A Dynamic Virtual Memory Management under Real-Time Constraints

Martin Böhnert
Department of Computer Science
University of Freiburg, Germany
Email: boehnert@informatik.uni-freiburg.de

Christoph Scholl
Department of Computer Science
University of Freiburg, Germany
Email: scholl@informatik.uni-freiburg.de

***Abstract*—In this work we describe a new memory management concept which allows the use of both virtual and dynamic memory management at the same time in the context of real-time systems. For a fixed size of the virtual address space, the operations of memory allocation, de-allocation and access have a constant complexity. Therefore our approach is highly suited for real-time environments with hard deadlines. We employ efficient data-structures to yield runtimes that are close to traditional static memory management concepts, and – at the same time – provide the user with the full flexibility of both virtual and dynamic memory management. Our approach is based on novel operating system components and a novel *real-time aware virtual memory management unit* (RTMMU) in hardware.**

Our experimental results demonstrate the applicability of our concept and compare its performance with a classical approach. The results show that our new approach does not only provide constant-time memory management operations, but is also able to reduce the memory footprint to a large extent.

I. Introduction

Complex embedded microsystems which contain microcontrollers and processors for controlling and data processing usually depend on standard functions for their communication with the hardware / software interface. To be able to develop application software in a convenient and efficient manner these standard functions should be realized by system calls of a real-time operating system (i.e. by pre-designed components). For embedded microsystems with limited resources this approach is only possible if it does not imply a large overhead leading to a violation of efficiency requirements (wrt. energy and memory consumption) or to a violation of real-time constraints.

The focus of this work lies on memory management under real-time constraints. Since large memories are opposed to the need for miniaturization and low power consumption, memory resources of embedded microsystems are often strictly limited. The goals of our memory management methods are the following:

1) Dynamic memory allocation [1], [2] should be supported, e.g., for being able to react to dynamic changes in the system environment.
2) Available memory resources should be used as efficiently as possible. Wasting memory (e.g. by fragmentation, delayed release of unused memory) should be avoided as much as possible.
3) Virtual memory management (VMM) [3], [4], [5] should be supported. VMM provides independent address spaces for different tasks, is able to realize memory protection, and helps to prevent fragmentation of the physical memory when memory is allocated and de-allocated dynamically.
4) In order to fulfill real-time requirements, basic operations like memory allocation, access and release should be as fast as possible; the complexity of these operations should be data independent and limited by (small) upper bounds.

Both dynamic and virtual memory management are widely used concepts in desktop and server environments. Despite the benefits of these concepts (e.g. making software development much more convenient), their use in the context of embedded real-time systems has been limited so far. Existing virtual memory management systems have runtimes that depend on the size of the requested memory. For this reason embedded real-time systems refrain from using virtual memory, because these mechanisms are viewed as a main obstacle to tight estimates of worst-case execution times [6] and thus to an efficient scheduling of processes with deadlines – see Sect. II-B for a more detailed discussion. In many cases, fixed (and contiguous) memory regions are assigned to each process and there is a static memory management strategy for each process [7], [8], [9], [10], [11], [12]. Clearly, this typically leads to a large overestimation of the memory resources which are really needed.

Here we present an approach which makes dynamic and virtual memory management available for real-time systems. Our approach introduces algorithms and data structures for memory management in the operating system as well as our *real-time aware virtual memory management unit* (RTMMU) supporting the operating system.

Our proposed RTMMU allows, for the first time, to combine dynamic and virtual memory management with constant runtime bounds that do not depend on the memory usage profile. For this purpose we introduce an efficient data structure named *Virtual Regions Tree* for constant-time management of typical memory operations including de-allocations. Unlike standard approaches, our approach does not rely on delayed memory de-allocations to achieve its runtime efficiency. In contrast, every de-allocation is immediately executed, leading to minimized memory footprints. Insofar our methods pave the way for tight estimations of worst-case execution times and worst-case memory consumption.

Our experimental results clearly confirm the theoretical expectations concerning the applicability of our approach in the real-time domain.

The rest of the paper is organized as follows: In Sect. II we give a brief introduction to dynamic memory management, virtual memory management and basic data structures used in our memory management approach. Sect. III reviews related work. Our approach is presented in Sect. IV followed by the

experimental results in Sect. V. Finally, Sect. VI summarizes the results of our paper.

II. Preliminaries

A. Dynamic Memory Management

Dynamic memory management [1], [2] is a concept, where processes can allocate and de-allocate memory at runtime. When processes use data structures whose memory consumption changes over time, then it is highly desirable to have an operating system providing routines for dynamic memory allocation and de-allocation. Fixed memory assignments to processes are usually based on overestimates for the memory consumption, waste memory and make software development more complex than needed. Memory de-allocation can be performed explicitly or implicitly by garbage collection. In the context of our paper we restrict ourselves to *explicit* de-allocations.

B. Virtual Memory Management

In a virtual memory management (VMM) environment [3], [4], [5] each process may *"virtually"* operate on the complete address space. Moreover, VMM helps to prevent fragmentation of physical memory when memory is allocated and de-allocated dynamically, and it facilitates memory protection mechanisms. Operating system and memory management hardware work together to map virtual memory pages to the physical memory at runtime as soon as they are actually needed by a process [5]. First of all, the physical memory and the virtual address space are partitioned into pages. These pages are numbered and the numbers are part of the memory address. A single memory cell within a page can be addressed with an offset which forms the rest of a physical memory address.

Now, a connection between the virtual and the physical memory has to be established. This is achieved with a page table, which simply is a look-up table, indexed with the virtual page number. The result of the look up is the corresponding physical page number. Combined with the offset (which is not part of the translation process) the virtual address is transformed into the physical one. The translation is done in hardware by the memory management unit (MMU). Hierarchical and inverted page tables are alternatives to the *direct* page tables as described here [5], [13]. We have generalized our concepts to the case of hierarchical page tables as well. However, in order to keep the presentation as clear as possible, we confine ourselves to direct page tables in this paper.

When a memory page is accessed for the first time, the MMU reports a *page fault* (i.e. it reports that the page is not yet present in the physical memory) and the page is mapped to a page in the physical memory. When a page of memory is de-allocated, then the mapping to the physical memory may be undone and the physical memory may be returned to the system. In general purpose systems with secondary memory, the return of physical memory to the system is usually not performed explicitly, but implicitly via page replacement strategies. Unused memory pages are usually not returned to the system before the system runs out of free physical memory. Although this approach is certainly efficient on the average, the runtimes increase when the memory load is high. For this reason (and because most embedded systems do not have secondary memory at all) embedded real-time systems usually refrain from using virtual memory with implicit page replacement strategies.

