2014 IEEE International Conference on High Performance Computing and Communications (HPCC), 2014 IEEE 6th International Symposium on Cyberspace Safety and Security (CSS) and 2014 IEEE 11th International Conference on Embedded Software and Systems (ICESS)

A Temporal Partition-based Linux CPU Scheduler

Xingliang Zou+, Albert M. K. Cheng, Yu Li
Department of Computer Science
University of Houston
Houston, USA
xzou@uh.edu, cheng@cs.uh.edu, liyupku2000@gmail.com

Yu Jiang+
School of Computer Science and Technology
Heilongjiang University
Harbin, China
jiangyu@hlju.edu.cn

***Abstract*—Resource partitioning is an important research area in real-time virtualization. The algorithm that maps a virtual resource to a portion of a physical resource is part of the research. When the resource refers to the CPU, allocating CPU time to different virtual partitions is accomplished by the CPU scheduler. This paper reports the implementation of a temporal partition-based CPU algorithm in a general Linux kernel. The implementation is expected to be a prospective testbed for a set of new algorithms for research on resource isolation, virtualization, and fault tolerance.**

Keywords—Fault Tolerance; Isolation; Partitioning; Real-Time; Resource Temporal; Virtualization

I. Introduction

A real-time task is distinguished from general tasks by its timing properties: successfully finishing a real-time task requires obtaining the result not only correctly, but also timely, which means before a specified deadline, even with particular restrictions on delay and timing jitters. Scheduling real-time tasks for execution in a timely manner has been intensively studied. Abeni and Buttazzo [4] have proposed the Constant Bandwidth Server (CBS) scheduling framework for certain kinds of Continuous Media applications as a means of achieving two goals: per-thread performance guarantees and inter-thread isolation. Lipari and Baruah [3] have later proposed an extension to the CBS framework, which permits the partitioning of the set of threads comprising the system into subsets representing individual applications, and extends timeliness guarantees to these applications as well. Under the bandwidth server control, threads have dedicated access to CPU. Åsberg *et al.* [5] unleashed the temporal isolation components research on AUTomotive Open System ARchitecture (AUTOSAR). Inam *et al.* [6] discussed the concept of how virtual nodes are applicable for ProCom, AUTOSAR and Architecture Analysis and Design Language (AADL).

Feng *et al.* [7] investigate an approach for temporal resource partitioning to implement in the open system environment [Deng and Liu, 13], an idea where a physical resource must be time-shared by different task groups and each task group is scheduled as if it has exclusive access to resources without interference from other task groups. A hierarchical two-level scheduler is used in the resource partitioning model. The resource level scheduler divides the shared resource into temporal partitions with predefined constraints and requirements. A task group runs in the partition it belongs to, and each partition schedules the tasks in it with a task-level scheduler. Li *et al.* [10] extend Feng's [9] regularity-based resource partitioning model from a single resource to a uniform multiresource platform. Li and Cheng [11] later extend their work to derive an optimal SAA (Static Approximation Algorithms) called Magic7 for resource level scheduling. They give a resource partition model and prove that the schedulability bound of any feasible SAA is at most 0.5, and their Magic7 optimal algorithms improve the resource utilization by 10% or more in simulations.

However, the partition-migration overhead is not taken into account in their simulations. In this paper, we implement this hierarchical scheduling algorithm in the Linux kernel 3.5 to further explore the performance of Magic7, and provide a research platform for testing theories and algorithms for Dynamic Approximation, resource isolations, fault tolerance, etc. Dynamically changing partition mapping and migrating tasks across partitions are also implemented.

The rest of this paper is organized as follows. Section II reviews the related work, including the work in virtualizations and the Linux CPU scheduler. We then describe our implementations, and the preliminary experiments in Section III. Finally, we overview our work and draw the conclusions.

II. Related Work

A. Open Systems

The open system problem was first researched in Deng and Liu [13] where tasks are assumed to use the Liu and Layland task model [2]. Feng and Mok [8] proposed an alternative approach. The resource allocation problem was treated as the scheduling of the individual task on a dedicated but virtual processor, and the dedicated processor can run at a non-constant speed. They introduced the concept of an RTVP (Real-Time Virtual Processor). The speed variation of a virtual processor in RTVP is constrained by a jitter boundary. RTVPs can be implemented by splitting and scheduling a physical processor time into partitions respecting the jitter boundary of a RTVP with each partition consuming a specified fraction of physical processor time. RTVP does not require the knowledge of the scheduling of the tasks. A task-level scheduler that schedules tasks successfully on a dedicated physical processor will schedules the tasks successfully on a RTVP as long as the RTVP has same capability as the physical processor.

