Towards Implementing Multi-resource Server on Multi-core Linux Platform

Rafia Inam, Joris Slatman, Moris Behnam, Mikael Sjödin, Thomas Nolte
Mälardalen University, Västerås, Sweden
Email: {rafia.inam, moris.behnam, mikael.sjodin, thomas.nolte}@mdh.se

***Abstract*—In this paper we present our ongoing work on implementing the multi-resource server technology in the Linux operating system running on multi-core architectures. The multi-resource server is used to control the access to both CPU and memory bandwidth resources such that the execution of real-time tasks become predictable. We are targeting Legacy applications to be migrated from single to multi-core architectures. We investigate the available techniques and mechanisms that can support our multi-resource servers and we discuss the potential problems that needed to be tackled considering the requirements of legacy applications.**

I. Introduction

Scheduling of real-time tasks on multi-core architectures is inherently unpredictable, and activities that are located in different cores can still interfere with others in an unpredictable manner hence imposing negative impact on system performance and real-time guarantees. A main source of such unpredictable negative impact is the contention for shared physical memory. In commercially existing hardware, there are currently no mechanisms that allow a core to protect itself from negative impact if another core starts stealing its memory bandwidth. Hence, there is a need to develop software technologies to track, and eventually police, the consumed memory bandwidth in order to achieve predictable multi-core software.

We focus on the problem of supporting migration of real-time legacy (single-core) systems to multi-cores without adding negative or unpredictable performance penalties using Commercial Off-The-Shelf (COTS) hardware [1]. For the single core architecture we have proposed the use of a two-level hierarchical scheduling framework, considering only the CPU resource, to support the execution of legacy applications sharing the same CPU [1]. However, when considering the multi-core architecture, the memory bandwidth should be considered to guarantee predictable performance of legacy applications that are located in different cores. To achieve this goal, we have proposed the Multi-Resource (MR) server technology [2] that allows a subsystem (i.e. legacy application tasks allocated to a server) to be tested and verified independently of other subsystems that will share the same hardware in the final system.

This work is supported by the Knowledge Foundation (KKStiftelsen), via the research programme PPMSched.

[1]Legacy systems are systems which have been developed and maintained for many years. The reuse of legacy code can have enormous cost, reliability, and safety benefits. The trend is observed in embedded software appliation domains such as automotive, consumer electronics and avionics.

While the focus in [2] was to provide an analysis framework to assess the composability of subsystems, in this paper, we present our on-going work towards implementing the MR server technology in the Linux operating system. We review some existing techniques, implementations and mechanisms that might be used for this purpose and we discuss their suitability to our targeted systems.

II. The Multi-Resource Server

In this section we present the multi-resource server. To bound the memory accesses for a server, we need some mechanisms of *memory throttling* to control the activity of memory requests of the server such that each server can access the memory according to its pre-reserved memory bandwidth limit. Later in this section, we first describe the means to implement memory throttling by measuring the memory bandwidth, and then we explain how we can monitor and enforce the consumed memory bandwidth m_s.

A. The concept of multi-resource server

Our scheduling model for the multi-core platform is a two level hierarchical scheduling framework which can be viewed as a tree of nodes, with one parent node and many leaf nodes per core, as illustrated in Figure 1. The parent node is a *node scheduler* and leaf nodes are the subsystems (servers). Each subsystem contains its own internal set of tasks that are scheduled by a *local scheduler*. The node scheduler is responsible for dispatching the servers according to their bandwidth reservations (both CPU- and memory-bandwidth). The local scheduler then schedules its task set according to a server-internal scheduling policy. For both levels of schedulers, including the node and server level, the *Fixed Priority Preemptive Scheduling (FPPS)* policy is assumed.

Each server maintains two different budgets; one is the CPU-budget and the other one reflects the number of memory requests. The server is of periodic type, meaning that it replenishes both budgets to the maximum values periodically every P_s period and this is done as $q_s := Q_s, m_s := M_s$. Here Q_s is the amount of CPU-time allocated, and M_s is the number of memory requests in each period. During run-time each server is associated with two dynamic attributes q_s and m_s which represent the amount of available CPU- and memory-budgets respectively.

