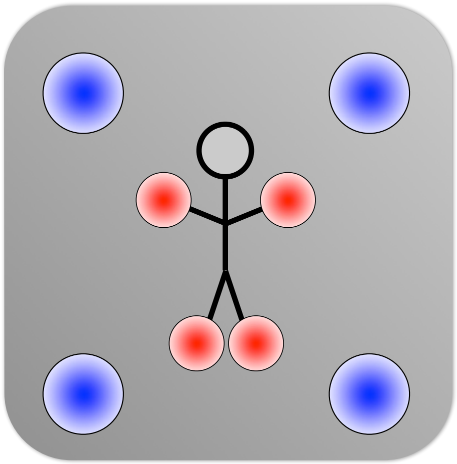
Adaptive Clustered EDF in LITMUS^{RT}

Aaron Block, *Austin College.* Sherman, Texas
William Kelley*, BAE Systems.* Ft. Worth, Texas

Austin College

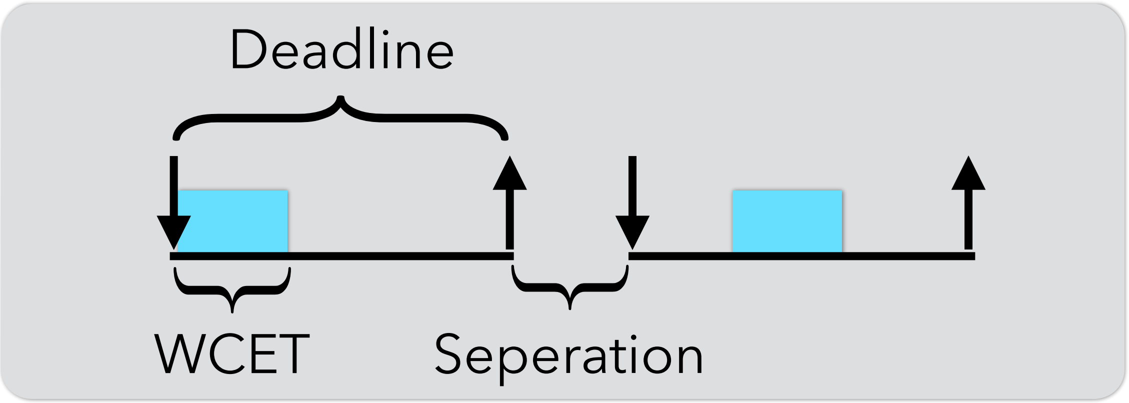
Adaptable System: Whisper

- **Whisper is a motion tracking system**
  - **Speakers** are placed on users hands and feet
  - **Microphones** are placed in the room
  - **Speed of sound computations** can calculate relative position of each speaker.
- **Location a speaker takes more work if**
  - The room is **noisy**
  - The microphone is **far** from the speaker
- **It needs**
  - A **real-time** system
  - A **multiprocessor** system
  - Needs to be able to **adapt** to changing workload.



Classical Sporadic Task Model

- **Worst case execution time** (WCET).
- **Actual execution time**, the actual execution of a job.
  - Upper Bounded by WCET
  - May be different for each job of a task
- **Period,** which defines the
  - **Relative Deadline** of each job (aka, *period*)
  - **Minimum Separation** between each job (≥ *relative deadline*)
- **Weight** of a task: the WCET divided by the period
  - Represents the utilization required by the task to meet all deadlines.
- **Actual Weight** of a job**:** Actual execution time divided by the period.



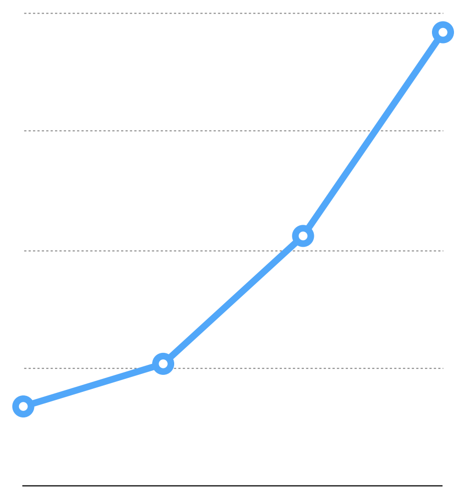
Austin College

Adaptable Model

- **Each task is comprised of several Service Levels. Each of which has:**
  - A **period**
  - A **code segment**
    - Changing the code **changes the execution time**.
  - A **Quality of Service** (QoS)
    - This represents the value to system the task running at this service level
    - Higher QoS = Better
- **The goal of an adaptable task system is to maximize the total QoS of all tasks without "over utilizing" the system.**

