# ECE 5550G Real-Time Systems
Group Project Specifications

## 1 Project Guidelines

Each project group will consist of two students. For your project you may select from the project topics listed below, or you may propose your own topic. You are encouraged to work on a topic that is related to your research.

## 2 Project Deadlines

There are two project deadlines:

- Friday April 6, 2018 at 11:59pm : Project proposal due: Write a 1-2 page proposal that describes your project. The proposal should include the title of your project, and the names and e-mail addresses of both students. The proposal should describe the project by clearly stating a) the goals (i.e., objectives) of the proposed effort, b) how you to plan to achieve those goals, c) anticipated outcomes (e.g., experimental measurements), d) deliverables (e.g., project report, source code, etc.) and e) a tentative schedule with reasonable milestones toward the submission date. Each of these should be described in a separate paragraph. We will evaluate the proposal and will provide feedback.
- May 8, 2018 at 11:59pm (last day of classes): Project deliverables due.

## 3 Project Topics

### 3.1 Implementation and Evaluation of Real-Time Dynamic Voltage and Frequency Scaling (DVFS) Schedulers

This project involves implementing and evaluating a set of real-time DVFS schedulers (e.g., 2-3) in a simulator or an RTOS. Real-time DVFS is motivated by the property that, a task's power consumption is a function of the processor frequency at which the task is executed. This function is a complex function and is highly dependent of the specific characteristics of the hardware and the task at hand. Broadly speaking, lower the frequency, lower is the power consumption (but higher the task execution time) and vice versa. A task's energy consumption is the product of the power consumption and the task execution time. Thus, in principle, one could run a task at a "sufficiently" lower frequency such that the power consumption is reduced, but the increase in execution time is not large enough to degrade the task timing property (e.g., no deadline is missed) and will also reduce the energy consumption. This has motivated significant research on real-time DVFS scheduling. However, many such schedulers have not yielded significant power and energy savings in practice because of the complex relationship between power and frequency (e.g., they are highly non-linear), and the platform and application dependencies of that relationship. pyTimechart (http://pythonhosted.org/pytimechart) or equivalent online/offline monitoring tool can help visualize that your work is correct (pyTimechart graphs the CPU frequency over time for each CPU if adequately configured).

The project has the following goals:

- Explore how to implement real-time DVFS schedulers It may be especially rewarding to work with Linux's CPUFreq Governor and implement the algorithms on an RTOS.
- Read and understand technical literature about real-time DVFS and implement a set of them (e.g., 2-3).

- Evaluate the scheduler implementations' power and energy consumptions and timing properties.

References:


- [1] An experimental evaluation of real-time DVFS scheduling algorithms. S. Saha and B. Ravindran. Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR '12). 2012. http://dl.acm.org/citation.cfm?id=2367604
- [2] Energy-Aware Scheduling for Real-Time Systems: A Survey. M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. ACM Transactions on Embedded Computing Systems (TECS), Volume 15 Issue 1, February 2016. http://dl.acm.org/citation.cfm?id=2808231
- [3] CPUFreq Governor - The Linux Kernel Archives. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`
- [4] Measuring application power consumption on the Linux* operating system - Intel Developer Zone. `https://software.intel.com/en-us/blogs/2013/06/18/measuring-application-power-consumption-on-linux-operating-system`


## 3.2 Real-time Media Player on ChronOS

This project consists of rewriting an open source multimedia application (e.g., Mplayer, mpv, x264, and VLC) in order to exploit real-time scheduling policies implemented in ChronOS. Multimedia applications are one of the most important category of applications that benefits from real-time scheduling policies. When we are listening a song or watching a video on our portable device or personal computer we are always very disappointed if the play is not smooth. Multimedia applications, together with the operating system do their best to deliver the content at a constant rate to the user. However, because multimedia application don't usually rely on a real-time scheduler many times the audio or video is not delivered as smoothly as expected. The actual performance degradation depends on whether there are other currently running applications in the system.

In order to make multimedia applications run smoothly this project proposes to rewrite an open-source multimedia player to use ChronOS real-time scheduling API and, therefore, avoiding deadline misses. Note that audio and video player are built around a main periodic loop in which the decoding of the audio and video runs and passes the results of this decoding to the audio and video cards respectively.