Each core has its own node scheduler which schedules the servers on that core. The node scheduler maintains a state of each server on the core; the state is either *runnable* or

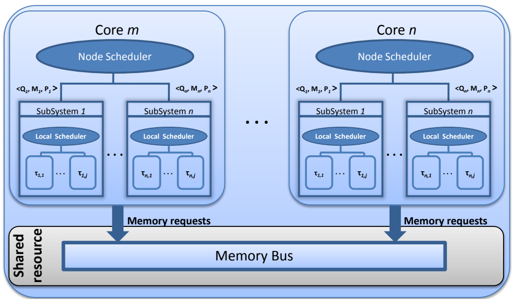

Fig. 1. A multi-resource server model

suspended. A server is in the *runnable* state if it still has allocated resources to use.

A server which depletes any of its resources is placed in the *suspended* state, waiting for replenishment which will happen at the beginning of the next period. The node scheduler selects a *runnable* server for execution. When a server S_s is executing, q_s decreases with the progression of time, while m_s decreases when a task in the server issues a memory request using the shared memory-bus.

A scheduled server uses its local scheduler to select a task to be executed according to the local scheduling mechanism. When a task τ_i, running inside a server S_s, issues a memory request, the associated core is stalling until the cache-line is fetched from memory. A higher priority task can pre-empt the execution of lower priority tasks but not during the core stalling. A memory request is managed even if the budget of S_s has depleted. When the memory budget is depleted, the remaining CPU budget will be dropped and the state of the server is changed to *suspended*.

For the CPU part of the server, any type of periodic server (i.e. idling periodic [3] or deferrable servers [4]) could be used to determine the consumption of allocated CPU budget of each server. The memory part of the server is modelled as a deferrable server since the memory-budget is consumed only when a memory request is made. More details on the execution of a multi-resource server are explained in [2].

B. Online monitoring and policing of m_s

In many cases, a continuous determination and tracking of the consumed memory bandwidth is very difficult without using a dedicated external hardware that monitors the memory-bus. Since we target the use of standard hardware, we use a software-based memory throttling technique.

Most modern processors host a range of Performance Monitor Counters (PMCs) which can be used to infer the amount of resources consumed. The hardware performance counters are registers frequently used for off-line and on-line performance analysis without slowing down the system. The counters are used to track certain low-level operations or events within the processor accurately and with minimal overhead. The performance counters can be used when evaluating performance of a computer system. For example, Eranian in [5] describes the use of performance counters to measure four interesting memory-related metrics: measuring cache misses, measuring memory bandwidth, measuring latency and access locality for the x86 platform.

We will implement servers that *enforce/police the consumed memory bandwidth* and thus accurate and non-intrusive estimates of the bandwidth consumptions become even more important. For policing purposes, using interrupts that will be generated when cache misses occur could give the most accurate approach for accounting of the consumed bandwidth.

C. Online monitoring and policing of q_s

The CPU-budget consumption of the servers should be monitored to properly handle server budget depletion (the tasks of the server should execute until its budget depletion).

In general, there are two methods to achieve this; the first one is based on decreasing q_s of each active server at every system tick and to check if $q_s = 0$. For example, the works in [6], [1] use this method. The second method is based on using a timer that generates an interrupt after q_s time units whenever a server becomes active and the value of q_s is updated whenever the server is preempted. Implementation examples that use this method are presented in [7], [8], [9].

III. Hierarchical Scheduling in Linux

In this section we investigate some available implementations to incorporate the multi-resource server for real-time applications. We present our choice of using the Linux operating system and its real-time patch to schedule hard real-time tasks. We review the existing work in implementing real-time hierarchical scheduling using the Linux operating system, and software throttling implementations using performance counters to limit memory bandwidth access of the server. We present their shortcomings with respect to providing memory-aware hierarchical scheduling.

