Implementing Adaptive Clustered Scheduling in LITMUS$^{\mathrm{RT}}$

Aaron Block
Department of Mathematics and Computer Science
Austin College
Sherman, TX
Email: ablock@austincollege.edu

William Kelley
BAE Systems
Fort Worth, TX

***Abstract*—In this paper, we develop an adaptive scheduling algorithm for changing the processor shares of tasks on real-time multiprocessor systems where tasks are assigned to *clusters* of processors. Additionally, we implement this adaptive algorithm as a LITMUS$^{\mathrm{RT}}$ plugin. Our focus is on adaptive systems that are deployed in environments in which tasks may *frequently* require *significant* share changes. Prior work on enabling real-time adaptivity on multiprocessors has focused primarily on systems where tasks are scheduled from a *global* priority queue. The algorithm proposed in this paper use *feedback* and *optimization* techniques to determine at runtime *which* adaptations are needed.**

I. Introduction

Real-time systems that are *adaptive* in nature have received considerable recent attention for both uniprocessor and multiprocessor environments [1], [4]–[6], [10], [11]. In prior work [5], we designed and implemented an adaptive multiprocessor scheduling algorithm (A-GEDF), in which all tasks are scheduled from a single *global* priority queue and can freely migrate between processors. As shown in [9], global systems have the advantage that they can fully utilize a multiprocessor system and still guarantee that deadlines will miss their deadlines by at most a bounded amount. However, Bastoni *et al.* [2] demonstrated that for soft-real time systems with *many* processors (*i.e.*, 12 or more cores), global scheduling algorithms are inferior to algorithms in which tasks can migrate between a *cluster* of processors that share a common cache. In this work, we designed and implemented (as a LITMUS$^{\mathrm{RT}}$ plugin) an alternative adaptive multiprocessor real-time scheduling algorithm (A-CEDF), which is a modification of A-GEDF that uses a clustered scheduling algorithm as its basis rather than a global scheduling algorithm. In this paper, we showed that adaptive behavior (which can improve the Quality of Service of a soft real-time systems) can be enabled on a clustered system without substantially increasing the scheduling cost.

II. Task Model

In this section, we describe our system model and the CEDF scheduling algorithm, upon which A-CEDF is based.

A. Sporadic Tasks

A *sporadic task* is defined by a *worst-case execution time* and *period*. The fraction of a processor required by a task is called the *weight* of the task, and is defined as the worst-case execution time divided by the period. The first job of a task may be invoked or *released* at any time at or after time zero. Successive job releases of task must be separated by at least the period of the task. The *deadline* of a job is period time units after the job is released. A job is said to be *active* if it has been released, but is not yet completed. In this work, we are concerned with *soft real-time systems* where it is acceptable for a job to miss a deadline as long as the amount of time that a job can miss a deadline is bounded (such a system is said to have *bounded tardiness*).

The *actual execution time of job* is the amount of time for which the job is actually scheduled. The *actual weight of a job* is the share of a processor that a job actually requires and is defined by the actual execution time of a job divided by the period of the task. We assume that actual execution time and actual weight for a job are unknown prior to the completion of the job since both values may differ between job releases.

The multiprocessor sporadic task scheduling algorithm that is the most relevant to this work is *clustered earliest deadline first.* (CEDF). Under CEDF, tasks are permanently assigned to "clusters" of processing cores that share a common cache. Jobs with work remaining are prioritized for scheduling on a cluster by their deadline. Jobs can be scheduled on any processor in their cluster, but cannot be scheduled outside of their cluster. As shown in [2] for soft real-time systems, CEDF-based scheduling tends to perform better than non-clustered approaches when clusters contain at least six cores.

B. Adaptable Sporadic Tasks

The *adaptable sporadic task system* [7] extends the notion of a sporadic task system in three major ways. First, *worst-case* execution times are not assumed. Second, each task has a set of *service levels*, which represent a different *Quality of Service* (QoS) levels of a task. Third, tasks have a *weight translation function*, which uses the actual weight and current service level of a task to estimate the actual weight of the task if it changed service levels.

Each service level of a task has three characteristics: a *QoS value*, a *period*, and a *code segment*. When a job is released, it is released at a given service level. That service level determines the code segment that the job will execute,

the deadline of the current job (via the period) and the earliest possible release time of the next job (again via the period).

The weight translation function of a task is an empirically-determined function that takes as an input the current active weight and service level of a task and provides an estimate of what the weight of task would be if it changed to a new service level. For example, if service level 2 for a task required twice as much computation as service level 1 and the current weight of a task was 0.25, then changing from service level 1 to 2 would change the weight of the task to 0.5. It is unnecessary for the weight translation function to be perfectly accurate, but the more accurate it is, the better an adaptive algorithm will be optimizing system QoS. It is important to note that tasks with lower QoS values must have lower estimated weights. Thus, an adaptive algorithm can trade QoS for schedulability.