B. Virtualizations

Since computer technologies have developed rapidly for dozens of years on both software and hardware, large-scale systems with various resources are very popular now, and many tasks are deployed on these systems in order to utilize the resources as fully as possible. Virtualization technology is used to separate resources that software requires to their physical

The work in this paper is supported in part by the United States National Science Foundation (NSF) under awards No. 0720856 and No. 1219082, and the State Scholarship Fund of China under award No. 201308230034.
+ Corresponding authors.

counterparts. Virtualization, in computer area, is a term that refers to various theories and technologies of creating a virtual version of a physical object, such as virtual hardware, operating system, particular kind of devices like storage, network and so on, and other resources[14]. We restrict the scope to CPU in this paper.

CPU virtualization consists of application virtualization and processing virtualization. *Application virtualization* breaks the dependencies between applications, the operating system (OS) and the hardware that hosts the OS. Examples are XenApp [16], App-V [18], ThinApp [20] and AppZero [15]. *Processing virtualization* is a range of technologies that makes many computers to appear like a single computer (such as a cluster) or a single computer appear like many computers (such as virtual machines). Examples are XenServer [16], VMware vSphere [20], Hyper-V[18]. There are two types of virtualization: full virtualizations and paravirtualizations. *Full virtualization* completely simulates the hardware. The OS that running on it is not aware of the virtualization. *Paravirtualization* does not offer complete simulations of hardware. The hosted OS is typically modified to support paravirtualizations. A *hypervisor* is a virtualization software layer that decouples the OS and the applications from the physical hardware. There are two types of hypervisors: Type 1 hypervisors are also known as bare metal hypervisors, and run directly on the hardware; Type 2 hypervisors are known as hosted hypervisors that run on a host OS and provide virtualization services such as CPU, I/O and memory.

A Real-time system is outlined by its timing requirements on the execution of tasks. The hypervisor used in the Real-Time systems is distinguished from the one that used in a general environment. One of the most significant differences is the strong spatial and temporal isolations between applications even hardware in a real-time system's hypervisor. There are several real-time virtualization products in the market: WindRiver Hypervisor [21], Green Hills INTEGRITY Multivisor [17] and Pike OS [19].

C. The Linux CPU Scheduler

Linux has been a highly popular and evolved operating system for years. When we mention Linux in this paper, it refers to Linux 3.5 unless we indicate otherwise. Linux has a time sharing pre-emptible CPU scheduler. It uses the term schedule class to denote different scheduling modules that use different algorithms. The word class is almost the exact as the semantics that are used in programming languages such as C++. There are four classes in the Linux system: Stop, Real-Time, CFS and Idle classes. All tasks, equivalent to processes, are classified into one of the four classes. The Stop and Idle classes contain one task only for each. The Real-Time Task class implements two algorithms, FIFO and Round Robin (RR). The CFS Task class implements an algorithm called Complete Fair Scheduler for non-real-time tasks.

The preemption of the Linux CPU scheduler is based on tasks' priorities. Linux has a priority array with 140 priorities, and the lowest number means the highest priority. 100 priorities (0 to 99) are assigned to real-time tasks. A real-time task is queued in one of the 100 arrays upon its priority. A bitmap is created to indicate the arrays that have outstanding tasks. Nice value of a non-real-time task is converted to the value in the range of 100 to 139, and also be mapped to the range 0 to 39 which is called user priorities. The user priorities are used to calculate the run-time priorities by CFS algorithm which is the key of a red-black tree that contains all the non-real-time tasks in the system, and the leftmost leaf is the one that has the highest priority to be picked up for executing. By using the priority bitmap and red-black tree, Linux achieves an $O(1)$ scheduler.

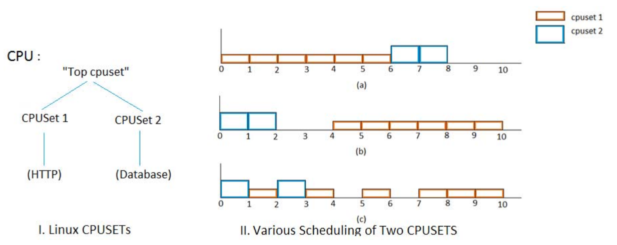

Fig. 1. Linux Control Groups on CPUSETs and various scheduling

The PER-CPU variable is an important Linux kernel feature. When a PER_CPU variable is defined and declared, the compiler makes sure that each processor on the system gets its own copy of that variable. Accessing of a PER-CPU variable requires no locks that cross over processors. An intensively accessed PER-CPU run queue (TABLE I) is employed by the Linux scheduler to manage all tasks that allocated to a particular CPU. All four schedule classes have their own entries in the run queue by which the scheduler manages the entire lifecycle of a task including arrival, activating, inactivating and dispelling. The scheduler relies on the run queue on its most important job, picking the next task to run when it is activated for scheduling. The run queue is the fundamental brick and stone of the scheduler and currently stores variously necessary entities of domain control, load balance (bandwidth) control, etc. that hinders a clean and effective hierarchical solution from deploying on top of it.

