Journal of Systems Architecture 57 (2011) 638–654

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

An overview of interrupt accounting techniques for multiprocessor real-time systems

Björn B. Brandenburg *, Hennadiy Leontyev, James H. Anderson

Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC 27599-3175, United States

ARTICLE INFO

Article history:
Available online 2 June 2010

Keywords:
Schedulability analysis
Overhead accounting
Global EDF

ABSTRACT

The importance of accounting for interrupts in multiprocessor real-time schedulability analysis is discussed and three interrupt accounting methods, namely quantum-centric, task-centric, and processor-centric accounting, are analyzed and contrasted. Additionally, two special cases, dedicated interrupt handling (*i.e.*, all interrupts are processed by one processor) and timer multiplexing (*i.e.*, all jobs are released by a single hardware timer), are considered and corresponding analysis is derived. All discussed approaches are evaluated in terms of schedulability based on interrupt costs previously measured on a Sun Niagara multicore processor. The results show that there is no single "best" accounting technique that is always preferable, but rather that the relative performance of each approach varies significantly based on task set composition, *i.e.*, the number of tasks and the maximum utilization.

1. Introduction

System overheads such as time lost to task switches and scheduling decisions must be accounted for in real-time systems if temporal correctness is to be guaranteed [1,2]. Of these overheads, *interrupts* are notoriously troublesome for real-time systems since they are not subject to scheduling and can significantly delay real-time tasks.

In work on uniprocessor real-time systems, methods have been developed to account for interrupts under the two most-commonly considered real-time scheduling policies [3]: under *static-priority* scheduling, interrupts can be analyzed as higher-priority tasks [2], and under *earliest-deadline-first* (EDF) scheduling, schedulability can be tested by treating time lost to processing interrupts as a source of blocking [4].

Properly—but not too pessimistically—accounting for interrupts is even more crucial in multiprocessor real-time systems. Due to their increased processing capacity, such systems are likely to support much higher task counts, and since real-time tasks are usually invoked in response to interrupts, multiprocessor systems are likely to service interrupts much more frequently. Further, systematic pessimism in the analysis has a much larger impact on multiprocessors, as is shown below.

Unfortunately, interrupts have not received sufficient attention in work on multiprocessor real-time systems. The first and, to the best of our knowledge, only published approach to date was proposed by Devi [5]. Devi presented a quantum-centric accounting method in which the length of the system's scheduling quantum is reduced to reflect time lost to overheads. In this paper, we consider this method, as well as two others, in the context of global scheduling algorithms. When the quantum-centric method is applied in this context, it is usually necessary to assume that all possible interrupts occur on all processors in every quantum. This assumption is obviously quite pessimistic and motivates the consideration of other approaches.

Motivating example. In a recent case study on a 32-processor platform involving up to 600 light[1] tasks, the release overhead (*i.e.*, the time taken to process a timer interrupt and invoke a real-time task) of a *global EDF* (G-EDF) implementation was measured to exceed 50 μs in the worst case [6]. Given the system's quantum size of 1000 μs, the quantum-centric method would have deemed any task set of 20 or more tasks unschedulable—with fewer tasks than processors, this is clearly excessively pessimistic.

In the above case study, a new, less pessimistic "task-centric" accounting method (see Section 4) was developed. However, even with "task-centric" accounting, G-EDF performed worse than expected. Suspecting high job release overhead to be the cause, we conducted simulations to estimate performance assuming reduced overhead. Surprisingly, we found that even with a 75% reduction in release overhead, schedulability increases very little (see Fig. 1). However, the experiments also confirmed release overhead as the leading cause of reduced schedulability—performance improved dramatically assuming overhead-free releases. This

* Corresponding author.
E-mail addresses: bbb@cs.unc.edu (B.B. Brandenburg), leontyev@cs.unc.edu (H. Leontyev), anderson@cs.unc.edu (J.H. Anderson).

[1] The utilization of the tasks was distributed uniformly in [0.001, 0.1]. Please see Section 6.1 and [6] for a detailed description of these experiments.

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.05.011

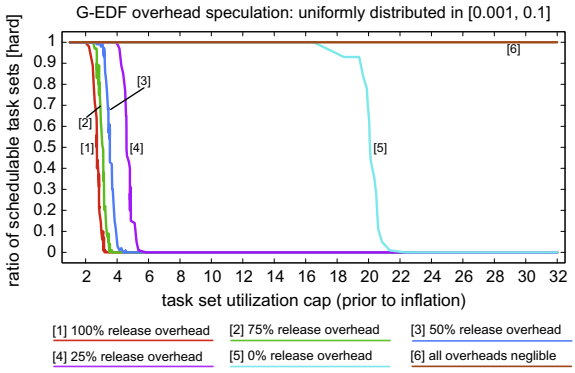

Fig. 1. Hard real-time schedulability under G-EDF on a 32-processor platform assuming reduced release overhead, which is accounted for using the task-centric method. Note that all task sets are schedulable if all overheads are assumed to be negligible. (This graph corresponds to Fig. 2a in [6]; see Section 6 for details.)

discrepancy stems from quadratically-growing pessimism in the "task-centric method," as is shown in Section 4.3.3. Fig. 1 highlights that accurate accounting for overheads is crucial to multiprocessor real-time performance, and that systematic pessimism can have a significant impact on multiprocessor systems due to high task counts.