Running Time/Weight Translation Function

- **The running time for each code segment is variable**
  - For example, in Whisper the same code segment may need to perform additional computations if the room is noisy
- **We assume that there is a relationship between the running time of the code segments at different service levels**
  - For example, in Whisper, even if we change the code segment, the room is still noisy
- **We assume that the developer of the task system provides a Weight Translation Function that given the weight of a task at one service level, produces an estimate weight at another level.**



Soft Real Time System

- **In our model, we assume that tasks can miss deadlines by a bounded amount.**
  - This model allows us to fully measure the actual execution time for job upon competition.
- **Other soft real-time models are possible to use**
  - We can discuss this off-line if y'all want

Prior Work: Adaptable GEDF

- **In our prior work, we produced an adaptive Global Earliest Deadline First scheduling algorithm. Which consisted of the following components**
  - **A Feedback Predictor**
    - Uses the previous actual weight of jobs and a Predictor-Integral (PI) controller to **predict the actual weight of the next job**.
  - **An optimizer**
    - Uses the estimated weight of all jobs to determine the "Best" service level for each task
  - **Reweighting rules**
    - Enacts the service level changes dictated by the optimizer
  - **A GEDF scheduler**
    - Schedules the system using a Global Multiprocessor Earliest Deadline First Scheduling algorithm.

Prior Work: When we adapt

- **If the system or a task is over utilizing the resources.**
- **After a user-defined interval of time since the last reweighting event.**
- **We do not change service levels under the following conditions**
  - **During the first few seconds**
    - so that the feedback predictors can determine an initial estimated weight.
  - **During a user-defined duration of time after a reweighting event**,
    - so that the feedback predictors can determine an new estimated weight

Prior Work: How we optimize

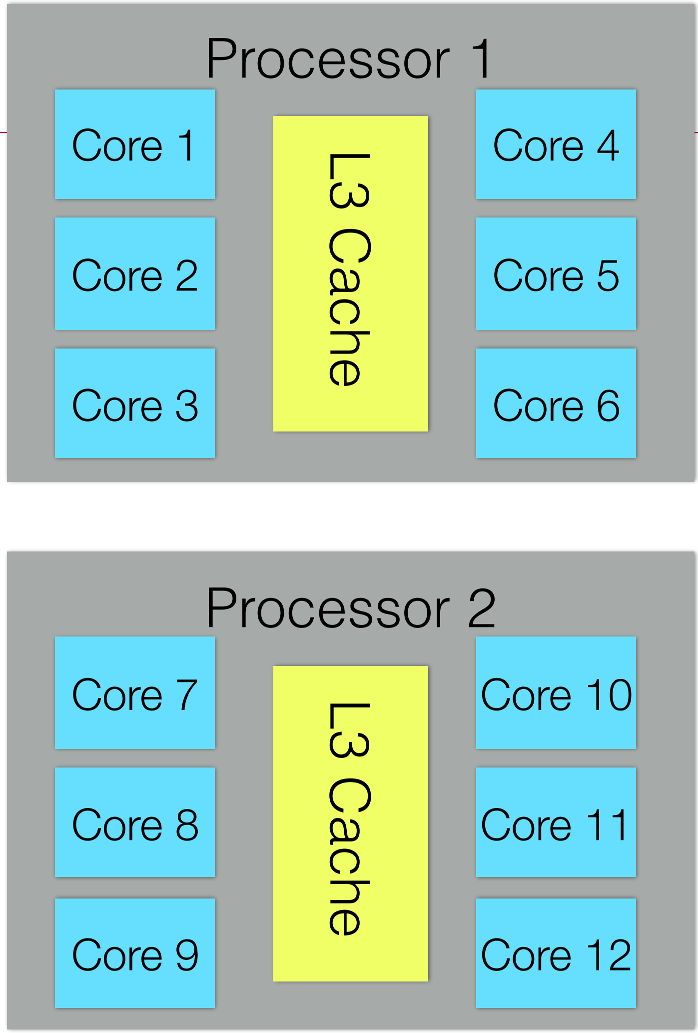
- **Using the value, QoS-to-Weight ratio, rank all tasks from highest-to-lowest**
- **In order, assign each task its highest possible service level that does NOT violate the following conditions**
  - No task has a weight greater than one processor
  - The system is not over utilized
  - Every task is at least assigned its lowest service level.

