Shared-Page Management for Improving the Temporal Isolation of Memory Reservations in Resource Kernels

Hyoseung Kim and Ragunathan (Raj) Rajkumar
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, USA
hyoseung@cmu.edu, raj@ece.cmu.edu

***Abstract*—Memory reservation provides real-time applications with guaranteed memory access to a specified amount of physical memory. However, previous work on memory reservation primarily focused on private pages, and did not pay attention to shared pages, which are widely used in current operating systems. With previous schemes, a real-time application may experience unexpected timing delays from other applications through shared pages that are shared by another process, even though the application has enough free pages in its reservation. In this paper, we describe problems with shared pages in real-time applications, and propose a shared-page management mechanism to enhance the temporal isolation of memory reservations in resource kernels that use resource reservation. The proposed mechanism consists of two techniques, Shared-Page Conservation (SPC) and Shared-Page Eviction Lock (SPEL), each of which prevents timing penalties caused by the seemingly arbitrary eviction of shared pages. The mechanism can manage shared data for inter-process communication and shared libraries, as well as pages shared by the kernel's copy-on-write technique and file caches. We have implemented and evaluated our schemes on the Linux/RK platform, but it can be applied to other operating systems with paged virtual memory.**

***Keywords*-Real-Time Memory Management, Shared Page, Memory Reservation, Temporal Isolation**

I. Introduction

Virtual memory can play an essential role in real-time and cyber-physical systems. Since the systems interact with external environmental factors, such as unexpected workload surges and severe weather changes, they will get increasingly large and complex and will need flexible memory management to meet their varying memory requirements. For example, Boss [20], the autonomous vehicle that won the 2007 DARPA Urban Challenge, executes multi-level perception and prediction algorithms along with running software for data fusion from tens of sensors equipped within the vehicle. In this system, the data calculation time and memory requirement of each software component can vary according to external road conditions. We therefore need to provide not only temporally predictable but also flexible memory management mechanisms for real-time and cyber-physical systems.

Much research has been conducted on real-time memory management. Specifically, memory reservation [5] provides guaranteed and predictable memory access performance to real-time applications. It allows an application to reserve a portion of the total system memory for its exclusive use, and these reserved memory pages can never be stolen by other applications even if the system is under memory pressure. If the application needs more memory than the size of its reservation, pages are swapped within its own reservation area to satisfy subsequent requests. This makes the primary performance effect of the application's memory access be contained within its reservation set, thereby providing temporal isolation to each application. Moreover, as memory reservation provides an abstraction for a logically partitioned physical memory area, other real-time memory techniques such as page replacement [7], prefetching [2], and dynamic memory allocation [11] can run on top of memory reservation, and each memory reservation set could be allowed to choose the best-suited memory technique and parameters for its associated applications.

Table I
Number of shared pages used by each process in Linux

Application	Private (code/data)	Shared (code/data)	Total	Shared/Total (%)
Rhythmbox *MP3 Player*	13722 (3999/9723)	4958 (4635/323)	18680	26.5
Minitube *YouTube Client*	11664 (5128/6536)	10375 (5140/5235)	22039	47.1
Sound-Rec *GNOME Voice Recorder*	3849 (862/2987)	2898 (2571/327)	6747	42.9
Sound-Rec x 2 (#1)	2990 (2/2988)	3758 (3431/327)	6748	55.7
Sound-Rec x 2 (#2)	2978 (2/2976)	3765 (3426/339)	6743	55.8
MPlayer *Video Player*	3866 (1160/2706)	777 (771/6)	4643	16.7
MPlayer x 2 (#1)	2800 (0/2800)	1937 (1931/6)	4737	40.9
MPlayer x 2 (#2)	2804 (0/2804)	1937 (1931/6)	4741	40.8

In previous approaches including memory reservation [5] and other similar schemes [8][1], shared pages are implicitly assumed to be an insignificant factor in the real-time performance of applications, so the impact of shared pages has not been appropriately studied. However, recent operating systems with paged virtual memory widely use shared pages for inter-process communication (IPC) and shared libraries. Besides, some operating systems like Linux aggressively share memory pages for efficiency, e.g., the copy-on-write technique for data pages and code page sharing for multiple

instances of a single application. Table I presents private and shared page usage of multimedia applications in an Ubuntu 10.04 Linux system. The numbers in the table are the number of pages residing in physical memory, and the size of each page is 4096 bytes. An interesting aspect to observe is that the number of shared pages is changed when we execute the same application twice. For example, in the fourth row of the table, the number of shared pages of a single instance of the *Sound-Rec* is 2898, but the number of shared pages is increased to 3758 when we launch another instance. Similarly, in the case of *MPlayer*, the percentage of shared pages to total pages is changed from 16.7 to 40.9%, after we run the second instance of *MPlayer*. The usage of shared pages may vary according to system load and application characteristics, but it is fairly obvious that shared pages are not negligible in current operating systems.

In this paper, we first show how shared pages can affect the performance of real-time applications across the boundaries of memory reservation sets. Existing techniques cannot avoid temporal interference across reservation sets, if they share memory pages and any of them swaps out the shared pages. Then, we propose a shared-page management scheme for memory reservation to provide temporally predictable and isolated memory access to real-time applications using shared pages. We propose two techniques, Shared-Page Conservation and Shared-Page Eviction Lock, each of which avoids any unexpected timing penalties from shared pages, thus promising improved temporal isolation to memory reservation. Our techniques reside in the same layer as memory reservation so that they do not restrict existing page replacement policies, and can run with other higher-level real-time memory techniques. We have implemented and evaluated our techniques by extending the Linux/RK platform [14][18]. The experimental results show the practicality and the effectiveness of our approach.

II. Related Work and Background

We discuss related work on real-time memory management and describe the background on the memory reservation [5], on which our approach is based.

A. Related Work

Eswaran et al. [5] proposed the memory reservation technique for real-time applications. It ensures that a specified number of physical pages is reserved for an application, enforcing the maximum memory usage of the application and protecting the reserved pages from other applications. In [5], the authors also provided compatibility with the Resource Kernel abstraction [18] and studied energy-aware memory management by using the reservation approach. Kato et al. [8], inspired by the memory reservation abstraction, enhanced it for interactive real-time applications by introducing a common buffer page pool for memory reservations. Hand [6] focused on an application-level self-paging technique within the Nemesis operating system, which is similar to the memory reservation approach. All these previous schemes, however, do not consider how to deal with shared pages. Only the Linux Memory Cgroup [1] manages shared pages by using the *first-touch* approach, but it is insufficient to provide real-time applications with guaranteed access to shared pages, because the shared pages can be arbitrarily evicted by an application that first accessed the pages.

