Proc. 2nd IEEE Int'l Conf. on Engineering of Complex Computer Systems (jointly with CSESAW '96, RTAW '96, & SES '96), Montreal, Canada, Oct. 1996, pp.60-67.

An Experimental Investigation of the Potential of BLF-driven Scheduling of Real-time Threads

K. H. (Kane) Kim & Yuseok Kim
University of California, Irvine
kane@ece.uci.edu

Thomas Lawrence
USAF Rome Laboratory

Cuong Nguyen & Richard Scalzo
USN Naval Surface Warfare Center

Abstract

Resource allocation in complex real-time computer systems (RTCS's) cannot be adequately handled by straightforward extensions of the CPU and peripheral device scheduling techniques used in non-RTCS's. Rather, all the resources including CPU's, peripheral devices, communication channels, etc. should be handled in an integrated manner. As an idealistic framework, we laid out a globally optimal resource allocation methodology based on the benefit loss function (BLF), which includes both static resource allocation and design of dynamic resource allocation algorithms. As a small experimental part of this research, we also designed and implemented a BLF-driven thread scheduler (CPU scheduler) on Solaris 2.3. This experiment showed that existing processor scheduling approaches were not competitive against a quickly developed BLF-driven scheduling approach with respect to minimizing the total benefit loss. Therefore, the BLF-driven resource allocation approach opens doors for many new worthwhile research efforts.

Index Terms: resource allocation, real-time systems, complex systems, globally optimal, benefit loss, scheduling.

1. Introduction

In spite of the continuing decline of computer hardware in its influence in determining the overall costs of computer-embedded application systems, allocation of computing resources is still a major issue in challenging complex RTCS's such as those for defense command and control [1, 2, 5]. In such systems, the failure rates of computing components are non-negligible and thus tight resource conditions are expected to arise. Moreover, real-time recovery of the computation disturbed by component malfunctioning involves resource allocation actions.

Complex RTCS's are invariably of the distributed system type and the percentage of them containing parallel computers as their components has been steadily growing. Therefore, in general, the resource allocation should be done in a hierarchical manner, starting from the WAN level and continuing through the LAN level and the node level down to the processor level.

The techniques for static (design-time) and dynamic (run-time) resource allocation of various computation segments to execution machine components in such systems have been advancing rather slowly. Two fundamental limitations in established approaches are as follows:

(1) Most are based on the use of simple characterizations of computation-segments competing for use of the execution resource. To improve, information on the global system-wide resource needs must be more vigorously exploited. System engineers must endeavor to understand the impacts of inaccurate outputs on the application success. The damaging impacts may be called *benefit losses.* Whenever resources are tight and competitions arise, allocation decisions must be made in the direction toward minimizing the benefit losses.

(2) Fault tolerance requirements must be an integral part of the initial system requirements. Selection of appropriate system designs to meet the fault tolerance requirements involves both static allocation of some execution resources and design of some dynamic allocation algorithms. Advances in this area will have much bigger impact on the cost-effectiveness of RTCS's than further improvement of conventional low-level task schedulers operating with little information on system-wide resource requirement patterns.

Scheduling each local resource or managing each service infrastructure (e.g., communication, shared data storage, etc.) to handle a maximal number of requests from clients of different importance, has been studied for long but each resource domain has been studied independent of others. A group of resource managers of which the service policies are not well coordinated cannot create a *globally optimal resource allocation*, i.e., a situation where available resources are used the most

0-8186-7614-0/96 $5.00 © 1996 IEEE

efficiently to meet the objectives of the applications. Therefore, the most challenging issue in globally optimal resource allocation is to *determine the urgency of each service request to each shared resource in an accurate quantitative form*.

In addition, in many safety-critical applications, it is too dangerous to accept a system design based solely on average performance measures. The worst-case (i.e., the peak-load case worth considering) performance measures are often more important.

Since the resource allocation problem is equivalent to the problem of making some of static allocations (which are design activities) plus designing some dynamic allocation algorithms, it is essentially a design problem. Therefore, realizing an optimal resource allocation means an optimal design of a system.