III. A-CEDF

In this work, we introduce the *adaptive clustered earliest deadline first* (A-CEDF) scheduling algorithm. A-CEDF is a clustered-scheduled variant of the *adaptive global earliest deadline first* (A-GEDF) scheduling algorithm, which we proposed in prior work [7]. A-CEDF is designed to schedule adaptable sporadic task with the objective of maximizing the total QoS while maintaining bounded tardiness. A-CEDF consist of five primary components: **(1)** the *predictor*, which uses a *proportional-integral* (PI) feedback controller to estimate the weights of future jobs using the actual weights of previously completed jobs; **(2)** the *optimizer*, which given estimated job weights, attempts to determine an optimal set of functional service levels; **(3)** the *repartitioner*, which given the estimated job weights attempts to determine the optimal assignment of tasks to clusters; **(4)** several *reweighting rules*, which are used to change the functional service level of a task to match that chosen by the optimizer; and **(5)** the CEDF *algorithm*. At a high level, these components function as follows.

- *At each instant*, tasks are scheduled via CEDF.
- *At a job's completion*, the predictor is used to estimate the weight for the next job release.
- *After a developer-specified threshold based on task weight and time elapsed*, the optimization component uses the estimated weight to determine new service levels for each task. If the service level of a job changes, then the reweighing rules will enact it.
- *If the clusters are "imbalanced"*, then the repartitioner will correct this behavior by migrating tasks between clusters. If necessary, the optimization and reweighting rules may be run as part of this process.

The primary difference between A-CEDF and A-GEDF is that A-GEDF allow tasks to freely migrate between *all* processors. Thus, A-GEDF does not need or have a repartitioner component. That being said, A-CEDF and A-GEDF have similar predictors, optimizers, and reweighting rules.

IV. Implementation

To better understand A-CEDF, we implemented this algorithm in the LITMUS$^{\mathrm{RT}}$ version 2014.2 (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which is an extension of Linux (currently, version 3.10.41) that allows different multiprocessor scheduling algorithms to be linked as plug-in components [3], [8]. Our implementation of A-CEDF consists of both a user-space library and kernel support added to LITMUS$^{\mathrm{RT}}$. Our implementation of A-CEDF required 1,227 lines of code. Most of these changes were in modifying LITMUS$^{\mathrm{RT}}$'s default CEDF implementation. In prior work [7], we discussed how to modify LITMUS$^{\mathrm{RT}}$ to support adaptable sporadic tasks scheduled via *global* scheduling algorithms. In this work, we focus on the *additional* challenges that arise when implementing *clustered* real-time adaptive scheduling algorithms.

A. Challenge: Defining "Imbalanced"

As we mentioned in Sec. III, A-CEDF repartitions when the clusters are *imbalanced*. Informally, the clusters become imbalanced if one cluster is doing more or less work than another. However, it is not obvious how we should formally define "imbalanced." There are two metrics that we can use to measure the quality of a partitioning: **(1)** the total weight of all tasks assigned to a given cluster and **(2)** the total QoS of all tasks assigned to a given cluster. Under either metric, a system is "imbalanced" if the metric value (*i.e.*, total weight or QoS) of one cluster is higher than a *user-defined threshold* the metric value of another.

In this work, we repartition the system when there is an imbalance between the *QoS* of tasks assigned to different clusters. We chose to use a QoS-based metric because the objective of A-CEDF is to maximize the QoS without causing unbounded tardiness. Thus, the weight balance by itself is not useful if the system could run at a higher QoS after rebalancing. For example, consider the following scenario. Suppose that an external event occurs that increases the execution time for all tasks on a given cluster. The optimizer component of A-CEDF will reduce the service level (and hence the QoS) for every task on the cluster. If this reduction in QoS is larger than the user-defined threshold, then this will trigger the repartitioning to occur. It is possible that such an event would be unnoticed by a weight-based metric; particularly, if the total weight of all tasks was approximately the same before the external event and after the optimizer ran.

B. Challenge: Enacting a Repartioning

When the system determines that tasks should be repartitioned, the next question is when should that repartitioning be enacted. There are two primary approaches to this problem: **(1)** migrate all tasks to new clusters immediately or **(2)** gradually migrate tasks between clusters over time. In our implementation of A-CEDF, we migrate tasks gradually. Specifically, after a repartitioning event, we migrate each task when it finishes its active job. We chose this approach because, based on a simple extension to our work in [6], it is possible to show that frequently moving *incomplete* jobs between clusters can cause unbounded tardiness.

Additionally, it is worth noting that since repartitioning occurs because of QoS imbalances, the quicker the repartition is enacted, the better it is for the overal QoS for the system. Yet, quickly enacting a repartitioning is *not* crucial for the functioning of the system. Thus, while gradually migrating tasks between clusters will reduce the QoS of the system, we believe this tradeoff is worth the cost to preventing unbounded tardiness from occurring.