The project has the following goals:

- Investigate how to rewrite an open-source multimedia application (e.g., Mplayer, mpv, x264, and VLC) in order to exploit real-time scheduling policies.
- Implement a prototype that exploits ChronOS scheduling.
- Evaluate the implementation (with multiple scheduling algorithms).

References:


- [1] http://www.irmosproject.eu/
- [2] Integrating multimedia applications in hard real-time systems. L. Abeni ; G. Buttazzo, Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE. pp 4-13 1998
- [3] Isovic, D.; Fohler, G. "Quality aware MPEG-2 stream adaptation in resource constrained systems", Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on, On page(s): 23 - 32

### 3.3 Implement an optimal multiprocessor real-time scheduler in KairosVM

KairosVM [1] is a real-time hypervisor developed by the SSRG lab. This project involves designing, implementing and evaluating a guest-scheduler overwriting prototype in KairosVM [1], a project aiming to develop hypervisor-level real-time resource management polices. The goals of these policies include ensuring temporal isolation of guest virtual machines and satisfy (guest) application time constraints, in particular, for guest OS/application workloads with dynamically uncertain properties (e.g., on execution times, event arrivals). The aim of the project project is to develop a real-time hypervisor architecture that encapsulates the resource management policies, and an implementation based on Linux-based hypervisors (e.g., KVM).

KairosVM supports guest real-time systems with different real-time scheduling algorithms and this currently cannot be achieved without sacrificing some CPU time. Thus this project involves implementing a solution in KairosVM to overwrite the real-time scheduling decisions taken by the guests, such that the host decides which task is executed in the guest. This would allow guests with different policies to be scheduled on the same machine while using less CPU time than the current solution. Currently, KairosVM schedules its guest virtual machines using EDF and uses the SCHED_DEADLINE scheduling class of Linux [2]. However, EDF is known to have poor performance on multiprocessor systems, especially regarding its utilization bound. Other multiprocessor real-time scheduling algorithms have better theoretical results, like fair schedulers (e.g. PFair [3], BF [4], ...) or RUN [5]. These scheduling algorithms are even optimal for periodic task sets on multiprocessor systems.

Fair scheduling algorithms do not schedule tasks according to priorities but they explicitly require tasks to make proportional progress based on their utilization. The processor time is divided into quanta and the scheduler decides at each quanta which task to execute according to the utilization of all the tasks of the task set. The RUN scheduling algorithms follows a different logic. Its objective is to reduce the problem of multiprocessor scheduling to a set of uniprocessor scheduling problems. Offline, it packs tasks to servers in order to get only one server to schedule. Then, online, the opposite step is done to know where and when each task should be scheduled.

This project involves implementing one of these optimal multiprocessor real-time schedulers such as RUN inside KairosVM. The current implementation of KairosVM has the fundations to ease this implementation. KairosVM already knows which task is being executed inside each guest and also knows their execution time, deadline and period. In KairosVM, each guest task is a scheduling entity associated with its vCPU process. SCHED_DEADLINE picks the scheduling entity with the earliest deadline and the associated vCPU task is executed.

The project has the following goals:

- Implement an optimal real-time scheduling algorithm in KairosVM.
- Evaluate the implementation.

References:


- [1] `http://www.ssrg.ece.vt.edu/kairos`
- [2] `https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt`
- [3] Proportionate progress: a notion of fairness in resource allocation. Baruah, S. K. and Cohen, N. K. and Plaxton, C. G. and Varvel, D. A. Proceedings of the twenty-fifth annual ACM symposium on Theory of computing. 1993.
- [4] Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? Zhu, Dakai and Mossé, Daniel and Melhem, Rami. Proceedings of the 24th IEEE International Real-Time Systems Symposium. 2003
- [5] RUN: Optimal Multiprocessor Real-Time Scheduling via Reduction to Uniprocessor. Regnier, Paul and Lima, George and Massa, Ernesto and Levin, Greg and Brandt, Scott. Proceedings of the IEEE 32nd Real-Time Systems Symposium. 2011.