Global EDF Limitations

- Scheduling costs can be very high because all tasks need to be scheduled
- At scheduling time, all tasks are synchronized on a single processor.
- So, as the processors counts get higher, Global EDF becomes worse.

Clustered EDF

- **Alternative, don't schedule all tasks from a SINGLE priority queue**
- **Instead group processors in to "clusters" that share a common cache**
- **Then schedule each cluster independently using an Earliest Deadline First Algorithm.**
- **This is Clustered EDF (CEDF).**



CEDF

- **CEDF Pros**
  - **Each cluster is independent**. So, scheduling costs and synchronization issues are *much* lower.
- **CEDF Cons**
  - In theory, **cannot fully utilize the system** with bounded deadline misses
  - In reality, **few situations** where we cannot fully utilize.
- **Prior work by *Bastoni et al.* suggests that CEDF may be superior to GEDF if we have more than six cores.**

Adaptable Clustered

- **In this work, we made an adaptable clustered EDF scheduling algorithm.**
- **At a high level the changes from GEDF to CEDF are relatively simple.**
  - Introduce a **repartitioner** to reassign tasks to clustered when the clustered become "imbalanced"

A Feedback Predictor
An optimizer
Reweighting rules
A GEDF scheduler

Adaptive GEDF

A Feedback Predictor
An optimizer
Reweighting rules
A CEDF scheduler
A Repartitioner

Adaptive CEDF

Reality…

- **In reality, moving from a globally scheduled system to a clustered introduces a host of other questions**
  - How do we determine if two clustered are "imbalanced"?
  - How and when do we enact a repartitioning?
  - How do we migrate a single task between two clusters?

Imbalanced

- **We state a Clustered EDF system is Imbalanced if the total QoS in two different clusters differs by a user-defined threshold.**
  - We use QoS instead of weight, because the weight of tasks is constantly changing whereas the QoS determines how well the system is performing.
- **When that threshold is passed, the system is repartitioned.**

Enacting a repartition

- **When do we enact a repartitioning?**
  - All at once?
  - Gradually move tasks one at a time.
- **If we enact it all at once then partially executed tasks will either be…**
  - be abandoned
  - restarted,
  - or could miss their deadline by an unbounded amount.
- **If we move tasks to their new processor upon completion of the current job, the process is slower but that's the only downside.**
- **Therefore, we move tasks gradually.**

Moving a Task

- **How do we move a task between two processors?**
- **Each cluster is protected by a spin lock, but migrating between clustered requires acquiring both simultaneously.**
- **To prevent deadlock, we use the following process:**
  - We introduced a new spin lock (called secondary) for each cluster.
  - Then we created a global order for all secondary spin locks.
  - When a cluster makes a scheduling decision it acquires both its primary and secondary spin locks (and releases them when done)
  - When a cluster moves a task from cluster A to cluster B it runs the following locking code.

Migrate task from Cluster A to B

```
Release Cluster A's second lock
if Cluster A's ID is less than Cluster B's ID then
    Acquire Cluster A's second lock
    Acquire Cluster B's second lock
else
    Acquire Cluster B's second lock
    Acquire Cluster A's second lock
fi
Actually move task from cluster A to B
Release Cluster B's second lock
```

Implementation Details

LITMUS^{RT} Framework

- **We implementing our Adaptive CEDF scheduling using LITMUS^{RT}**
  - **LITMUS^{RT}**, (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems)is an open source framework allows for researchers to create their own "plugin" scheduling algorithms and evaluating them.
    - Created by the research group at UNC-Chapel Hill
    - Currently maintained (and primary developed) by Björn Brandenburg
- **More about LITMUS^{RT} can be found here: http://www.litmus-rt.org**

LITMUS^{RT} plugin

- **Generally, implementing scheduling plugin is fairly "simple"**
  - You create the code that should be executed during scheduling events (releases, job completions, etc.)
  - You let LITMUS^{RT} know about your plugin
  - Recompile/reboot
  - RUN!