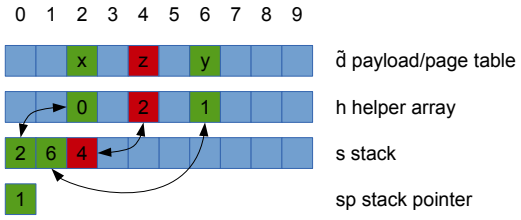


Fig. 1: Principle of the initialization-free array

If implicit page replacements are not used, memory has to be de-allocated explicitly when using VMM in a dynamic memory management context. Then de-allocation with un-mapping of memory pages leads to runtimes which are linear in the number of deallocated memory pages. In order to avoid this overhead, processes often do not return memory pages to the system, but use them internally for subsequent memory allocations. Nevertheless, from time to time processes shrink their memory footprint (e.g. when memory blocks at the end of the heap are released) to avoid too much memory overhead [14]. When shrinking the memory footprint, un-mapping of virtual memory pages still leads to runtimes which are linear in the number of de-allocated pages. Such an approach has two undesired effects: First, runtimes of memory de-allocations become highly unpredictable, since from time to time they trigger a large runtime overhead when shrinking the memory footprint. However, predictability is of uttermost importance for hard real-time systems. Second, such a delayed memory de-allocation may lead to a large memory overhead.

Due to these difficulties most embedded real-time systems do not use VMM at all. Instead, in many cases, fixed (and contiguous) memory regions are assigned to each process and there is a static memory management strategy for each process. Clearly, this typically leads to a large overestimation of the memory resources which are really needed.

C. Data Structures

In the following we briefly review two existing data structures which we use in our implementation.

1) Initialization-free Arrays: In our work the idea of *initialization-free arrays* [15] is mainly used to be able to invalidate whole blocks of page mappings in constant time when memory is de-allocated. Initialization-free arrays are usually employed when the (linear) cost for array initialization should be avoided. To implement initialization-free arrays, as in Fig. 1, the contents of the original array $d[0 \ldots n-1]$ are represented by means of a data array $\tilde{d}[0 \ldots n-1]$, a helper array $h[0 \ldots n-1]$ and a stack. The stack stores at most n elements and can thus be implemented by another array $s[0 \ldots n-1]$ and a stack pointer sp. Suppose that the default initialization value for $d[i]$ is -1. The following invariant is maintained:

$d[i] = c$, if $\tilde{d}[i] = c$ and $0 \leq h[i] \leq sp$ and $s[h[i]] = i$, otherwise $d[i] = -1$.

'Reading from $d[i]$' is performed using this invariant. For initialization of d we set $sp = -1$ with the effect $d[i] = -1$ $\forall 0 \leq i < n$. Writing a value c to $d[i]$ works as follows: If $0 \leq h[i] \leq sp$ and $s[h[i]] = i$, we set $\tilde{d}[i] := c$, otherwise we set $sp := sp+1$, $h[i] := sp$, $s[sp] := i$ and $\tilde{d}[i] := c$. It is easy to see that this maintains the invariant and that all operations

read, *write* and *initialize* can be performed in constant time. The illustration in Fig. 1 shows an initialization-free array with $d(2) = x$, $d(6) = y$, and $d(i) = -1$ else. Although it holds ("by accident") that $s[h[4]] = 4$, this connection is not secured by the stack, since $sp = 1$. Thus $d[4] = -1$ and not z.

2) Layered Trees: The concept of layered trees is used to manage intervals of virtual addresses which correspond to allocated or free regions of virtual memory. Layered trees solve the union-split-find problem [15], [16], [17]. They can be used to handle dynamic partitions of sets $\{0, \ldots, N-1\}$ into disjoint intervals $I_1, \ldots, I_k$. In our application the integers represent virtual page numbers. The operation *union* replaces two neighboring intervals I_i and I_{i+1} by $I_i \cup I_{i+1}$. The operation *split* splits an interval $I_i = [a, c]$ at position $b \in [a, c]$ into $[a, b-1]$ and $[b, c]$ and replaces I_i by these new intervals. For $b \in \{0, \ldots, N-1\}$ the operation *find* determines the unique interval $I_i = [a, c]$ with $b \in I_i$. [1] For layered trees the runtime complexity of the operations *union*, *split* and *find* is in $O(\log \log N)$.

III. Related Work

In [18] an approach for real-time systems is presented which organizes virtual memory regions into intervals and uses a linear mapping of the virtual memory regions to physical memory. This approach suffers from an increasing fragmentation of the physical memory, although it uses virtual memory. Moreover, the runtime complexity is not independent from the number of entries in the page table.

Other approaches use virtual memory management only for components without hard real-time constraints (such as the virtual ROM concept for mobile phone applications [19]).

Bennett and Audsley [20] introduce a specialized virtual memory management for hard real-time tasks by so-called modular page tables, but they do not solve the problem of explicit de-allocations in constant time.

For real-time systems without virtual memory management the *two-level segregated fit* (TLSF) allocator [21], [22] is a widely known and accepted implementation of dynamic memory management. TLSF combines fast response times for allocation and release of memory with a low memory consumption (by restricting the memory fragmentation). All operations (allocation, release, splitting and coalescing adjacent free memory areas) can be performed with a number of machine operations bounded by a constant, if one assumes that certain common bit operations are available in the instruction set. Otherwise the number of machine operations is logarithmic in the address width of the underlying architecture, i.e., it is in $O(\log \log N)$, if N is the number of addressable memory cells. However, the original TLSF concept for embedded real-time systems [21], [22] only works directly on the physical memory. Later on, TLSF has been used in desktop and server environments as well and has been extended for virtual memory usage. However, in these extensions TLSF uses the `mmap`/`sbrk` system calls of POSIX compatible operating systems to allocate and de-allocate virtual memory pages. These calls are used without consideration of the runtimes of `nmap` and `sbrk`. Potential paging situations where memory pages are stored on hard drive are also neglected. Thus, TLSF loses its hard real-time capabilities in this context and runtimes of allocation and de-allocation are hard to predict as long as the state of the underlying operating system is not known. In this paper we present – for the first time – a concept which provides the underlying operating system and hardware support for an dynamic memory allocator in a way that hard real-time capabilities of allocators such as TLSF can be retained in a virtual memory management context.

[1] In our application, it is sufficient to find the left bound a of I_i.

IV. Dynamic Real-time aware VMM

Our goal of a dynamic real-time aware virtual memory management is achieved by sophisticated data structures used in operating system routines supported by the proposed new real-time aware management unit (RTMMU). All operations (allocation, de-allocation, access) are bounded by small and strict upper runtime limits. These bounds are not related to the size of the memory regions, but only to the fixed size of the virtual address space in the given memory architecture.

In the following we assume that a virtual address space A_i with a size of n memory cells is assigned to process p_i. When p_i is started, the operating system provides a new virtual address space, copies the program to a predefined location and starts the program. If the program needs additional memory at runtime, then the operating system dynamically allocates new virtual pages for process p_i. To save as much memory as possible *physical* memory corresponding to the virtual memory addresses is only allocated by the page fault mechanism, if there are accesses to the allocated virtual pages.