Earlier in [4], we laid out an idealistic high-level methodolody framework for globally optimal resource allocation in complex RTCS's, which includes both static resource allocation and design of dynamic resource allocation algorithms. The notion of timed value accuracy of each output action was formalized there. Thereafter, an approach for using the quantitative specification, supplied by the system engineer, of the relationship between the loss in the timed value accuracy of each output action (due to faults, resource shortage, etc.) and the damage to the application mission as a key driver in globally optimal resource allocation, was formulated. The quantitative specification of the relationship between the accuracy loss of an output action and the damage is called the benefit loss function, or BLF for short, of that output. (The time-value function proposed in [3] is one special case of the benefit loss function.)

The purpose of this paper is to address two issues both of which are relevant to converting the abstract methodology framework into concrete resource allocation schemes:

(1) The first issue is how the designer might generate quantitative specifications of urgencies of various resource requests, starting from the initial BLF specifications. Some general directions are discussed in this paper but practical implementation techniques are a major subject for future research.

(2) The second issue is whether the urgency specifications can be honored more effectively by existing local scheduling policies or by new local scheduling policies. As a small experimental effort aimed for addressing this question, we designed and implemented a BLF-driven thread scheduler (CPU scheduler) on Solaris 2.3. The BLF-driven thread scheduler schedules real-time threads towards the objective of minimizing the total benefit loss where the threads dynamically present their worst-case computation time estimates, deadlines, and BLF's associated with their next computation-segments to be executed. This experiment showed that existing processor scheduling approaches such as fixed priority or deadline-driven scheduling approaches were not competitive against a quickly developed BLF-driven scheduling approach with respect to minimizing the total benefit loss. Therefore, there is room for future research on local scheduling policies with BLF-driven urgency specifications.

Section 2 provides a discussion on the issues in deriving the specifications of urgencies of resource requests. The experimental implementation of the BLF-driven thread scheduler on Solaris 2.3 is described in Section 3. Section 4 then discusses the measurement results. Section 5 is a conclusion.

2. System-level BLF's and object-level BLF's

2.1 Real-time functional requirements and accuracy of outputs

Complex real-time computer systems (RTCS's) consist of many multi-purpose modules which share the workload for providing a variety of time-critical service functions to achieve application objectives. The typical service actions of a RTCS are combinations of

(a) outputting control values to devices in the application environments and

(b) storing newly computed values into a database which is shared by users outside the control domain of the RTCS.

Ideally, an RTCS must take every service action "accurately" not only in "time dimension" but also in "logical value dimension". The value of an output of an RTCS is a two-dimensional value called a "timed logical value". The accuracy of an output of an RTCS is the closeness of both time and logical value attributes of the output to the time and logical value attributes of the desired output. Similarly, the temporal accuracy of any data transmission or storing action in an RTCS is the closeness of the time attribute of the action to the time attribute of the desired action, while the logical value accuracy of the action is the closeness of the logical value attribute of the data involved in the action to the logical value attribute of the desired action.

Examples of the manifestations of inaccurate output observed often in practice are:

(1) Omission of a desired output, i.e., output with zero degree of temporal accuracy;

(2) Too late output, i.e., output with an unacceptably low degree of temporal accuracy;

(3) Unacceptable logical value, i.e., output with an unacceptably low degree of logical value accuracy.

2.2 Major causes for loss of output accuracy and resultant benefit loss

A big challenge in the field of optimal design of complex RTCS's is the resource allocation to maintain output accuracy in the presence of component failures. Figure 1 depicts three major causes for run-time resource allocation actions: (1) fault detection, (2) repair and reincorporation of previously unusable components, and (3) entering of the applications into a new phase where resource requirements are significantly different from the preceding phase.

Given a set of service functions defined as the mission of an RTCS under development, the basic required resource set can be defined as a minimum set of resources which collectively provide enough processing capacity to support the mission when the possibility of any component failure during the application life-time can be ruled out. Therefore, mobilizing any additional resources beyond the basic required resource set will return no benefits as long as no components in the basic set fail. However, component failure probabilities are non-negligible in practice and thus some extra resources not included in the basic required resource set must be incorporated into the RTCS.

When a sequence of component failures causes the system resource condition to fall below the level to form a basic required resource set, a very difficult resource allocation problem is posed. For example, a choice may have to be made between sacrificing a certain service function completely, thereby effectively cutting off the output channel for that function, and degrading the output accuracy of multiple service functions to some extent. In principle, such a trade-off must be made on the basis of a rigorous evaluation of which option will cause less damage to the application mission.

