On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors

Piyush Garyali, Matthew Dellinger, and Binoy Ravindran

ECE Dept., Virgina Tech, Blacksburg, VA 24061, USA
piyushg@vt.edu, mdelling@vt.edu, binoy@vt.edu

Abstract. We consider the problem of scheduling dependent real-time tasks for overloads on a multiprocessor system, yielding best-effort timing assurance. The application/scheduling model includes tasks with time/utility function time constraints, mutual exclusion constraints, and arbitrary arrival, execution-time and resource access behaviors, with the timeliness optimization objective of maximizing the total accrued utility while ensuring mutual exclusion constraints and deadlock-freedom. Since this problem is NP-hard, we develop a class of polynomial-time heuristic algorithms, called the *Global Utility Accrual* (GUA) class of algorithms, and present two algorithm instances, namely, *Non-Greedy Global Utility Accrual* (NG-GUA) and *Greedy Global Utility Accrual* (G-GUA). We establish several properties of the algorithms including conditions under which optimal total utility is accrued, mutual exclusion constraints are satisfied, and deadlock-freedom is achieved. We develop a Linux-based real-time kernel called ChronOS, extended from the `CONFIG_PREEMPT_RT` real-time Linux patch. ChronOS provides a framework for the implementation and plugging-in of a variety of multiprocessor schedulers. Our experimental study with ChronOS reveals the effectiveness of GUA algorithms under a broad range of workloads.

Keywords: real-time, multiprocessors, scheduling, time/utility function

1 Introduction

Recently, there has been a shift in the computer industry from increasing clock rates to designing multi-core and hyper-threading architectures in a quest to produce faster computers [29]. Motivated by heat and power issues, most chip manufacturers have chosen the route of increasing system- and chip-level parallelism, as opposed to increasing clock rates, to improve performance. Consequently, the design of multiprocessor real-time scheduling algorithms has become important in order to allow real-time applications to take advantage of these emerging architectures.

One unique aspect of multiprocessor real-time scheduling is the degree of run-time migration allowed for job instances of a task across processors (at scheduling events). Example migration models include: (1) *full migration*, where jobs are allowed to arbitrarily migrate across processors during their execution. This

usually implies a global scheduling strategy, where a single shared scheduling queue is maintained for all processors and a processor-wide scheduling decision is made by a single (global) scheduling algorithm; (2) *no migration*, where tasks are statically (off-line) partitioned and allocated to processors. At run-time, job instances of tasks are scheduled on their respective processors by processors' local scheduling algorithm, such as single processor scheduling; and (3) *restricted migration*, where some form of migration is allowed—e.g., at job boundaries.

The Pfair class of algorithms [7] that allow full migration and fully dynamic priorities have been shown to be theoretically optimal—i.e., they achieve a *schedulability utilization bound*, U, below which all tasks meet their deadlines, that equals the total capacity of all m processors i.e., $U = m$. Under Pfair, tasks are decomposed into several small uniform segments, which are then quantum-scheduled and may cause frequent scheduling and migration. Thus, algorithms other than Pfair have also been intensively studied though their utilization bounds are lower. Examples of global algorithms include global-EDF [8], global-non-preemptive-EDF [6] with a bound $U \approx m/2$, LLREF [11], LRE-TL [15], PG/PCG [9], NVNLF [14] with a bound $U = m$, and global-RMS with a bound $U \approx 3m/8$ [4]. In the partitioned space, examples include partitioned dynamic priority algorithms, such as partitioned-EDF [3] with a bound $U \approx m/2$, and fixed priority algorithms such as partitioned-DMS with a bound $U \approx m/2$ [26] and PDMS-HPTS-DS with a bound of 65% [22].

Majority of these scheduling efforts focus on application contexts where key aspects of application behavior—e.g., task arrivals, execution times, resource accesses—are deterministically bounded or known. Although this is an extremely important subspace of the real-time problem space, there also exist some real-time applications with behaviors outside this envelope—e.g., unpredictable task arrival and execution-time behaviors, caused due to data- and context-dependent executions, resulting in transient and permanent overloads (i.e., $U > m$) [13, 2]. During overloads, applications such as [13] desire graceful timeliness degradation and "best-effort" timing assurance in the sense that as many processor cycles as needed are assured to be allocated to the most important task, less so are allocated to the least important task, and so on [25, 19]. (Note that task importance may be orthogonal to task urgency.) An interesting feature of these applications is that their task execution-time magnitudes are relatively longer—e.g., milliseconds to minutes. This allows relatively time-expensive real-time scheduling.

Past works on scheduling for overloads with best-effort timing assurances (e.g., LBESA [25], DASA [12], GUS [23, 24], D^{over} [20]) have focused on single processor systems, with a few exceptions.[1] The only efforts in this space that consider multiprocessors include MOCA [21] and gMUA [10]. Both these algorithms, however, exclude task dependencies that arise due to synchronization constraints.

---

[1] LBESA's and DASA's design were directly motivated by the "best-effort" real-time notion, and have been transferred to the application in [13] due to its matching operational requirements.

In this paper, we focus on this multiprocessor problem space, directly motivated by applications such as [13, 2]. We consider tasks with time/utility function (TUF) time constraints [18] that subsume deadlines and allow task urgency to be expressed independent of task importance. Tasks have unknown arrival behaviors and are subject to execution overruns, causing overloads. In addition, tasks have mutual exclusion constraints; they use lock-based concurrency control, with unknown lock-access and release behaviors. We consider the timeliness objective of maximizing the total accrued timeliness utility, while satisfying mutual exclusion constraints and freedom from deadlocks. This problem is NP-hard. We develop a class of polynomial-time heuristic algorithms called the GUA class of algorithms, and present two algorithm instances, namely, NG-GUA and G-GUA. We establish several properties of the algorithms including conditions under which optimal total utility is obtained, mutual exclusion constraints are satisfied, and deadlock-freedom is achieved.