Adding Service Levels to LITMUS^{RT}

- **The adaptive algorithms that we are implementing have more interplay between user space and kernel space**
  - Specifically, when we change the service level of a task, the **code segment** also needs to change
- **To enable this we had to modify the LITMUS^{RT} Framework prior to implementing our plugin.**
  - Specifically, LITMUS^{RT}, has a per-task data structure, `struct control_page`, defined in `rt_param.h`, that is shared between user space and kernel.
  - We extended this data structure to include the **current service level number**.
  - When the scheduling algorithm changes the service level of a task, this number is also changed.
  - Each time a job **begins a new job it reads this number**, which lets the job know **which code segment it should execute**.
  - While a task may change its code segment with each execution of a job. Jobs **DO NOT** change their code segment once they have begun.

Additional LITMUS^{RT} modifications

- **Additionally, to enable adaptive behavior a few additional modifications had to be made to LITMUS^{RT} as well**
  - In `rt_param.h`, the `struct rt_task`, (which contains the information about the **execution time**, **deadline,** and **assigned CPU/Cluster** of a task) had to be extended to include
    - **An array of service levels**
    - A variable (called `target_cpu` for historic reasons) that indicates which cluster the task should migrate to.
    - A `target_service_level` that is used to store the service level that the task should be operating at (and will be changed to shortly).
  - In `jobs.c`, the function `setup_release()` was modified to allow for tasks changing their period at every job release.

Changes to Clustered EDF

- **Our implementation of Adaptive CEDF is a modification of default CEDF plugin**
- **The primary changes we had to make were were upon a job completion the following actions occurred**
  - Use the **feedback predictor** to estimate the execution time of the tasks's next job.
  - Update task's position in a per-cluster list sorted by **QoS/Estimated Weight**
  - Determine if tasks on a cluster should have their service level "optimized."
  - If the tasks should **change service levels**, then do so now.
  - Determine if the **clusters are imbalanced**
    - If so, "repartition" the tasks onto clusters.
  - If a task should change clusters, then **migrate that task**

Feedback Predictor Code

- **The code for predicting the weight of a task is relatively simple**
  - alpha and beta are determined by the developer based on the desired characteristics of the feedback predictor (i.e., stead state error, instantaneous response, etc.)

```
void cacluate_Estimated_execution_time(struct task *t, double alpha, double beta){
    t->cumulative_estimated_actual_difference += t->current_difference
    t->current_difference = t->current_actual - t->current_estimated
    t->current estimated = alpha * t->cumulative_estimated_actual_difference +
                              beta * t->current_difference
}
```

Optimizer

- **The optimizer consists of Four distinct phases**
  1. Go through the cluster's list of tasks sorted by QoS-to-Estimated weight ratio
  2. In order, increase the service level of all tasks as high as possible until the cluster is fully utilized (or set a lower threshold)
  3. Mark each task in the cluster as having a new `target_service_level`.
     - For some, the `target_service_level` will be the same as their `current_service_level`.
     - For others, their `current_service_level` will change at their next job competition.
  4. The system is now marked as "stable" and cannot be re-optimized for a developer-specified duration of time.

Repartitioner

- **The repartitioner both determines which tasks should be assigned to which cluster and optimizes the service level of each task**
- **As a result, it is similar to the optimizer, and as such consists of the following phases**
  - Merge each cluster's list of sorted tasks into a single list
  - Go through the master list, assigning tasks to clusters based on which cluster has the largest capacity available. Use the estimated **minimum service level** to determine the amount of capacity available
  - For each cluster, optimize the service levels assigned to it
  - For each task that changed service level and/or cluster, change the associated target service level and/or target cluster
  - The system is now marked as "stable" and cannot be re-partitioned for a developer-specified duration of time.

Running Time

- **Aside from the optimizer and the repartitioner, the running time of adaptive CEDF is incrementally more than the running time of non-adaptive CEDF.**
  - The running time of the optimizer is **O(C)** where **C** is the number of tasks assigned to the cluster
  - The running time of the repartitioner is **O(N)**, where **N** is the number of tasks in the system
- **Both of these times stem from having to go through all of the tasks in the cluster/system**
- **Repartitioning is also costly because clusters involved are "paused" while the repartitioning is occurring.**
- **It is possible on systems with many clusters, to devise an improved repartitioner that only attempts to repartition 2 or 3 clusters at a time.**
  - This would substantially reduce the overhead of repartitioning the system.

Future Work

- **In the future, we plan to the following**
  - Produce a full comparison of adaptive CEDF and GEDF
  - Deliver the adaptable GEDF and adaptable CEDF plugins and LitmusRT modifications as an open source project.
  - Integrate synchronization protocols into CEDF and GEDF.