2.3 Determination of BLF's and object-level BLF's

Each time an output action occurs with degraded accuracy, some damage may occur to the application mission. The damage is called the *benefit loss*. When the accuracy loss Δ_v in an output action v occurs, the benefit loss $BL(\Delta_v)$ results. The benefit loss function (BLF) is thus the mapping of every accuracy loss event to a consequent benefit loss value. The determination of $BL(\Delta_v)$ for every output action v in an RTCS is the responsibility of the system engineer. When the benefit losses accumulated over a certain period of time exceed a certain predetermined threshold, we can treat the RTCS as having failed in its application mission. The threshold can thus be called a system failure threshold and it should be determined by the system engineer.

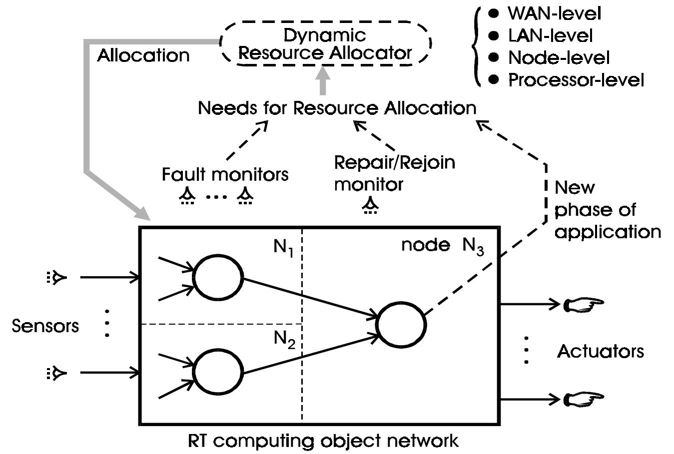
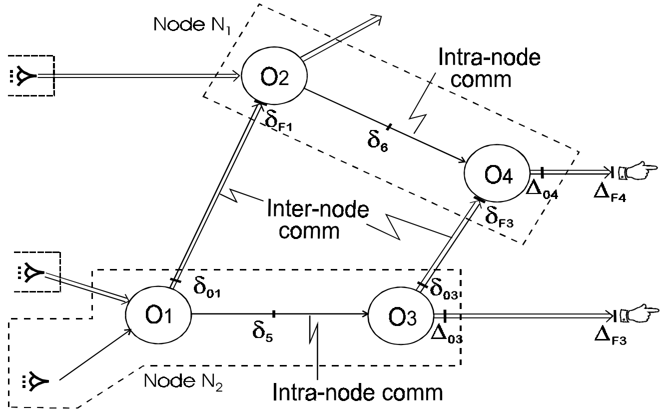


Figure 1. Needs for run-time resource allocation

Given the specification of *expected* component failures (or various fault sources) as probabilistic variables, the *expected benefit loss* caused by the fault sources is also a probabilistic variable. If the system engineer provides the specification of the benefit loss function in addition to the expected component failure specification, then the expected benefit loss can be calculated. Or if only frequency bounds on component failures are provided without information on probability distributions, then just the *maximum expected benefit loss* can be calculated. In principle, the choice between two or more options for degrading selected service functions due to the resource shortage can be made on the basis of expected benefit loss values under each expected configuration. In the rest of this paper, we consider only the accuracy loss in the time dimension, i.e., the cases of late or missing outputs.

We assume that RTCS's are structured as object networks (See Figure 2). Each subsystem can then be viewed as a computation object (C-object). Some outputs from C-objects are system outputs going to devices in application environments or to the database outside the control domain of the RTCS. Other outputs from C-objects go to other C-objects. Accuracy loss at a C-object may thus contribute to loss in accuracy of system outputs.

Initially, an RTCS can be represented as a single C-object O which interacts with the external environment by accepting inputs from the sensors and producing outputs to the actuators. The important prerequisite for the BLF-based resource allocation is to require the system engineer to determine the $BLF(\Delta_F)$ for each action of a system output reaching an actuator in the environment, where Δ_F is the accuracy loss occurred in a system output reaching the actuator. In the system where the C-object and the actuator are located in the same node (i.e., the communication delay between the C-object and the actuators is negligible), we can assume $\Delta_F = \Delta_O$ where Δ_O is the accuracy loss occurred in a system output produced.