We further assume that the operating system provides the following system calls for allocation and release of virtual memory: The call `malloc`(A_i, x) reserves an arbitrary number of x contiguous virtual addresses inside the virtual address space A_i. Upon success, the call returns a virtual address a; the virtual memory at addresses a to $a+(x-1)$ is reserved by this call. The call `free`(A_i, a) releases the memory formerly allocated with start address a.

Our memory management is designed for a hierarchical use. The higher level systems calls `malloc`(A_i, x) and `free`(A_i, a) make use of corresponding procedures at a lower level which allocate and release whole *pages* of virtual memory. This higher level can be served by an allocator like binary buddy [2] or TLSF. In this paper we focus on the lower level, i.e., memory management on the basis of virtual pages.

A. Memory Management Based on Virtual Pages

For our memory management, we consider allocation and releases of whole memory *pages*. As already mentioned above, we would like to make the runtimes for allocation and de-allocation independent from the size of the allocated / de-allocated memory when we combine dynamic and virtual memory management for embedded real-time systems.

Allocations do not form any problem here, if the page fault technique is used and a mapping is not done before a virtual memory page is accessed for the first time.

De-allocation could be supported by page replacement strategies in a memory hierarchy using secondary memory. In our approach we consciously do without secondary memory, because delays caused by swapping pages to the hard disk are difficult to estimate. Thus, swapping pages is not desirable for embedded real-time systems. Instead, such systems need to rely on the assumption that the available amount of main memory is sufficient to meet the memory demands of the running processes. Upper bounds for memory consumption may be computed statically (similarly to worst-case execution time computations [6]). As we will see later, such analyses are

simplified to a great extent by our eager and explicit memory de-allocation mechanisms.

For de-allocation of p memory pages in a memory-efficient system *without* secondary memory a straightforward option which makes de-allocated memory available to other processes would be as follows: Upon de-allocation explicitly return the free physical memory pages to the system and mark the corresponding mappings in the page table as invalid. However, this needs a runtime linear in p for de-allocation which we would like to avoid.

Our system implements system calls similar to the `mmap`(A_i, a, x) and `sbrk`(x) functions from standard POSIX operating systems. Here, A_i denotes the virtual address space, where x contiguous virtual pages should be mapped at address a. In contrast to `sbrk`(x) which shrinks the heap of a process by releasing x pages at the end, our system is able to release arbitrary regions of contiguous memory pages and it does this in constant time.

The new system call `mapVRH`(A_i, a) marks a new virtual memory region in virtual address space A_i at virtual address (respectively page) a. For all virtual pages in this memory region (which reaches up to the next following virtual memory region) an unmapping together with a release of the corresponding physical memory pages can be performed using one single operation in constant time. This is done with the `unmapRegion`(A_i, a) system call. For every first access to a virtual page, `mapPage`(A_i, a) needs to be called. Usually this is done as part of a page fault handling routine. `mapPage`(A_i, b) allocates a new physical memory page, maps it to the virtual page b in address space A_i and links it to the corresponding memory region established by `mapVRH`(A_i, a).

We would like to emphasize that our new system calls provide an efficient opportunity to unmap virtual memory (and free the corresponding physical memory) not only at the end of the virtual memory space used for a process, but at any arbitrary position. Our experimental results show that this property is essential to maintain a small memory footprint.

Like in a standard environment, the higher level memory allocator communicates the memory requests on the basis of pages to the operating system with system calls, while the according processes directly access their corresponding virtual address space using hardware support (for an illustration see Fig. 2).

In order to efficiently realize our system supporting dynamic virtual memory management for real-time systems we introduce the data structures as illustrated in Fig. 3. We derive the need for these data structures step by step from the requirements of the different operations of our memory-efficient real-time approach.

1) Releasing virtual pages: To support a constant-time operation `unmapRegion`(A_i, a) releasing x memory pages with numbers a to $a + (x - 1)$ (which are the pages up to the beginning of the next region of memory pages) we have to fulfill two requirements:

1) We have to invalidate all mappings from virtual pages $a, \ldots, a+(x-1)$ to physical pages in constant time.
2) We have to release all physical memory pages which have been mapped to virtual addresses b with $a \leq b \leq a+(x-1)$ by some call `mapPage`(A_i, b). This operation should be performed in constant time, too.

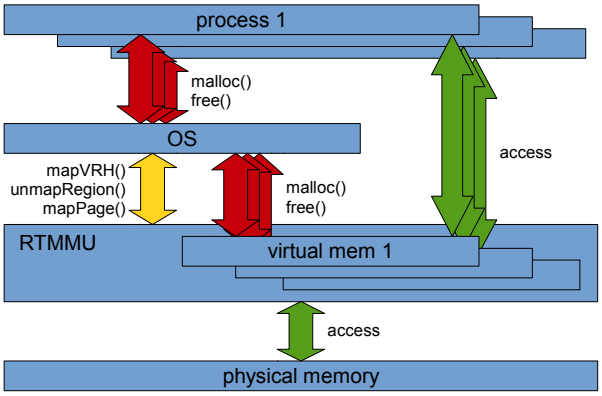


Fig. 2: Dataflow between processes, OS/memory management, RTMMU and physical memory. The red arrows indicate control flows for function calls like `malloc` and `free`, while the green ones indicate memory accesses.

To fulfill Requirement 1 we customized the idea of initialization-free arrays (see Sect. II-C1). Each region of memory previously allocated by a call `mapVRH`(A_i, a) has its own initialization-free array consisting of a data array $\tilde{d}_a[0 \ldots x-1]$, a helper array $h_a[0 \ldots x-1]$, a stack array $s_a[0 \ldots x-1]$, and a stack pointer sp_a. x denotes the number of memory pages from the beginning of this memory region to the next region. Invalidating all mappings from virtual addresses $a, \ldots, a+(x-1)$ to physical pages at once can be performed by setting $sp_a := -1$. It is easy to see that the initialization-free arrays for *all* regions of allocated memory in a virtual address space A_i with N pages numbered by $0, \ldots, N-1$ can be combined into global arrays $\tilde{d}[0 \ldots N-1]$ (for data arrays), $h[0 \ldots N-1]$ (for helper arrays), $s[0 \ldots N-1]$ (for stacks), and $sp[0 \ldots N-1]$ (for stack pointers). These global arrays replace the original page table $pt[0 \ldots N-1]$. For each allocated virtual memory region with virtual addresses $a, \ldots, a+(x-1)$ we have a stack pointer $sp[a]$ and sub-arrays $\tilde{d}[a \ldots a+(x-1)]$, $h[a \ldots a+(x-1)]$, and $s[a \ldots a+(x-1)]$. For $a \leq b \leq a+(x-1)$ we maintain the invariant "$pt[b] = c$, if $\tilde{d}[b] = c$ and $a \leq h[b] \leq sp[a]$ and $s[h[b]] = b$, otherwise $pt[b] = -1$" just as described in Sect. II-C1. Invalidating all mappings from this region can then be performed by $sp[a] := a - 1$, e.g. (we always interpret the stack beginning at index a as empty, if $sp[a] < a$).