The Linux CPU scheduler was barely a task scheduler before the adoption of Group Scheduling, which is showed in Figure 1.I. The Control Groups provides a mechanism for aggregating sets of tasks and their future children into hierarchical groups on demand. A group of tasks shares a proportion of the total CPU time (bandwidth). Unlike the temporal partitioning we propose in this paper, the Linux Control Groups distributes the CPU time to groups by a means of statistics. The concerns of this technology are to share the CPU bandwidth statistically. The timing jitters of the tasks' executing instances are not constrained, and the technology is hence not suitable for certain types of real-time applications.

III. Temporal Partition-based CPU Scheduler

Consider the example in Fig.1.I, we assume CPUSET 1 is assigned 60% CPU time, and CPUSET 2 is assigned by 20%. The existing Linux scheduler schedules these two CPUSETs with multiple possibilities (Fig.1.II). The response time of CPUSET 2 is in the range of (0, 8). The performance of the tasks within these CPUSETs is thus unstable. Even every possible scheduling out of the three has its merits and fits for some particular applications, the problem is that an application cannot be guaranteed to be scheduled by its desired manner.

There are intensive research on resource partitioning and partition scheduling [1] [7] [9] [10] [11], where a resource partition property called regularity has been proposed. Li and Cheng [11] have proposed a novel algorithm, Magic7, for scheduling on a special resource partition with regularity equal to 1, where the partitioning requires the CPU time it gets to be distributed as evenly as possible. Recall the RTVP and RRP (Regular Resource Partition) model, they both require the boundaries at the integer number of a time unit, which is not implemented in general hypervisors. The real-time hypervisors somehow implement their own temporal real-time characteristics, and their temporal scheduling are in a fashion of Round Robin, which is less complicate and less efficient than Magic7 and other SAA algorithms for RRP model. Beside, those real-time hypervisors are commercial products, hence less documented and opened to be revised. This is a part of the motivation of the work we present in this paper. For the sake of code maintaining with the evolutions of Linux, we have virtualized the CPU in the scheduler by dividing its computation time into partitions in the sense of TDMA (Time Division Multiple Access) to CPU. The isolation property between partitions is guaranteed by Definition 4.2. We have implemented a straight forward version (in subsection A.) and an improved version (in subsection B.).

We add a linked list in the run queue structure to manage the partitions that are partitioning the CPU (*struct sched_vp *vps* in Table I). Per the output of the partition scheduling algorithm (Magic7 for instance), several entries serve together to one partition and are called to form an entry-set. An entry-set contains the information of a partition including the partition ID and current running mode, *active, standby* or *surpassing* exclusively, of the partition. Processes of a partition run on any entry of the same entry-set. Entries in the same entry-set shares the same set ID which is the same as the partition ID. Each entry also records its lifetime in ticks.

Keeping the scheduler in *O*(1) time complexity is critical. We use a red-black tree to trace the normal processes and the same arrays for real-time processes that belong to a partition. Since the timing is changed due to the partitioning, we also create a virtual clock in each partition. An EDF(Early Deadline First) class is developed on top of the real-time class. We also create two system calls to facilitate the partition creating, deleting, migrating and scheduler settings.

In the *sched_tick*() entry, we decrease the remaining life *current_partition*, which denotes the active partition, and switch to next partition when it consumes all the life and select a task in the new *current_partition* to preempt the current task. When the *scheduler*() entry is activated to select the next task to execute, we find it in *current_partition*.

A. Strict Temporal Partitionning

The strict temporal partitioning switches partitions strictly at the boundaries. It is straightforward and demonstrates the temporal partitioning of the CPU time. The timing characteristics show a smooth and stable output of CPU computation bandwidth on a specified partition, and have potentials for the timing sensitive real-time tasks which have less demand on CPU bandwidth.