Δ_F : BLF for a DCS output arrival
Δ_0 : derived BLF for a DCS output generation
δ_F : derived BLF for an intermediate result arrival
δ_0 : derived BLF for an intermediate result generation
δ : $\approx \delta_F \approx \delta_0$ where the result communication delay is negligible

* In general, $x := f(\{x^{t1}, x^{t2}, x^{t3}, \ldots\})$ where $x = \Delta_F, \Delta_0, \delta_F, \delta_0$, or δ and x^{ti} represents the BLF for a particular action of type x needed at time ti

Figure 2. System-level BLF and object-level BLF

The design process decomposes the RTCS initially represented as a single C-object into a set of several C-objects, say **O** = $\{O_1, O_2, \ldots, O_n\}$, which interact with one another as illustrated in Figure 2. Each C-object produces outputs going to other C-objects or the environment. Outputs going to other C-objects are called intermediate outputs. As a part of the design process, the benefit-loss potential function BLPF (δ_F), where δ_f is the accuracy loss occurred in the intermediate output, must be determined for each intermediate output reaching another C-object. This involves the determination of an appropriate deadline for the intermediate output reaching the receiver C-object as well as estimated probabilities of a late execution of the intermediate output leading to deadline violations of system outputs. In principle, the BLPF of each intermediate output function w is

$$\sum_{u_k \in U} \{\text{benefit loss that can occur through a deadline miss by the system output } u_k \text{ due to the lateness or missing of } w\},$$

where **U** is the set of all system output functions. Therefore, unless the BLF for each system output function u_k is a simple function (e.g., single step function), it does not seem practical to derive an effective BLPF from a general-form BLF. Nevertheless, this is a meaningful topic for future experimental research.

The decomposition process will continue until the C-objects become the basic units for resource allocation. For a CPU scheduling, the basic unit is typically a process. After the RTCS is decomposed into a network of C-objects which are basic competing units for resources, each case of a C-object presenting to a local resource allocator its estimates for worst-case execution times before its output actions and the BLPF's associated with its output actions, is analyzed to ensure that the local resource scheduler is never over-taxed under fault-free circumstances.

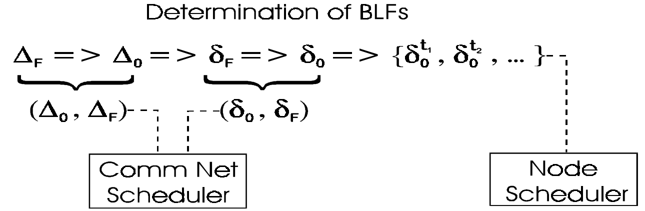


Figure 3. Derivation sequence for BLF's

Figure 3 summarizes the typical sequence of producing the scheduling parameters which were discussed above. The system engineer initially determines Δ_F, the BLF for each system output reaching the environment. From this and the available knowledge on the channel connecting the RTCS to the environment, the designer derives Δ_O, the BLPF for each system output being produced. As the system is decomposed, δ_F the BLPF for each (intermediate) output of a C-object reaching another C-object is derived. From δ_F and the available knowledge on the channel connecting the producer C-object to other receiver C-objects, the BLPF δ_O is derived. δ_O can be further refined into $\delta_{O,t1}$ $\delta_{O,t2}$, …, if the deadline and the BLP value differ among executions of the same output function at different times. Whenever Δ_F and Δ_O (similarly, δ_F and δ_O) are different, then they are used by the communication network scheduler in scheduling the transmission of each output value generated.

3. An experimental implementation of a BLF-driven thread scheduler

As a small experimental part of this research, we designed and implemented a BLF-driven thread scheduler (CPU scheduler) under Solaris 2.3 operating system. The BLF-driven thread scheduler schedules real-time threads with the objective of minimizing the total benefit loss where the threads dynamically present their worst-case computation time estimates, deadlines, and BLPF's associated with their next computation-segments to be executed.