The resulting data structure (illustrated in Fig. 4) is called *Pages Validity Array (PVA)* in our context.

To be able to fulfill Requirement 2, after each mapping of a physical page to a virtual page b with $a \leq b \leq a+(x-1)$, the corresponding virtual page has to be included into a linked list $l[a]$ of physical pages. A pointer to this linked list is stored at address a. Then the operation `unmapRegion`(A_i, a) needs only constant time to add this linked list to the list of free physical memory pages.

2) Access to Virtual Pages: When we make an access to a virtual memory page b in a region $a, \ldots, a+(x-1)$, we have to check first, whether this virtual page is already mapped to physical memory. To decide this question using our Pages Validity Array we need information on the beginning a of the memory region. If we have the page number a, then we can access the stack pointer $sp[a]$, check whether $a \leq h[b] \leq sp[a]$, $s[h[b]] = b$, and $\tilde{d}[b] \neq -1$. If all these conditions are true,

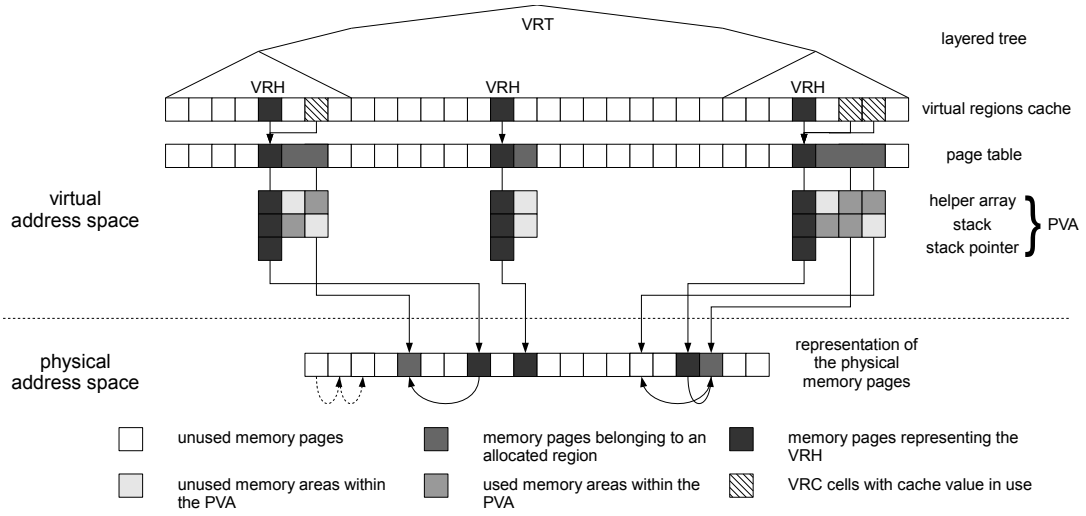


Fig. 3: Data structures for memory management based on virtual pages

then $\tilde{d}[b]$ provides a valid page mapping, otherwise we have observed a page fault.

It is important to note that we have to be able to compute for each page number b the beginning a of the memory region $a, \ldots, a+(x-1)$ with $a \leq b \leq a+(x-1)$. For this purpose, we make use of layered trees as described in Sect. II-C2. The set of virtual pages with adresses $A_i = \{0, \ldots, N-1\}$ is partitioned into disjoint intervals of virtual page numbers corresponding to contiguous allocated or free regions of virtual pages. For a page number b the operation *find* provides exactly the left bound a of the interval $a, \ldots, a+(x-1)$ with $a \leq b \leq a+(x-1)$. The complexity of the *find* operation in layered trees is $O(\log \log N)$ (see also Sect. IV-A5).

a) First access to a virtual page: When we access a virtual page b for the first time, we obtain the information that the virtual page is not yet mapped to physical memory. Then the page fault mechanism assigns a free physical page to this virtual page, updates the initialization-free array accordingly, and collects the mapped page in the linked list $l[a]$ (in order to prepare the release operation as described above). Note that we need the left interval bound a corresponding to b to identify the head pointer of the linked list, too.

We call the auxiliary data stored at the left bound a of a memory region the *Virtual Region Header (VRH)*. The Virtual Region Header represents an allocated but not necessarily mapped memory region and points to the head of the linked list $l[a]$. Since the VRH represents a memory region, it can also be used to store additional information not immediately needed for memory management, e.g. access rights to the region.

The layered tree, which is able to compute the beginning of a memory region and thus the corresponding VRH, is called *Virtual Regions Tree (VRT)*.

b) Subsequent accesses to a virtual page: After a virtual page b has been accessed for the first time, all subsequent accesses will not lead to a page fault any more as b has now been mapped to a physical memory page. However, if we would use the same method as for the first access, then we would need $O(\log \log N)$ operations again for the *find* operation in the VRT to be able to evaluate the initialization-free array and thus to find the physical memory page mapped to b. $O(\log \log N)$ additional operations are acceptable in the case of page faults, but any delays on regular memory accesses are definitely unacceptable. For that reason we introduce a cache to store the results of find operations in the *VRT*.

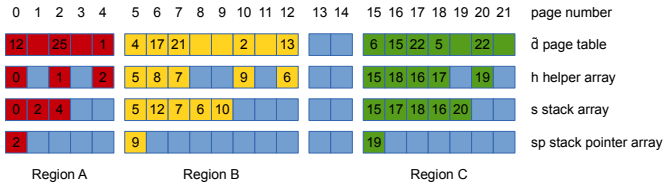


Fig. 4: The generalization of the initialization free array to global arrays for each task – the *PVA*. Regions A, B and C denote allocated and partly mapped memory regions.

This cache $c[0, \ldots, N-1]$ (which is global to the address space) is called *Virtual Regions Cache (VRC)*. When a virtual page b is accessed for the first time and the *find* operation in the VRT results in a, then the page fault handler sets $c[b]$ to a.

During an access to a page b, lookups in the data array $\tilde{d}[b]$, in the VRC $c[b]$, and in the helper array $h[b]$ can be parallelized. If $c[b]$ is valid, then $s[h[b]]$ and $sp[c[b]]$ can be computed in parallel in a second step and the page mapping can be evaluated. If we assume that the VRC c, the helper array h, the stack array s and the stack pointers sp are implemented by fast memory with double speed compared to original page table, then the original memory access times of a traditional page table based system are preserved.[2]

We want to make clear, that the VRC only stores the result of the find operation in the Virtual Regions Tree. In contrast, the translation lookaside buffer (*TLB*) of a standard MMU

[2]If fast memory is not available for that purpose, one could also think of a pipelined application of the two steps described above. Then the latency of memory accesses is delayed by one read cycle, but the speed is preserved.

stores the result of the address translation.[3]

3) Allocating virtual pages: The operation `mapVRH` (A_i, a) performs a *split* operation in the VRT at position a, just as the de-allocation operation performs a *union* operation in the VRT (which has not been mentioned yet in the description above). Then we 'initialize' the initialization-free array by $sp[a] := a - 1$ and we initialize the linked list in the VRH by the empty list.

