An Adaptive Framework for Multiprocessor Real-Time Systems*

Aaron Block†, Björn Brandenburg†, James H. Anderson†, and Stephen Quint‡
Department of Computer Science† and Department of Biomedical Engineering‡
University of North Carolina at Chapel Hill

Abstract

In this paper, we develop an adaptive scheduling framework for changing the processor shares of tasks—a process called reweighting*—on real-time multiprocessor platforms. Our particular focus is adaptive frameworks 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 exclusively on scheduling algorithms that can* enact *needed adaptations. The algorithm proposed in this paper uses both feedback and optimization techniques to determine at runtime* which *adaptations are needed.*

1. Introduction

Real-time systems that are *adaptive* in nature have received considerable recent attention [3, 6, 7, 8, 15, 20]. As multicore platforms become ever more ubiquitous, such systems will be deployed with increasing frequency on multiprocessor platforms. In the multiprocessor case, prior work on real-time adaptivity has focused exclusively on scheduling algorithms that can *enact* needed adaptations [6, 7, 8]. This paper is directed at the related issue of devising mechanisms that determine at runtime *which* adaptations are needed.

Under traditional task models (*e.g.*, periodic, sporadic, *etc*.), the scheduability of a system is based on each task's *worst-case execution time* (WCET), which defines the maximum amount of time each of its jobs can execute. The disadvantage of using WCETs is that systems may be deemed unschedulable even if they would function correctly most of the time when deployed. Adaptive real-time scheduling algorithms allow per-task processor shares to be adjusted based upon runtime conditions, instead of always using constant share allocations based upon WCETs. It is desirable that the share of each task be selected in a way that attempts to maximize overall quality-of-service (QoS). Moreover, the allocation scheme should not "over-react" in adjusting shares when transient overloads occur. To the best of our knowledge, the problem of devising adaptation mechanisms that satisfy these requirements has not been considered before in work on multiprocessor systems. In this paper, we remedy this shortcoming by presenting a multiprocessor framework in which share allocations are adjusted using feedback and optimization techniques. This framework is directed at multiprocessor workloads specified as soft real-time sporadic tasks for which bounded deadline tardiness is acceptable.

Whisper. To motivate the need for this work, we consider the Whisper tracking system, which performs full-body tracking in virtual environments [22]. Whisper tracks users via an array of wall- and ceiling-mounted microphones that detect white noise emitted from speakers attached to each user's hands, feet, and head. Like many tracking systems, Whisper uses *predictive techniques* to track objects. The workload on Whisper is intensive enough to necessitate a multiprocessor design. Furthermore, adaptation is required because the computational cost of making the "next" prediction in tracking an object depends on the accuracy of the previous one. In fact, the processor shares of the tasks that are deployed to implement these tracking functions may vary by as much as *two orders of magnitude*.

Adaptive feedback-based scheduling algorithms. In *adaptive feedback-based scheduling algorithms*, each task's processor share is defined as a function of its *estimated execution time*, which is calculated by using its prior jobs' *actual execution times*. Moreover, a user can fine-tune a feedback-based system to achieve desired behaviors.

Lu *et al.* [14] were the first to propose such a scheduling algorithm, which was directed at uniprocessor systems.[1] Under their algorithm, called FC-EDF^2, each task has multiple versions (called *service levels*), each of which has a different level of QoS and a different *nominal processor share*, representing the fraction of a processor the task will require on average if it executes at that service level. A task can only execute at one service level at a time. (The idea of using multiple service levels to control a task's behavior dates back to work by Tokuda and Kitayama [21] and was later expanded upon by other authors [2, 9, 10].) In order to control the system, FC-EDF^2 monitors the system's *utilization* and *miss-ratio*, *i.e.*, the fraction of jobs with missed deadlines. In order to maximize utilization subject to the constraints of a target miss-ratio, FC-EDF^2 adjusts the set of scheduled tasks and their service levels. More recently, Lu *et al.* extended this work to create a comprehensive feedback

*Work supported by Intel and IBM Corps., NSF grants CCR 0408996, CCR 0541056, and CCR 0615197, and ARO grant W911NF-06-1-0425. The first author was also supported by the UNC Alumni Fellowship.

[1] Specifically, the first *correct* feedback algorithm was proposed in [14].

scheduling framework [15] that more explicitly incorporates the value to the system associated with each service level. This framework is the basis for the approach we propose in this paper. One drawback of FC-EDF^2 is that, because only the utilization and the *system-wide* miss-ratio are monitored, the system cannot identify whether an individual task has an actual execution time that deviates substantially from its estimated execution time. Thus, the system can only respond to differences between the actual and estimated execution times of tasks by changing the entire system instead of only a few tasks.

Alternatively, Abeni *et al.* have proposed a uniprocessor feedback algorithm in which each task has its own feedback-controller rather than one controller for the entire system [3]. In order to attempt to maintain an accurate processor share for each task, their algorithm monitors *for each task* the difference between the estimated and actual execution times of each job. Once the system has calculated a new estimated execution time for a future job, it adjusts the task's weight. More recently, Cucinotta *et al.* extended this approach to provide stochastic guarantees concerning per-task processor shares [12]. One drawback of their approach is that it ignores the possibility that some tasks are more important than others.

In addition to general-purpose real-time scheduling algorithms, feedback-based scheduling has become increasingly important for managing *control tasks*, *i.e.*, tasks that control external devices. In work by Martí *et al.* [16], an approach is proposed that is similar to that of Abeni *et al.*, except that in [16], each period of each task has an associated "importance value" that denotes the task's value to the system in that period. By using importance values, Martí *et al.* determined the optimal period for each task via standard linear programming techniques. One limitation of this approach is that the system can only adjust task periods, and not execution times (as done by Lu *et al.* [15]).

There has also been substantial work on using feedback mechanisms within distributed real-time systems [1, 19]. However, because such systems are "loosely coupled," and because distributed systems must deal with issues not present in multiprocessor designs (*e.g.*, end-to-end delays), work in this area is less directly applicable to multiprocessor feedback systems.

In the multiprocessor case, there has been relatively little work on feedback algorithms. The little multiprocessor-based work that has been done has focused on non-preemptive systems where WCETs, best-case execution times, and deadlines are static [4, 18]. As such, this work is too restrictive to apply in applications like Whisper.

Multiprocessor scheduling. All multiprocessor scheduling algorithms can be classified as either *partitioned* or *global*. Under partitioned algorithms, each task is permanently assigned to a specific processor and each processor independently schedules its assigned tasks using a uniprocessor scheduling algorithm. An example of such an algorithm is the partitioned EDF (PEDF) algorithm, wherein the uniprocessor earliest-deadline-first (EDF) algorithm is used as the per-processor scheduler. The advantage of partitioned approaches is that tasks have lower runtime overheads since they never migrate among processors. The disadvantage of such approaches is that, to ensure that every job completes within a bounded range of its deadline, overall utilization may need to be restricted substantially (by up to approximately 50%). In global scheduling algorithms, *e.g.*, global EDF (GEDF), tasks are scheduled from a single priority queue and may migrate among processors. While global algorithms incur higher overheads than partitioned algorithms due to migration, recent work has shown that global algorithms (in particular, GEDF) can guarantee that every job completes within some time bound of its deadline in any system that is not overutilized [13].