The thread library in Solaris 2.3 provides a set of system calls to create and destroy user-level thread, voluntarily yield the CPU to another user-level thread, resume the execution of a specific user-level thread, etc. Since the UNIX environment (including the Solaris environment) is not a single-user environment, disturbances caused not only by other user processes but also by the kernel threads under a typical configuration are not negligible. As a result, it is hard to expect any credible measures in this situation. Therefore, in order to remove the system's interventions completely, user-level

threads in our implementation are created as *real-time threads*. While real-time threads are under execution, all the kernel threads executing system functions as well as other user processes are completely prevented from executing.

3.1 Experimental Configuration

Figure 4 shows the overall architecture of our BLF-driven thread scheduler implementation under Solaris 2.3. Multiple user-level threads are created as real-time threads share the same address space. In other words, all these threads have access to the global data structures declared in our program. One of the multiple threads is designated as a *scheduler thread* (ST), and the others are designated as *worker threads* (WT's) which behave as if they are application threads scheduled by the scheduler thread. The WT's dynamically present their scheduling parameters (e.g., worst-case computation time estimates, deadlines, BLF's, etc.) to the ST through a global data structure called the *scheduling parameter table* (SPT) at initialization time and at the end of each cycle, and then, suspend themselves. The ST periodically checks the SPT, determines which WT to execute next, and informs the Solaris 2.3 Kernel Thread Scheduler of the selected WT's identification via a thread library system call. The Solaris 2.3 Kernel Thread Scheduler in turn dispatches the WT requested by the ST. The ST thus emulates the BLF-driven thread scheduler to replace the Solaris Kernel Thread Scheduler.

3.2 Thread Model

(a) Worker threads (WT)

A WT behaves as if it is an application thread that is scheduled by the ST. The scheduling status of a WT is one of {RUNNING, READY, RECOVERING}. This status is recognized by the ST, not by the Solaris Kernel Thread Scheduler. After multiple WT's are created at the beginning, each WT, in turn, stores its computation time estimate, deadline, and BLF (to calculate benefit loss upon deadline miss) into the SPT, sets its status to

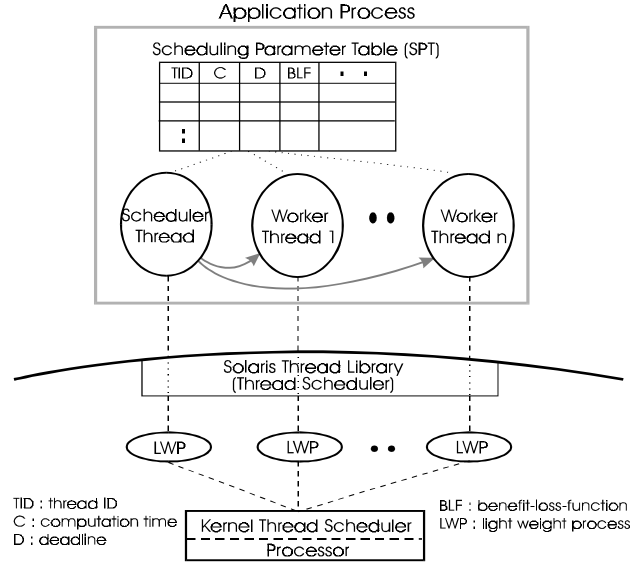


Figure 4. Experimental setup

READY, and suspends itself. When resumed (by the ST), it loops for a certain number of iterations. The number of iterations is determined such that the duration of the loop approximates the computation time estimate of the WT. After the WT finishes the iterations, it generates new computation time estimate, new deadline, and new BLF for the next computation-segment to be executed, and stores them into the SPT, sets its status to READY, and then suspends itself.

In this implementation every WT is assumed to be independent of each other. We modeled four types of WT's which are distinguished based on their preemptibilities and the type of BLF's associated with them:

(Type I) The WT is non-preemptible and the BLF f_i^{BL} associated with each deadline d_i imposed by the WT T_i has the following simple cliff form:

$= bl_i^{max}$ if T_i did not finish before its deadline d_i,
$= 0$ if T_i finished by its deadline d_i.

(Type II) The WT is preemptible and the BLF is same type as in Type I.