The motivation of using Linux is that it is an open source kernel under the GNU General Public License and it is developed worldwide, and the use of Linux in embedded systems is increasing rapidly [10]. The stock 2.6 kernel version allows pre-emption when Linux is running in user space, when 1) Linux returns from a system call or an interrupt back to user space, 2) the systems blocks on a mutex, 3) the system calls yield. This makes the kernel suitable for running soft real-time tasks only [11] but it lacks the hard real-time tasks support.

A. The real-time support

A lot of works have been done to support hard real-time behaviour in Linux. Some options are discussed below:

1) The real-time patch (RT_PREEMPT): The real-time behaviour can be added to Linux by patching the kernel with the real-time patch called RT_PREEMPT [12]. This patch allows for running hard real-time tasks by making the Linux kernel a fully preemptible kernel, because it allows for preemption by higher priority tasks outside code that is not protected by a special kind of non preemptible spinlocks. Furthermore, there are several tools and built-in test-and debugging mechanisms that can be used for measuring the performance of the preemptible kernel.

The scheduler of the kernel is constructed in a modular fashion and it consists of several scheduling classes. Three native Linux scheduling classes are used to schedule the tasks. The first one is the `SCHED_NORMAL` that is a non real-time scheduling class. The other two scheduling classes, `SCHED_FIFO` and `SCHED_RR` are real-time scheduling classes that schedule real time tasks. Each scheduling class has its own priority. The real time scheduling classes have a higher priority than the `SCHED_NORMAL` class. `SCHED_RR` implements a round robin algorithm. It is an enhancement of the `SCHED_FIFO` (first in first out) policy and it schedules tasks according to the time slice. In this manner Fixed Priority Pre-emptive Scheduling (FPPS) is supported for the real-time tasks.

2) SCHED_DEADLINE: Another potential solution we investigated is SCHED_DEADLINE [13]. It is a non intrusive modification to the stock Linux scheduler done by the company Evidence. It provides a scheduling class based on the real time scheduling algorithm Earliest Deadline First (EDF). The recent version of SCHED_DEADLINE supports cgroups thus it presents a kind of real-time server-based implementation to support the temporal isolation property.

3) ExSched: ExSched [9] is a framework to develop real-time schedulers without the need to patch or modify the main kernel itself. Using ExSched the scheduling policies can be implemented as external kernel modules for different operating systems. Its main advantage is its portability. It can be connected to different operating systems, currently it is working with Linux and VxWorks operating systems. ExSched provides the traditional scheduling algorithms FPPS and EDF scheduling. It provides implementations for global, partitioned, and semi-partitioned scheduling on multi-core platforms.

B. Hierarchical scheduling support

Here we investigate different implementations supporting server-based scheduling in Linux.

1) Linux Control groups: Linux provides a functionality to group together tasks in a hierarchy called *Linux Control groups* or simply *cgroups* as a Linux kernel feature. This mechanism facilitates allocation of a resource or a combination of resources such as CPU runtime, system memory and or network bandwidth to the hierarchy [14]. A cgroup can be configured with a quota for every resource where its hierarchy is attached to. A task set can be attached to a particular cgroup. In this manner groups of tasks can be forced to use only their own share of the resources. Cgroup usage can be monitored and changed during runtime.

Cgroups provide a fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources. It is also possible to divide hardware resources among tasks and users, that increases the overall efficiency [14]. The CPU resource also called *cpu* can be attached to a cgroup. It uses the `SCHED_CFS` or `SCHED_RT` schedulers to divide the CPU time (CPU bandwidth) proportionately between cgroups (groups of tasks). The `SCHED_CFS` divides the proportional share of the scheduler depending on the priority/weight of the task assigned to cgroups. The `SCHED_RT` is used for real-time tasks in which the runtime execution and period of real time tasks can be specified in microseconds as `cpu.rt_running_us` and `cpu.rt_period_us`.