In recent work, Calandrino *et al.* [11] constructed a multiprocessor real-time extension of Linux called LITMUS$^{\mathrm{RT}}$ that allows different scheduling policies to be linked as plugins. They showed that, for soft real-time systems, *i.e.*, systems in which bounded deadline tardiness is acceptable, global scheduling algorithms tend to be better than partitioned approaches from the standpoint of schedulability when real overheads are considered. Furthermore, recent work by Block *et al.* [8] has shown that partitioned approaches are ill-suited for adaptive scenarios because such systems would likely require frequent repartitioning to prevent unbounded tardiness. Frequent repartitioning mitigates the primary advantage of partitioned systems, which is that tasks do not migrate. Thus, in this paper, we focus our attention on global scheduling algorithms, in particular, GEDF.

Contributions. Our novel contributions are:

- We present an *adaptive* GEDF *framework* called A-GEDF that consists of three components: a *predictor*, which uses feedback techniques to estimate the processor share of future jobs; an *optimizer*, which uses the estimated processor shares of tasks to determine a new set of service levels; and several *reweighting rules*, which change the service levels to match that determined by the optimizer. To the best of our knowledge, this is the first such adaptive global framework for multiprocessor systems to be proposed. Our focus on *multiprocessor* platforms is the main distinguishing aspect of our work compared to prior efforts.

- We (briefly) discuss an implementation of A-GEDF in LITMUS$^{\mathrm{RT}}$.

- We present an experimental evaluation of our A-GEDF implementation using code derived from Whisper. In this evaluation, A-GEDF proved to be an extensible

framework that can be easily configured to support different optimization criteria. Also, it performed well in scenarios where a non-adaptive GEDF algorithm would result in significant system over-utilization. (We also evaluated A-GEDF using code derived from a night-vision system and found similar results, which we omit due to space constraints.)

We developed a new adaptive framework because of shortcomings associated with other approaches:

- Our experiments show that Whisper *will not work properly if partitioned or non-adaptive scheduling is used*.
- For GEDF, the miss-ratio, proposed by Lu *et al.* [14], cannot be used to determine if a system is overutilized because deadline misses are expected behavior.
- Because the approach proposed by Abeni *et al.* [3] assumes that all tasks are equally important, it is insufficient for Whisper, which values some tasks over others.

The rest of this paper is organized as follows. In Sec. 2, we discuss needed background. In Sec. 3, we describe the A-GEDF algorithm in greater detail. In Sec. 4, we discuss our implementation of A-GEDF in LITMUS$^{\mathrm{RT}}$. In Sec. 5, we present our experimental evaluation. We conclude in Sec. 6.

2. Definitions

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

Sporadic tasks. We denote the i^{th} task of a task system T as T_i (where tasks are indexed by some arbitrary method), and denote the j^{th} job of the task T_i as T_i^j (where jobs are ordered by the sequence in which they are invoked). A *sporadic task* is defined by a *worst-case execution time*, denoted $\mathsf{e}(T_i)$, and *period*, denoted $\mathsf{p}(T_i)$. The fraction of a processor required by T_i is called the *weight of* T_i, denoted $\mathsf{w}(T_i)$, and is defined as $\mathsf{e}(T_i)/\mathsf{p}(T_i)$. The first job of a task may be invoked or *released* at any time at or after time zero. The release time of job T_i^j is denoted $\mathsf{r}(T_i^j)$. Successive job releases of task T_i must be separated by at least $\mathsf{p}(T_i)$ time units. The *absolute deadline* (or just *deadline*) of job T_i^j, denoted $\mathsf{d}(T_i^j)$, equals $\mathsf{r}(T_i^j)+\mathsf{p}(T_i)$. The *actual execution time of job* T_j^j, denoted $\mathsf{Ae}(T_i^j)$, is the amount of time for which T_i^j is scheduled; this value is upper-bounded by $\mathsf{e}(T_i)$. We assume that the value of $\mathsf{Ae}(T_i^j)$ is not known until T_i^j finishes execution.

Adaptable sporadic tasks. Our notion of an *adaptable sporadic task system* is based on a task model presented by Lu *et al.* [15] and extends the notion of a sporadic task system in two major ways. First, *worst-case* execution times are not assumed. Second, each task T_i has a set of *service levels*, denoted $\mathsf{SL}(T_i)$, each of which represents a different level of QoS for T_i, and a *weight translation function*, denoted $\mathsf{g}(T_i, e, k, q)$, which is used to compute the weight of T_i at different service levels, as explained below.

The k^{th} service level of $\mathsf{SL}(T_i)$ is defined by an *importance value*, $\mathsf{v}(T_i, k)$, a *period*, $\mathsf{p}(T_i, k)$, and a *code segment*. Without loss of generality, we assume that the service levels in $\mathsf{SL}(T_i)$ are indexed by increasing importance value from 1 to $|\mathsf{SL}(T_i)|$, where $|\mathsf{SL}(T_i)|$ is the number of elements in $\mathsf{SL}(T_i)$. The importance value represents some user-defined notion of "goodness," where 0.0 represents a service level that has no value and 1.0 represents the maximal possible value associated with any service level of any task in the system.

At any point in time t, one service level in $\mathsf{SL}(T_i)$ is said to be the *functional service level of* T_i. The index of the functional service level of T_i at time t is denoted $\mathsf{f}(T_i, t)$. For now, we assume that the functional service level of a task T_i does not change within $(\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$ for any job T_i^j of T_i. In Sec. 3.3, we discuss how to change the functional service level of a task at any time. If k is the functional service level at $\mathsf{r}(T_i^j)$, then T_i^j is said to be *functioning at service level* k. If T_i^j is functioning at service level k, then both $\mathsf{r}(T_i^{j+1}) \geq \mathsf{r}(T_i^j) + \mathsf{p}(T_i, k)$ and $\mathsf{d}(T_i^j) = \mathsf{r}(T_i^j) + \mathsf{p}(T_i, k)$ hold. We consider a task T_i to be *active* at time t if there exists a job T_i^j (called T_i*'s active job*) such that $t \in [\mathsf{r}(T_i^j), \mathsf{d}(T_i^j))$. We use $\mathsf{ACT}(t)$ to denote the set of active jobs at time t.

The code segment associated with the k^{th} service level is the code segment that a job T_i^j will execute if T_i^j is functioning at service level k. There are numerous different methods for defining such code segments, depending on the specific application. For some applications, each service level may execute the same code segment, and the only difference between service levels is the period. For other applications, the difference between service levels may be something as simple as the number of iterations in a loop, while for others, each service level may use entirely different code. As we will discuss in Sec. 3, how the code segment is implemented will impact the efficacy of A-GEDF at adapting tasks.