## 3.4 Implementation of gMUA and Comparison with G-EDF

A number of multiprocessor real-time scheduling algorithms have been recently developed. These algorithms can be broadly classified based on the migration model – i.e., the degree of run-time migration that is allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) full migration, where jobs are allowed to arbitrarily migrate among processors during their execution. This usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors, and a system-wide scheduling decision is made by a single (global) scheduling algorithm at each scheduling event; (2) no migration, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled on their respective processors by processors' local scheduling algorithm, like single processor scheduling; and (3) restricted migration, where some form of migration is allowed e.g., at job boundaries. Example algorithms in the full-migration category include G-EDF or Global EDF (Global Earliest Deadline First) and gMUA (global Multiprocessor Utility Accrual scheduling Algorithm).

It turns out that EDF's optimality on single-processor systems does not hold on multiprocessors. Let $m$ denote the number of processors of a multiprocessor. Thus, $m$ equals 100% of the total processing capacity of a multiprocessor. For multiprocessors, the total utilization demand below which all deadlines are met by G-EDF is approximately $\frac{m}{2}$ – i.e., approximately 50% of the total processing capacity of a multiprocessor. Note that, this utilization bound is 100% for EDF on a single-processor.

This project involves implementation of gMUA which is a global scheduling algorithm based on G-EDF [1]. gMUA uses Time Utility Functions (TUFs) and defaults to G-EDF during underloads and uses the value densities to provide a best effort utility accrual during overloads. The details of gMUA algorithm can be found in [2].

The project has the following goals:

- Implement the gMUA scheduling algorithm
- Compare the performance between G-EDF and gMUA for the under-load and overload scenarios.
- Compute the overhead of both algorithms and the impact on the results.

References:


- [1] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In ECRTS 05, pages 209-218, 2005.
- [2] H. Cho. Utility accrual real-time scheduling and synchronization on single and multiprocessors: models, algorithms, and tradeoffs. PhD thesis, Blacksburg, VA, USA, 2006. `http://scholar.lib.vt.edu/theses/available/etd-09022006-160653/`


## 3.5 Implementation of G-GUA and NG-GUA Multiprocessor Scheduling Algorithms

This project involves the implementation of the GUA and NG-GUA scheduling algorithms. The GUA class of algorithms [1] include two algorithms, namely, the Non-Greedy Global Utility Accrual (or NG-GUA) and Greedy Global Utility Accrual (or G-GUA) algorithms. One way to view these algorithms is as "DASA for multiprocessors" i.e., they maximize total accrued utility and minimize deadlines missed on multiprocessors.

NG-GUA and G-GUA differ in the way schedules are constructed towards meeting all task deadlines, when possible to do so. While NG-GUA constructs a global schedule that defaults to G-EDF's schedulability bound, G-GUA does not default to any algorithm. However, NG-GUA and G-GUA try to maximize the total accrued utility. The greediness in the name of these algorithms describes the tendency of the algorithms to accrue as much total utility as possible. As the names suggest, G-GUA is more greedy for utility accrual when compared to NG-GUA.

The project has the following goals:

- Implement G-GUA
- Implement NG-GUA
- Compare the performance of G-GUA and NG-GUA.

References:

- [1] On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors. Piyush Garyali. Master thesis. Virginia Tech. 2010.
  `http://scholar.lib.vt.edu/theses/available/etd-07222010-114202/`

## 3.6 Comparison and Analysis of Publicly Available Linux-Based RTOS

Four freely available Linux-based RTOS are VT's ChronOS, UNC Chapel Hill's Litmus, Milan Polytechnic's RTAI, and Open RTLinux, maintained by Wind River. These are widely differed, both in design and maturity. Since all were developed for different purposes, they poses a wide range of capabilities and features.

The goal of this project is for your group to compare these four RTOS across a wide range of characteristics. You should address issues including:

- The history and design goals of the RTOS.
- The structure of the system. What modifications were made to the standard Linux kernel?
- The features of the system. What does it do that no other system does?
- How does the system deal with interrupts? How much control does the user have over this? If I decided that my wireless card was less important than my real time task, could I disable handling of that interrupt while my task was running?
- How is process scheduling handled? Is it event based? Quanta based? Both? Are priority queues used, or is some other mechanism available? Is the system Posix-compliant? If not, is any advantage gained by the non-compliance?
- The non-standard APIs the system exposes to the user.
- How does the system scale in a multi-core environment? What areas of the system would limit this the scalability?