We develop a Linux-based real-time OS kernel called ChronOS, extended from the `CONFIG_PREEMPT_RT` real-time Linux patch, which provides optimized interrupt service latencies and real-time locking primitives. ChronOS provides a scheduling framework for the implementation of a broad range of scheduling algorithms as scheduler plugins. We implement the GUA algorithms and their competitors (e.g., G-EDF, G-NP-EDF, gMUA, P-EDF, P-DASA) in ChronOS and conduct experimental studies. Our results reveal the effectiveness of the GUA algorithms under a broad range of workloads.

Thus, the paper's contribution is the GUA class of algorithms that allow tasks to be subject to run-time uncertainties, overloads and dependencies, and yield optimal total utility (when possible) and best-effort timeliness behavior otherwise — the first such multiprocessor real-time scheduling algorithms to do so.

The rest of the paper is organized as follows: Section 2 describes our models and objective. Section 3 presents the GUA class of algorithms. The algorithms' rationale, design, and properties are described in this section. We report our experimental studies in Section 4. Finally, we conclude in Section 5.

2 Models and Objective

We consider Clark's phase abstraction [12] as the unit of scheduling. A phase describes a single flow of execution. Phases arrive arbitrarily and may be preempted arbitrarily.

Phases have time constraints. A time constraint has a "scope"—a segment of the phase control flow that is associated with the time constraint [28]. Such a scope is called a "scheduling segment". Each phase has a single scheduling segment. A phase i's scheduling segment's time constraint is specified using a TUF. TUFs can only be downward step-shaped—i.e., a constant maximum utility u_i is accrued if the segment completes before a deadline time d_i; zero utility otherwise.

A good-faith estimate of a phase i's scheduling segment's execution time, e_i, is available (through off-line measurements). This time estimate is not the worst-case; it can be violated at run-time (e.g., due to context dependence) and can cause processor overloads.

A phase enters and exits a scheduling segment by invoking scheduler APIs—e.g., Real-Time CORBA's [28] `begin_scheduling_segment` and `end_scheduling_segment` APIs. When a scheduling segment is entered, a phase passes its scheduling parameters (e.g., d_i,u_i,e_i) to the API.

Phases may access non-CPU resources, which in general, are serially reusable. Resources can be shared, and can be subject to mutual exclusion constraints. A phase may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped, or disjoint. Phases may request and release resources arbitrarily—i.e., which phase needs which resource and in what order is unknown.

An `abort_handler` is associated with each phase scheduling segment. We consider a termination model for all failures encountered during phase executions including time-constraint violations and logical errors, which raises an exception that is handled by the phase. The handler performs compensating actions that are necessary to avoid inconsistencies and ensure the safety of the external state.

Scheduling Objective. Our objective is to schedule the phases on an m-processor system such that the sum of the utility accrued by the completion of the phases is maximized, as much as possible, while satisfying phase mutual exclusion constraints and ensuring deadlock-freedom. Additionally, the number of phase deadlines missed must be minimized as much as possible.

This problem is NP-hard because its one-processor version is NP-hard [12]. Thus, the GUA class of algorithms that we present are polynomial-time heuristic algorithms.

3 GUA Class of Algorithms

3.1 Basic Rationale

Since the phase model is dynamic—i.e., when phases will arrive, how long they will execute, which set of resources will be needed by which phases, the length of time for which those resources will be needed, the order of accessing the resources are all statically unknown—future scheduling events such as new phase arrivals and new resource requests cannot be considered at a scheduling event. Thus, a schedule must be constructed solely exploiting the current system knowledge.

Since the primary scheduling objective is to maximize the total utility, a reasonable heuristic is a "greedy" strategy: Favor "high return" phases over low return ones, and complete as many of them as possible before phases' termination times and also as early as possible.

The potential utility that can be accrued by executing a phase is an indication of its "return on investment". We measure this using a metric called the *Potential Utility Density* (or PUD) pioneered in [12]. A phase's PUD measures the utility

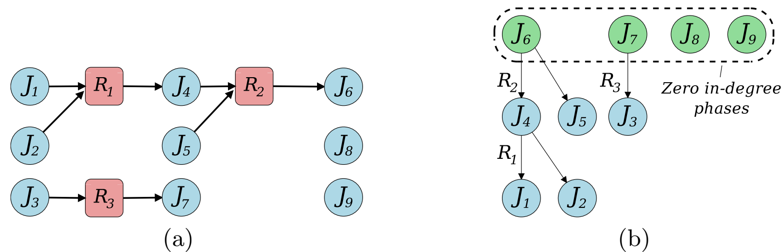


Fig. 1: (a) A phase and resource dependency chain (b) DAG representation showing the zero in-degree phases

that can be accrued per unit time by immediately executing the phase and those phase(s) that it (directly or transitively) depends upon for locked resources.

Ensuring Mutual Exclusion: The presence of phase dependencies can result in many phase dependency chains, similar to the single-processor case. But unlike the single-processor case, since there are $m > 1$ processors for the multiprocessor case, up to m of these chains (or phases at the head of those chains) can be potentially executed. In [12], the dependency chains are computed at the per phase level which works well for the single-processor case as only one phase needs to be selected. However, this method cannot be applied to the multiprocessor case as, in order to ensure mutual exclusion, two phases that are dependent on each other should not be allowed to be executed on processors concurrently. Hence, there is a need to find the dependency relationship of all phases in an effective way. We solve this problem by constructing a directed acyclic graph (DAG) to represent the dependency relationship between phases. Fig 1(a) shows the dependency relationship between phases and resources (e.g., phase J_1 requires a resource that is owned by phase J_4). In Fig 1(b), we represent the dependency relationship using a DAG (the node represents the phase and the edge represents the resource relationship). Thus, at the end of the graph creation, we consider the zero in-degree (ZID) phases, which have zero input edges and hence are not dependent on other phases, as eligible for the final schedule.

