of objects of a certain size and, like the general purpose allocator, use a bitmap to keep track of their allocation status. However, each pool of objects can only allocate objects of a certain size. This reduces search time. A pool contains one or more pages of memory, each with a bitmap and linked together in a list.

Allocation entails flipping the first set bit, while freeing requires identifying the page from which the object came from, identifying the index of the bit and then flipping the bit.

It is a simple, yet efficient mechanism, somewhat similar to SLAB, with the exception that objects are not pre-initialized, and metadata is simpler.

A case where smallpools made a large difference is the allocation of level 2 page table entries. This structure is 128 bytes long and is allocated very often. During boot and running of the test program, this structure had a peak of 1 438 instances allocated at once. If normal allocation was to be used, this would mean 1 438 kB of space used. However, using smallpools, 196 kB were used to store all these instances. The microkernel has 12 MB reserved for allocations, making the 1 242 kB saved 11.24 % of the total memory available for allocation.

6.2 On-Demand Allocation

After analysing what each data structure does, some objects were discovered to be allocated but not used. More specifically, the context for the floating point unit (named VFP on ARM CPUs), which most threads do not use, was allocated at thread creation. Since not all threads are allowed to use the coprocessor, when threads are created they are marked if they are able to use the VFP. However, space is no longer allocated for the coprocessor context, allocation being postponed until an exception arises and the function responsible for handling the exception is called. This function allocates memory if it was not previously allocated and the thread is allowed to use the floating point unit.

As an example, during Linux boot and while running the test program (see Section 7.2.2 for information on the test program) only seven threads allocated the VFP, which is negligible compared to all the threads created during this time. On a more complex setup this percentage would be higher, but the majority of programs only use integer arithmetic. Assuming 100 threads, and 20 % of them utilize the FPU, 8 kB would be needed to store the context, if smallpools are used. If all threads allocated the FPU context, 28 kB would be needed to store it, which results in 20 kB of wasted memory. This is not much, but on real systems the number of threads can be in the thousands, resulting in a larger amount of unused, allocated memory.

However, space is not the only gain, but also cycles spent allocating these contexts would be wasted, slowing down thread creation.

7 ANALYSIS OF THE BITMAP-BASED ALLOCATOR

Testing and performance evaluation are important for proving the allocator is functioning well and is both efficient and fast. The platform for most of the tests and benchmarks was a development board called PandaBoard, based on a Texas Instruments OMAP4430 system on a chip.

Because the allocator is critical to the stability of the kernel, its functionality must be tested and proved to be correct and without bugs. The allocator must be able to use all the memory reserved, not overlap any areas and offer a properly aligned memory area when it is required. Failing to meet these criteria can cause system instability, possibly even leading to random, hard to debug crashes and makes the system unusable for any real life scenario, where stability is one of the biggest priorities.

In the course of this section we present various testing scenarios for our memory allocation algorithm. We did not perform any comparative computational experiments with list scheduling algorithms [14]. We tried to see what were the best current memory allocation algorithms used in operating systems and how they behave in a particular environment, i.e. a microkernel-based system.

7.1 Testing

7.1.1 ktest

The **ktest** provides a number of tests that allocate and use a large number of kernel structures. It does so in a controlled environment and provides a feedback for the tests that fail. This makes it a really useful regression testing tool [15]. Failed tests indicate problems in the microkernel's functionality and it is easier to trace their cause than in a real world scenario.

7.1.2 Unit Testing the Allocator

The ktest only provides partial coverage for microkernel usage, and thus it does not show all the possible problems which the allocator might have. Taking the allocator out of the microkernel's context and running allocate and free operations in a controlled environment could reveal how it behaves in corner cases and under a multitude of other circumstances.

The unit test is a program that runs under a GNU/Linux operating system. The allocator code was copied from the microkernel, and minor modifications were made to aid testing, such as making asserts in the allocator throw exceptions and removing some of the debugging code that was not useful.