4) Correctness and Complexity:

Theorem 1. *Our algorithm handles accesses (with and without page faults) and allocation, de-allocation of pages correctly. Allocations, de-allocations and handling of page faults are performed in $O(\log \log N)$ (with N equal to the size of the virtual address space).*[4] *Memory accesses without page fault are performed in constant time.*

The proof of Theorem 1 basically follows from the arguments given above and the correctness resp. complexity of the operations on our data structures (initialization-free arrays and layered trees). There is one subtle point concerning correctness which needs an additional consideration:

As described in Sect. IV-A1 we invalidate all mappings from a region with virtual addresses $a, \ldots, a + (x - 1)$ in constant time by setting $sp[a] := a - 1$. We neither perform assignments in the global arrays $\tilde{d}[0 \ldots N-1]$, $h[0 \ldots N-1]$, and $s[0 \ldots N - 1]$ (which is justified by the concept of initialization-free arrays), nor we perform any assignments in the global cache c (the *Virtual Regions Cache*) storing the results of *find* operations in the *Virtual Regions Tree*. Here the question arises whether we need to initialize (or re-initialize) the cache c in order to avoid problems with old (or uninitialized) cache entries, when the same virtual pages are used again after following allocations of memory regions. Re-initializing every entry in c for the whole memory region which has been released is clearly not suitable, as this would introduce a linear runtime and destroy our real-time properties. Fortunately, we can show that initializations in the cache c are not needed.

Lemma 1. *As long as the stack pointer array sp is initialized when the complete system is started,*[5] *mapping of memory pages can be implemented correctly without any initializations to the Virtual Regions Cache c.*

Proof (Sketch): Consider a page b within a memory region of allocated pages $a, \ldots, a + (x - 1)$ ($a \leq b \leq a + (x - 1)$). Possibly the page b has been part of another memory region which has been allocated and freed before. If b has been accessed after the previous allocation (leading to a valid entry in $c[b]$ at that time) and not accessed after the current allocation, then $c[b]$ may hold an outdated value. We have to prove that even in this case it is possible to detect that a current access to b is the first one after the new allocation, without interference by the old value $c[b]$.

In order to prove this fact we have to consider several cases:

- Case 1: $c[b] > b$. In this case it is clear that $c[b]$ is invalid. Every valid entry $c[b]$ points to the beginning of a memory region $c[b], \ldots, c[b]+(x'-1)$ with $c[b] \leq b$. Thus we can detect that the access to b is the first one after the allocation and the entry $c[b]$ is invalid (in that case due to a lack of initializations to the array c).
- Case 2: $c[b] \leq b$. Here we have to differentiate between two sub-cases:
  Case 2.1: $sp[c[b]] < c[b]$. This inequation indicates that the stack beginning at $c[b]$ is empty (see Sect. IV-A1). Since the array sp is initialized when the system is started (precondition of the lemma) and only changed in a correct way during page faults, page allocations, and page releases, we can rely on the entries in sp. Thus, in case of $c[b] = a$ the stack for region $a, \ldots, a + (x - 1)$ is empty and the access to b is the first one after the allocation (as there has not been any access to the region at all). In case of $c[b] \neq a$, the entry $c[b]$ is outdated. In any of the two cases a page fault handling takes place, the correct position a of the Virtual Regions Header is found in the Virtual Regions Tree during page fault handling and finally $c[b]$ is overwritten by a.
  Case 2.2: $sp[c[b]] \geq c[b]$. With the same argument as in the case before we can rely on the fact that the position $c[b]$ is the beginning of a non-empty stack $s[c[b]], \ldots, s[sp[c[b]]]$. This stack definitely belongs to a memory region $c[b], \ldots, c[b]+(x'-1)$. There may be two cases for the corresponding intervals: $[c[b], c[b] + (x' - 1)] = [a, a + (x - 1)]$ and $c[b] + (x' - 1) < a$.
  Case 2.2.1: $c[b] + (x' - 1) < a$. In this case $c[b]$ is definitely outdated. We have to show to be able to detect that the access to page b is the first one after the allocation. Due to the construction of the initialization-free array we know that $s[i] \leq c[b] + (x' - 1) < a$ for all $c[b] \leq i \leq sp[c[b]]$. Even if $c[b] \leq h[b] \leq sp[c[b]]$, we have $s[h[b]] < a$ and thus $s[h[b]] \neq b$. So we detect that the mapping at b is not valid, handle the page fault, perform a lookup in the Virtual Regions Tree during page fault handling, and update $c[b]$ to the correct value a.
  Case 2.2.2: $[c[b], c[b] + (x' - 1)] = [a, a + (x - 1)]$. In this case the value in $c[b]$ is correct (either by chance or since there has been an access to page b after the last allocation of the region containing b). The algorithm uses the initialization-free array to check whether the mapping at page b is valid. In every case the value of $c[b]$ remains correct after the access to page b.

■

5) Optimizations and Cost: Of course, for achieving the runtimes given in Theorem 1 we need additional memory for our data structures.

For a more detailed analysis we first consider the VRT which is based on layered trees. In general, layered trees are used to handle dynamic partitions of totally ordered sets. Implementations according to [15], [16] assume these totally ordered sets to be lists of dynamic size. Elements can be added to or erased from these lists, i.e., there are operations *add* and

[3] Of course, if needed, our memory address translation can simply be extended by a translation lookaside buffer for caching address translation results. If TLBs are used in the final memory system, then the WCET (worst-case execution time) analysis has the task to analyze situations when there are definite hits in the TLB just as in the standard analysis for hits in the data and instruction cache [6]. Since the additional use of TLBs is orthogonal to our approach, we omitted them in our exposition.

[4] This is called "quasi-constant" in [21], [22], since N is constant in a given architecture.

[5] Remember that the stack for a region with virtual addresses $a, \ldots, a+(x-1)$ is considered to be empty, if $sp[a] < a$. Thus, in a practical implementation one option for initialization of sp would be to ensure $sp[i] = 0$ at power-up of the memory and to exclude virtual page 0 from further usage.

erase in addition to the operations *union*, *split*, and *find*. This leads to a rather involved data structure where the different nodes of the list and of the layered tree are linked by several pointers (for successor and predecessor relations, links to tree structures at higher layers etc.) as well as several copies of the original list at different hierarchy levels have to be stored. The overall space complexity is in $O(N \cdot \log\log N)$ (where N is the size of the list). In contrast, since the size of the virtual address space is fixed in our work,[6] we have to handle partitions of a fixed set $\{0, \ldots, 2^k - 1\}$ where $N = 2^k$. This allows for rather sophisticated measures to simplify the layered tree and to reduce its size. Due to lack of space we only mention a few optimizations without going into detail:

1) First of all, our layered tree has a static structure.
2) Moreover, we can do without pointers between nodes at different hierarchy levels, since the connections between these nodes are given implicitly in the static structure, i.e., they can be computed based on node numbers.[7]
3) In contrast to [15] we can avoid to copy the representation of the original set $\{0, \ldots, 2^k - 1\}$ for several times at different hierarchy levels and we can combine other nodes in the layered tree into one representative without losing any information.