Maximizing Accrued Utility: Once the ZID phases are found, we need to determine the m ZID phases that have the highest execution eligibility. The PUD metric [12] has been shown to be highly effective in determining phase execution eligibility for the single-processor case as a single phase needs to be selected at the end of the schedule. On multiprocessors, unlike the single processor case, many non-dependent phases can be concurrently dispatched for execution. However, the PUD metric alone cannot be used to pick one phase over another, as there could be a ZID phase that currently owns a resource, blocking other phases in the system, but has a lower PUD. As a result, the phase could be pushed to the back of the queue preventing other eligible phases that are currently blocked on it, from executing. The challenge is to find a metric that provides a way to represent the overall benefit the system can accrue if a particular phase is selected for execution. To solve this, we define two metrics—*Local Value Density* (LVD), which is equivalent to the PUD of a phase, and *Global Value Density*

(GVD), which is defined as the sum of the LVDs of the individual phases that are in a dependency relation with a ZID phase. For example, in Fig 1(b), GVD(J_6) = LVD(J_6) + LVD(J_4) + LVD(J_5) + LVD(J_1) + LVD(J_2). The GVD for a ZID phase J represents the aggregate value density for the entire dependency chain of J. This gives a fair representation of the dependency relationship for that phase, and thus provides the highest execution eligibility for a phase that is currently blocking other phases.

Deadlock Detection and Resolution: A deadlock can occur when a phase J_a, which owns resource R_a, makes a request for another resource R_b, owned by a phase J_b, wherein phase J_b directly or through its dependency chain requests for the resource R_a (owned by J_a). Thus, a deadlock represents a cycle in the dependency chain of a phase which can be detected using a cycle detection algorithm. In [12], deadlocks are detected at the phase level when the individual phase dependency chains are being computed. However, as we construct a DAG to represent the dependency relationship of all the phases, deadlock detection and resolution can be integrated with DAG construction. A DAG can be created for a phase and all its dependencies in a single pass. During each step, phases in the dependency chain can be maintained in a list such that if a dependent phase is again added to the list, a deadlock is detected. In order to resolve the deadlock, one of the phases needs to be rejected. In the design of GUA algorithms, we select the least LVD phase as it contributes the least utility to the total accrued utility.

3.2 Overview

GUA's scheduling events include the arrival of a phase, completion of a phase, a resource request and a resource release. To describe the algorithms we define the variables and auxiliary functions. For a phase `J`, `J.RemExec` is the estimated remaining execution cost of the phase and `J.Utility` denotes the TUF at the time of the scheduling event. The following auxiliary functions are used:

`InsertEdge(J, DepJ)` inserts an edge between phases `J` and `DepJ`.
`RemoveEdge(J)` removes all in-degree and out-degree edges of phase `J`.
`InsertList(J,` σ`)` inserts the phase `J` in the list σ.
`InsDeadLnPos(J,` σ`)` inserts phase `J` in the list σ at its deadline position.
`FindZIDPhases(`σ`)` returns the ZID phases from the list σ.
`RemoveLeastLVD(`σ`)` removes the phase with the least LVD from the list σ.
`RemoveLeastGVD(`σ`)` removes the phase with the least GVD from the list σ.
`FindPIPDeadLn(`σ`)` finds the earliest deadline amongst the dependents of the ZID phases in list σ to ensure Priority Inheritance Protocol (PIP) behavior.
`ComputeGVD(`σ`)` computes the GVD for the ZID phases in the list σ.
`SortByGVD(`σ`)` sorts the list σ by the decreasing value of GVD.
`IsPresent(J,` σ`)` returns `true` if the phase `J` is present in the list σ.
`IsFeasible(`σ`)` returns `true` if schedule in σ is feasible, i.e., the predicted completion time of each phase in σ must never exceed its deadline.
`Owner(R)` returns the phase that holds resource `R`.
`ResRequested(J)` returns the resource requested by phase `J`.

Algorithm 1: Creation of DAG with detection/resolution of deadlocks

```
1:  Procedure: CreateDAGwithDRD (σ_T)
2:  Input: σ_T              // List of released phases
3:  Vars: J, V, next        // Phase pointers
4:  Vars: σ_J               // Phase J's list of dependents
5:  for each phase J in σ_T do
6:      σ_J = φ ;
7:      J.Lvd = J.Utility / J.RemExec ;
8:      InsertList(J, σ_J);
9:      next = Owner(ResRequested(J));
10:     while next ≠ φ do
11:         if IsPhaseAborted(next) then
12:             break;
13:         if IsPresent(next, σ_J) == false then
14:             InsertEdge(J, next) ;
15:             InsertList(next, σ_J);
16:             J = next;
17:             next = Owner(ResRequested(next));
18:         else
19:             V = FindLeastLVD(σ_J);
20:             AbortPhase(V);
21:             RemoveEdge(V);
22:             break;
```

`FindProcessor()` returns the ID of the processor on which the currently assigned phases have the shortest sum of allocated execution times.

`FindProcessor(cpu_mask)` is an extended version of `FindProcessor()` that takes a `cpu_mask` (mask of processors that have been checked earlier and should be avoided). If all the processors are in the mask, it returns `NULL`.

`AddCpuToMask(p, cpu_mask)` adds processor `p` to the `cpu_mask`.

`FindLeastLVD(J)` finds the phase with the least LVD in dependency chain of `J`.