Contributions. This paper provides an overview of all currently-known multiprocessor interrupt accounting techniques for global schedulers. In particular, we

- highlight the importance of accurate interrupt accounting for multiprocessor real-time systems and survey commonly-encountered interrupt sources in current multiprocessor systems (Section 2, formalized in Section 3);
- summarize "quantum-centric" interrupt accounting (Section 4.1) and describe "task-centric" interrupt accounting (Section 4.3), which has been used—but not described in detail—in previous studies [6–8] to overcome the limitations of "quantum-centric" accounting, and show that the "task-centric" method still over-estimates interrupt delays by a factor that is quadratic in the number of tasks (Section 4.3.3);
- show how interrupt accounting is fundamentally different from previous work on reduced-capacity scheduling and, by addressing these differences, propose "processor-centric" interrupt accounting, which is designed to overcome the pessimism inherent in "task-centric" accounting (Section 4.4);
- discuss two common special cases, namely how job releases are delayed if interrupts are handled by a dedicated processor (Section 5.2), and further how said delays change if all job releases are triggered by a single hardware timer (Section 5.3); and
- evaluate the effectiveness of all considered approaches in terms of schedulability for both hard and soft real-time systems and show that none of the proposed approaches strictly dominates the others, *i.e.*, there is no single "best" approach (Section 6).

This paper extends a previous conference version [9]; in particular, a discussion of delays due to "inter-processor interrupts" (Section 4.2) and common special cases (Section 5) has been added, and the experiments (Section 6) were augmented to include all special cases and repeated with more-recent overhead measurements [8], thus reflecting over a year of implementation improvements.

Next, we provide a survey of common interrupt types and describe how they are usually serviced by operating systems.

2. Interrupts

To motivate our system model, we begin by providing a high-level overview of interrupts in a modern multiprocessor architecture. We focus on Intel's x86 architecture because it is in widespread use and well-documented [10,11], but the discussion similarly applies to other multiprocessor architectures as well [12,13].

Interrupts notify processors of asynchronous events and may occur between (almost) any two instructions. If an interrupt is detected, the processor temporarily pauses the execution of the currently-scheduled task and executes a designated *interrupt service routine* (ISR) instead. This can cause the interrupted task to incur undesirable delays that must be accounted for during schedulability analysis.

Most interrupts are *maskable*, *i.e.*, the processor can be instructed by the OS to delay the invocation of ISRs until interrupts are unmasked again. However, *non-maskable interrupts* (NMIs), which can be used for "watch dog" functionality to detect system hangs, cannot be suppressed by the OS [11].

In multiprocessor systems, some interrupts may be local to a specific processor (*e.g.*, register-based timers [12]), whereas others may be serviced by multiple or all processors.

Interrupts differ from normal preemptions in that a task cannot migrate while it is being delayed by an ISR, *i.e.*, a task cannot resume execution on another processor to reduce its delay. This limitation arises due to the way context switching is commonly implemented in OSs. For example, in Linux (and Linux-derived systems such as the one considered in [6]), there is only a single function in which context switching can be performed, and it is only invoked at the *end* of an ISR (if a preemption is required). From a software engineering point of view, limiting context switches in this way is desirable because it significantly reduces code complexity. In terms of performance, ISRs tend to be so short (in well-designed systems) that context-switching and migration costs dominate ISR execution times. Hence, delaying tasks is usually preferable to allowing migrations unless either migration and scheduling costs are negligible or ISR execution times are excessively long.

Delays due to ISRs are fundamentally different from scheduling and preemption overheads: the occurrence of scheduling and preemption overheads is controlled by the OS and can be carefully planned to not occur at inopportune times. In contrast, ISRs execute with a statically-higher priority than any real-time task in the system and cannot be scheduled, *i.e.*, while interrupts can be temporarily masked by the OS, they cannot be selectively delayed,[2] and thus are not subject to the scheduling policy of the OS.

2.1. Interrupt categories

Interrupts can be broadly categorized into four classes: *device interrupts* (DIs), *timer interrupts* (TIs), *cycle-stealing interrupts* (CSIs), and *inter-processor interrupts* (IPIs). We briefly discuss the purpose of each next.

DIs are triggered by hardware devices when a timely reaction by the OS is required or to avoid costly "polling" (see below). In real-time systems, DIs may cause jobs to be released, *e.g.*, a sensor may trigger a DI to indicate the availability of newly-acquired data, in response to which a job is released to process said data.

TIs are used by the OS to initiate some action in the future. For example, TIs are used to support high-resolution delays ("sleeping") in Linux. They can also be used for periodic job releases and

[2] Masking may create non-trivial timing dependencies because it usually affects physical *interrupt lines*, which are oftentimes shared among multiple interrupt sources.