(Type III) The WT is non-preemptible and the BLF f_i^{BL} associated with each deadline d_i imposed by the WT T_i has the following general form:

$= f_i^{BL}(\delta_i)$ if $\delta_i \geq 0,$
$= 0$ if $\delta_i = 0,$

where t_i is the completion time of T_i, δ_i is the temporal accuracy loss for T_i, and is

$= t_i - d_i$ if T_i fails to finish before deadline d_i,
$= 0$ otherwise.

(Type IV) The WT is preemptible and the BLF is the same type as in Type III.

(b) Scheduler thread (ST)

In general, the thread scheduler action consists of two parts: *selection* and *dispatching*. Our ST can be viewed as

a selector and the Solaris kernel thread plays the role of a dispatcher.

When the ST is activated, it checks if there is any WT in READY status whose deadline is going to be missed. This is determined based on the WT's *laxity value* (= absolute deadline of WT - current time - computation time estimate of WT). The negative laxity means that if the computation time estimate presented by the WT becomes a reality, the WT cannot finish its job before its deadline. In such a case, the ST computes an *actual execution time* for that WT by multiplying a random value d ($0.7 \leq d < 1$) and checks again if the deadline is going to be "really" missed. If so, the ST sets the status of the WT to RECOVERING. In the current implementation, the reincarnating time ($\supset$ recovery time) for the WT which has missed a deadline and is in RECOVERING status, is set to the absolute deadline of the WT, which will be missed by the WT, plus Δ. That is, Δ time units after the WT's deadline has arrived, the WT is reincarnated.

Next, the ST checks if there is either a WT in RECOVERING status which is to be reincarnated at the present time or any WT which has RECOVERING status for which the reincarnation time has already passed. If such WT is found, the ST changes the status of such a WT to READY, and sets a new deadline for the WT, which is determined based on the reincarnation time as if the WT is newly submitted (or created) at the reincarnation time.

Finally, the ST gets the ID of the ready WT which is selected under the current scheduling policy, sets the status of the new WT to RUNNING, and then resumes the execution of the new WT by calling the Solaris 2.3 Kernel Thread Scheduler to dispatch the selected WT.

In the case of non-preemptive scheduling of the WT's (Types I and III), the thread-time-quantum is set to infinite when the program begins execution. Thus, a WT is not preempted once it starts executing until it finishes and voluntarily suspends itself. When a WT finishes, the WT notifies the Solaris by calling thr_suspend() function. Then, the Solaris kernel thread scheduler executes the ST next since the ST is the only "ready" thread visible to the kernel thread scheduler at that time. The ST will in turn select a new WT, inform the kernel thread scheduler of the ID of the new WT, and then, voluntarily yields the CPU (i.e., the ST remains as a "ready" thread visible to the kernel thread scheduler) to the selected WT.

However, in the case of preemptive scheduling of the WT's (Types II and IV), the thread-time-quantum is set to a certain value q ($10\ msec \leq q \leq 100\ msec$). When the ST is activated, it first reads the current system clock value and calculates the elapsed time since the last activation of the currently running WT. According to the elapsed time, the ST adjusts the remaining computation time of the currently executing WT by subtracting the elapsed time from the remaining computation time of the currently executing WT. If the currently executing WT is selected again by the ST, the ST simply hands over the control back to the WT running. However, if a new WT is selected, the ST sets the status of the currently executing WT to READY and suspends the execution of that WT by calling thr_suspend() function. Thereafter, the ST sets the status of the new WT to RUNNING and resumes the execution of the new WT by calling the Solaris 2.3 Kernel Thread Scheduler to dispatch the selected WT.

3.3 Scheduling policies

A non-preemptive version and a preemptive version of the Least Laxity First (LLF) algorithm were implemented. Also, two non-preemptive BLF-driven scheduling algorithms and their preemptive variations were formulated and implemented. The effectiveness of these algorithms were compared in terms of the benefit losses incurred per millisecond.

(a) Non-preemptive/preemptive Least Laxity First (LLF)

In this algorithm, the WT which has the smallest laxity will get scheduled first. The laxity l_i of a WT T_i is defined as the time remaining to the deadline d_i minus the remaining execution time c_i of T_i, that is, $l_i = d_i - CT - c_i$, where CT is the current time. For the preemptive LLF scheduling, c_i is updated whenever T_i gets preempted. Let e_i be the elapsed execution time since T_i became the running process the most recently until the current time. Then, c_i is updated such that $c_i \leftarrow c_i - e_i$.