`UpdateCpuEC(p, J, b)` adds `J.RemExec` to the sum of remaining execution times for phases allocated on processor `p` if `b` is true, subtracts otherwise.

`AbortPhase(J)` sends an aborting signal to the phase `J`.

`IsPhaseAborted(J)` returns `true` if phase `J` has been marked for abortion.

`HeadOf(`σ`)` returns the phase `J` which is at the head of the list σ.

Algorithm 1 describes the pseudo-code for `CreateDAGwithDRD(`σ_T`)` that uses the list of phases, σ_T, and creates a DAG representation along with deadlock detection and resolution. We refer to the phase that has requested a resource as a *child* while the phase that owns the resource being requested as a *parent*. In lines 5-22, the algorithm iterates over the list, σ_T, and for each phase, J, checks if a *parent* node exists and adds an edge from the *parent* to the *child* (line 14). In lines 16-17, the algorithm sets the current *parent* node as the new *child* and checks if it has requested a resource. The steps are repeated for all the phases in the dependency chain of J.

In order to detect deadlocks we use list σ_J (line 6) to which we add all the dependencies for phase J (line 15). Before adding an edge between a *child* and a

parent, we check if the phase exists in σ_J (line 13). The existence of the phase in σ_J indicates that the phase has already been added to the graph, thus detecting a deadlock. To resolve the deadlock, we find the least LVD phase in σ_J, abort the phase and remove it from the graph (lines 19-21).

3.3 Non-greedy Global Utility Accrual (NG-GUA)

Algorithm 2 describes the NG-GUA scheduling algorithm. For a given list of phases σ_T, the DAG is created (line 4) using `ComputeDAGwithDRD()`. To ensure mutual exclusion we find the ZID phases and compute their GVD (lines 5-6). In the presence of dependencies, NG-GUA defaults to G-EDF with PIP. We compute the PIP deadlines for each of the ZID phases (line 7). The PIP deadline of a ZID phase J_z is the earliest deadline of a phase J_i which is dependent on J_z. In line 8, we sort the ZID phases by their PIP deadlines. The key idea here is to sort the ZID phases by the deadlines of the phases which have an earlier deadline but are currently blocked on a resource that is being held by the ZID phases, thus ensuring a PIP behavior[2].

In lines 9-11, we use `FindProcessor()` to assign phases to individual processor lists σ_p. The processor that yields the shortest sum of allocated execution times of all jobs in its local schedule is selected for assignment. The rationale for this choice is that the shortest summed execution time processor results in the nearest scheduling event for completing a job after assigning each job.

In lines 12-14, we check each of σ_p lists for schedule feasibility using the `IsFeasible()` method. During overloads, the schedule might not be feasible. Hence, NG-GUA attempts to maximize the total utility by allowing phases that have a higher value density to be executed. In line 14, we remove the phase in σ_p that has the least GVD and check the schedule for feasibility. Lines 12-14 are repeated until a feasible schedule is found. Finally, the head of the final feasible schedule, σ_p, for each processor p is dispatched (lines 15-17). In the absence of dependencies, all the phases are treated as ZID phases and the PIP deadlines for each phase is equivalent to the phase's deadline. Hence, after the sort (line 8), NG-GUA defaults to a G-EDF order.

Algorithm 2 is referred to as non-greedy because it defaults to a deadline order rather than a value density order along with support for priority inheritance protocol, thus following a G-EDF with PIP behavior during underloads and maximizing total accrued utility during overloads. A sample schedule for NG-GUA is presented in [16].

3.4 Greedy Global Utility Accrual (G-GUA)

Algorithm 3 describes the G-GUA scheduling algorithm. Lines 6-8 are similar to the NG-GUA algorithm, described in Section 3.3. We create the DAG, find the ZID phases and compute their GVD. G-GUA does not default to G-EDF with PIP. Hence, we do not need to find the PIP deadlines.

[2] In the absence of dependencies, the PIP deadline of a ZID phase J_z can be considered equal to the deadline of J_z.

Algorithm 2: NG-GUA: Non-greedy Global Utility Accrual

```
1: Input: σ_T          // List of released phases
2: Vars: σ_1 ··· σ_m   // Per processor ready queues for m processors
3: Vars: σ_z           // Zero in-degree phase list
4: ComputeDAGwithDRD(σ_T);
5: σ_z ← FindZIDPhases(σ_T);
6: ComputeGVD(σ_z);
7: σ_z ← FindPIPDeadLn(σ_z);
8: σ_d ← SortByPIPDeadLn(σ_z);
9: for each phase J in σ_d do
10:    p ← FindProcessor();
11:    InsertList(J, σ_p);
12: for each processor p do
13:    while IsFeasible(σ_p) == false do
14:       RemoveLeastGVD(σ_p);
15: for each p processor's schedule σ_p in m do
16:    Job_p ← HeadOf(σ_p) ;
17: return { Job_1, ··· , Job_m };
```

G-GUA differs from NG-GUA in two ways— (i) the ZID phases are sorted by GVD instead of the PIP deadlines (line 9); and (ii) instead of assigning phases to all the processors and then running the feasibility check, G-GUA follows a greedier approach to accrue total utility. For all individual GVD-sorted ZID phases in σ_d (lines 10-24), G-GUA assigns the phase to a processor which has the smallest sum of total phase remaining execution cost and checks for feasibility of schedule on that processor. If the schedule is feasible, G-GUA moves to the next phase in σ_d. However, during overloads, if the schedule is not feasible (after the phase was added to the first processor it was assigned to), G-GUA removes it from that processor's list and tries the same phase on all the other available processors (using `cpu_mask`, lines 18-22). The key idea is to ensure that a high GVD phase is checked on all processors before being rejected. In lines 25-27, the head of the final feasible schedule (σ_p) for each processor p is taken and dispatched to the individual processor for scheduling. In the absence of dependencies, all the phases are treated as ZID phases. Thus, GVD for each phase is equivalent to the phase's LVD.