In addition, you should thoroughly test the real-time claims of the system by measuring the jitter of the system (the time from the creation of a task to the time when it is placed in the schedule by the system). Based on a statistical analysis of your measurements and an analysis of the system's code, you should determine whether jitter is bounded, and if it is not bounded, what probabilistic statements may be made about it in relation to the jitter of the standard Linux kernel.

## 3.7 Analysis of Linux Preemption Levels

One of the largest detriments to determinism on modern operating systems is the fact that most modern operating systems kernels contain huge chunks of code which do not allow preemption. Depending on the OS implementation, hardware and software interrupts may or may not be handled. This means that the maximum latency for the system to respond to an event, such as thread creation or a software timer, is the length of the longest non-preemptible section. For real-time systems, this obviously creates a problem if the length of the longest section is greater than the minimum time granularity needed for all tasks to meet their timeliness constraints.

The Linux Kernel currently offers four preemption levels: no forced preemption, voluntary kernel preemption, preemptible kernel, and complete preemption. Each of these changes the preemption characteristics of the kernel, but also changes the performance characteristics of the preemptible sections.

The goal of this project is for your group to explore the effect of these various levels of preemption on a real-time system, and their effects on event handling latencies and overall kernel performance. Several questions that your group should address include:

- What changes to the code are made at each level of preemption? Are any broad changes made to the structure of the kernel to cause the changes? Are interrupts handled differently at different levels?
- How is locking handled at each level? What synchronization protocols are used?
- Does changing the level of preemption result in an increase or decrease in determinism? How is this measured? Is there any performance difference between the levels?
- Since it is impossible to create a completely preemptible kernel, there must still be non-preemptible sections in the "completely preemptible" kernel. What sections are there and why would they have been left non-preemptible?
- How should (theoretically) each of these systems scale as the number of cores increases? Where are the system bottlenecks in a multi-core environment?
- How do the four preemption models affect the behavior of a scheduler like DASA-full? What if any changes are needed on the scheduler such that the resulting behavior is as close as possible to the original design?

## 3.8 Evaluating Work Stealing/Sharing Overhead in Linux and Scaling it Up

To support different scheduling policies the Linux scheduler is built with a set of scheduling classes. The most recent Linux kernel supports up to 5 scheduling classes, in decreasing priority order: STOP, DEADLINE, RT, FAIR, and IDLE. Only DEADLINE, RT, and FAIR are usable by the application developer and respectively implements GEDF/EDF/CBS, FIFO/RR, and CFS. (Please refer to the classical Linux literature for more information.) Linux is an operating system for SMP platforms, thus the scheduler deploys techniques to share the workload among processors. To accommodate the workload in the platform each scheduling class implements its own work stealing and/or work sharing technique. Previous simulation (queue theory) and empirical works demonstrated that work stealing and work sharing have different advantages when compared to a global queue solution. In fact a solution that employs a global queue is bottlenecked by the required exclusive access to the global queue when multiple CPUs try to access it. (Imagine that all CPUs want to pick the next task to execute at the same time.) Moreover, theory shows that work stealing is more competitive than work sharing. However, both have practical drawbacks when implemented. For example, in the RT class work stealing requires to access other's CPU run queues, this, in the worst case translates in a O(n) cost every time the local run queue is empty, where n is the number of processors in the system. Finally, some scheduling classes, as FAIR, to reduce the work stealing/sharing overhead, and promote the reduction of cache misses, implements scheduling domains. Scheduling domains groups CPUs together based on the topology, e.g., two CPUs core/threads that shares the same L1 data cache are grouped together. The goal of this project is to evaluate the overhead of work stealing/sharing in Linux's RT scheduling class, to propose possible new heuristics to scale it up, to evaluate the proposed modifications, and to compare with the original implementation. The group should:

- Create different periodic tasksets for sched_test_app. The tasksets should be characterized by long-running, medium-running and short-running tasks. A minimum of 5 tasksets is required, the tasksets should run for a couple of minutes to collect meaningful results. (Note that a common

methodology is to create a taskset with only short-running, another with only medium-running, ..., the other twos mixed.)