(b) Non-preemptive/preemptive Highest Benefit-Loss First (HBLF)

In the case of the WT's of Type I and II the HBLF scheduling algorithm gives the highest priority to the WT whose benefit loss to be incurred upon missing the deadline is the largest among those of all the WT's. For the WT's of Type III and IV, BLF is modeled as a simple linear function, i.e., $f_i^{BL}(\delta_i) = a_i\delta_i$, where δ_i is a *temporal accuracy loss* (i.e., amount of time by which the WT misses the deadline). Here, the WT whose a_i is the largest among those of all the WT's is given the highest priority. In the case of using general-form BLF's, the problem of selecting a WT with the largest potential benefit loss becomes an infeasible problem. Thus, the heuristics for selecting a WT under different types of BLF's is a meaningful topic for future research.

(c) Non-preemptive/preemptive Highest Benefit-Loss-to-Laxity-Ratio First (HBLLRF)

The HBLLRF is a combination of the LLF and HBLF. In this algorithm, the WT which has the largest benefit-loss-to-laxity-ratio will get scheduled first.

4. Result Analysis

Whenever it becomes clear that a WT is going to miss its deadline (ref. Section 3.2 (b)), its benefit loss value is calculated based on its BLF and temporal accuracy loss, and is added to the total benefit loss. If the total benefit loss exceeds some predetermined benefit loss limit, the ST displays the total benefit loss incurred and the benefit loss incurred per millisec (BL/msec), and then exits.

In this experimentation, the scheduling parameters were set as follows:

(1) 10 msec ≤ computation time estimate ≤ 200 msec
(2) 0.7 * computation time estimate ≤ actual computation time < computation time estimate
(3) deadline = computation time estimate * 4.
(4) The BLF of WT T_i of type I or II was defined as the following constant function:

$f_i^{BL}(\delta_i) = bl_i^{max}$, where $100 \le bl_i^{max} \le 1000$,

and for WT T_i of type III or IV, the BLF was defined as the following simple linear function:

$f_i^{BL}(\delta_i) = a_i\delta_i$, where $1 \le a_i \le 10$.

(5) benefit loss limit = 10000.
(6) For the preemptive scheduling, the time quanta of 10 msec and 100 msec were used.

We ran each scheduling algorithm 10 times for each type of the WT model, and measured the performance in terms of the BL/msec value. The smaller the BL/msec is, the better it performs. Figure 5 shows the result with the WT's of type I and II. In the case of non-preemptive scheduling, the HBLLRF always performed better than the LLF. However, the performance of the HBLF which reflected the potential benefit loss but not any other scheduling parameters such as deadline, computation time estimate, and laxity, was highly fluctuating, i.e., sometimes it performed better than the HBLLRF and sometimes it performed worse than the LLF. In the case of preemptive scheduling, the HBLLRF always performed better than the HBLF and the LLF. Figure 6 shows the result with the WT's of type III and IV. Here, the result is almost identical to that in Figure 5. In the case of the time quantum set to 100 msec, the HBLLRF performed always better than the LLF and the HBLF. However, in the case of the time quantum set to 10 msec, the LLF sometimes performed better than the HBLLRF even though the average performance of HBLLRF was better than that of the LLF. The following is a summary of the results:

(1) HBLLRF always outperformed the LLF with the time quantum set to 100 msec;
(2) Most of the time the HBLLRF outperformed the LLF with the time quantum set to 10 msec. However, the LLF performed better than the HBLLRF once in a while. This is because of the irregularities caused by the scheduling overhead with a small time quantum size;
(3) The average performance of the HBLLRF was better than that of the HBLF and the LLF with both sizes of time quanta, and
(4) The average performance of the LLF was better than that of the HBLF with both sizes of time quanta.

5. Conclusion

In this paper we addressed two issues: one is how the designer might generate quantitative specifications of urgencies of various resource requests, starting from the initial BLF specifications, and the other is whether the urgency specifications can be honored more effectively by existing local scheduling policies or by new local scheduling policies. As a small experiment, we designed and implemented a BLF-driven thread scheduler (CPU scheduler) on Solaris 2.3, which schedules real-time threads towards the objective of minimizing the total benefit loss. This experiment results suggest that the urgency specifications can be exploited better by new local scheduling policies.

