Journal of Systems Architecture 60 (2014) 165–178

Contents lists available at SciVerse ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Memory reservation and shared page management for real-time systems

Hyoseung Kim *, Ragunathan (Raj) Rajkumar

Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, USA

ARTICLE INFO

Article history:
Available online 15 July 2013

Keywords:
Real-time memory management
Shared page
Memory reservation
Temporal isolation
Resource kernels

ABSTRACT

Memory reservations are used to provide real-time tasks 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 task may experience unexpected timing delays from other tasks through shared pages that are shared by another process, even though the task has enough free pages in its own reservation. In this paper, we first describe the problems that arise when real-time tasks share pages. We then propose a shared-page management framework which enhances the temporal isolation provided by memory reservations in resource kernels that use the resource reservation approach. Our proposed solution consists of two schemes, 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 framework 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 also be applied to other operating systems with paged virtual memory.

1. Introduction

Future real-time and cyber-physical systems are expected to interact with the external environment and be able to deal with dynamic factors such as unexpected workload surges and severe weather changes. As a result, such systems 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.

Many researchers in the real-time systems community have studied memory management schemes that enable the use of virtual memory in a predictable manner. Among a variety of approaches, memory reservation [5,21] has been considered a viable approach to provide guaranteed and predictable memory access performance to real-time tasks. It allows a task to reserve a portion of the total system memory for its exclusive use, and these reserved memory pages can never be stolen by other tasks even if the system is under memory pressure. If the task 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 task's memory access be contained within its reservation set, thereby providing temporal isolation to each task. 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 tasks.

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 tasks, so the impact of shared pages has not been extensively 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

* Corresponding author. Tel.: +1 4129965417.
E-mail addresses: hyoseung@cmu.edu (H. Kim), raj@ece.cmu.edu (R. (Raj) Rajkumar).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.07.002

multiple instances of a single application. Our measurements indicate that the fraction of shared page usage by an application can be as high as about 56%, which captures the importance of shared pages in current operating systems.[1]

The use of shared pages, however, may cause a real-time task to experience unexpected timing penalties due to the behavior of other tasks. Even though a task has its own memory reservation with a sufficient number of free pages, the task's shared pages can be arbitrarily swapped out to disk by other tasks that have access to the shared pages. Hence, when the task accesses the evicted shared pages, it needs to spend extra time to swap the shared pages in from disk. This can significantly weaken the temporal isolation capability of memory reservation.

We propose a shared-page management framework for memory reservation to provide temporally predictable and isolated memory access to real-time tasks that use shared pages. Our framework avoids any unexpected timing penalties from the arbitrary eviction of shared pages, thereby providing significantly better temporal isolation between tasks when memory reservation is used. Our framework operates 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 management techniques.

Contributions: The contributions of this work are as follows:

- We show how shared pages can affect the performance of real-time tasks 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.
- Our shared-page management framework consists of two schemes, Shared-Page Conservation and Shared-Page Eviction Lock, each of which avoids any unexpected timing penalties from shared pages. These two schemes have different memory usage characteristics. We analyze the system-wide memory usage with these schemes and describe their combined use to minimize overall system memory usage.
- We have implemented and evaluated our schemes by extending the Linux/RK platform [14,18]. The experimental results show the practicality and the effectiveness of our approach.

Organization: The rest of this paper is organized as follows. Section 2 presents some background on the memory reservation approach and presents our measurements on the shared page usage of multimedia applications. Section 3 explains the problems with shared pages under existing memory partitioning and reservation approaches. Section 4 describes our shared-page management approach. Section 5 discusses the implementation of our framework. A detailed evaluation of our framework is provided in Section 6. Related work is discussed in Section 7. Finally, we conclude the paper in Section 8.

2. Background

In this section, we provide a high-level overview of the memory reservation paradigm [5] upon which our approach is based. We also measure the shared page usage of multimedia applications in order to understand the importance of managing shared pages.

2.1. Memory reservation

Memory reservation aims to provide predictable and enforced access to memory resources. With memory reservation, a real-time task can specify its memory demands, and after memory is reserved, the requested memory pages are exclusively used by the task. Other tasks cannot take the reserved pages away; thus, the task with the memory reservation can expect constant performance regardless of the existence of co-running tasks. Memory reservation also ensures that the potential abuse of memory by any task does not affect the performance of other tasks. In the case that a task 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 task 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 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, a task bound to the reservation 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, a task 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. Fig. 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 tasks belonging to the reservation.

2.2. Shared page usage of multimedia applications

In order to understand the performance impact of shared pages in current systems, we measured the shared page usage of multimedia applications. Table 1 presents private and shared page usage of the multimedia applications in a Linux system running Ubuntu 10.04. 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.

3. 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 tasks.

[1] Please see Section 2.2 of this paper for additional details.