Just as for sporadic tasks, the value of $\mathsf{Ae}(T_i^j)$ denotes the amount of time for which T_i^j is actually scheduled. The *actual weight of a job* T_i^j, denoted $\mathsf{Aw}(T_i^j)$, represents the actual fraction of a processor that T_i^j requires and is defined by $\mathsf{Aw}(T_i^j) = \mathsf{Ae}(T_i^j)/\mathsf{p}(T_i^j, k)$, where T_i^j is functioning at service level k. Just as with sporadic tasks, we assume that the value of $\mathsf{Ae}(T_i^j)$ (and by extension $\mathsf{Aw}(T_i^j)$) is not known until T_i^j finishes execution.

As we will discuss in Sec. 3.1, since the actual weight of a job is not known until it completes, A-GEDF uses an *esti-*

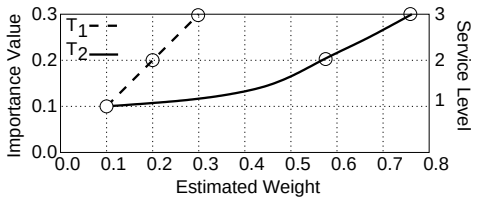

Figure 1. Estimated weight vs. importance value/service level for two tasks: T_1 and T_2, each of which have three service levels with importance values of 0.1, 0.2, and 0.3. For T_1, $\mathsf{g}(T_1, e, 1, 2) = 2e$ and $\mathsf{g}(T_1, e, 1, 3) = 3e$, and for T_2, $\mathsf{g}(T_2, e, 1, 2) = e^{1/4}$ and $\mathsf{g}(T_2, e, 1, 3) = e^{1/8}$, where e is the estimated weight while functioning at service level one. For both T_1 and T_2, this graph depicts the scenario where $e = 0.1$.

mated weight for incomplete jobs, denoted $\mathsf{Ew}(T_i^j)$. When A-GEDF calculates the estimated weight for a job T_i^j it does so for a specific service level (typically, the same service level at which T_i^{j-1} was functioning). The weight translation function is used to map the estimated weight as calculated by A-GEDF for a specific job T_i^j functioning at a specific service level to what the estimated weight of T_i^j would be if it functioned at a *different* service level. Specifically, if e is the estimated weight of T_i^j assuming that T_i^j is functioning at service level k, then the weight translation function, $\mathsf{g}(T_i, e, k, q)$, returns the estimated weight of T_i^j if it *were to be functioning at q^{th} service level instead of the k^{th} service level*. For example, consider the two tasks T_1 and T_2 described in Fig. 1. In this example, if $e = 0.1$, $k = 1$, and $q = 3$, then $\mathsf{g}(T_1, e, k, q) = 0.3$, which is the estimated weight of a job of T_1 if it had been calculated for the third service level instead of the first. Also, for T_2, $\mathsf{g}(T_2, e, k, q) \approx 0.75$, which is the estimated weight of a job of T_2 if it had been calculated for the third service level instead of the first.

As we will discuss in Sec. 3.2, the weight translation function is used by the optimizer to determine the effect on the system caused by changing the functional service level of a task. We make only two assumption about the behavior of $\mathsf{g}(T_i, e, k, q)$: if $q < k$, then $\mathsf{g}(T_i, e, k, q) \leq e$; and if $\mathsf{g}(T_i, e_1, k, q) = e_2$, then $\mathsf{g}(T_i, e_2, q, k) = e_1$. It is important to note that the function $\mathsf{g}(T_i, e, k, q)$ can return approximate values; however, the accuracy of $\mathsf{g}(T_i, e, k, q)$ will impact the performance of the optimizer. Like service levels and code segments, the weight translation function is defined by the application developer and can be determined empirically.

The primary difference between the task model presented by Lu *et al.* [15] and our task model is that in [15] each service level of a task T_i has a *static* notion of "estimated weight" that represents the nominal fraction of a processor

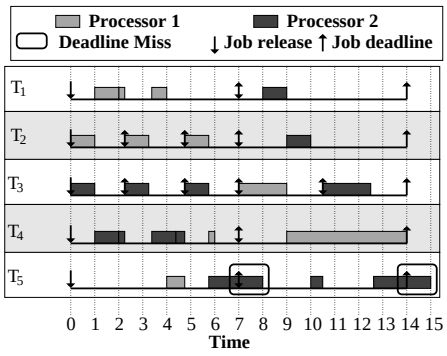

Figure 2. A two-processor system T scheduled by GEDF with five tasks: T_1, with a period of 7, for which $\mathsf{Ae}(T_1^1) = 2$ and $\mathsf{Ae}(T_1^2) = 1$; T_2, with an initial period of $7/3$ that changes to 7 at time 7 (via a functional service-level change), for which $\mathsf{Ae}(T_2^j) = 1$ for any j; T_3, with an initial period of $7/3$ that at time 7 changes to $7/2$, for which $\mathsf{Ae}(T_3^j) = 1$ for $1 \leq j \leq 3$ and $\mathsf{Ae}(T_3^j) = 2$ for $j \geq 4$; T_4, with a period of 7, for which $\mathsf{Ae}(T_4^1) = 3$ and $\mathsf{Ae}(T_4^2) = 5$; T_5 with a period of 7, for which $\mathsf{Ae}(T_5^j) = 3$ for any j. Notice that T_5^1 misses a deadline by one quantum unit at time 7, and T_5^2 misses a deadline by one time quantum at time 14.

required by T_i. Statically assigning an estimated weight to a task implies that the task has a typical behavior and that if it requires a smaller or larger fraction of a processor, then such a scenario is an anomaly. While this may be true for many applications, for systems like Whisper, predetermining the nominal weight of a task can be difficult if not impossible. Thus, as we will discuss in Sec. 3, rather than statically determining estimated weights, A-GEDF will dynamically calculate the estimated weight for each job.

Scheduling. Under GEDF, "ready" jobs (see below) are prioritized by deadline, with earlier deadlines having higher priority. Deadline ties are resolved arbitrarily, but consistently. Additionally, an arriving job with higher priority preempts the executing job with the lowest priority if no processor is available. The preempted job may later resume execution on a different processor. It is important to note that if a sporadic task system, adaptive or not, is scheduled by GEDF then deadlines may be missed; however, in the absence of over-utilization, such misses are by bounded amounts only [13]. As an example, consider the two-processor system depicted in Fig. 2 (described in the caption). Notice that T_5 misses a deadline at times 7 and 14.

For an arbitrary scheduling algorithm $\mathcal{A}$ and task system T, we let $\mathcal{S}$ denote an M-processor schedule $\mathcal{A}$ of T, and let $\mathsf{A}(\mathcal{S}, T_i^j, t_1, t_2)$ denote the total time allocated to T_i^j in $\mathcal{S}$ in $[t_1, t_2)$. Similarly, we use $\mathsf{A}(\mathcal{S}, T_i, t_1, t_2)$ to denote the total time allocated to all jobs of T_i in $\mathcal{S}$ over the in-

terval $[t_1, t_2)$. We say that the value of $\mathsf{A}(\mathcal{S}, T_i^j, 0, t)$ is the amount for which T_i^j has *executed by* t. For example, in Fig. 2, $\mathsf{A}(\mathcal{S}, T_5^1, 0, 8) = 3$, $\mathsf{A}(\mathcal{S}, T_5^1, 0, 14) = 3$, and $\mathsf{A}(\mathcal{S}, T_5^2, 7, 8) = 0$ (since T_5^1 and not T_5^2 is executing over the range $[7, 8)$).