The sum of these measures leads to a data structure with space complexity $O(N)$ instead of $O(N \cdot \log\log N)$.

Moreover, the constant factors both for space and runtime requirements are reduced by using bit arrays instead of list elements for representing the virtual pages $\{0, \ldots, 2^k - 1\}$, by using bitvector operations (like *least significant one bit*, *most significant one bit*), and by inlining recursive calls.

For instance, for a system with 2^{20} virtual pages, this leads to a memory consumption of $(\frac{2^{20}}{8} + 2^{10} \cdot 4 \cdot 3 + 2^5 \cdot 4 \cdot 3 + 4 \cdot 3)$ Bytes $= 143756$ Bytes $\approx 140KiB$ for the layered tree, whereas a lower bound to the memory consumption of the original implementation [15] (for the case of a complete layered tree, corresponding to the finest possible partition of the set, e.g.) is given by $2^{20} \cdot 5 \cdot 10 \cdot 4$ Bytes $= 200MiB$. This means that we managed to reduce the VRT to a size below the size of the page table for realistic memory architectures (e.g. for the one used in our experiments in Sect. V).

The memory requirement for the Virtual Regions Cache, the Pages Validity Array and the page table altogether exceeds the memory requirement for the usual page table only by a constant factor of 5. Bearing in mind that the memory consumption for the page table is *far* below the size of the managed memory in a realistic memory architecture, the additional effort for our data structures is negligible.

Finally, we would like to emphasize that

1) the runtime $O(\log\log N)$ for *union*, *split* and *find* in the VRT does not depend on the sizes of the released, allocated or accessed regions, but only on the number of memory pages in the system, and
2) it is a small constant given a fixed and realistic memory architecture.

Thus, the runtimes of these operations can be estimated independently from the sizes of the memory regions and can be bounded by small constants.

[6] As the total amount of memory in a system does not change during runtime.

[7] Provided that we additionally pay attention to build the levels in the layered tree in a way that the nodes are distributed to substructures in a uniform way and the numbers of nodes at different levels are powers of two.

B. The Real-Time Memory Management Unit

Up to this point we presented the algorithms which are used in our new dynamic virtual memory management scheme. Here we will briefly introduce the Real-Time Memory Management Unit (*RTMMU*).

The RTMMU is located between the CPU and the memory subsystem. Each memory access of the CPU leads to three parallel memory requests which are initiated by the RTMMU. To this end the RTMMU has three independent bus ports to the memory subsystem. One port first accesses the VRC to get the position of the actual VRH and subsequently accesses the stack pointer in the PVA at the found position. The second port reads the helper array in the PVA, followed by an access to the stack in the PVA at the found position. The third port reads the page table and waits for the other two ports to finish and decide whether the physical memory address is correct. If no page fault is detected, the third port finally makes an access to the translated physical memory address.

If the memory accesses through all three ports can be performed fully in parallel (e.g. using a crossbar switch and dedicated memory modules) and one can guarantee that VRC and PVA are accessible twice as fast as the page table (or if pipelining is used accordingly as mentioned in Sect. IV-A2b), then there is no additional cost for accessing a physical memory address compared to an access using a standard page table approach.

V. Experimental results

A. Experimental setup

To evaluate our approach we built an experimental platform based on an OpenRISC CPU softcore [23] which is run on an Altera Stratix III FPGA development board. We replaced the original Memory Management Unit (MMU) in the OpenRISC core by the proposed RTMMU supporting our approach. The bus system used in this microcontroller system is an instance of the Avalon Bus developed by Altera [24]. Basically the Avalon Bus forms a cross bar switch for all components connected to the bus. This enables a fully parallel access of the RTMMU to different memory components.

The Virtual Regions Cache (VRC) and the stack pointer array of the PVA are located in one discrete DDR2 DRAM chip, the stack array and the helper array of the PVA are located in a second discrete DDR2 DRAM chip. The page table is located in a DDR2 DRAM module on the board. This enables parallel access to the VRC, the helper array, and the page table on the one hand, and to the stack array and the stack pointer array on the other hand (see Sect. IV-B). The discrete DDR2 DRAM chips are operated at double speed compared to the DDR2 DRAM module. By this measure the access times of the original approach can be preserved.

For the experiments we did not integrate the memory management completely into an operating system, rather we provided software implementations of the operations `malloc()` and `free()` by implementing the TLSF allocator with some effective modifications which make use of our explicit deallocation scheme with tremendous effects on the memory footprint of a task. The software parts are located in a static RAM. Memory allocations, read/write accesses and releases are performed for memory located in the DDR2 DRAM module. To be able to collect statistical data we trigger the execution of commands via a network interface, measure execution times using a timer component located in the microcontroller and

communicate the measured results via the network interface. The controller program running on a workstation is able to execute a given trace of de-/allocations and memory accesses.

For an evaluation of our approach, we ran several benchmarks. First we used a synthetic workload which enables an extensive evaluation with a large number of allocations, de-allocations and accesses. In addition to this, we used traces from different real programs.

The synthetic workload consists of about one million randomly chosen actions. Allocations and de-allocations had a probability of 10 percent whereas read and write accesses had a probability of 40 percent each. After the random decision which action will be taken, we either decided the size of the allocation, which allocation has to be deallocated or the address of the access.

We gathered the real world traces with the help of one of the Valgrind tools [25]. The Dynamic Heap Analyzer Tool (DHAT) traces every allocation and de-allocation as well as every access to this allocated heap memory. So we ran different programs together with Valgrind/DHAT and logged all relevant memory actions. Afterwards we used these data and reran the logs using our experimental setup. With this toolchain we are able to execute the memory operations of real programs on our RTMMU hardware without the need of an implementation in a fully featured operation system. The applications we used for our benchmarks are as follows: *avconv* is a Linux command line tool for audio/video-transcoding and we ran it with a short video. *FFT* is a simple fast fourier transformation which runs on a small input vector. *mc* is a program which successively processes several input streams and computes for each input stream and each element in the input stream the number of its occurrences. *sharpSAT* and *antom* are both tools from the satisfiability domain and they are run on SAT solving benchmarks. Finally, we ran a PHP programm which models the training phase of a neural network.