Cgroups are good for system administrators by providing interfaces to configure the system, but they provide limited functionality for application developers and hence not suitable to implement a real-time two-level hierarchical structure. To execute the hard real-time tasks preemtibaly, we need some real-time patch like RT_PREEMPT. But this patch is not yet supported by the cgroups. Using cgroups, only a limited amount of memory RAM can be allocated to a task. Another limitation in using cgroups to implement a two-level hierarchical system is a limited assignment of subgroup (server in our case) periods. The hierarchical structure makes the cgroups less flexible. This hierarchical organization can be compared to the hierarchical organization of processes in Linux. Child groups inherit some of the attributes of their parents. Hence the subgroup must have a period smaller or equal than its parent cgroup's period [15].

Considering the limitations of the Linux control groups we move our attention to find a better solution for our implementation.

2) CPU-based server implementations: Many server-based techniques for CPU time have been completely implemented for single-core and multi-core platforms using different operating systems like [16], [17], [7], [6], [1], [8], [9].

A kernel-level implementation to support partitioning and hierarchical scheduling in ARINC 653 for Linux is provided in [18]. It uses a partitioned scheduler at the global-level to schedule partitions, and a local-level process scheduler is used to schedule processes within each partition using. A timer is used to trigger the scheduling events based on the scheduling table. A POSIX compliant implementation of sporadic server for the Linux kernel is provided in [19].

For ExSched, a Hierarchical Scheduling Framework (HSF) plugin has been implemented as an HSF kernel module, which provides a two-level hierarchical scheduling implementation for temporal isolation on single core processors. The HSF-FP plugin module can communicate with the core module of ExSched through call-back functions. These call-back functions are used to tell the HSF-FP plugin about different events like task releases, task executions and task completions.

C. Memory-based server implementations

Memory servers are used to limit memory requests generated by tasks of the server. Since this issue has arisen with the advent of multi-core platforms, not much work is found in this area. A software-based memory throttling approach is used to limit the memory-accesses of the interfering cores. Hardware performance counters are used to measure last level cache misses to measure memory accesses and the scheduler is updated to enforce the memory throttling.

Recently a server-based approach to bound the memory load of low priority non-critical tasks executing on non-critical cores was presented by Yun et al. i [20] for an Intel architecture running Linux. In their model, the system consists of one critical core executing a critical application and a set of non-critical cores executing non-critical applications. One memory server is implemented on each non-critical core to limit memory requests generated by tasks that are located on that core. Hence the interference from other non-critical cores on the critical core is bounded.

The servers are implemented on Linux using cgroups in [20]. Since in this approach only one server per core is considered, one cgroup (without any hierarchy of cgroups) is used on each core and there is no need to assign periods to the child groups which makes the problem easier. This work has been extended in [21] by using a memory reclaiming technique when a core is not fully utilizing its allocated memory budget. The memory reclaiming algorithm is presented and is implemented in Linux kernel 3.6, as a dynamic loadable kernel module.

IV. Discussion

In this section we will discuss the suitability of using the existing techniques and implementations presented in the previous section to implement the MR servers.

Looking at the implementations of memory or CPU resources servers based on Linux cgroups, we believe that cgroup is not a good solution due to its inherited limitations as explained earlier. In addition, adding both resources, memory and CPU, for servers implementations might add more limitations especially when considering the effect of each resource on the other. For example, when any of the resources budget depletes the corresponding server should be suspended. For this reason we have decided to implement the server from scratch. This problem was not considered in the memory server implementation presented in [20] because only one memory server is considered in each core which makes the use of cgroup possible. For our case, our legacy applications are executed in different servers and that means multiple servers may share a single core, which interfere with each other through the sharing of both memory and CPU resources. While servers that are located in different cores will interfere with each others through the sharing of the memory bandwidth resource.

The next decision is on which of the Linux real-time extensions that is more appropriate for our legacy applications. ExSched seems to be more attractive since it provides a platform independent solution, which is more appropriate for our targeted legacy applications[2].