Definition (Completed, Pending, and Ready). If $\mathcal{S}$ is an M-processor schedule of the task system T, then a job $T_i^j \in T$ is said to have *completed by time t in $\mathcal{S}$* iff T_i^j has executed for at least $\mathsf{Ae}(T_i^j)$ time units by t in $\mathcal{S}$. For example, in Fig. 2, T_1^1 is complete by time 4, and T_5^1 is complete by time 8. For an arbitrary scheduling algorithm $\mathcal{A}$, if $\mathcal{S}$ is an M-processor schedule of the task system T under $\mathcal{A}$, then a job T_i^j is said to be *pending at time t in $\mathcal{S}$* if $\mathsf{r}(T_i^j) \leq t$ and T_i^j is incomplete at t in $\mathcal{S}$. For example, in Fig. 2, T_1^1 is pending until time 4, and T_5^1 is pending until time 8. Note that a job can be pending, but not active, if it misses its deadline. A pending job T_i^j is said to be *ready at time t in $\mathcal{S}$* if all prior jobs of task T_i have completed by t. For example, T_5^2 is not ready until time 8 because T_5^1 is incomplete until time 8. A job T_i^j can be pending but not ready if T_i^{j-1} is incomplete at $\mathsf{r}(T_i^j)$.

3. A-GEDF

We now present the A-GEDF scheduling algorithm and its three components: the *predictor* (Sec. 3.1), which uses feedback-based techniques to estimate the actual weights of future jobs; the *optimizer* (Sec. 3.2), which given estimated job weights, attempts to determine an optimal set of functional service levels; and several *reweighting rules* (Sec. 3.3), which are used to change the functional service level of a task to match that chosen by the optimizer. In the following, we assume that A-GEDF is used on an M-processor system.

The major components of A-GEDF are depicted in Fig. 3. At a high level, these components function as follows.

- *At each instant*, the M pending jobs with the smallest deadlines are scheduled.
- *At T_i^j's completion*, the predictor is used to estimate the weight for the next job release of T_i. If maintaining a constant weight is important, then the reweighting rules may change T_i^{j+1}'s functional service level.
- *After some user-specified threshold*, the optimization component is run to determine new service levels for each task. Then, the following two steps are performed. First, if some tasks require an estimated weight *decrease*, then the reweighting rules are used to change the service levels of those tasks. This creates spare capacity in the system. Second, as the spare capacity created by weight decreases becomes available, if some tasks require an estimated weight *increase*, then the reweighting rules are used to change the service levels of those tasks.

It is worthwhile to note that the optimization component (and hence large-scale changes to task functional service levels) is only executed after some user-specified threshold. We offer some guidelines for choosing this threshold in Sec. 3.4.

3.1. The Feedback Predictor

Before continuing, we first review the basics of *feedback systems*. Most feedback systems consist of the following components, most of which are labeled in the model of our system in Fig. 3(b): the *input value*, the *output value*, the *actuator*, the *error*, the *plant*, and the *controller*. The input value is the reference value for the system, while the output value is value computed by the system. The actuator calculates the error by subtracting the output from the input. The plant is the system we wish to control. The controller modifies the input to change the behavior of the output.

The performance of a feedback system is measured in terms of *transient response*, *steady-state error*, and *stability*. The transient response of a system is the initial output of the system to a change in input. The steady-state error denotes the difference between the output and the input of the system as time increases. A system is considered to be *stable* if every bounded input causes the system's steady-state error to be bounded.

It is worthwhile to note that while feedback-based techniques are primarily used to control the behavior of a plant for which the (reference) input is known, another viable use for such techniques is to *predict* future values of a changing and unknown input. The design of such a system is exactly the same as the typical feedback system, *except that the feedback loop does not directly impact the behavior of the system*. (Thus, the plant and the controller can be one-in-the-same.) In such a system, the transient response describes the initial accuracy of predictions after there has been a change in the input, and the steady-state error describes the difference between the predicted and actual values as system time increases.

The feedback predictor. Since the predictor in A-GEDF uses feedback-based techniques to predict the weight of future jobs instead of using a simpler approach, such as setting $\mathsf{Ew}(T_i^j) = \mathsf{Aw}(T_i^{j-1})$, the predictor both produces values of $\mathsf{Ew}(T_i^j)$ that are less susceptible to ephemeral fluctuations in the workload and is capable of closely tracking trends in the actual weight (*e.g.*, when the actual weight of the task changes at a constant rate). Using a feedback loop to predict the weight of future jobs is similar to the approach by Abeni *et al.* [3].

As depicted in Fig. 3(a), in the predictor, each task has its own feedback loop. Also, as depicted in Fig. 3(b), for each feedback loop, the input is the actual weight; the output is the

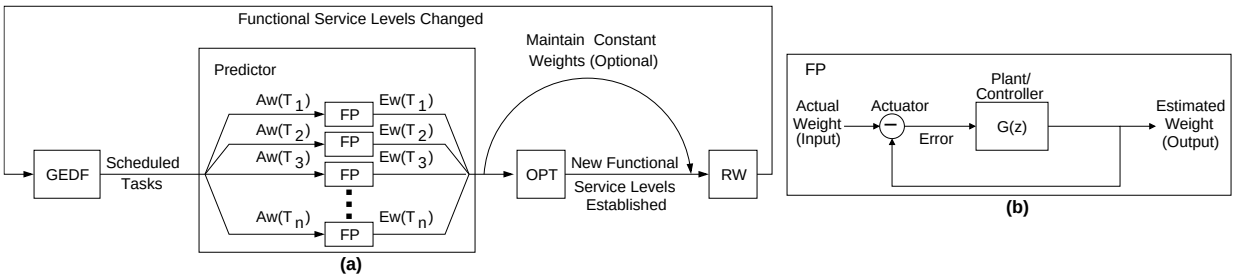

Figure 3. **(a)** The A-GEDF scheduling algorithm. **(b)** The model of A-GEDF's feedback component.