TABLE I. PART OF THE STRUCT RQ

Member	Description
unsigned long cpu_load[]	CPU loading
struct cfs_rq *cfs	runqueue of CFS class
struct rt_rq *rt	runqueue of real-time class
struct task_struct *curr, *idle, *stop	task pointer
int cpu	processor id of this runqueue
struct root_domain *rd	domain is a concept used in SMP scenario
*struct sched_vp *vps*	*a list of virtual patitions*

We give a set of formal definitions following the literal descriptions here. Definition 4.1 defines a set of sequential time slices which repeats itself by a period. Theses time slices are dived into groups and used by a partition. Definition 4.2 tells that a taskset is running exclusively in a partition. Definition 4.3 describes a concept which is essential in this paper. Observation 4.4 and the respective explanations show the importance of the scheduling interval and its jitter.

Definition 4.1 *A virtual CPU partition assignment $\mathcal{P}$ refers to the scheduling of a set of partitions during a period. Any of these partitions can be split into sub-partitions, and all the sub-partitions in the system are executed exclusively. $\mathcal{P}$ is formally defined as a tuple (S, P) where P is the period of $\mathcal{P}$, and $S=\{\langle S_i^j, P_i^j\rangle$: $S_i^j \geq 0$, $S_i^j+P_i^j \leq S_i^{j+1} < P$, $S_i^j < S_k^l$ or $S_i^j > S_k^l+P_k^l$, $i \neq k$ or $j \neq l$, $(0 < i, k \leq N)$, $(0 < j, l \leq N_i)$ $\}$. S_i^j and P_i^j are the beginning and length of the j-th section of partition i. N is the number of time slices. N_i is the number of time slices that used by the i-th partition.*

Definition 4.2 *$\mathcal{T}$ is an aggregation of sets of tasks running on a $\mathcal{P}$, $\mathcal{T} = \{\langle T_i^j\rangle, 0 < i \leq N, 0 < j \leq M_i\}$. One set of tasks runs inside a partition: T_i^j runs in some $\langle S_i^k, P_i^k\rangle$, $0 < k \leq N_i$. M_i is the number of task runs on the i-th partition.*

Definition 4.3 *Schedule interval V_i^j of T_i^j is defined as the time elapsed between the last time the task was scheduled and time it is scheduled again.*

Observation 4.4 *Jitter, the deviation of schedule interval in our temporal partition-based algorithm could be smaller than that in the Linux group scheduling algorithm.*

Explanation *In our algorithm, V_i^j has two contributors: $V_i^j = V_{sw}{}_i^j + V_{intra}{}_i^j$, where $V_{sw}{}_i^j$ is brought by partition switching, $\min(P_i^j) \leq V_{sw}{}_i^j \leq P$, $j=1, 2, \ldots, M_i$, and $V_{intra}{}_i^j = \sum_{k=1}^{M_i} V_i^{k,j}$, where $V_i^{k,j}$ is the sub-interval resulting from T_i^k. In the Linux group scheduling, $V_i^j = V_{gs} + V_{inter}{}_i^j$, where V_{gs} is caused by group bandwidth control, and $V_{inter}{}_i^j$ is caused by all others tasks in the system. Jitter in group scheduling tends to be greater.*

Unlike simulations, frequent switching of virtual partitions causes heavy overhead. Our implementation uses a Linux tick as the minimum unit. A subsidiary red-black tree and real-time task arrays are attached to each partition to keep the tasks that belong to the partition in order to maintain the *O*(1) scheduler. Preliminary experiments are performed on behalf of our implementation in Ubuntu 12.10. The comparisons are done between our implementation and the Linux group scheduling.

TABLE II. RUN TIME(COUNT =1,000,000,000)

Time Cost	**partition_1**	**partition_2**
sec:usec	2:172750	14:410219

In our experiments, two tested virtual partitions have the lengths of 13 and 2 ticks, respectively, and two Linux CPU groups are created with the same proportion. The target of the experiments is to measure the runtime cost with the different system configurations. We run an interfering task to the CPU:

void burn_cpu(int count) {for(int i=0; i< count; i++);}

From Table II, we can see that the task's running time differs proportionally to the length of the partition on which it runs. However, it is not precisely 2/15 to 13/15 which are the reciprocals of the partition lengths. Therefore we are here observing the partition switching overheads as well as the task switch overheads. Notice that the task running on partition 2 is switched more times than that of the task on partition 1.

In another set of experiments, we run the same interfering task to the CPU in partition 1 and run a set of tasks in partition 2 simultaneously. Then repeat the same experiment to Linux group 1 and group 2. From Fig. 2, we find that both schedule interval and task running time vary widely when using the Linux group scheduling. Meanwhile, the variations are in simple patterns when using our implementation.