Even though ExSched supports a CPU resource hierarchical scheduling framework, the provided implementation is only limited for single core, which must be extended for multi-core to be suitable for the MR server technique. In addition, the implementation of memory part of the MR server must be added and the interaction between two resources (memory and CPU) should be considered which is our current work.

References

[1] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam. Hard Real-time Support for Hierarchical Scheduling in FreeRTOS. In *7th Annual Workshop (OSPERT' 11)*, pages 51–60, Porto, Portugal, July 2011.

[2] M. Behnam, R. Inam, T. Nolte, and M. Sjödin. Multi-core composability in the face of memory bus contention. In *5th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS'12)*, December 2012.

[3] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some Practical problems in Prioritised Preemptive Scheduling. In *Proc. 7th IEEE Real-Time Systems Symposium (RTSS' 86)*, pages 181–191, December 1986.

[4] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-time Environments. *IEEE Transactions on Computers*, 44(1), 1995.

[5] S. Eranian. What can performance counters do for memory subsystem analysis? In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC'08)*, pages 26–30. ACM, 2008.

[6] R. Inam, J. Mäki-Turja, M. Sjödin, S. M. H. Ashjaei, and S. Afshar. Support for Hierarchical Scheduling in FreeRTOS. In *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA' 11)*, France, September 2011.

[7] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril. Towards hierarchical scheduling on top of vxworks. In *Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, pages 63–72, July 2008.

[8] M.M.H.P. van den Heuvel, M. Holenderski, R. J. Bril, and J. J. Lukkien. Constant-bandwidth supply for priority processing. *IEEE Transactions on Consumer Electronics (TCE)*, 57(2), 2011.

[9] M. Asberg, T. Nolte, S. Kato, and R. Rajkumar. ExSched: an external cpu scheduler framework for real-time systems. In *IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA' 12)*, 2012.

[10] R. Love. *Linux Kernel Development 3rd edition*. Prentice Hall, 2006.

[11] Paul McKenney. A realtime preemption overview, 2005. http://lwn.net/Articles/146861/.

[12] C. Hallinan. *Embedded Linux Primer: A Practical, Real-World Approach*. Pearson Education, Inc, 2010.

[13] J. Lelli, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT' 11)*, pages 6–15, July 2011.

[14] M. Prpić, R. Landmann, and D. Silas. *Resource Management Guide - Managing system resources on Red Hat Enterprise Linux 6*. Red Hat, Inc, 2013. https://access.redhat.com/site/documentation/.

[15] Real-Time group scheduling - The Linux Kernel Archives. https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt.

[16] D. Kim, Y-H Lee, and M. Younis. Spirit-ukernel for strongly partitione real-time systems. In *Proc. of the 7th International conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[17] S. Saewong and R. Rajkumar. Hierarchical reservation support in resource kernels. In *Proc. 22th IEEE Real-Time Systems Symposium (RTSS' 01)*, December 2001.

[18] Sanghyun Han and Hyun-Wook Jin. Kernel-level ARINC 653 partitioning for Linux. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*, 2012.

[19] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari. Design and implementation of a POSIX compliant sporadic server for the Linux kernel. In *proceedings of the 10th Real-Time Linux Workshop (RTLWS' 08)*, 2008.

[20] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory-access Control in Multiprocessor for Real-Time Systems with Mixed Criticality. In *Proc. of the 24th Euromicro Conf. on Real-Time Systems (ECRTS' 12)*, July 2012.

[21] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proc. 19th IEEE Real-Time Technology and Applications Symposium (RTAS' 13)*, 2013.

[22] J. Calandrino U. Devi H. Leontyev B. Brandenburg, A. Block and J. Anderson. Litmus^{RT}: A status report. In *9th Real-Time Linux Workshop*, pages 107–1232, November 2007.

[23] M. Mollison and J. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, pages 283–292, April 2013.

[2]A number of additional works have been presented to support real-time in Linux for multi-cores which may impose less overhead than ExSched, e.g., the works in [22] and [23], however, most of these works either require changing of the kernel of Linux or they provide platform dependent solutions.