The first series of tests check the assertions in the allocator, to prove that they are correct and work when they have to. The assertions are meant to help programmers find mistakes in their code or show errors in the allocator's functionality.

Another series of tests allocate as much memory as possible from the memory pool given to the allocator, to prove that all memory is available for allocation and the allocator does not over allocate. The size of the allocations varied from one chunk to 64 chunks, to cover the whole range of functions specified in Section 5.

The last test made a large number of random allocations and frees, keeping track of which chunks were allocated and which were not, to test possible overlaps in allocations, bad frees or aligned allocations which were not aligned properly.

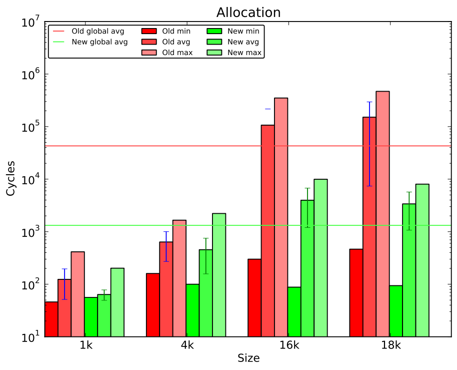
These tests helped to discover and fix a number of bugs, some of which occurred under very specific circumstances.

7.1.3 Booting and Using Linux and Android

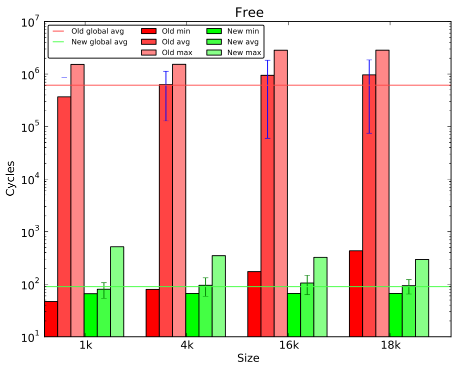
The previous tests were indicative of functionality under an artificial set of circumstances, that proved the algorithm correct in terms of its functionality as designed, but not if the design is correct. Several tests were done under Linux and Android to prove that the system works in a real life scenario as well. We used a minimal filesystem from `Angstrom` distribution for Linux test-case and the `Ice Cream Sandwich` version for Android. In both cases the `3.0.8` version of the kernel was used.

Booting and running several programs under Linux proved that the allocator does not break any functionality. We used the Linux Test Project [16] benchmark suite to validate that Linux system calls function correctly. The results yielded no differences between a Linux system running natively on the development board and the one running on the L4-based setup.

Android proved a similar point while booting up. Installing and using applications under Android worked without problems.



a) allocation



b) deallocation

Figure 10. ktest benchmark overview

7.2 Performance Evaluation

Evaluating the allocator can be done in a vacuum, but it is not necessarily representative. This is why the evaluation of the bitmap-based allocator was done against the previous linked-list allocator. This shows the improvements of the new allocator and served to point out where improvements could still be made during the process of optimization.

The data for these plots was collected using a trace buffer mechanism provided by the L4 microkernel. The metrics collected were number of cycles, the size for which that number of cycles was recorded, and the type of operation (allocation or free). These were chosen because they provide a complete image of performance behaviour, including cache misses and any other possible cause of performance loss. We did not include the number of cycles needed to acquire the spinlocks that protect the metadata, because lock acquisition is non-deterministic, but also because it is not representative of algorithm performance. Additionally we omitted the time to zero the allocated area since it is constant across both algorithms.

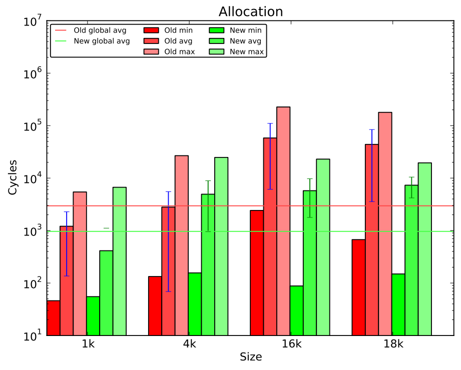
7.2.1 ktest

ktest is a synthetic benchmark, as it allocates a large number of objects, not necessarily representative of how a real life system might behave. However it does show how a system might behave under stress.