The memory locking mechanism [9][19] has been used in some real-time systems in which locked pages cannot be swapped out by others. However, page locking by itself does not provide any isolation between applications, because locking may incur pages of other applications to be swapped out when the system is under memory pressure. Besides, locking all pages of real-time applications can easily lead to wasted memory resources.

Instead of statically locking all pages, the insertion of paging hints based on compile-time application analysis [15][3][10] was proposed to efficiently control page lock and release at run-time. Hardy et al. [7] and Puaut el al. [16] suggested the explicit specification of page loading and eviction points in real-time applications. These approaches assume that their target application is temporally independent of other co-running applications, which is not true when shared pages are used.

Real-time memory management techniques in the context of dynamic memory allocation have also been studied in [11][17][12][13]. They aim to meet changing memory demands of user-level applications under real-time timing constraints. Belogolov et al. [2] suggested scheduling-assisted prefetching to improve the performance of demand paging. Our work does not conflict with these approaches, and we expect that the memory reservation architecture can embrace them to provide sophisticated and customized memory management for each application.

B. Memory Reservation

As we mentioned in the previous section, memory reservation aims to provide predictable and enforced access to memory resources. With memory reservation, a real-time application can specify its memory demands, and after memory is reserved, the requested memory pages are exclusively used by the application. Other applications cannot take the reserved pages away; thus, the application with the memory reservation can expect constant performance regardless of the existence of co-running applications. Memory reservation also ensures that the potential abuse of memory by any application does not affect the performance of other applications. In the case that an application needs more memory than its reservation, pages are swapped within its own reservation. Currently, memory reservation is implemented as a part of the Linux/RK platform [14].

The parameters for memory reservation include the size of memory required by the application in bytes and the reservation type. The reservation size is translated to the number of pages depending on the page size supported by the target system architecture. The memory reservation type

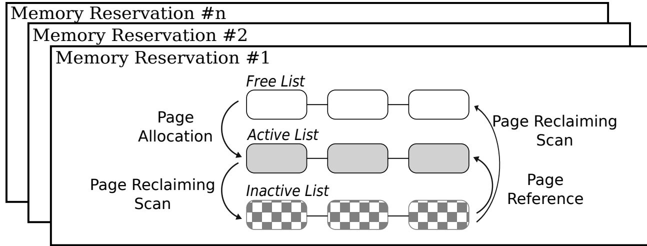


Figure 1. Page lists and movement in memory reservation

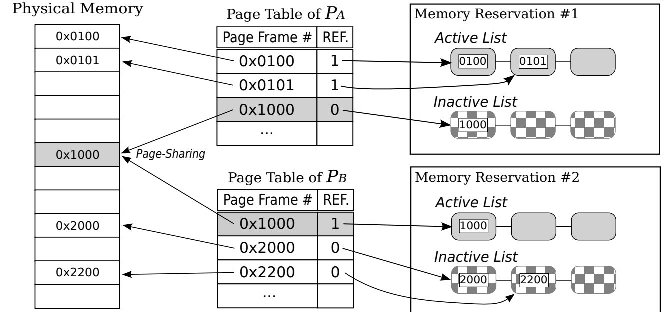


Figure 2. Page sharing among multiple memory reservations

specifies the handling policy when every free page in the reservation is exhausted. The type can be chosen as *hard* or *firm*. With a hard reservation, an application bound to it is not allowed to use pages beyond the reserved size of memory, even though unreserved free pages are available in the system. This provides a strict sand-boxing mechanism for enforcing memory usage. With a firm reservation, an application is allowed to use free pages from an unreserved area when the reserved pages are exhausted, but these borrowed pages will be retrieved in the case of a system's memory pressure. In this paper, we only consider the hard reservation type for simplicity.

Memory reservation can adopt any page replacement policies, but the current implementation approximates the Least-Recently-Used (LRU) algorithm by using the FIFO with Second-Chance algorithm. Figure 1 shows the active, inactive, and free lists in a memory reservation, following the design of the Mach page replacement implementation [4]. Each memory reservation has these lists and moves pages according to page references. The memory reservation allocates pages from the free list when a page fault has occurred. A page scan is triggered when a free page is needed and the free list is empty. Page reference information is obtained by examining the Page Table Entries (PTE) of the applications belonging to the reservation.

III. Problems with Shared Pages

In this section, we describe three problem scenarios caused by shared pages. With existing memory partitioning and reservation approaches, shared pages can be arbitrarily swapped out to disk even though they are being used by other real-time applications.

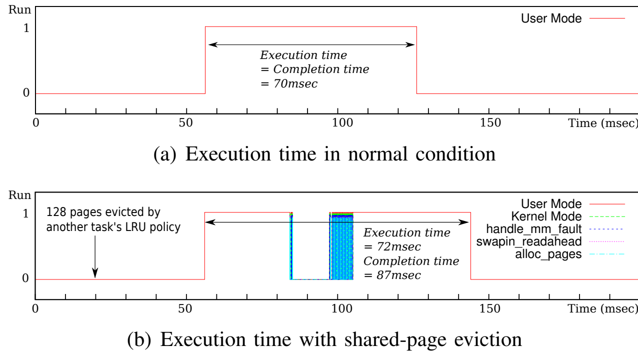
A. Shared Pages under Global Memory Management

The first case occurs when the memory reservation or partitioning technique ignores shared pages and lets the global memory management policy manage them. Most current operating systems adopt the LRU policy for page replacement, and the shared pages are handled along with other pages by the LRU policy. Under memory pressure, the kernel scans pages in use and selects victim pages by observing whether the page has been recently accessed. Since the global paging policy is unaware of the importance of each page, the shared pages being used by a real-time application are handled in the same way as other (shared and private) pages; hence, they can be evicted if they are relatively less recently accessed than other pages. The real-time application therefore can experience timing penalties when it tries to access the evicted shared pages, though the application has enough reserved free pages in its reservation. The global paging policy may lock every shared page to avoid this problem. However, it will unnecessarily consume excessive memory resources, because the size of shared pages in a system is not trivial as shown in Table I, and many of them may not be actually used by real-time applications. Therefore, shared pages for real-time applications should be managed using other schemes.