G-GUA is greedier than NG-GUA for accrued utility during overloads. It does not default to any deadline-based scheduling algorithm and attempts to maximize accrued utility both during underloads and overloads. A sample schedule for G-GUA is presented in [16].

3.5 Algorithm Properties

The properties of NG-GUA and G-GUA are summarized in this section. For brevity, the proofs have been omitted and provided in [16].

Theorem 1 *NG-GUA, without dependencies, defaults to G-EDF during underloads.*

Algorithm 3: G-GUA: Greedy Global Utility Accrual

```
1: Input: σ_T          // List of released phases
2: Vars: σ_1 ··· σ_m // Per processor ready queues for m processors
3: Vars: σ_z          // Zero in-degree phase list
4: Vars: cpu_mask
5: Vars: not_fes
6: ComputeDAGwithDRD(σ_T);
7: σ_z ← FindZIDPhases(σ_T);
8: ComputeGVD(σ_z);
9: σ_d ← SortByGVD(σ_z);
10: for each phase J in σ_d do
11:    cpu_mask = 0; not_fes = true;
12:    while not_fes == true do
13:       p ← FindProcessor();
14:       if p == φ then
15:          break;
16:       InsDeadLnPos(J, σ_p);
17:       UpdateCpuEC(p, J, true);
18:       if IsFeasible(σ_p) == false then
19:          RemoveList(J, σ_p);
20:          UpdateCpuEC(p, J, false);
21:          AddCpuToMask(p, cpu_mask);
22:          not_fes = true;
23:       else
24:          not_fes = false;
25: for each p processor's schedule σ_p in m do
26:    Job_p ← HeadOf(σ_p) ;
27: return { Job_1, ··· , Job_m };
```

Theorem 2 *NG-GUA, with dependencies, defaults to G-EDF with PIP during underloads.*

Theorem 3 *The gMUA scheduling algorithm is a special case of NG-GUA.*

Theorem 4 *Both NG-GUA and G-GUA ensure mutual exclusion.*

Theorem 5 *For both algorithms considered, an application always makes progress, i.e., executes application-specific code, if there is work offered and the application is not deadlocked.*

Property 1 *In [17], Theorem 16.3.1 shows that when the schedule length is used as a criteria, a greedy algorithm that schedules the ZID nodes in a DAG produces a schedule that is within a factor of two from being optimal. Further, for a multi-threaded application with P threads, work T_1 and critical path length T_∞, the length of the schedule is bounded by $\frac{T_1}{P_A} + \frac{T_\infty(P-1)}{P_A}$, where P_A is defined as the average number of threads executed at each scheduling interval.*

Theorem 6 *Property 1 applies for both NG-GUA and G-GUA.*

Theorem 7 *For m processors and n phases, the asymptotic cost for both NG-GUA and G-GUA is $O(mn \log n)$.*

4 Experimental Evaluation

4.1 ChronOS Real-time Linux

In order to implement and evaluate the performance of NG-GUA and G-GUA with other state-of-the-art scheduling algorithms, we created a real-time Linux kernel, called ChronOS [1], based on the `CONFIG_PREEMPT_RT` patch [27]. The key motivation was to take advantage of the `CONFIG_PREEMPT_RT` real-time patch which enables complete preemption in Linux and improves interrupt latencies. ChronOS provides a set of APIs and a scheduler plugin infrastructure using which various single-processor and multiprocessor (including utility accrual and non-utility accrual) scheduling algorithms can be implemented. ChronOS is the first academic real-time Linux kernel based on the `CONFIG_PREEMPT_RT` patch. The architectural details of ChronOS are discussed in [1, 16].

4.2 Experimental Setup

Both NG-GUA and G-GUA do not assume any specific task arrival model (e.g., periodic, aperiodic, sporadic). Tasks can arrive at any time in the system and generate scheduling events. However, in order to evaluate NG-GUA/G-GUA against other state-of-the-art algorithms, we use a periodic model which helps quantify the schedulability criteria of the algorithms and allows us to compare the performance with other scheduling algorithms. We create a synthetic real-time test application in ChronOS which enables evaluation using a wide range of workloads. Tasks are represented as threads and the application periodically fires threads with specified time-constraints. For each task, we use a `burn_cpu(exec_cost)` method, which takes the execution cost of the task as an input and burns processor cycles for that amount of time.

We conduct the experiments on a quad-core platform based on AMD Phenom 9650 processor with 2.3 GHz frequency and 2 MB L3 cache, and measure the Deadline Satisfaction Ratio (DSR) and the Accrued Utility Ratio (AUR). At a given utilization load U, the DSR is measured as the ratio of the tasks that met their deadlines to the total number of tasks released in the system. In a similar fashion, the AUR is measured as the total accrued utility of the tasks that met their deadlines to the total possible accrued utility in the system.

We consider two types of task-sets in this paper—12 tasks (12T) with periods in the range $[300ms - 20000ms]$ and 27 tasks (27T) with periods in the range $[50ms - 7500ms]$. The utilization load per task are in the range $[0.01 - 0.5]$. We use a downward "step" TUF and consider three models: (i) *Increasing Utility* (IU), utilities assigned to the tasks are proportional to their deadlines. The task with the earliest deadline has the least utility and vice-versa; (ii) *Decreasing Utility* (DU), utilities assigned to a task are inversely proportional to their deadlines. The task with the earliest deadline has the highest utility and vice-versa; and (iii) *Random Utility* (RU), tasks are assigned random utilities with no two tasks having the same utility. These models are used to ascertain whether, irrespective of the TUF ordering, our algorithms perform comparable to the competitors and

to ensure that we do not create a bias based on the TUF assignment against the deadline-based algorithms. The data points on all results are shown as an average of ten samples along with the standard deviation.