estimated weight; the error is the actual weight minus the estimated weight; and the controller is a *proportional-integral* (PI) controller that uses information about the current error and the sum of all previous errors in order to calculate a new estimated weight. Specifically, the controller is defined as

$$\mathsf{Ew}(T_i^{j+1}) = a \cdot \epsilon(T_i^j) + b \sum_{k=1}^{k=j-1} \epsilon(T_i^k), \tag{1}$$

where $\mathsf{Ew}(T_i^1) = 0$, $\epsilon(T_i^j) = \mathsf{Aw}(T_i^j) - \mathsf{Ew}(T_i^j)$ and both a and b are user-defined values that we discuss shortly. Taking the Z-transform[2] of (1) and rearranging the values we get,

$$\mathsf{G}(z) = \frac{a(z-c)}{z(z-1)}, \tag{2}$$

where $c = (a-b)/a$. (The convention is to write Z-transformed functions as a function of the complex value z.) In control theory parlance, (2) is called the *open-loop transfer function* because it represents the behavior of the controller *ignoring the feedback loop*. The *closed-loop transfer function*, which incorporates both the behavior of the controller and feedback loop is given by

$$\mathsf{H}(z) = \frac{\mathsf{G}(z)}{1+\mathsf{G}(z)} = \frac{a(z-c)}{z^2+(a-1)z-ac}. \tag{3}$$

From the above equation, the predictor has a *closed-loop zero* (the value of z for which $H(z) = 0$) at $z = c$, and *closed-loop poles* (the values of z for which $H(z)$ is undefined) at

$$\frac{(1-a) \pm \sqrt{(a-1)^2 + 4ac}}{2}. \tag{4}$$

Let $\mathcal{P}_m$ denote the pole from (4) that is the farthest from the origin. We henceforth use $\mathcal{R}(\mathcal{P})$ and $\theta(\mathcal{P})$ to denote, respectively, the radius and angle (in radians) of the pole $\mathcal{P}$ in polar-complex form. Because the predictor has two closed-loop poles, it is a *second-order system*. This is the reason why we chose to use a PI controller instead of a *proportional-integral-derivative* (PID) controller. Since PID controllers are *third-order systems*, *i.e.*, such systems have three poles, the transient response analysis is substantially more complex. In fact, the typical means for determining the transient response of a third-order system is to approximate it as a second-order system.

Feedback characteristics. The stability, steady-state error, and transient response of the predictor can be easily derived from (2), (3), and (4) by using standard feedback control techniques. Due to space constraints, we present a limited overview of these characteristics in the body of this paper, and refer the reader to an appendix of the paper, which can be found on the third author's webpage at `http://www.cs.unc.edu/~anderson`, for a thorough discussion of these topics. The predictor is *stable* if $\mathcal{R}(\mathcal{P}_m) < 1$ holds. The predictor is *unstable* if $\mathcal{R}(\mathcal{P}_m) \geq 1$ holds. If the predictor is unstable, then it cannot make meaningful predictions. Since we are using a PI controller, the steady-state error for the predictor in response to a *step input* (*i.e.*, a sudden change from one actual weight to another) is zero and the steady-state error for a *ramp input* (*i.e.*, the actual weight changes at a constant rate) is a non-zero value given as a function of a and c (given in the appendix). The transient response is also a function of a and c (as specified in the appendix). The fact that a PI controller has zero steady-state error for a step response is the reason why we chose a PI controller instead of a *proportional-derivative* (PD), which would have a superior transient response but would have a non-zero value for a step input (*i.e.*, if the actual weight is constant, a PD controller would still have error).

3.2. Optimization

As mentioned above, the optimization component of A-GEDF uses the estimated weights of tasks in order to choose service levels for each task. There are a variety of different methods for implementing this component depending on what metric the user wants to optimize and the behavior of $\mathsf{g}(T_i, e, k, q)$.

For example, suppose the objective is to optimize the total importance value in the system. In this case, if the relation-

[2]The *Z-transform* is a function used for analytic purposes that maps a formula to the *frequency-domain*, *i.e.*, as a function of frequencies. Discussing the Z-transform in detail is beyond the scope of this paper, and we refer the reader to [17] for a review of this subject.

ship between the importance value and weight is linear (like T_1 in Fig. 1), then an approximate solution for this objective could be achieved by assigning the highest service level possible to those tasks with the highest *value density*, as given by,

$$\frac{\mathsf{v}(T_i, |\mathsf{SL}(T_i)|) - \mathsf{v}(T_i, 1)}{\mathsf{g}(T_i, \mathsf{Ew}(T_i^j), k, |\mathsf{SL}(T_i)|) - \mathsf{g}(T_i, \mathsf{Ew}(T_i^j), k, 1)},$$

while ensuring at least every task receives its minimum service level and the system is not over-utilized. (This approach is similar to the *highest-value-density-first* approach used by Lu *et al.* [15].) On the other hand, if the relationship between the importance value and weight is non-linear (like T_2 in Fig. 1), then an approximate solution for this objective could be achieved by using nonlinear programming techniques such as *steepest descent* or *Newton's method* [5]. If exact solutions are required, then techniques like *branch-and-bound* can be used offline, and the optimization component could then switch between several predetermined system states.

It is important to note that the use of weight translation functions is the reason why the optimization component is extensible because it allows any optimizing function to assess the impact of changing the functional service level. In prior work on adaptive real-time systems, the two primary methods for optimizing service levels have been to assume either that each service level has a "nominal" utilization [15] or the relationship between the service level and importance value is linear [16]. As we discussed in Sec. 2, the problem with the first approach is that assessing a meaningful "nominal" utilization may be difficult if not impossible for many applications. The problem with the second approach is that there exist applications for which linearity cannot be assumed. For example, consider any video application in which each service level corresponds to a different resolution. Typically, in such a system, as the service level (and by extension the resolution) increases, the amount of benefit to user perception per pixel added decreases. It is easy to see that in such a scenario, the relationship between importance value and estimated weight is nonlinear.

3.3. Reweighting

Whenever a task is reweighted (*i.e.*, changes its functional service level) either by the optimization component or by the main A-GEDF algorithm, its code segment and/or period may change. If no job of a task T_i is active when T_i changes its functional service level from the ℓ_0^{th} to ℓ_1^{th} service level at time t, then the change is simple—the next released job of T_i has the period and code segment associated with the ℓ_1^{th} service level. If a job of T_i is active at t, then the situation is more complicated. For the remainder of this section, let T_i^j denote the active job of T_i at t. When a task with an active job reweights, there can be a difference between when it "initiates" the change and when the change is "enacted." The time at which the change is *initiated* is defined externally to the reweighting component (by either the optimization component or the main A-GEDF algorithm); the time at which the change is *enacted*, *i.e.*, the functional service level is changed, is dictated by reweighting rules.