B. Shared Pages in Memory Reservation

The second case happens when shared pages are under the control of the existing memory reservation approach, which handles shared pages in the same way as private pages. Figure 2 shows an example of shared pages with multiple memory reservations. There are two applications, P_A and P_B, having their own reservation set, and they share two physical pages; hence, each of the two corresponding page table entries indicates the same physical memory address to access the shared page. According to the current memory reservation scheme, the shared pages belong to each memory reservation set which in turn has the right to control the shared pages in the same way as private pages. In this example, the reference bit of P_A's page table entry for the shared page 0x1000 is zero, because P_A did not access the page recently. From P_A's perspective, it is reasonable to evict the page 0x1000 under its memory pressure. Conversely, P_B frequently accesses the page 0x1000, so the reference bit of the P_B's page table entry is one. Here, if P_A evicts the shared page 0x1000, P_B will experience a timing penalty that does not appear when it runs alone in the system. If P_A does not evict this shared page, P_A needs to evict another page instead and different temporal results may result. In effect, the shared page weakens the temporal isolation capability of memory reservation.

The timing penalty caused by the unexpected eviction of shared pages consists of page fault exception handling delay and page swapping-in delay. The page fault handler performs free page allocation and page table mapping for an application, and it leads to extended kernel time for the application. Page swapping-in needs I/O operations, which



(a) Execution time in normal condition

(b) Execution time with shared-page eviction

Figure 3. Self-suspension caused by arbitrary eviction of shared pages

may produce unwanted jitter. Also, the application needs to be blocked until the completion of an I/O operation, inducing self-suspension delay. The important thing here is that this self-suspension is not inherent in the application itself. Even if the real-time application has sufficient memory reservation and runs with high CPU scheduling priority, it may experience self-suspension due to co-running applications at any time. Figure 3 shows a simple test result illustrating the self-suspension due to shared-page eviction. We created 10MB of shared memory area using the *shmem* system call and a test application accessing the shared memory area periodically. We ran the application with 20MB of memory reservation and a real-time scheduling priority on Linux/RK. Figure 3(a) shows the per-period execution time of the test application. The y-axis of the graph represents the execution state of the task; 1 means that the task is currently running; 0 means another task is being scheduled. During execution, neither page faults nor context switching occurred, so the completion time was same as the execution time. While this application was running, we ran the same application again with a normal priority. We intentionally gave the *second* instance 10MB of memory reservation, which was not enough to keep all the shared pages and its code pages. Figure 3(b) shows the execution time of the first instance of the application, when 128 shared pages were evicted by the second instance. The LRU policy of the second instance kept selecting pages in the shared memory area as victims to page out because its private code and data pages are more frequently accessed than each of the shared pages. Unlike the case of running alone, both the execution time and the completion time were noticeably extended, even though the number of evicted pages was much less than the number of shared pages. This simple test implies that the impact of self-suspension by shared pages can become more severe in complex real-time systems. Moreover, a real-time application may not meet its deadlines due to unpredictable self-suspension delay.

C. Shared Pages in Linux Memory Cgroup

The Linux kernel provides the memory *cgroup* function to isolate and limit the memory usage of applications. It is conceptually similar to memory reservation, but it handles shared pages using the "first-touch" approach. In a memory cgroup, every shared page is accounted for and managed by a single owner group which first accessed the page. This means that other applications also can use the pre-owned page, but the single owner has a right to control the page. This approach has two drawbacks. First, when the owner is under memory pressure, shared pages can be evicted if they have not been recently used by the owner. This will cause timing penalties to other groups accessing the shared pages. Second, this approach makes it hard to accurately account for the memory usage of applications. If an application uses some pages owned by other applications, its memory usage may seem smaller than when it runs alone.

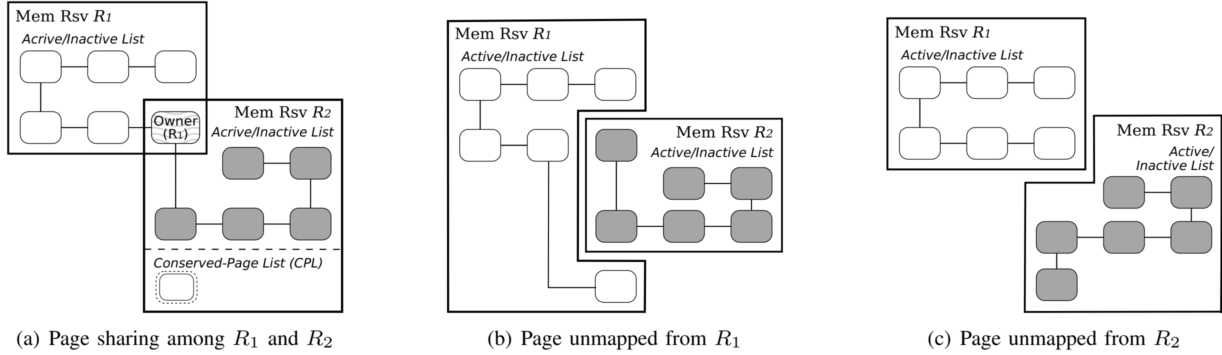
IV. Shared-Page Management for Temporal Isolation

Our proposed shared-page management schemes include Shared-Page Conservation (SPC) and Shared-Page Eviction Lock (SPEL). Both techniques prevent the self-suspension and the page fault handling delays caused by the shared-page eviction of co-running applications. We also describe a combined use of SPC and SPEL, which reduces the total memory usage of the system.

A. Shared-Page Conservation (SPC)

A real-time application using shared pages should ideally have guaranteed memory access timing delays, regardless of the existence of co-running applications. The key idea behind SPC to solve this problem is that it does not really swap out pages when an application selects a shared page as a swap-out victim. This technique merely unmaps the shared page from the application's page table and maintains the page in physical memory. From the application's point of view, one page is removed from its reservation; hence, it can get a new free page. The other applications sharing the page are not affected, because the page remains in physical memory and no changes are made to their page tables. Under the control of this technique, only private pages are evicted to secondary storage.