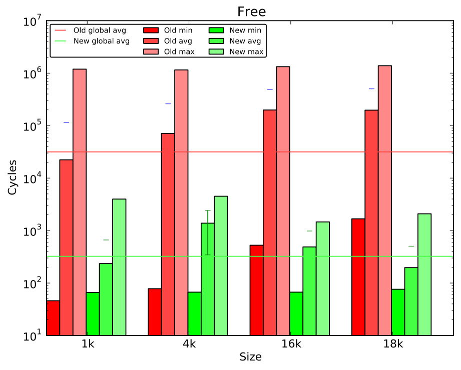
Figure 10 a) shows that under the previous allocator, performance would decrease dramatically with allocation size. The new allocator also suffers from this, but the decrease is not as substantial. Overall, the new allocator behaves better under all circumstances, and the average in number of cycles is 32 times smaller.

Figure 10 b) shows what was expected: with the old allocator freeing could be very costly, while under the bitmap-based allocator it is basically constant time, regardless of size. The new allocator is 6 852 times faster on average, which is a significant improvement.

On average, considering both allocation and deallocation, the new allocator is 465 times faster than the previous one, under these circumstances.



a) allocation



b) deallocation

Figure 11. Linux benchmark overview

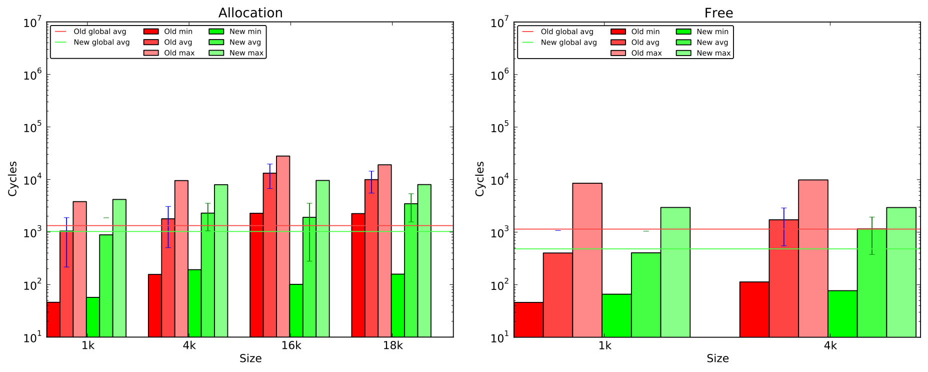
7.2.2 Linux

Under Linux, data was gathered from boot and from running a program that forks, creates threads, allocates memory and opens files. This is meant to emulate how a user might interact with the system, but in a short time.

Figure 11 a) shows that unlike ktest, the differences are less accentuated, although they still exist. The plot looks very similar to the previous plot, and probably, under a more stressful test, the averages would become even closer. Average allocation time is only 3 times better.

Figure 11 b) is, again, comparable to what is seen under ktest, although the performance of the new allocator is slightly decreased. This may be because ktest did all the deallocations in a row, preserving cache locality better. Average deallocation is about 98 times faster.

On average, considering both allocation and deallocation, the new allocator is 26 times faster than the previous one, under these circumstances.



a) allocation b) deallocation

Figure 12. Android benchmark overview

7.2.3 Android

Under Android, data was gathered from the boot process, running a game and browsing the Internet. Since these tests were run by a user, less data was collected and it is not identical between the two cases. However, this is still useful data and can provide interesting information, because the averages should remain more or less the same, if the pattern of usage continues.

Both Figure 12 a) and Figure 12 b) show that the difference between the two allocators is not as impressive as it was with Linux or ktest. Android only has one process in the foreground, at any one time, and a small number of background processes, which makes stressing the allocators difficult. Especially allocating big objects seems to be a very rare process, as Android was built to be used on a system with limited resources and be battery efficient.