For comparison, we used a second system containing an MMU with a simple traditional page table instead of our RTMMU. To obtain a better traceability of the results, neither our RTMMU nor the standard MMU use a TLB. Since we did not want to obscure the results of the comparison by the use of different allocation schemes, we used TLSF as the higher-level memory allocator here as well. In the second system we use a conventional approach where de-allocation with un-mapping of memory pages needs linear effort in the number of memory pages. As usual, in order to avoid this effort as far as possible, memory pages are not un-mapped and returned to the system with every de-allocation. Instead they are kept for internal re-use in the same process and only returned to the system by a special shrink operation `sbrk()` [14] when the last page of the allocated memory footprint is de-allocated. In the following we denote this system as "conventional MMU" in brief.

B. Results

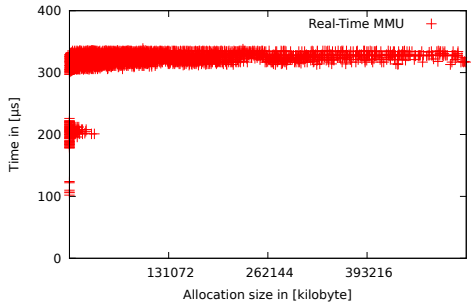
Both systems were run on exactly the same traces. Figures 5, 6, and 7 show the results of the synthetic trace with the same axis scaling for both systems.

Fig. 5a and Fig. 5b show the measured times of roughly 100,000 allocation calls, Fig. 5a for our approach and Fig. 5b for the conventional MMU. The times are ordered by the size of the allocated memory region, i.e., for a single data point the x-value gives the size of the allocated region and the y-value the runtime in μs. Both approaches show similar structures in their runtime behavior. The most frequent situation occurs when a free virtual memory block needs to be split and also a page mapping takes place for a new "boundary tag"[8] needed in the TLSF algorithm [21], [22]. This refers to the upper thick band in the diagrams. The thin, lower band represents the case, when a free block does not need to be split and all necessary pages are already mapped, because this block is surrounded by used ones. Another case, mostly happening with smaller allocation calls, occurs, when a free block needs to be split, but no mapping has to be done. This situation is reflected by the middle band. Concerning the maximum runtimes, a small offset compared to the conventional approach is added for our new scheme due to the operations in the Pages Validity Array and the Virtual Regions Tree.

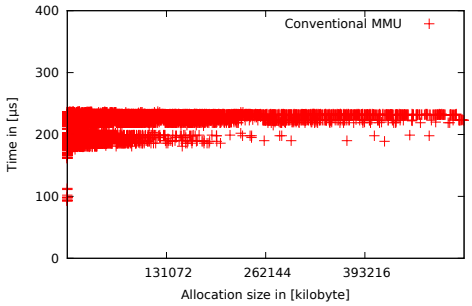
In contrast to allocations, the de-allocations in Fig. 6a and Fig. 6b behave completely different for the two variants. Again, x-values are ordered wrt. the sizes of the de-allocated memory regions, y-values correspond to runtimes in μs. Our memory management scheme shows a runtime behavior which is clearly bounded by a strict upper limit (see Fig. 6a). In the conventional MMU (see Fig. 6b) the runtimes lose their predictability due to the scheme how allocated pages are handed back to the systems. The data points near the x-axis are the cases where an allocation somewhere in the middle of all active allocations is returned to the system. But whenever a block at the end of the heap is deallocated and the memory footprint is shrunk, a linear runtime in relation to its allocation size is introduced. This can be observed from the rising linear band in the middle of Fig. 6b. Further, de-allocation times get even worse in the case where a used allocation at the end of the heap is preceded by free memory, which is not used. In this situation, these free memory pages are also handed back to the system, resulting in an even worse runtime. For an extreme case consider the single upper left data point in Fig. 6b which corresponds to the de-allocation of a small memory region at the end of the heap which triggers a large number of page un-mappings when the memory footprint is shrunk. This behavior heavily depends on the allocator which is used and even if the internals of the allocator are known, is it hard to predict when this will happen. Considering this unpredictability, almost no runtime guarantees can be given. In contrast to this, our approach is well behaved in every situation and gives strict upper bounds which depend on the underlying computer architecture but neither on allocation sizes nor on special situations.

In Fig. 7a and Fig. 7b we compare the memory footprints of the two variants. In this comparison the x-axis represents the time, whenever an allocation or de-allocation takes place, the y-axis represents the number of memory pages mapped into the virtual memory at that time. Starting with Fig. 7b one can basically see three rising curves, each ended with a drop down to almost no mapped memory pages. The rising arcs are formed by the fact that only de-allocations at the end of the heap result in memory handed back to the system. As long as the last allocation is not freed, the arc tends to grow because of page faults in the allocated regions. Ultimately this would lead to a plateau in the chart where a steady state of mapped memory is reached. The random trace eventually frees almost every used memory which leads to the high drops. Interestingly, the single upper left data point in Fig. 6b mentioned above (with an excessively large runtime for a small de-allocation) corresponds to the de-allocation causing the highest drop in Fig. 7b. In contrast to this we can see

[8] A small data structure situated at the beginning of an allocation.

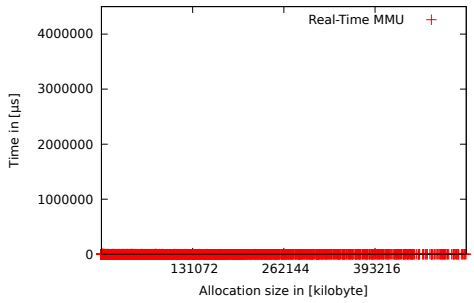


(a) Allocation runtimes RTMMU

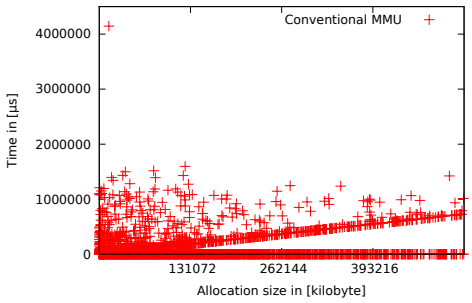


(b) Allocation runtimes conventional MMU

Fig. 5: Allocation runtimes

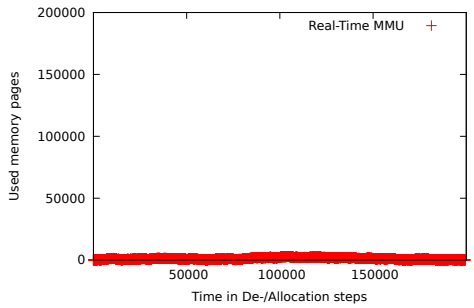


(a) De-allocation runtimes RTMMU

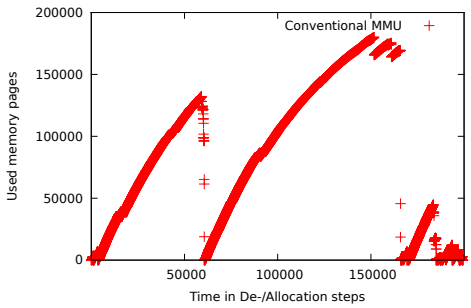


(b) De-allocation runtimes conventional MMU

Fig. 6: De-allocation runtimes



(a) Memory footprint RTMMU



(b) Memory footprint conventional MMU

Fig. 7: Memory footprint

TABLE I: Maximum runtimes for allocations and de-allocations, maximum memory footprint during the runs.