A new data structure called the *Conserved-Page List (CPL)* is used to manage these unmapped shared pages. When an application running with its memory reservation accesses a page for the first time, a page fault exception occurs, and the kernel inserts a virtual/physical address mapping into the application's page table. If a free physical page allocation is required, the memory reservation uses one free page from its free page list. Otherwise, if the target page already resides in physical memory, the memory reservation includes the page in its local LRU list. In the case of the existing memory reservation approach, it returns one free page from the reservation's free list to the global memory manager, to maintain the constant maximum effective size of the reservation. SPC, however, does not return the free page when the newly added page is being used by another reservation. Instead, the technique conserves the free page in the Conserved-Page List (CPL). The conserved page is not allowed to be used for page allocation, so the



(a) Page sharing among R_1 and R_2

(b) Page unmapped from R_1

(c) Page unmapped from R_2

Figure 4. Page sharing and unmapping with Shared-Page Conservation (SPC)

effective memory size of the reservation does not change. The minimum number of pages in a CPL can be zero, when the reservation owns every shared page it uses, or the reservation does not use any shared pages. The maximum possible number of pages in a CPL is equal to the size of the memory reservation; this is the case when the reservation uses only the shared pages which are owned by others. Figure 4(a) shows an example of page sharing of two memory reservations under SPC. Each reservation has six pages, and they share one page. Since the shared page was initially allocated from R_1, the reservation R_2 stores one free page in its CPL.

As mentioned before, when an application selects a shared page for a swap-out victim, the proposed technique unmaps the shared page from the application's page table and maintains the page in physical memory. Here, it needs to provide a free page to the application. If the application requests a free page from the global memory manager, it may take a long time because of competition with normal non-realtime applications. To avoid this delay, the technique therefore makes the application obtain a free page from a CPL. When the shared page was allocated from the application's reservation, one of the other reservations using the shared page is selected. Any policy can be used for selecting a reservation, but it may depend on the implementation of how the reservations sharing the page are managed, e.g. a FCFS list for each page. The selected reservation takes a free page out of its CPL and gives the page to the application. The selected reservation then becomes a new owner for the shared page. Figure 4(b) is an example of this case. R_1 selects the shared page for a victim and unmaps it from its active/inactive list. The shared page becomes a private page of R_2, and R_2 gives one free page from its CPL. On the other hand, when the victim shared page is owned by another reservation, the application takes and uses a free page from its CPL. Because the application is not the owner of the shared page, the application has conserved a free page since it accessed the shared page for the first time, so there is at least one free page in its CPL. After this procedure, if the shared page is no longer used by more than one reservation, it becomes a private page. In Figure 4(c), now R_2 selects the shared page for a victim. The shared page is unmapped from R_2 and becomes a private page of R_1. Then, R_2 takes a free page out of its CPL and includes the page in its active/inactive list.

The advantage of this technique is that it runs automatically without any user intervention. Since the technique uses the page-table modification points, it supports not only user-visible shared pages, such as IPC and shared libraries, but also user-invisible shared pages, copy-on-write and file caches. The disadvantage of this technique is that it cannot get the benefit of reduced memory usage of page-sharing, because it conserves free pages for each shared page. However, since it does not permanently lock the entire shared page area, it can provide better memory availability to private pages with a given amount of physical memory, when the working set size of shared pages changes at run-time. We will further discuss this issue in the Section IV-C.

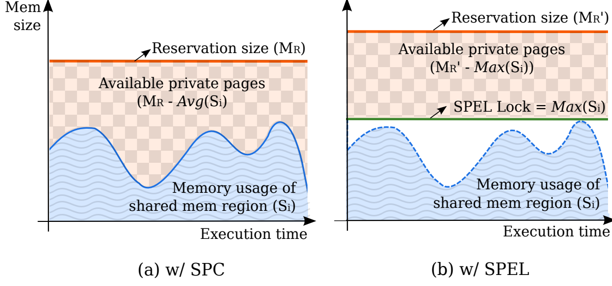
B. Shared-Page Eviction Lock (SPEL)

The SPEL scheme prevents the arbitrary eviction of shared pages by locking specified shared pages. Since locking all private/shared pages is infeasible due to the limited size of physical memory, we suggest locking only shared pages which may cause inter-task interference. The locked shared pages are never evicted during the execution of applications. Using this lock information, we can reduce the size of physical pages occupied by memory reservations. For example, suppose that there are n memory reservations, each of which reserves M_R pages and shares M_S pages. If a user specifies an eviction lock for the shared pages, we can safely deallocate $(n-1)M_S$ pages from the reservations, so the total size of physical pages for the reservations becomes $nM_R - (n-1)M_S = n(M_R - M_S) + M_S$. Run-time cancellation of SPEL is undesirable, because it needs to request free pages from the global memory manager, which may introduce an unexpected delay when the system is under memory pressure, thereby nullifying the benefit of memory reservation scheme.

SPEL can be specified with the virtual memory information of a process, which can be easily obtained from memory-map reporting tools, such as the `pmap` command. Figure 5 is an example snapshot of the memory map. It shows a starting address, size, and a symbol of each virtual

```
0000000000400000      8K r-x--  /opt/testapp_shm
00007f5880761000  10240K rw-s-    [ shmid=0x1a0014 ]   ------>  Shared data (shmem)
...
00007f5881161000     96K r-x--  /lib/libpthread-2.11.1.so  ------>  Pthread Lib.
...
00007f588137e000   1512K r-x--  /lib/libc-2.11.1.so        ------>  C Lib.
...
00007f5881701000    520K r-x--  /lib/libm-2.11.1.so        ------>  Math Lib.
...
00007f5881984000     28K r-x--  /lib/librt-2.11.1.so       ------>  POSIX RT Lib.
...
```

Figure 5. Memory map example captured by using pmap command



(a) w/ SPC (b) w/ SPEL

Figure 6. Memory usage compasiron of SPC and SPEL

memory area, including shared data and shared libraries.

The advantage of SPEL is that it yields a system-level memory benefit by reducing the size of physical memory usage of memory reservations. However, since we are not able to reclaim the locked pages at run-time, the inappropriate use of eviction locks, such as locking infrequently-used pages, could introduce poor memory efficiency. Moreover, a memory reservation is required to exceed the size of locked shared pages, so it may increase the minimum memory requirement for executing an application. We therefore next consider using SPC and SPEL together.

C. Combined Use of SPC and SPEL

As shown in Figure 5, multiple shared memory regions exist in a process's virtual memory space, and the SPC and SPEL schemes can be applied to each of these shared memory regions. Both techniques are able to avoid timing penalties from the shared page eviction, but they have different memory usage characteristics. Here, we explain how to apply either SPC or SPEL for each shared memory region. The purpose of the combined use of SPC and SPEL is to reduce the total physical memory usage of memory reservations, while providing the same amount of memory availability to private pages.