On average, considering both allocation and deallocation, the new allocator is only 1.5 times faster than the previous one, under these circumstances.

7.2.4 Fragmentation

Running Linux with a program that spawns threads and programs which allocate and open files, has shown a maximum of 29 % external fragmentation. However, during ktest, external fragmentation was 49 %. This is not ideal, but it is expected of first-fit allocators to cause external fragmentation.

Since most objects allocated are either 1 or 4 kB in size, external fragmentation should not pose a large problem.

Internal fragmentation is possible, but through **smallpools** (see Section 6.1) it is eliminated almost entirely. It might be possible to improve fragmentation by adding some heuristics to not stop at the first possible solution, but find a few, and choose one that is a better fit.

In Bonwick's paper on the SLAB allocator [1] there is a table with total fragmentation for allocators in multiple kernels that were available at the time. For some kernel allocators, total fragmentation does reach nearly 45 %, making 45 % of memory inaccessible.

It is difficult to compute exactly what the total wasted memory in the L4 microkernel is, but memory allocator metadata provides a negligible overhead and the only real waste arises from space that remains unoccupied in smallpools, which is at most one byte less that the structures allocated from each pool. Right now, the structure allocated predominantly is 128 bytes in size, other small objects having significantly lower concurrent instance counts. If 128 byte structures are used as a measure, only 3 % of space is lost per page, which is less than SLAB, (between 11 % and 14 % total fragmentation). It is entirely possible that there are pages from which only a few objects are allocated, making 3 % the minimum and the maximum dependent on the order of allocations and deallocations, with the worst case where there are multiple pages, each with only one object. This is however a very unlikely scenario.

8 CONCLUSION

The goal of this paper was to present the steps involved in improving memory usage in an L4-based microkernel kernel. By reducing memory usage through the use of smallpools, removing unnecessary allocations and improving the allocator this goal was achieved.

Using the smallpools mechanism reduces the memory usage and makes costly allocations (from the general allocator) more rare. This translates to better internal and external fragmentation and better performance.

The new allocator is faster than the previous one, at the cost of a small memory overhead. The new allocator performs better under fragmentation, and has a smaller average and maximum time for allocation and deallocation, as seen in the previous section, where under Linux it performed significantly better.

Better allocation time can be very important in automated tasks, such as HTTP daemons, internet routing or databases, leading to a better throughput and lower latency. It can be beneficial for a user experience as well, even if a user is not able to quantify or even observe the difference.

Reducing the number of allocations for certain structures further decreases latency and memory usage, making thread creation faster and removing additional overhead for threads that do not need the resources.

Removing extraneous safety code from the allocator could further improve performance. This code is useful for development, but not for a stable, tested system. Reducing the size of certain structures could reduce memory usage and even allow for them to be moved to the smallpools mechanism.

It might be possible to modify the smallpolls mechanism, make it more general purpose, and use it to create caches of objects, considering that objects are of well known sizes. This would reduce the number of allocations and deallocations that have to go through the general purpose allocator, and could allow increasing chunk size to a page, improving performance whilst maintaining minimal internal fragmentation. This would make the allocator more similar to SLAB.

The smallpools mechanism might be improved by adding heuristics, to reduce the number of page allocations and deallocations.

Memory zeroization is done in the most straightforward way, each word being written separately, with a store instruction. It could be improved through DMA transfers or other architecture-specific methods.

Acknowledgment

The work has been funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

REFERENCES

[1] Bonwick, J.: The Slab Allocator: An Object-Caching Kernel Memory Allocator. In USENIX Summer, 1994, pp. 87–98.

[2] L4 microkernel family. `http://en.wikipedia.org/wiki/L4_microkernel_family` [Online; accessed 2 July 2013].

[3] Liedtke, J.: On Microkernel Construction. `http://l4ka.org/publications/`, December 1995.