We have additional extensive results using a wide range of task-sets on two, four, and eight processor platforms and those have been excluded here due to space limitation. We present those results in [16].

4.3 Results Without Dependencies

Comparison with global scheduling algorithms. Fig 2(a) and Fig 2(b) show the AUR and DSR results, respectively, for 27T using RU on a 4-core platform. No locks have been used. We observe that both NG-GUA and G-GUA are able to accrue higher utility during overload conditions as compared to the deadline-based scheduling algorithms. As a consequence, the algorithms are able to satisfy more task deadlines during overloads when compared to the deadline-based scheduling algorithms. On a 4-core platform, G-EDF is able to meet all deadlines upto $\approx$ 380% CPU utilization load, after which it suffers from a domino effect. G-NP-EDF starts missing deadlines earlier than G-EDF. On the other hand, NG-GUA not only defaults to G-EDF during underloads, it is able to sustain higher DSR during overloads. As NG-GUA defaults to a deadline-based order, we observe that it is able to meet more deadlines than G-GUA in Fig 2(b). The performance improvement for AUR is manifold. We observe $\approx$ 900% improvement in AUR during overloads for both G-GUA and NG-GUA over the deadline-based algorithms.

Comparison with partitioned scheduling algorithms. Figs 2(c) and 2(d) show the AUR and DSR results, respectively, for 27T using RU on a 4-core platform with the task-set partitioned using Baruah's optimized first-fit (BF) heuristic [5]. The task-set was partitioned off-line and assigned to the individual processors using ChronOS APIs. We compare our algorithms against P-EDF and P-DASA. In Fig 2(d), we observe that P-EDF is able to meet all deadlines upto $\approx$ 390% CPU utilization, after which it suffers from a domino effect. P-DASA uses the single-processor utility accrual scheduling algorithm, DASA, on individual processors. DASA defaults to EDF during underloads and maximizes accrued utility during overloads. We observe a similar behavior in Fig 2(c). However, both NG-GUA and G-GUA perform better than P-EDF and P-DASA during overloads by yielding a "best-effort" utility accrual behavior, with an improvement of $\approx$ 50% in AUR over P-DASA.

4.4 Results With Dependencies

To compare the performance of G-GUA and NG-GUA in the presence of dependencies against global scheduling algorithms, we consider three models: (i) *varying utilization load*, keeping the number of locks and critical section length fixed; (ii) *varying critical section length*, keeping the utilization load and the number of locks fixed; and (iii) *varying number of locks*, keeping the utilization load and the critical section length fixed. We implement locks using `futexes`, which allow us

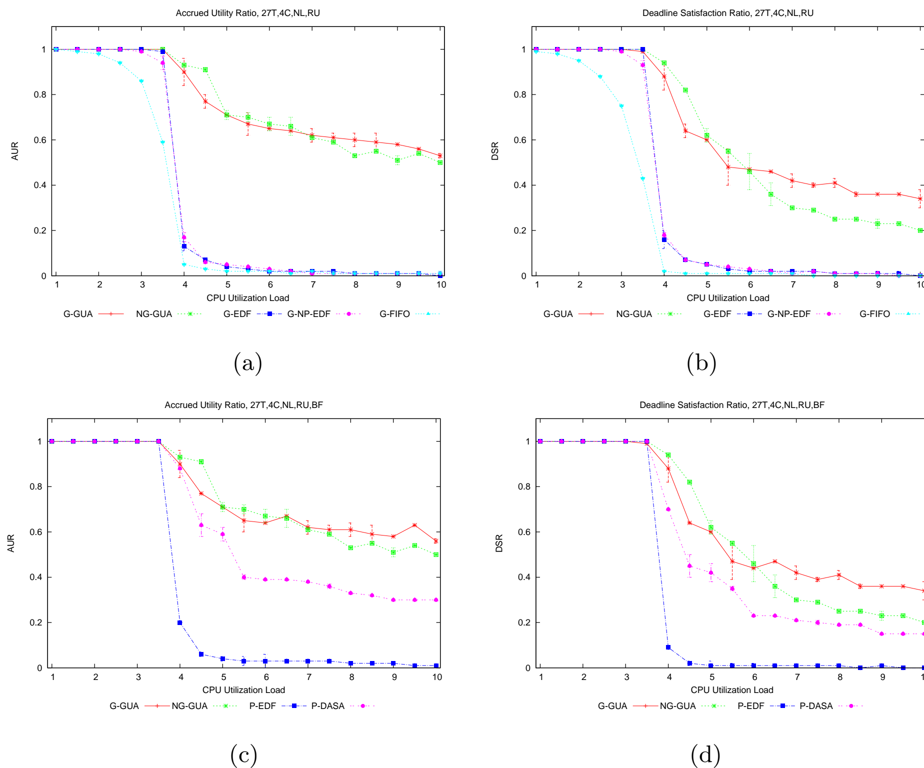


Fig. 2: Experimental results without dependencies against (a-b) global scheduling and (c-d) partitioned scheduling algorithms

to share a context between kernel-space and user-space. We consider the critical section length as a percentage of the total execution cost of the task. The tasks request the locks sequentially.