Figure 6 compares the memory usage of the SPC and SPEL schemes. X-axis presents the execution time of an application, and Y-axis corresponds to the size of memory. The working set size of a shared memory region S_i changes during the application's execution. Figure 6(a) shows the available memory size for private pages under the given reservation size M_R with the SPC scheme. Since SPC does not permanently allocate the entire shared memory region, the application can use $M_R - Avg(S_i)$ of memory for private pages on average. In Figure 6(b), SPEL locks the shared memory region S_i, and the locked memory size is equal to the maximum usage of S_i. With the given

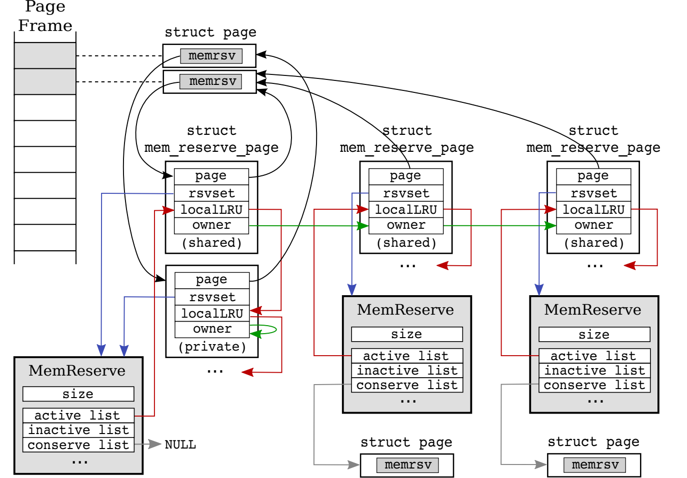


Figure 7. Data structure used by our implementation

reservation size ${M_R}'$, the available pages for private memory is ${M_R}' - Max(S_i)$. In order to provide the same memory availability for private pages as the SPC scheme, ${M_R}'$ should be larger than ${M_R}'$ and can be represented as ${M_R}' = M_R + Max(S_i) - Avg(S_i)$. Even though SPEL may increase individual memory reservation's size ${M_R}'$, it can reduce the total memory usage of memory reservations. When n memory reservations share S_i, SPC requires nM_R regardless of the size of the shared region. On the other hand, SPEL requires $n{M_R}' - (n-1)Max(S_i) = nM_R + Max(S_i) - nAvg(S_i)$. If $Max(S_i)$ is less than $nAvg(S_i)$, SPEL consumes fewer physical pages than SPC. The opposite case can also happen when n is small or the maximum usage is much larger than the average usage. Based on this observation, we can selectively apply SPC and SPEL to each shared memory region to reduce the total physical memory usage of memory reservations.

V. Implementation

In this section, we describe our implementation on Linux/RK [14] which is based on the Linux 2.6.32.42 kernel. We have targeted the 64-bit x86 architecture and use the typically used configuration of 4KB page frames.

Our proposed shared memory management scheme needs to maintain a link to individual physical pages and to support multiple ownership control for each page. To make this possible, we have designed its data structure as shown in Figure 7. We declare and use `struct mem_reserve_page` for our own purpose, which contains a local LRU-list entry, an ownership list entry, and a pointer variable to reference a corresponding physical page. The Linux kernel maintains a page descriptor, `struct page`, for each physical page to keep track of the current status of the page. The pointer in `mem_reserve_page` indicates its corresponding page descriptor to access the page's status. Since a page can be allocated by any application and be dynamically mapped to another application, we need to be able to access `mem_reserve_page` from a page descriptor. Hence, we

have inserted a pointer variable into the page descriptor to enable cross-referencing. When a page is shared by two or more memory reservations, `mem_reserve_page` is connected to another `mem_reserve_page` via its ownership list entry. All shared references in the ownership list point to the same page descriptor, and they are reachable from the page descriptor via a two-step reference, which finds one of them using a pointer in the page descriptor and traverses its ownership list. The pointer to `mem_reserve_page` in the page descriptor is also used to identify its owner, which is necessary for the SPC scheme. Except the owner, each of the other memory reservations in the ownership list conserves one free page in its conserved-page list.

Like previous implementations of memory reservation [5][8], we have inserted hooks into the page allocation function and the deallocation function of the kernel for implementing memory reservation. In addition, we have put hooks into the PTE mapping/unmapping functions to monitor dynamic page sharing.

For implementing the SPEL scheme, we need to identify whether a page has an *eviction lock*. The Linux kernel uses a flag variable in a page descriptor to describe the status of a physical page. Each bit of the flag represents a certain condition, and many bits of the flag remain unused. Hence, we have used one of the unused bits to specify an eviction lock. We inserted a system call for users to set and cancel an eviction lock. The system call parameters include a task ID, a start address of a virtual memory area, and a size of the area. It first tests whether the task's memory reservation is greater than the requested area. Then, it makes all pages of the area present in physical memory, and deallocates free pages gained by using SPEL.

The system call and the hook functions are implemented as a loadable kernel module. Though our implementation is based on 64-bit x86 architecture, we expect that they also can be used in other architectures, because we have not modified any architecture-dependent code except the system call table.

VI. Performance Evaluation

This section presents our experiments on the proposed schemes and analyzes their effects on real-time applications using shared pages. The target machine for the experiment is equipped with the Intel Core-i5 2540M processor running at 2.6GHz, 4GBytes of RAM, and a 2.5inch, 500GBytes 7200rpm hard-disk drive.