	RTMMU			*Traditional*		
Benchmark	Allocation	Deallocation	Memory Footprint	Allocation	Deallocation	Memory Footprint
Synthetic	339	945	2 971	237	4 145 462	180 132
avconv (A/V-Transcoding)	327	547	621	233	2 263	781
FFT	326	524	6 263	233	94 305	6 263
mc	336	747	206	232	2 899 292	96 035
sharpSAT	327	702	4 522	233	73 454	4 718
antom	325	668	2 942	233	37 854	3 176
PHP (neural net)	322	437	476	233	1 423 306	476

considerable smaller memory footprints in Fig. 7a. We can clearly see that our strategy to hand virtual memory pages back as fast as we can pays off and that we only need a fraction of the memory used in the conventional MMU.

Next we consider the traces of real programs in Table I. The allocation and deallocation columns show the maximum runtime in μs observed for these operations in the different benchmark traces (including the synthetic trace). The memory footprint gives the maximum number of memory pages mapped into the virtual memory of the task. The results for the traces produced by real programs basically confirm our observations made for the synthetic trace: The allocation times both in our and the conventional approach are strictly bounded, with a small runtime penalty for our approach. The differences wrt. the maximal memory footprint are still apparent, but not always as extreme as in the synthetic benchmark (apart from the benchmark mc). Again, the strongest effect can be observed with respect to the de-allocation times.

VI. Conclusions and Future Work

We presented a new memory management unit which provides for the first time both dynamic and virtual memory management for real-time systems. We achieve runtimes which do not depend on the size of allocated and de-allocated memory regions for memory allocations, de-allocations, and page faults. In addition, for simple read/write accesses the application of the virtual regions cache and a parallel hardware access to data structures in small fast memories enables the new real-time aware approach to act as fast as the traditional page table based MMU. Especially for de-allocations we reached the goal to reduce the runtimes from a linear behavior to a small bound by utilizing the VRT and the PVA.

Another key advantage of our approach using virtual memory is its increased memory efficiency: Physical memory is only used (mapped), if it is really needed for read or write accesses, and unused physical memory is immediately returned to the system.

We implemented the memory management algorithms in software and designed a novel real-time aware virtual memory management unit (RTMMU) in hardware to evaluate the approach. Our experimental results confirm the theoretical expectations concerning the applicability of our approach in the real-time domain.

Using non-delayed memory de-allocations our approach paves the way for tight estimations of worst-case execution times *and* worst-case memory consumption for systems using dynamic and virtual memory. For the future we plan to profit from this feature in the context of real-time scheduling with hard memory bounds.

Acknowledgment

This work has partly been supported by the German Research Foundation (DFG) within the Research Training Group 1103.

The author would also like to thank Matthias Sauer and Thorsten Zitterell for their constructive discussions and the time they sacrificed to support this work.

References

[1] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management, International Workshop IWMM 95*, ser. Lecture Notes in Computer Science, H. G. Baker, Ed., vol. 986, 1995, pp. 1–116.

[2] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. Addison-Wesley Longman, 2005, vol. 1 Fundamental algorithms, ch. 2.5. Dynamic Storage Allocation, pp. 435–456.

[3] J. Fotheringham, "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store," *Commun. ACM*, vol. 4, no. 10, pp. 435–436, 1961.

[4] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, 1970.

[5] B. Jacob and T. Mudge, "Virtual memory: Issues of implementation," *Computer*, vol. 31, no. 6, pp. 33–43, 1998.

[6] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[7] L. Ramachandran, D. Gajski, and V. Chaiyakul, "An algorithm for array variable clustering," in *EDAC-ETC-EUROASIC*, 1994, pp. 262–266.

[8] E. de Greef, F. Catthoor, and H. D. Man, "Array placement for storage size reduction in embedded multimedia systems," in *ASAP*, 1997, pp. 66–75.

[9] P. Grun, F. Balasa, and N. D. Dutt, "Memory size estimation for multimedia applications," in *CODES*, 1998, pp. 145–149.

[10] Y. Zhao and S. Malik, "Exact memory size estimation for array computations without loop unrolling," in *DAC*, 1999, pp. 811–816.

[11] J. Zhu, "Static memory allocation by pointer analysis and coloring," in *DATE*, 2001, pp. 785–790.

[12] J. Ramanujam, J. Hong, M. T. Kandemir, A. Narayan, and A. Agarwal, "Estimating and reducing the memory requirements of signal processing codes for embedded systems," *IEEE Transactions on Signal Processing*, vol. 54, no. 1, pp. 286–294, 2006.

[13] W. Stallings, *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011.

[14] W. Mauerer, *Professional Linux Kernel Architecture*. Wiley Publishing, Inc., 2008, ch. 4.9 Managing the Heap, pp. 327–329.

[15] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1984, vol. 1.

[16] P. van Emde Boas, R. Kaas, and E. Zijlstra, "Design and implementation of an efficient priority queue," *Mathematical Systems Theory*, vol. 10, pp. 99–127, 1977.

[17] K. Mehlhorn, S. Näher, and H. Alt, "A lower bound on the complexity of the union-split-find problem," *SIAM J. Comput.*, vol. 17, no. 6, pp. 1093–1102, 1988.

[18] X. Zhou and P. Petrov, "The interval page table: virtual memory support in real-time and memory-constrained embedded systems," in *SBCCI '07: Proceedings of the 20th annual conference on Integrated circuits and systems design*. New York, NY, USA: ACM, 2007, pp. 294–299.

[19] H. Kim, J. In, D. Ham, S. Yoon, and D. Shin, "Virtual-ROM: A new demand paging component for rtos and nand flash memory based mobile devices," in *Proc. of International Symposium on Computer and Information Sciences*, ser. LNCS, vol. 4263. Springer, 2006, pp. 677–686.

[20] M. D. Bennett and N. C. Audsley, "Predictable and efficient virtual addressing for safety-critical real-time systems," in *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2001, p. 183.

[21] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–86.

[22] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings, "Implementation of a constant-time dynamic storage allocator," *Software: Practice and Experience*, vol. 38, pp. 995–1026, August 2008.

[23] *OpenRISC project*, OpenCores.org, http://opencores.org/or1k/Main_Page.

[24] *Avalon Interface*, 11st ed., Altera, May 2011.

[25] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 89–100.