Fig 3(a) and Fig 3(b) show the AUR and DSR results for 12T with 4 locks per task, using RU on a 4-core platform. The tasks use the locks sequentially. The locks have a fixed critical section length of 5% of the total task execution cost. In the presence of locks, none of the scheduling algorithms are able to meet all deadlines during underloads. We observe that both G-GUA and NG-GUA provide a better accrued utility as well as deadline satisfaction during overloads. In Fig 3(a), G-GUA and NG-GUA provide a consistent 80% accrued utility benefit when compared with the deadline-based algorithms. Fig 3(c) provides the AUR results for a fixed utilization load of 800% and 4-locks by varying the critical section length. With an increase in the critical section length, the overall AUR decreases. G-GUA is able to provide an improvement of $\approx 5\%$ over NG-GUA. Fig 3(d) shows the AUR results for a fixed utilization load of 800% and critical section length of 5% while varying the number of locks. We observe that both NG-GUA and G-GUA consistently accrue higher utility when compared to the deadline-based algorithms.

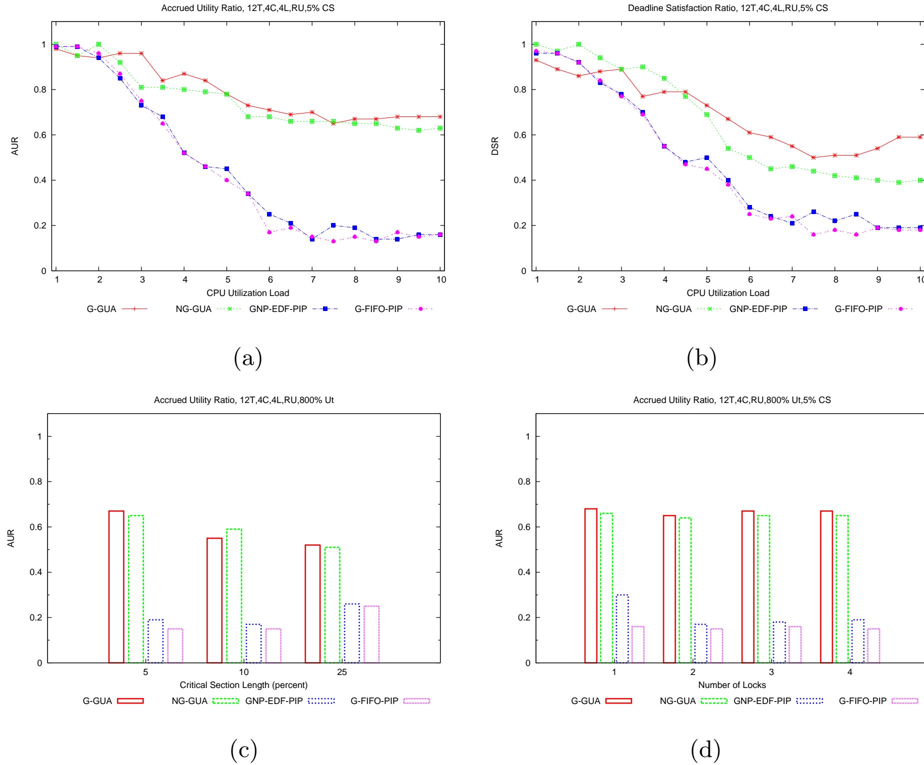


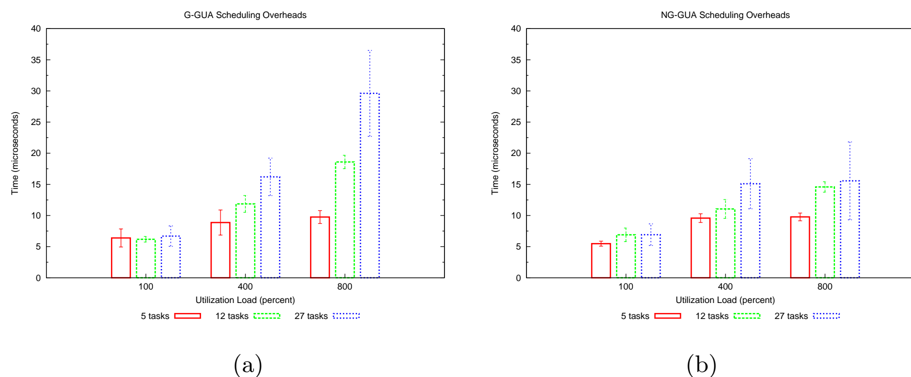
Fig. 3: Experimental results with dependencies by (a-b) varying utilization load (c) varying critical section length (d) varying number of locks

4.5 Scheduling Overheads

As mentioned earlier, both NG-GUA and G-GUA have a worst-case asymptotic cost of $O(mn \log n)$. Figs 4(a) and 4(b) show the scheduling overheads for G-GUA and NG-GUA on ChronOS. We observe that with an increase in the number of tasks and also the task utilization load, the scheduling overhead of both algorithms increase. In particular, G-GUA is seen to have a higher overhead compared to NG-GUA. This is primarily because G-GUA is more greedy for accruing overall utility as compared to NG-GUA. For a 27T task-set, we observe $\approx 30\mu$s overhead for G-GUA and $\approx 15\mu$s overhead for NG-GUA.

5 Conclusions

This paper focuses on the dynamic, multiprocessor real-time scheduling problem space—i.e., those characterized by execution overruns, unpredictable task arrivals, causing transient and permanent overloads. The paper demonstrates that it is possible to design scheduling algorithms for this problem space, such that they yield an optimal timeliness behavior (e.g., meeting all deadlines; obtaining maximum total utility), when total utilization demand does not exceed



(a) (b)

Fig. 4: Scheduling overheads under variable CPU utilization loads and various task-sets for (a) G-GUA (b) NG-GUA

the algorithms' utilization bound, and a best-effort timeliness behavior at all other times. This approach was pioneered in the Alpha OS kernel [19], which included two generations of TUF scheduling algorithms for scheduling single-processor systems [25, 12]. At its core, the paper's algorithms demonstrate that a similar approach can also be successfully extended for multiprocessors. Additionally, the paper's ChronOS real-time Linux kernel, provides a framework for implementing and plugging-in a broad range of multiprocessor real-time schedulers, while taking advantage of `CONFIG_PREEMPT_RT` patch's optimized interrupt service latencies and real-time locking primitives. Ongoing work is transitioning the GUA algorithms and the ChronOS kernel to a US Department of Defense system.