- Disable RT throttling or force it to a minimum value. Eventually disable sched_setaffinity in sched_test_app.
- Exploit the Linux tracing framework, ftrace, to measure the scheduling overhead due to work stealing/sharing. An alternative to ftrace is to manually instrument the Linux source code.
- Evaluate the percentage of the time tasks are going to pull vs to push, for different workloads.
- Design heuristics to improve work stealing/sharing. The following ideas can be explored for work stealing: randomize the CPU picking order, use a modulo order, exploit the topological order. Eventually other ideas apply here, like how to minimize the lock contention on each local queue, using more scalable data structures, or using data replication (e.g., multiple queues for work sharing).
- Present a comparison of at least one method to the original implementation on different workloads. Another suggested comparison is with a static placement method (enabled sched_setaffinity in sched_test_app) that should show near 0s overhead for the work stealing/sharing mechanism.

References :


- [1] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. J. ACM 46, 5 (September 1999), 720-748.
- [2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling parallel programs by work stealing with private deques. SIGPLAN Not. 48, 8 (February 2013), 219-228.


### 3.9 Messaging Based Scheduling for Popcorn Replicated-kernel OS (implemented on vanilla Linux)

Some recent research efforts at VT involves work on an extended multikernel OS design [1] called replicated-kernel OS. Popcorn Linux [2] is the implementation of this idea. When a replicated-kernel OS is made to run on a multicore platform each core or group of cores run an OS kernel. However, a user-space application runs as it was running on a traditional SMP operating system, this illusion is called single system image. A replicated-kernel OS can exploit shared memory, but the main idea is that every kernel is the local administrator of a set of the platform resources and memory is not used to shared data or for synchronization amongst CPUs. In this design kernel on different CPUs communicate by message passing, messaging is implemented on shared-memory and this is the only memory that two kernel shared amongst each other. However, on multicore as an optimization shared memory is used by user-level applications. Because kernels running on different CPUs cannot share memory by design one of the fundamental questions is how to implement scheduling in replicated-kernel OSes.

This project should be implemented on vanilla Linux, therefore simulating a replicated-kernel OS in a more familiar environment. The project requires developing a message-passing scheduler algorithm for the Linux's RT scheduling class and breakdown the costs. Refer to Project 3.8 for a description of the Linux's scheduler infrastructure. This scheduling algorithm should establish which messages have to be exchanged between CPUs in order to implement global FIFO (Linux's SCHED_FIFO). Messages can carry tasks or load/other information. An example algorithm, that the group should consider to extend, is the following:

- Anytime the CPU scheduler initiates a task push, this should be done via a message. (Evaluate if this message should be synchronous or asynchronous.)

- Anytime the CPU scheduler should pull from other CPUs it will send a message to them and wake them up (IPI, check the ChronOS source code). Each CPUs will respond with a message indicating the work they would like to share.
- CPU should communicate their load to other processors with messages. (Periodically?)

The group should first implement an in-kernel messaging layer, [3] can be used to start with. The messaging layer is a kernel method to communicate amongst different CPUs. Obviously shared memory will be used as the messaging buffer. Because the project focuses on Linux's RT scheduling class, the code of kernel/sched/rt.c should be understood first and then modified accordingly with the designed algorithm.

To avoid un-necessary complication it is strongly suggested to deploy this experiment on a dual-core setup, this is more than enough for a successful implementation and evaluation.

The final design should be explained, discussed, motivated, evaluated, and compared with the vanilla Linux on different load cases, note that this only covers FIFO_SCHED. Sched_test_app should be used for the evaluation; different set of workloads should be generated. The scheduling overhead can be extracted via the execution time of the application, code instrumentation, or tracing (Linux's ftrace). The group should also answer to the following question: is the execution order of the tasks maintained? If not, why? Provide evidence to your answer. Evaluate the percentage of the time tasks are going to pull vs to push, for different workloads and the additional message overhead.

References:


- [1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09). ACM, New York, NY, USA, 29-44.
- [2] http://www.popcornlinux.org/
- [3] http://git.chronoslinux.org/mklinux.git/blob_plain/cf60f3dd61d0083aafc10b8d8a45de84fadb4de7:/ipc/mcomm.c http://git.chronoslinux.org/mklinux.git/blob_plain/cf60f3dd61d0083aafc10b8d8a45de84fadb4de7:/ipc/mcomm.h