Changing the code segment. Whether the code segment of the task T_i that released T_i^j can change depends on the implementation of T_i. For example, if the ℓ_0^{th} and ℓ_1^{th} service levels have substantially different code segments, then T_i^j cannot change its code segment. On the other hand, suppose that the difference between the code segments for the ℓ_0^{th} and ℓ_1^{th} service levels is simply the number of iterations in a loop. Then, as long as T_i^j is not complete and this change would not cause either $\mathsf{Ew}(T_i^j) > 1$ or $\sum_{T_a^b \in \mathsf{ACT}(t)} \mathsf{Ew}(T_a^b) > M$, T_i^j can change its code segment immediately. Moreover, if the code segment is changed, then $\mathsf{Ew}(T_i^j)$ is changed to

$$\frac{\max(\mathsf{Nw} \cdot \mathsf{p}(T_i, \ell_1), \mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t))}{\mathsf{p}(T_i, \ell_0)},$$

where $\mathcal{S}$ is the A-GEDF schedule, and $\mathsf{Nw} = \mathsf{g}(T_i, \mathsf{Ew}(T_i^j), \ell_0, \ell_1)$. Notice that the estimated amount of time for which T_i^j will execute as a consequence of changing its code segment is the larger of the amount of time it has already been schedule by time t, *i.e.*, $\mathsf{A}(\mathcal{S}, T_i^j, \mathsf{r}(T_i^j), t)$, and the amount of time that T_i^j would have been scheduled if the ℓ_1^{th} service level was the functional service level at $\mathsf{r}(T_i^j)$, $\mathsf{Nw} \cdot \mathsf{p}(T_i, \ell_1)$. Thus, the estimated weight of T_i^j is the estimated amount of time that T_i^j will be scheduled divided by $\mathsf{p}(T_i, \ell_0)$. For example, consider the three-processor system depicted in Fig. 4. In this example, $\mathsf{Ew}(T_1^1)$ is changed to $4/7$, $\mathsf{Ew}(T_2^1)$ is changed to $2/7$, and $\mathsf{Ew}(T_3^1)$ is changed to $1/7$. Notice that $\mathsf{Ew}(T_3^1)$ is changed to $1/7$ even though $\mathsf{g}(T_3, 3/7, \ell_0, \ell_1) = 1/14$. The reason for this is that $\mathsf{A}(\mathcal{S}, T_3^1, 0, 1) > \mathsf{Nw} \cdot \mathsf{p}(T_i, \ell_1)$.

Changing the period. The rules for changing the period of a task have been presented in prior work [8]. Unfortunately, due to space constraints, we cannot describe these rules in full detail. For our purposes, the most important aspect of these rules is that weight changes cannot always be immediately enacted. This is because doing so may cause unbounded tardiness. Thus, in A-GEDF, whenever the optimization component establishes new service levels for all tasks in the system, service level decreases must be *enacted* before service level increases can be *initiated*. If service level increases were immediately enacted, then A-GEDF would knowingly overload the system, which is against the design objectives of A-GEDF. For example, consider the one-processor system depicted in Fig. 5. In this example, T_1, T_2, and T_3 are allowed to decrease their weights immediately after being scheduled. Thus, the system is never over-

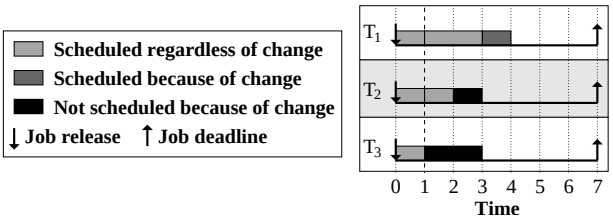

Figure 4. A three-processor system T scheduled by A-GEDF with three tasks, all of which have a period of 7, an estimated weight of $3/7$, and in the absence of a weight change, would be scheduled for 3 time units. At time 1, all three tasks experience a service level change that changes the code segment for each job. Moreover, the value of $\mathsf{g}(T_i, \mathsf{Ew}(T_i^j), \ell_0, \ell_1)$ as a result of the functional service level change is $4/7$, $2/7$, and $1/14$, for T_1, T_2, and T_3, respectively. Thus, as a result of the change, T_1^1 executes for 4 time units, T_2^1 executes for 2 time units, and T_3^1 executes for 1 time unit.

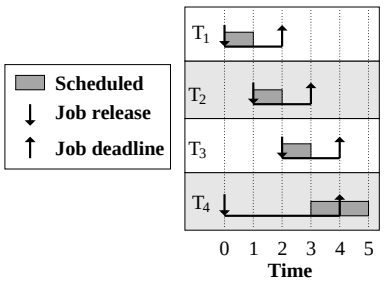

Figure 5. A one-processor system T scheduled by EDF with four tasks. T_1, T_2, and T_3 all have a period of two, an actual execution cost of one, decrease their weights to zero immediately after being scheduled, and have one job released at, respectively, times 0, 1, and 2 (each increases its weight from zero to 0.5 when its first job is released). T_4 has an actual execution cost of two and a period of four. Since tasks are allowed to decrease their weights immediately, the system is never over-utilized, yet T_4^1 misses a deadline.

utilized; however, T_4 misses a deadline by one time unit, which should not occur under EDF.

3.4. User-Defined Threshold

Choosing a specific user-defined threshold for invoking the optimizer will depend largely on the targeted application. Some possible thresholds could include a duration of time, a substantial change in the estimated weight for one task, or a substantial change in the total estimated weight for all tasks. While running the optimizer more frequently will increase the accuracy of the system, it will also increase the amount of time the scheduler is active with the system not producing "useful" work. Additionally, as we discussed in Sec. 3.3, the reweighting rules cannot always be enacted immediately. Thus, if the optimizer is called before all changes have been enacted, then it may produce an inaccurate result. Notice that, if the weight translation function is accurate, then after all reweighting events have been enacted the system will remain in an "optimal" state, unless the actual weight of a task changes. Thus, if the separation between optimizer invocations is sufficiently large for all tasks to enact their functional service level changes (*i.e.*, at least the largest period of a job in the system), then it is possible to guarantee that no task will unnecessarily "thrash" between service levels.

4. Implementation

Recently, our research group developed a testbed called LITMUS$^{\mathrm{RT}}$ (**LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems), which is an extension of Linux (currently, version 2.6.20) that allows different multiprocessor scheduling algorithms to be linked as plug-in components [11]. Our implementation of A-GEDF consists of both a user-space library and kernel support added to LITMUS$^{\mathrm{RT}}$. In this section, we briefly discuss both parts. Unfortunately, a detailed description is not possible, due to space constraints.

Because LITMUS$^{\mathrm{RT}}$ was designed for sporadic tasks provisioned using WCETs, several modifications were needed to support adaptable sporadic tasks. These included: adjusting the internal structure of the task control block to allow each task to have multiple service levels; disabling the enforcement of WCETs to allow tasks to overrun their expected allocation; and modifying LITMUS$^{\mathrm{RT}}$ to allow task statistics such as actual execution times to be gathered.

After making these changes, we implemented A-GEDF by changing the GEDF scheduling algorithm (which had already been implemented in LITMUS$^{\mathrm{RT}}$) in two ways. First, we introduced a system call to query the kernel in order for a task to determine its current service level. Second, we implemented the feedback, optimization, and reweighting components in kernel space. Since in Linux floating point operations cannot be used in kernel space, these components were implemented using fixed-point calculations instead.

5. Experiments

In this section, we report on experiments conducted using LITMUS$^{\mathrm{RT}}$ to evaluate the performance of A-GEDF when running the core operations of Whisper.