B. Relaxed Temporal Partitionning

The restrict implementation suffers from the efficiency problem when there are plenty of idle times in the partitions because the partitions keep active and occupy CPU even they have no ready tasks waiting for executing. We then implement an improved version. Unlike the PikeOS having foreground partitions and background partitions, there are no background partitions in our implementation and all partitions are equal to others. When there is no ready task in the *current_partition*, we put the partition into the *standby* mode, and let the next partition goes to *surpassing* mode and is eligible to use the slack times. However, once there is a task becomes ready in the *current_partition*, it returns *active* immediately and preempts any running task that does not belong to it.

The enhancement improves the performance in great scales since it eliminates the slack times. Most importantly, it remains fast reactive to tasks that assigned to *current_partition*.

IV. CONCLUSION AND FUTURE WORK

We describe our implementations of a temporal partition-based Linux CPU scheduler. We also discuss the timing jitter which is critical for a smooth output of CPU computation capabilities with experimentations. We have implemented an EDF policy in the task scheduler, and an EDF acceptance test function. However, a more sophisticate clocking mechanism would benefit the scheduler more and is under developing. We are also considering multicore implementations that requiring taking into considers the load balancing.

The work we've done in this paper and our ongoing enhancements to the existing implementations allow us to study the performance of resource-level task scheduling in the Regularity-based Resource Partitioning Model with newly developed first-ever partitioned scheduling techniques [12]. Furthermore, new extensions also make it possible to investigate the potential faults occurring at the physical resource(s) on virtual resource(s). A better implementation that built on top of the run queue is a worthy of hard work.

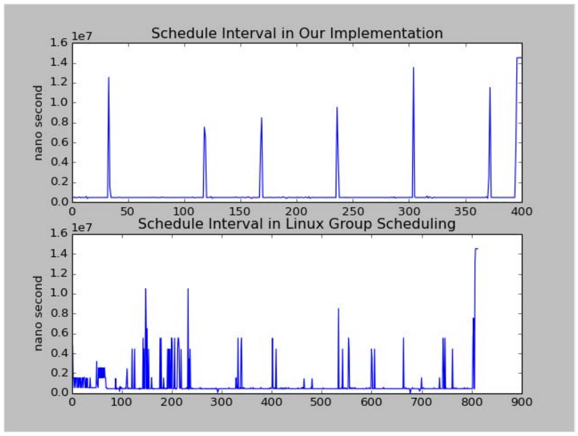

Fig. 2. Schedule intervals

REFERENCES

[1] A. K. Mok and X. Feng. Towards compositionality in real-time resource partitioning based on regularity bounds. RTSS, 2001.
[2] C. L. Liu, J. Layland, Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. Journal of the ACM 20, PP. 46-61.
[3] G. Lipari, S. Baruah, A Hierarchical Extension to the Constant Bandwidth Server Framework. RTAS 2001.
[4] L. Abeni, G. Buttazzo, Integrating Multimedia Applications in hard Real-Time Systems. RTSS 1998.
[5] M. Åsberg, M. Behnam, F. Nemati, and T. Nolte, Towards Hierarchical Scheduling in AUTOSAR, in ETFA 2009.
[6] R. Inam, J. Mäki-Turja, J. Carlson, and M. Sjödin. Using temporal isolation to achieve predictable integration of real-time components. ECRTS10 WiP Session, July 2010.
[7] X. Feng, A. Mok, D. Chen, Resource Partition for Real-Time Systems. RTAS 2001.
[8] X. Feng, A. Mok, Real-time Virtual Resource: a Timely Abstraction for Embedded Systems. The Second ICES, Lecture Notes in Computer Science, LNCS 2491, pp. 182-196.
[9] X. Feng, Design of real-time virtual resource architecture for largescale embedded systems. Ph. D. dissertation, Department of Computer Science, The University of Texas at Austin, 2004.
[10] Y. Li, A. Cheng, A. Mok, Regularity-based Partitioning of Uniform Resources in Real-Time Systems. RTCSA 2012.
[11] Y. Li, A. Cheng, Static Approximation Algorithms for Regularity-based Resource Partitioning. RTSS 2012.
[12] Y. Li, A. Cheng, Partitioned Scheduling of Transparent Real-time Virtual Resources. Submitted to ECRTS 2014.
[13] Z. Deng and J. Liu. Scheduling Real-Time Applications in an Open Environment. RTSS 1997.
[14] http://en.wikipedia.org/wiki/Virtualization/
[15] http://www.appzero.com/
[16] http://www.citrix.com/
[17] http://www.ghs.com/products/rtos/integrity_virtualization.html/
[18] http://www.microsoft.com/
[19] http://www.sysgo.com/
[20] http://www.vmware.com/
[21] http://www.windriver.com/products/hypervisor/