A. Microbenchmark

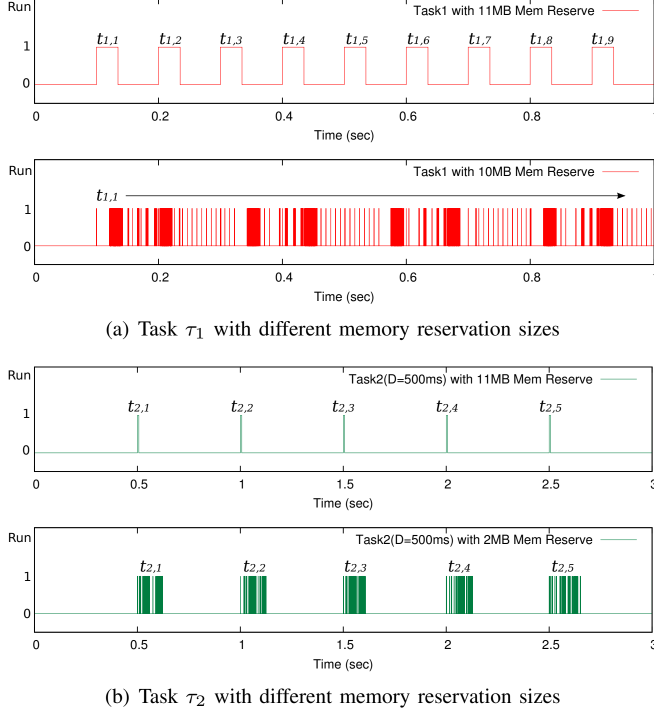
Our shared-page management mechanism may introduce spatial and computational overheads, due to managing an additional data structure for each page and hooking the kernel's memory functions. Firstly, we present the increased memory usage from our schemes. Since we have inserted one pointer variable to the page descriptor in the Linux kernel, the size of the page descriptor is increased from 56 bytes to 64 bytes. This means 0.195% of memory overhead in the system using 4KB pages. For instance, in the target system equipped with 4GB RAM, 8MB of physical memory is additionally consumed regardless of the creation of memory reservations. This overhead can be considerably reduced if we configure to use bigger pages, such as a 4MB page mode in x86. When we create a memory reservation, we allocate a `mem_reserve_page` for each page frame. The size of `mem_reserve_page` in the current implementation is 56 bytes. Hence, it needs an additional 1.37% of memory for the requested reservation size. This overhead comes from both our shared-page management mechanism and memory reservation. The pure spatial overhead caused by our mechanism is 24 bytes per each page, meaning 0.59% of the reservation size.

Table II
Computational overhead of Shared-Page Management

Items	Linux kernel (Global Mgmt)	Mem Reserve w/o SP-Mgmt	Mem Reserve w/ SP-Mgmt
Page allocation (`alloc_page`)	400.4	192.4	202.3
Page deallocation (`__free_page`)	208.6	187.2	188.6
Mapping a shared page to a page table	1197.0	1220.4	1417.3
Unmapping a shared page from a page table	816.2	834.3	1059.7

** All values are in CPU cycles*

Next, we determine the computational overhead of our mechanism. SPEL does not incur run-time overhead after it is set. SPC, however, performs page ownership control and monitors shared-page mapping at run-time. Therefore, we compared the page allocation and deallocation times in our mechanism with not only the times in the global memory management of the original Linux kernel but also the times in the memory reservation scheme lacking our mechanism. We also measured the times for mapping and unmapping a shared page in our scheme. We used the `rdtsc` instruction for time measurement, which returns the number of elapsed CPU-clock cycles. The measurement was conducted under no memory pressure and no page-swapping happened. Table II presents the results. Every number in the table is an average observed value. The memory reservation scheme without and with shared-page management (SP-Mgmt) took less time than the global memory management of the original Linux kernel for allocating and deallocating a page, because the memory reservation manages a relatively small number of isolated pages. The last column of the table shows the execution time of the memory reservation with our mechanism. For page allocation and deallocation, it spent almost the same clock cycles as the cycles in the memory reservation without our mechanism. For mapping and unmapping a shared page, it spent approximately 200 cycles more than the results in the original Linux kernel. However, page mapping happens when the page is accessed for the first time, and page unmapping happens when page replacement occurs. We therefore conclude that the overheads induced are either negligible or acceptably small.



(a) Task τ_1 with different memory reservation sizes

(b) Task τ_2 with different memory reservation sizes

Figure 8. Identifying minimum schedulable memory sizes for τ_1 and τ_2

Table III
PHYSICAL MEMORY USAGE COMPARISON

Task-set	Technique	Per-task Memory Reserve (MBytes)	Total Physical Memory Usage (MBytes)
$\{\tau_1, \tau_1\}$	No-SP-Mgmt	Each τ_1 = 11	(11+11)-10 = 12
	SPC		11+11 = 22
	SPEL		(11+11)-10 = 12
$\{\tau_2, \tau_2\}$	No-SP-Mgmt	Each τ_2 = 2	(2+2)-1 = 3
	SPC		2+2 = 4
	SPEL	Each τ_2 = 11	(11+11)-10 = 12
$\{\tau_1, \tau_2\}$	No-SP-Mgmt	τ_1 = 11, τ_2 = 2	(11+2)-1 = 12
	SPC		11+2 = 13
	SPEL	τ_1 = τ_2 = 11	(11+11)-10 = 12

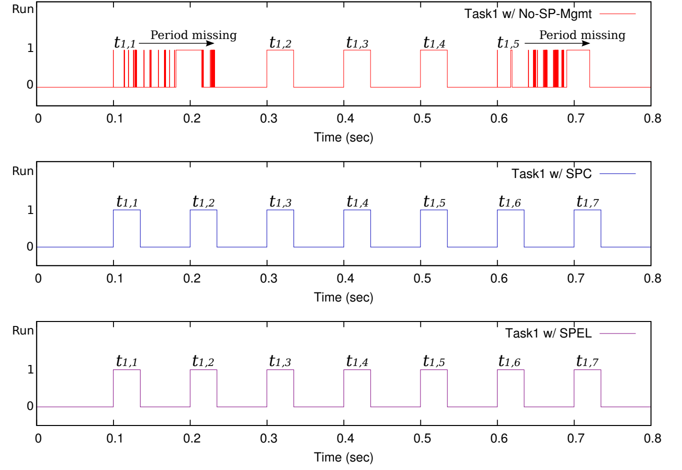


Figure 9. Execution history of τ_1 running with τ_2, under No-SP-Mgmt(top), SPC(middle), and SPEL(bottom)

B. The Effect of Our Proposed Mechanisms

To show the effectiveness of our schemes, we used two periodic tasks running at different periods and memory access patterns. The tasks' real-time periodic task parameters (C_i, T_i, D_i), a maximum computation time C_i per each period, period T_i, and deadline D_i, are specified with the Linux/RK APIs; Task τ_1's (C, T, D) is $(40_{ms}, 100_{ms}, 100_{ms})$; Task τ_2's (C, T, D) is $(100_{ms}, 500_{ms}, 500_{ms})$. Both tasks are given real-time scheduling priorities by Linux/RK; τ_1 has higher priority than τ_2 because $T_1 < T_2$. The system has 10MB of shared memory region created by the *shmem* system call. τ_1 sequentially accesses 10MB of shared memory in every period. τ_2 accesses 1MB per period, so it takes 10 periods to access the entire shared memory. The tasks are compiled by *gcc* with *-static* option, so they do not use shared libraries.