Whisper. As noted earlier, Whisper tracks users via speakers that emit white noise attached to each user's hands, feet, and head. Microphones located on the wall or ceiling receive these signals and a tracking computer calculates each

speaker's position by measuring signal delays. Whisper is able to compute the time-shift between the transmitted and received versions of the sound by performing a *correlation* calculation on the most recent set of samples. By varying the number of samples, Whisper can trade measurement accuracy for computation—with more samples, the more accurate and more computationally intensive the calculation. As the signal-to-noise ratio decreases, the number of samples is increased to maintain the same level of accuracy. As the distance between a speaker and microphone increases, the signal strength decreases. This behavior (along with the use of predictive techniques mentioned in the introduction) can cause frequent task share changes.

Experimental system set-up. Unfortunately, it is currently not feasible to produce experiments involving a complete implementation of Whisper, for two reasons. First, as Whisper currently exists, it is single-threaded (and non-adaptive) and consists of several thousand lines of code. Converting it to a multi-threaded implementation is a non-trivial task. Indeed, because of this, it is *essential* that we first understand the scheduling and resource-allocation tradeoffs involved. Second, support for *task synchronization* is required, and this issue is beyond the scope of this paper. For these reasons, we have chosen to conduct our evaluation using the core operation of Whisper (*i.e.*, a correlation computation). (We emphasize that the experiments discussed here involved running *real* code on a *real* OS kernel and are not merely simulations.)

The development platform used in our experiments is an SMP consisting of four 32-bit Intel(R) Xeon(TM) processors running at 2.7 GHz, with 8K L1 instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory. For each task, each job was implemented as a loop in which the core operation of Whisper is performed iteratively. The exact manner in which jobs behave is discussed below. In implementing the optimizing component, we assumed that there is a linear relationship between importance value and estimated weight, and attempted to maximize the total importance value for all tasks, as discussed earlier. The optimizer was configured to run at least once every second, and also whenever the estimated weight of a task changed by at least 50%, or upon a job completion, the total estimated system weight exceeded four. However, it was constrained to run at most once every 200ms. Note that in a full Whisper implementation, these choices could possibly be improved upon by carefully considering human-factors issues of relevance to virtual-reality systems.

In all experiments, we defined the PI controller using $a = 0.102$ and $c = -1.975$. We chose these values because we believe that they represent a good tradeoff between transient response and steady-state error (for a ramp input).

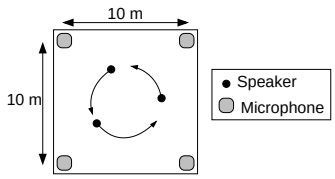

Figure 6. The simulated Whisper system.

Whisper experiments. In our Whisper experiments, we simulated three speakers (one per object) revolving at a speed of 2m/s (this is within the speed of human motion) in a 10m $\times$ 10m room with a microphone in each corner, as shown in Fig. 6. Tracked objects were sampled at a rate of 2,000 kHz, and the distance of each object from the room's center was set at 5m. While this test scenario may seem simple (since the path of each object is simple and pre-determined), it is actually a challenging test case for Whisper. This is because objects moving at a relatively high speed of 2m/s require significant computational resources to track. Moreover, while it is possible to simulate objects that start, stop, and change directions, such scenarios actually require *less* computational resources and adapt task service levels *less* frequently, because user motion is typically slower when motion is not continuous.

In the above scenario, one task is required per speaker-microphone pair, for a total of 12 tasks. Each task was configured to have three service levels, with periods/importance values of 66ms/0.25, 33ms/0.5, 22ms/0.75, respectively, and $\mathrm{g}(T_i, e, 1, 2) \approx 5e$ and $\mathrm{g}(T_i, e, 1, 3) \approx 9e$. The importance values were selected somewhat arbitrarily after some trial-and-error experimentation; in an actual deployment, user studies would be required to assess the impact of different settings. Since the weight/importance value relationship is linear, we used the approach in Sec. 3.2 to optimize the system. The other parameters were selected based upon the existing Whisper implementation. In Whisper, the QoS provided is directly related to the number of correlation computations (CCs) performed per second. When the signal-to-noise ratio decreases, the number of CCs must be increased to maintain the same QoS. Similarly, the QoS provided can be increased by increasing the number of CCs per second. A change in the functional service level of a Whisper task changes the number of CCs per second. We estimated that the existing Whisper implementation, if implemented on our test platform, would perform approximately 27,600,000 CCs per second in the average case. The task periods and $\mathrm{g}(T_i, e, \ell_1, \ell_2)$ values given above were defined so that the average number of CCs per second for the second service level matches this rate. Note that, because the code segments of the three service levels differ only in the number of CCs performed, the code segment of an active job can be changed.

The first experiment we discuss was conducted to see if adaptivity is even needed in implementing Whisper. In this

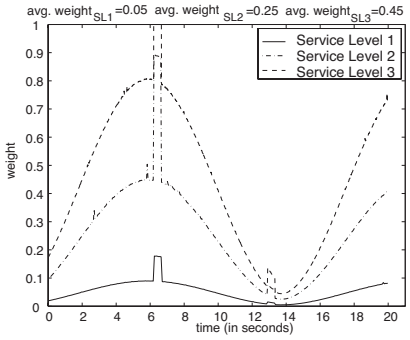

Figure 7. The actual weight of a Whisper task at three different service levels over a 20-second run with two bursts of noise at approximately times 6 and 13.

experiment, we ran one task as a normal Linux task for 20 seconds at all three service levels and measured its actual weight. Results are shown in Fig. 7. After approximately six and 13 seconds, the system experiences 48ms of noise that doubles the number of correlation computations required per job. The average weight of the task at the first, second, and third service level is, respectively, 0.05, 0.25, and 0.45. Notice that, for a system with 12 tasks, operating each task at its highest-service level and allocating it a processor share based on its worst-case weight (which is 1.0) gives a total actual weight of 12, which substantially over-utilizes the system. Even with an average-case provisioning, the system is still over-utilized, as the total actual weight is 5.4 in this case. On the other hand, configuring each task to run at its second service level using its average-case weight gives a total actual weight of 3.0, which does not over-utilize the system; however, using a constant average-case allocation would likely cause the system to be over-utilized when noise is encountered. Thus, from this experiment, we can infer that, in order for Whisper to schedule tasks at any service level higher than the lowest one, adaptive scheduling is needed (as is a multiprocessor).