[4] Knowlton, K. C.: A Fast Storage Allocator. Communication of the ACM, Vol. 8, 1965, No. 10, pp. 623–624, doi: 10.1145/365628.365655.

[5] Knuth, D. E.: The Art of Computer Programming. Volume 1: Fundamental Algorithms. 3rd Edition. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[6] Brodal, G. S.—Demaine, E. D.—Munro, J. I.: Fast Allocation and Deallocation with an Improved Buddy System. Acta Informatica, Vol. 41, 2005, No. 4, pp. 273–291, doi: 10.1007/s00236-004-0159-6.

[7] Slab Linux Implementation. `http://lxr.cpsc.ucalgary.ca/lxr/#linux+v3.9/mm/slab.c` [Online; accessed 2 July 2013].

[8] Rosenberg, D.: A Heap of Trouble: Breaking the Linux Kernel Slob Allocator. `http://vsecurity.com/download/papers/slob-exploitation.pdf`, January 2012 [Online; accessed 25 June 2013].

[9] Slob Linux Implementation.

[10] Slub Linux Implementation. `http://lxr.cpsc.ucalgary.ca/lxr/#linux+v3.9/mm/slub.c` [Online; accessed 2 July 2013].

[11] Corbet, J.: The Slub Allocator. `http://lwn.net/Articles/229984/`, April 2007 [Online; accessed 2 July 2013].

[12] OKL4 Microkernel – Reference Manual. `http://wiki.ok-labs.com/downloads/release-3.0/okl4-ref-manual-3.0.pdf` [Online; accessed 2 July 2013].

[13] L4Ka::Pistachio Microkernel. `http://www.l4ka.org/65.php` [Online; accessed 2 July 2013].

[14] Adam, T. L.—Chandy, K. M.—Dickson, J. R.: A Comparison of List Schedules for Parallel Processing Systems. Communication of the ACM, Vol. 17, 1974, No. 12, pp. 685–690.

[15] Buhaiu, A.—Moraru, C.—Cojocar, L.—Priescu, V.: Microkernel Virtualization Support for Multiprocessor Embedded System.

[16] Linux Test Project. `https://github.com/linux-test-project/ltp/wiki` [Online; accessed 10 July 2013]. `http://lxr.cpsc.ucalgary.ca/lxr/#linux+v3.9/mm/slob.c` [Online; accessed 2 July 2013].

Petre Eftime is a Master's student at University Politehnica of Bucharest. He is doing research on embedded operating systems and virtualization, supported by VirtualMetrix, Inc. His interests include distributed systems and IoT.

Lucian Mogoşanu is a Ph.D. student and Teaching Assistant at University Politehnica of Bucharest. He is also doing research on operating systems security at VirtualMetrix, Inc., his main focus is the usage of formal methods to verify security and safety properties of system software. Other interests include virtualization technology, programming languages and compilers.

Mihai Carabaş is Teaching Assistant and a Ph.D. student at University Politehnica of Bucharest and a researcher at VirtualMetrix, Inc. His main research interest is studying and developing mechanisms to improve virtualization in operating systems. Other interests include system administration, high performance computing and advanced network protocols.

Răzvan Deaconescu is Assistant Professor at University Politehnica of Bucharest. He is fond of teaching and doing research in operating systems-related topics, with interest in systems security. His recent activities have targeted mobile operating system security, virtualization and embedded systems.

Laura Gheorghe is Assistant Professor and Researcher at University Politehnica of Bucharest. She is passionate about security and operating systems. Her current research focused on malware detection in Android operating system is supported by VirtualMetrix, Inc.

Valentin Gabriel Voiculescu has been involved with embedded devices for the last nine years, with interest ranging from low level to middleware and high level OS in FPGA's, 8–16 bit microcontrollers, SoC's in modern smartphones and tablets. He has been with VirtualMetrix, Inc., a startup focused on reducing power in select embedded devices (lately in flagship smartphone and tablets), by fine grained performance management of the system.