To begin, we determined the minimum schedulable memory reservation size for each task. Figure 8(a) illustrates the execution history of τ_1 with different sizes of memory reservations. τ_1 with 11MB memory met its deadlines, but τ_1 with 10MB memory could not even finish its first period until 3sec; thus, we could understand that at least 11MB of memory is required for τ_1. Figure 8(b) shows the execution history of τ_2. The upper graph is the result of τ_2 with 11MB memory, and the lower graph is the result with 2MB memory. When τ_2 ran with 2MB memory, its completion time was retarded, but it could still meet its deadlines. Hence, we determined that 2MB of memory reservation is enough for running τ_2.

Next, we examined the physical memory usage when multiple tasks access the shared memory region. We erected and used three task-sets consisting of τ_1 and τ_2: $\{\tau_1, \tau_1\}$, $\{\tau_2, \tau_2\}$, and $\{\tau_1, \tau_2\}$. Running a same task twice was accomplished by executing a copy of the task's program image, so the kernel's code page sharing did not happen here. Table III shows the per-task memory reservation size and the total physical memory usage with the three task-sets, under the control of no shared-page management (No-SP-Mgmt), SPC, and SPEL. In case of $\{\tau_1, \tau_1\}$, the per-task memory reservation was 11MB with all techniques, but SPC spent 10MB more physical memory than No-SP-Mgmt and SPEL, due to its Conserved-Page List. In case of $\{\tau_2, \tau_2\}$, the per-task memory reservation was 2MB with both No-SP-Mgmt and SPC, but it was 11MB with SPEL. Though the average per-period shared memory usage of τ_2 was 1MB, SPEL needed to lock the entire shared memory region, which resulted in 9MB and 8MB more total memory usage than No-SP-Mgmt and SPC, respectively. In case of $\{\tau_1, \tau_2\}$, No-SP-Mgmt and SPC used a different reservation size for each task, but SPEL used the same size for both tasks. The total physical memory usage of SPEL was equal to that of No-SP-Mgmt and was 1MB less than that of SPC. This result implies that, to reduce the total memory usage, applying SPC and SPEL to a shared memory region should consider the tasks' access patterns to the shared region.

To verify the temporal isolation of our schemes from shared-page eviction, we executed the task-set $\{\tau_1, \tau_2\}$ with

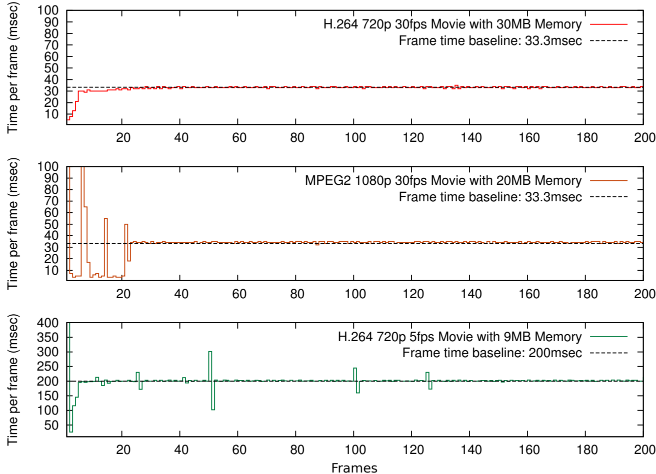


Figure 10. Time-per-frame of three different video files

No-SP-Mgmt, SPC, and SPEL. Figure 9 shows the execution history of τ_1. Without our mechanisms, τ_1 missed its periodic deadlines every 500msec, which is equal to τ_2's period T_2, because τ_2 swapped out the 1MB of shared pages during each of its invocations. τ_1 experienced self-suspension due to the eviction of shared pages, thereby affecting its schedulability. τ_1 had higher priority than τ_2, but τ_1 missed its deadlines, not τ_2, demonstrating that shared pages, if not managed correctly, can be a significant source of priority inversion. However, τ_1 with SPC and SPEL showed the same result as when it executed alone. τ_2 tried to evict the shared pages, but both SPC and SPEL prevented the eviction of shared pages being used by τ_1. Unlike τ_1, τ_2 was schedulable in all the three cases, but there was a difference between the cases. τ_2 without using our mechanisms showed similar execution history to when it ran alone with 2MB memory. With SPC and SPEL, however, τ_2's execution seemed like when it ran alone with 11MB memory.

C. Case Study: MPlayer

We now use the MPlayer open-source movie player to demonstrate the effectiveness of our schemes in a real-world application. We prepared three different video files: (i) a H.264 video with 1280x720 (720p) frame size and 30fps frame rate (33.3msec period), (ii) a MPEG2 video with 1920x1080 (1080p) frame size and 30fps frame rate (33.3msec period), and (iii) a H.264 video with 720p frame size and 5fps frame rate (200msec period). We checked the minimum schedulable memory reservation size for playing each video file. Figure 10 presents the time-per-frame histories of playing each video file with the minimum memory reservation size. The first movie file needed at least 30MB reservation to run properly. The number of major page faults, which need disk swapping and may suspend the task, was zero, and the number of minor page faults, which only sets a page table mapping to existing pages, was 8677. The second video with 20MB reservation caused 272 major page faults and 9258 minor page faults; it started with some fluctuations on frame time, but stabilized after playing 24 frames of the video. The third video with 9MB caused 741 major page faults and 8551 minor page faults; it showed more fluctuations, but it also stabilized after playing 130 frames of the video. Therefore, we picked these values as the sufficient reservation sizes for playing the associated video files.

Table IV
SHARED MEMORY USAGE OF *MPlayer* TASK

Shared area S_i	Max(S_i) (KBytes)	Avg(S_i) (KBytes)	Max(S_i) < nAvg(S_i) (*n=3*)
MPlayer code	13088	900	No
libc-2.11.1.so	1512	288	No
libSDL-1.2.so.0.11.3	420	4	No
libstdc++.so.6.0.13	984	8	No
libpthread-2.11.1.so	96	24	No
libm-2.11.1.so	520	92	No
libdl-2.11.1.so	8	4	Yes

By using the obtained memory reservation sizes, we played the three video files together, under the memory reservation approach without and with our shared-page management schemes. Table IV presents the total size ($Max(S_i)$) and the average usage ($Avg(S_i)$) of each shared memory area S_i, measured while running the three videos. MPlayer has more shared memory areas, but the table only shows the shared areas with $Avg(S_i) > 1page(= 4KB)$. In most shared areas, $Avg(S_i)$ is much less than $Max(S_i)$ because MPlayer uses only some portion of shared areas after its initialization. For these areas, SPC is better than SPEL to minimize the total physical memory usage. Only the shared library *libdl* in the last row shows that $Max(S_i)$ is less than $nAvg(S_i)$, meaning o SPEL is better than SPC for this area. Therefore, we applied SPEL only to this area and let other shared memory areas be managed by SPC.