In the second experiment, we ran all 12 Whisper tasks on LITMUS$^{\mathrm{RT}}$, scheduled by A-GEDF, for 20 seconds with 1,100ms bursts of ambient noise after 5.5 and 12.3 seconds that double the number of CCs required to maintain the same QoS. Insets (a) and (b) of Fig. 8 depict the actual and estimated weights and error for two different tasks as a function of time. It also shows the functional service level for each task as a function of time. (The other 10 tasks exhibited similar behaviors, but plots for them are omitted due to space constraints.) Inset (c) shows the total actual weight and the total importance value of the system as a function of time. There are several interesting things to notice about these graphs. First, for the tasks depicted in insets (a) and (b), error is typically within the range $[-0.05, 0.05]$. Second, whenever the functional service level changes or the system encounters noise, error briefly spikes but quickly falls back within the range $[-0.05, 0.05]$. Third, when the task in inset (a) encounters noise at time 5.5, its service level is changed and there is a substantial drop in its weight; however, when it encounters noise at time 12.3, its service level is not decreased because its actual weight is so low. Note that its low weight at this time is coincidental: its weight varies between times 8 and 20 as depicted because of the movement of the corresponding object, and that object happens to be closest to the microphone for which this task is defined at approximately time 14. Fourth, when a job of the task in inset (b) completes after the noise at time 12.3, the total estimated weight is greater than four, so the optimizer is invoked causing this task to decrease its service level. This is why the actual weight of this task is briefly greater than one. Fifth, the total utilization of the system is typically close to 4, and the system is briefly over-utilized when noise is encountered. Because the total actual weight is always close to 4, *this system would not be schedulable using a partitioning approach*. Sixth, the total importance value of the system is typically in the range $[7.5, 8]$ and drops below 6.0 only when noise is encountered. In contrast, if tasks were statically assigned their second service level and scheduled by GEDF, then the total importance value would never *exceed* 6.0.

6. Conclusion

We have presented an adaptive framework, A-GEDF, that uses feedback and optimization techniques, and allows the processor shares of tasks running on a multiprocessor to be controlled dynamically at runtime. We have also presented an evaluation of A-GEDF's performance that uses Whisper as a test case. These experiments clearly demonstrate the need for adaptivity in this application. They also show that A-GEDF is capable of enacting needed adaptations in a way that enhances overall QoS. In future work, we plan to incorporate synchronization support into A-GEDF and investigate the application of feedback control in other (non-GEDF-based) multiprocessor scheduling algorithms.

References

[1] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Sys. Magazine*, 23(3):74–90, June 2003.

[2] T. Abdelzaher, E. Atkins, and K. Shin. QoS negotiation in real-time systems and its application to automated flight control. *IEEE Trans. Comput.*, 49(11):1170–1183, 2000.

[3] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the 23rd IEEE Real-Time Sys. Symp.*, pages 71–80, 2002.

[4] R. Al-Omari, G. Manimaran, M. V. Salapaka, and A. K Somani. Novel algorithms for open-loop and closed-loop scheduling of real-time tasks in multiprocessor systems based

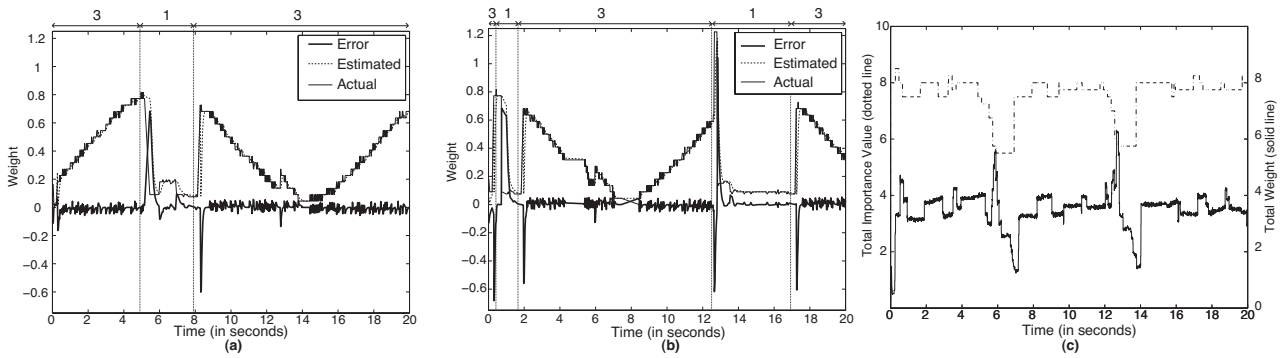

Figure 8. Results from executing 12 Whisper tasks for 20 seconds. **(a) & (b)** Actual and estimated weights and error (difference between the actual and estimated weight) for two tasks as a function of time. The service level for each task is depicted across the top of each inset. **(c)** Total actual weight and importance value as a function of time.

on execution time estimation. In *Proc. of the 2003 Int'l Parallel and Distributed Processing Symp.*, page 7, 2003.

[5] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, second edition, 1999.

[6] A. Block and J. Anderson. Accuracy versus migration overhead in multiprocessor reweighting algorithms. In *Proc. of the 12th Int'l Conf. on Parallel and Distributed Sys.*, pages 355–364, 2006.

[7] A. Block, J. Anderson, and G. Bishop. Fine-grained task reweighting on multiprocessors. *Journal of Embedded Computing*, special issue on multiprocessor real-time scheduling, to appear.

[8] A. Block, J. Anderson, and U. Devi. Task reweighting under global scheduling on multiprocessors. *Real-Time Sys.*, special issue on selected papers from the 18th Euromicro Conf. on Real-Time Sys., 39(1-3):123–167, 2008.

[9] S. Brandt, G. Nutt, T. Berk, and J. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proc. of the 19th IEEE Real-Time Sys. Symp.*, pages 307-316, 1998.

[10] S. Brandt and G. Nutt. Flexible soft real-time processing in middleware. *Real-Time Syst.*, 22(1-2):77–118, 2002.

[11] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Sys. Symp.*, pages 111–126, 2006.

[12] T. Cucinotta, L. Palopoli, L. Marzario, G. Lipari, and L. Abeni. Adaptive reservations in a linux environment. In *Proc. of the 10th IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 238–245, 2004.

[13] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *Real-Time Sys.*, 38(3):237–288, 2008

[14] C. Lu, J. Stankovic, T. Abdelzaher, T. Gang, S. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proc. of the 21st IEEE Real-Time Sys. Symp.*, pages 13–23, 2000.

[15] C. Lu, J. Stankovic, S. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Sys.*, 23(1-2):85–126, 2002.

[16] P. Marti, C. Lin, S. Brandt, M. Velasco, and J. Fuertes. Optimal state feedback based resource allocation for resource-constrained control tasks. In *Proc. of the 25th IEEE Real-Time Sys. Symp.*, pages 161–172, 2004.

[17] C. Phillips and H. Nagle. *Digital Control System Analysis and Design*. Prentice Hall, third edition, 1995.

[18] D. Sahoo, S. Swaminathan, R. Al-Omari, M. Salapaka, G. Manimaran, and A. Somani. Feedback control for real-time scheduling. In *Proc. of the American Control Conf.*, Vol. 2, pages 1254–1259, 2002.

[19] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proc. of the 22nd IEEE Real-Time Sys. Symp.*, pages 59–70, 2001.

[20] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proc. of the 17th IEEE Real-Time Sys. Symp.*, pages 288–299, 1996.

[21] H. Tokuda and T. Kitayama. Dynamic QoS control based on real-time threads. In *Proc. of the 4th Int'l Workshop on Network and Operating System Support for Digital Audio and Video*, pages 114–123, 1994.

[22] N. Vallidis. *WHISPER: A Spread Spectrum Approach to Occlusion in Acoustic Tracking*. PhD thesis, The University of North Carolina at Chapel Hill, North Carolina, 2002.