There are several directions for future work. Immediate directions include improving the algorithms' utilization bound from $\approx m/2$ and reducing their time overheads. Other directions include developing scalable and approximate algorithms for GUA's problem space with lower bounds on accrued utility and satisfied deadlines.

References

1. ChronOS Real-time Linux, `http://www.chronoslinux.org`
2. Allen, R., Garlan, D.: A Case Study in Architectural Modelling: The AEGIS System. In: IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design. p. 6. IEEE Computer Society, Washington, DC, USA (1996)
3. Anderson, J.H., Bud, V., Devi, U.C.: An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In: ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems. pp. 199–208. IEEE Computer Society, Washington, DC, USA (2005)
4. Andersson, B., Baruah, S., Jonsson, J.: Static-priority Scheduling on Multiprocessors. Tech. rep., Chapel Hill, NC, USA (2001)
5. Baruah, S., Fisher, N.: The Partitioned Multiprocessor Scheduling of Deadline-Constrained Sporadic Task Systems. IEEE Trans. Comput. 55(7), 918–923 (2006)
6. Baruah, S.K.: The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors. Real-Time Syst. 32(1-2), 9–20 (2006)

7. Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate Progress: A Notion of Fairness in Resource Allocation. Algorithmica 15(6), 600–625 (1996)
8. Bertogna, M., Cirinei, M., Lipari, G.: Improved Schedulability Analysis of EDF on Multiprocessor Platforms. In: ECRTS '05. pp. 209–218 (2005)
9. Chen, S.Y., Hsueh, C.W.: Optimal Dynamic-Priority Real-Time Scheduling Algorithms for Uniform Multiprocessors. In: RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium. pp. 147–156. IEEE Computer Society, Washington, DC, USA (2008)
10. Cho, H.: Utility Accrual Real-Time Scheduling and Synchronization on Single and Multiprocessors: Models, Algorithms, and Tradeoffs. Ph.D. thesis, Virginia Tech (2006)
11. Cho, H., Ravindran, B., Jensen, E.D.: An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In: RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium. pp. 101–110. IEEE Computer Society, Washington, DC, USA (2006)
12. Clark, R.K.: Scheduling Dependent Real-Time Activities. Ph.D. thesis, CMU (1990), CMU-CS-90-155
13. Clark, R., Jensen, E.D., Kanevsky, A., Maurer, J., Wallace, P., Wheeler, T., Zhang, Y., Wells, D., Lawrence, T., Hurley, P.: An Adaptive, Distributed Airborne Tracking System ("Process the Right Tracks at the Right Time"). In: In IEEE WPDRTS, volume 1586 of LNCS. pp. 353–362. Springer-Verlag (1999)
14. Funaoka, K., Kato, S., Yamasaki, N.: Work-Conserving Optimal Real-Time Scheduling on Multiprocessors. In: Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on. pp. 13–22 (2-4 2008)
15. Funk, S., Nanadur, V.: LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets. In: 17th International Conference on Real-Time and Network Systems. pp. 159–168 (2009)
16. Garyali, P.: On Best-Effort Utility Accrual Real-Time Scheduling on Multiprocessors. Master's thesis, Virginia Tech (2010), http://www.real-time.ece.vt.edu/Garyali_P_T_2010.pdf
17. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
18. Jensen, E., Locke, C., Tokuda, H.: A Time Driven Scheduling Model for Real-Time Operating Systems (1985), IEEE RTSS, pages 112–122, 1985.
19. Jensen, E., Northcutt, J.: Alpha: A Non-proprietary OS for Large, Complex, Distributed Real-Time Systems. In: Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on. pp. 35–41 (11-12 1990)
20. Koren, G., Shasha, D.: D-OVER; An Optimal On-line Scheduling Algorithm for Overloaded Real-Time Systems. In: Real-Time Systems Symposium, 1992. pp. 290–299 (2-4 1992)
21. Koren, G., Shasha, D.: MOCA: A Multiprocessor On-line Competitive Algorithm for Real-Time System Scheduling. Theor. Comput. Sci. 128(1-2), 75–97 (1994)
22. Lakshmanan, K., Rajkumar, R., Lehoczky, J.: Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In: Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on. pp. 239–248 (1-3 2009)
23. Li, P.: Utility Accrual Real-Time Scheduling: Models and Algorithms. Ph.D. thesis, Virginia Tech (July 2004)
24. Li, P., Wu, H., Ravindran, B., Jensen, E.D.: A Utility Accrual Scheduling Algorithm for Real-Time Activities with Mutual Exclusion Resource Constraints. IEEE Trans. Comput. 55(4), 454–469 (2006)
25. Locke, C.D.: Best-Effort Decision Making for Real-Time Scheduling. Ph.D. thesis, CMU (1986), cMU-CS-86-134
26. López, J.M., Díaz, J.L., García, D.F.: Utilization Bounds for EDF Scheduling on Real-Time Multiprocessor Systems. Real-Time Syst. 28(1), 39–68 (2004)
27. Molnar, I.: CONFIG PREEMPT REALTIME, "Fully Preemptible Kernel", vp-2.6.9-rc4-mm1-t4, http://lwn.net/Articles/105948/
28. OMG: Real-time CORBA 2.0: Dynamic Scheduling Specification. Tech. rep., Object Management Group (September 2001)
29. Patterson, D.: The Trouble With Multicore. IEEE Spectrum 47(7), 28–32 (July 2010)