The experiment scenario was as follows. At first, we started the H.264 30 fps movie with 30MB of memory reservation. After playing 300 frames of the movie, we started the MPEG2 30 fps movie with 20MB of memory reservation. Lastly, after playing 600 frames of the first movie, we started the H.264 5 fps movie with 9MB of memory reservation. Figure 11 shows the time-per-frame history of playing the first movie file. We can easily observe the performance difference with and without our scheme. With our scheme, the graph shows constant frame time, regardless of playing other movie files. However, without our scheme, the MPlayer instance playing the first movie experienced 143 major page faults due to the eviction of shared pages, which did not occur with our scheme. This result also implies that a real-world application can be easily affected by other applications sharing the same physical pages, and our schemes can provide enhanced temporal isolation to their memory access.

During the experiments, we learnt two lessons. The first one is that it is not easy to decide the optimal memory size of memory reservation, which minimizes the memory usage of a task and does not deteriorate the task's schedulability. The memory reservation size depends on the task's data locality and the execution period, so it needs some testing to get the optimal size before deploying it. On the plus

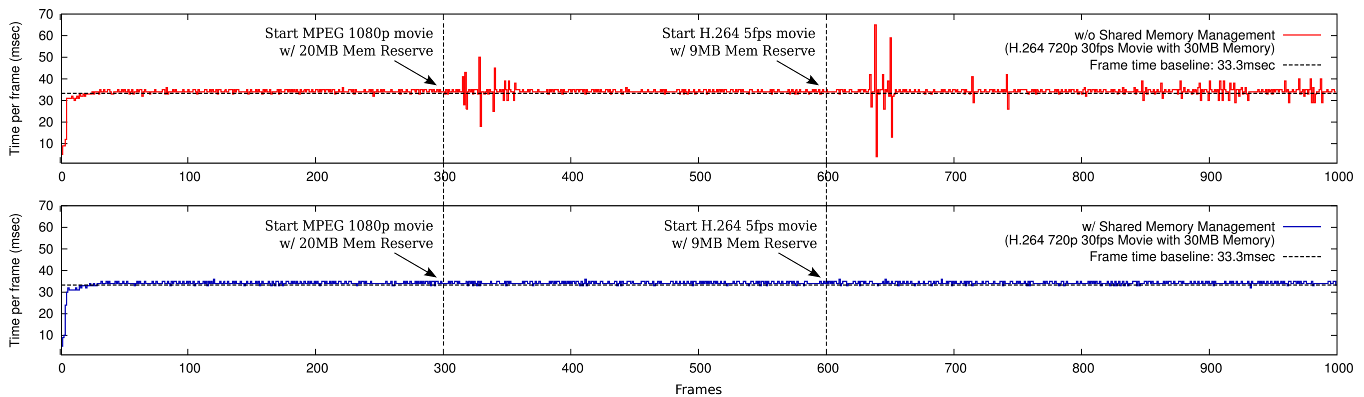


Figure 11. Time-per-frame w/ and w/o Shared-Page Management

side, our schemes enable the use of the memory configuration established during the testing phase, regardless of co-running applications. The second lesson learnt is regarding the impact of non-CPU resources on the CPU scheduling of multi-core systems. When shared pages are evicted, a task is suspended during page swapping. We tried to run each task on a different CPU core and observed the result, but it did not help at all. The impact of unexpected self-suspension to the schedulability and utilization of scheduling algorithms is likely more severe on multi-core systems. Therefore, the OS components managing non-CPU resources also need to respect the CPU scheduling algorithms and to consider the multi-core environment.

VII. Conclusions

We demonstrated potential problems of existing memory reservations with shared pages, and proposed two shared-page management schemes; Shared-Page Conservation and Shared-Page Eviction Lock, which address the shared-page problems. The proposed mechanism has been implemented in Linux/RK, a resource kernel. The experimental results show the effect of the proposed schemes not only with simple task models, but also with a real-world application. Our implementation is based on the Linux kernel, but we expect that our approach can be easily applied to other operating systems with paged virtual memory. Our shared-page management schemes improve the desirable property of temporal isolation in memory reservations. However, it cannot completely remove interference among applications. For example, if an application accesses several files on a disk drive, other applications accessing the disk, such as reading in files and swapping in/out, can be easily affected by the application due to the contention at the shared resource. The cache interference on multi-core systems is also a critical issue to be solved for providing temporal isolation. We plan to study these issues in the future.

References

[1] Linux cgroups. http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[2] S. A. Belogolov, J. Park, J. Park, and S. Hong. Scheduler-Assisted Prefetching: Efficient Demand Paging for Embedded Systems. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[3] A. D. Brown and T. C. Mowry. Taming the memory hogs: using compiler-inserted releases to manage physical memory intelligently. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[4] R. P. Draves. Page replacement and reference bit emulation in mach. In *USENIX Mach Symposium*, 1991.

[5] A. Eswaran and R. Rajkumar. Energy-aware memory firewalling for QoS-sensitive applications. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[6] S. M. Hand. Self-paging in the nemesis operating system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[7] D. Hardy and I. Puaut. Predictable code and data paging for real time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[8] S. Kato and Y. Ishikawa. CPU scheduling and memory management for interactive real-time applications. *Real-Time Systems*, pages 1–35, 2011.

[9] P. A. Laplante. *Real-time systems design and analysis - an engineer's handbook*. IEEE, 1993.

[10] M. Malkawi and J. Patel. Compiler directed memory management policy for numerical programs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1985.

[11] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory resource management for real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2007.

[12] M. Masmano, I. Ripoll, and A. Crespo. A comparison of memory allocators for real-time applications. In *International Workshop on Java Technologies for Real-time and Embedded Systems*, 2006.

[13] M. Masmano, I. Ripoll, A. Crespo, and J. Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2004.

[14] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *IEEE Real-Time Systems Sumposium (RTSS) Work-In-Progress*, 1998.

[15] A. Patil and N. Audsley. An efficient page lock/release os mechanism for out-of-core embedded applications. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[16] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2007.

[17] I. Puaut and P. Solidor. Real-time performance of dynamic memory allocation algorithms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2002.

[18] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[19] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Towards a predictable real-time system. In *USENIX Mach Symposium*, pages 73–82, 1990.

[20] C. Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Field and Robotics*, 2008.