C. Challenge: Migrating a Task

In the typical implementation of CEDF, each cluster has its own spin lock for protecting the priority queue containing all active jobs. This prevents a race condition in which multiple cores on the same cluster attempt to change the priority queue at the same time. Moreover, under CEDF tasks never migrate between clusters. This is not the case in A-CEDF.

To enable A-CEDF to migrate a task from Cluster A to Cluster B, we need two layers of synchronization: **(1)** one layer to prevent any core on Cluster B from corrupting Cluster B's priority queue and **(2)** one layer to prevent any core on Cluster A *that is migrating a task to Cluster B* from corrupting Cluster B's priority queue. Moreover, Cluster A cannot simply acquire Cluster B's spin lock or a deadlock could occur (*e.g.*, if Cluster B attempted to migrate a task to Cluster A at approximately the same time that Cluster A is attempting to migrate a task to Cluster B). To enable task migration, we need a more sophisticated approach to synchronization. We do so by employing the following method:

- Each cluster has a unique ID number.
- Each cluster has a `prime` and `second` spinlock.
- When entering into *any* critical section, a core first acquires its cluster's `prime` lock, then its `second` lock.
- When a task that is flagged for migration from Cluster A to Cluster B, it executes the pseudo-code given in Fig. 1.

There are three keys to this synchronization technique. First, the `prime` lock on each cluster protects the priority queue from corruption by all cores in the same cluster. Second, the `second` lock provides a means to protect a cluster's priority queue from external corruption (*i.e.*, Cluster A must acquire Cluster B's `second` before migrating the task). Third, by releasing and reacquiring `second` locks in a globally established order (*i.e.*, the code in Fig. 1), we prevent the circular chain of dependencies that is a prerequisite for deadlock. Notice that this ordering heuristic is similar to the double-lock used by Linux for its native run queues.

D. Cost of Implementation

To measure the cost of an implemented A-CEDF, we ran a *simulated* virtual reality human tracking system (called Whisper [12]) on a Mac Pro with two 2.66 Ghz 6-core Intel Xeon processors (12 cores total). Each core has 512KB of L2 cache and each processor has 12 MB of fully shared L3 cache. Our clustered implementation of A-CEDF had two clusters, one for each processor. Our simulated human tracking system had 96 tasks each of which had both gradual and sudden changes in weight. We found that the introduction of adaptive techniques slightly increased the average scheduling cost compared to a non-adaptive variant. Specifically, A-CEDF took on average 5.8μs per scheduling decision while CEDF took on average 4.3μs per scheduling decision. The increased running time was primarily becasuse our implementation of the of the optimizer and repartitioner involves sorting a large number of tasks. It is possible to reduce the running time of A-CEDF by using a faster, but less accurate implementation of these two components. It is worth noting that neither the feedback predictor nor the double-locking mechanism appreciably increased the scheduling time.

```
Migrate task from Cluster A to B
1:  Release Cluster A's second lock
2:  if Cluster A's ID is less than Cluster B's ID then
3:      Acquire Cluster A's second lock
4:      Acquire Cluster B's second lock
5:  else
6:      Acquire Cluster B's second lock
7:      Acquire Cluster A's second lock
8:  fi
9:  Actually move task from cluster A to B
10: Release Cluster B's second lock
```

Fig. 1. Pseudo-code defining task migration

V. Conclusions and Future Work

In this work, we designed and implemented A-CEDF as a LITMUS$^{\mathrm{RT}}$ plugin. In the process of implementing A-CEDF, we came across multiple issues with implementing any type of adaptive clustered real-time scheduling algorithm. We also established that adaptive behavior can be enabled in clustered soft-real time systems with only a small additional scheduling cost. In future work, we plan to compare the performance of A-CEDF to A-GEDF at maximizing the QoS for a system.

References

[1] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *RTSS.* '02.

[2] A. Bastoni, B. Brandenburg, and J. Anderson. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor Real-Time Schedulers. *RTSS*, '10.

[3] B. Brandenburg *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* PhD thesis, UNC, '11.

[4] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. *Journal of Embed Comp*, '11.

[5] A. Block, *Adaptive Multiprocessor Real-Time Systems.* PhD thesis, UNC, '08,

[6] A. Block, J. Anderson, and U. Devi. Task reweighting under global scheduling on multiprocessors. *Real-Time Sys.*, '08.

[7] A. Block, B. Brandenburg, J. Anderson, and S. Quint. An Adaptive Framework for Multiprocessor Real-Time Systems. *ECRTS*, '08.

[8] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *RTSS*, '06.

[9] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Sys.*, '08

[10] N. Khalilzad, F. Kong, X. Liu, M. Behnam, and T. Nolte. A feedback Scheduling Framework for Component-Based Soft Real-Time Systems. *RTAS.*, '15

[11] C. Lu, J. Stankovic, S. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Sys.*, '02.

[12] N. Vallidis *Whisper: A Spread Spectrum Approach to Occlusion in Acoustic Tracking.* PhD thesis, UNC, '02.