Acknowledgment: The research work reported here was supported in part by US Navy, NSWC Dahlgren Division under Contract No. N60921-92-C-0204, in part by the California Transportation Department via the UCI Institute for Transportation Studies, in part by USAF Rome Lab via Martin Marrietta ATL, in part by Hitachi, Ltd, and in part by ETRI.

References

[1] Halang, W.A. and Stoyenko, A.D., eds., ‘Real time Computing’, NATO ASI series F, Vol. 127, Springer-Verlag, 1994 (Proceedings of the NATO Advanced Study Institute on Real-Time Computing held in Sint Maarten, Oct. 1992).

[2] Howell, S., Hoang, N., Nguyen, C., and Karangelen, N., “Critical Issues in the Design of Large-Scale Distributed Systems”, Proc. IEEE Workshop on Advances in Parallel and Distributed Systems, Oct. 1993, Princeton, pp. 28-33.

[3] Jensen, E.D., Locke, C.D., and Tokuda, H., “A Time-Value Driven Scheduling Model for Real-Time Operating Systems”, Proc. IEEE CS Symposium on Real-Time Systems, Nov. 1985.

[4] Kim, K.H. et al., “A Methodology Framework for Optimal Design of Real-Time Dependable Computer Systems”, Proc. CSESAW ’94 (1994 Complex Systems Engineering Synthesis and Assessment Technology Workshop), US Navy NSWC, Dahlgren Div., July 1994, pp.251-259.

[5] Son, Sang H. ed., '*Advances in Real-Time Systems*', Prentice Hall, 1994.

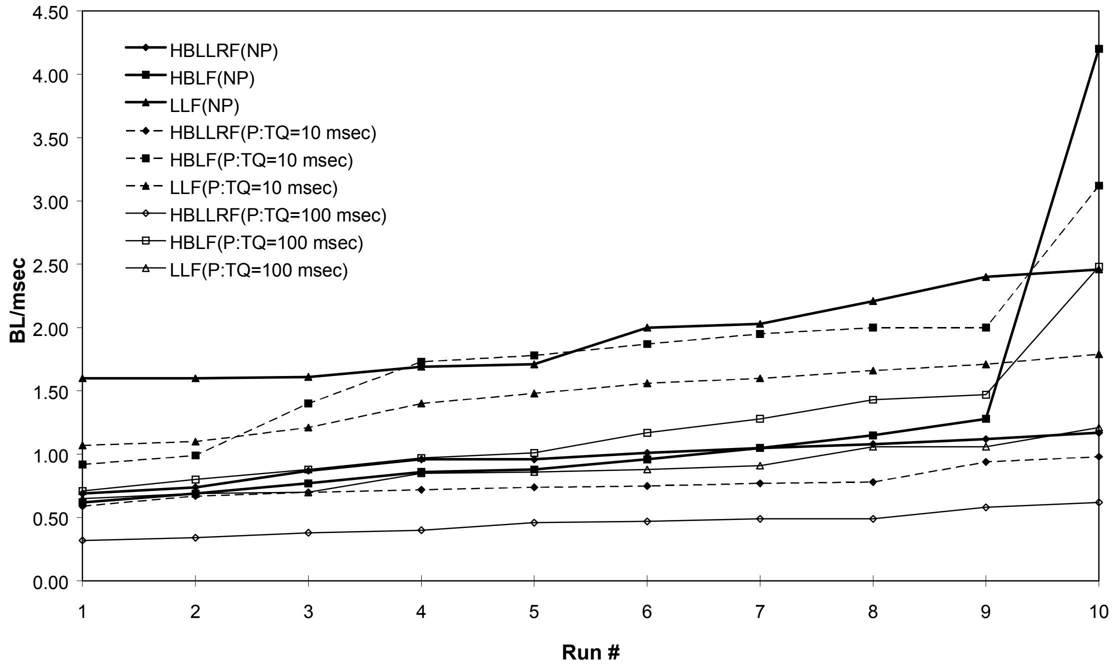


Figure 5. Comparison of three real-time scheduling strategies with constant BLF's

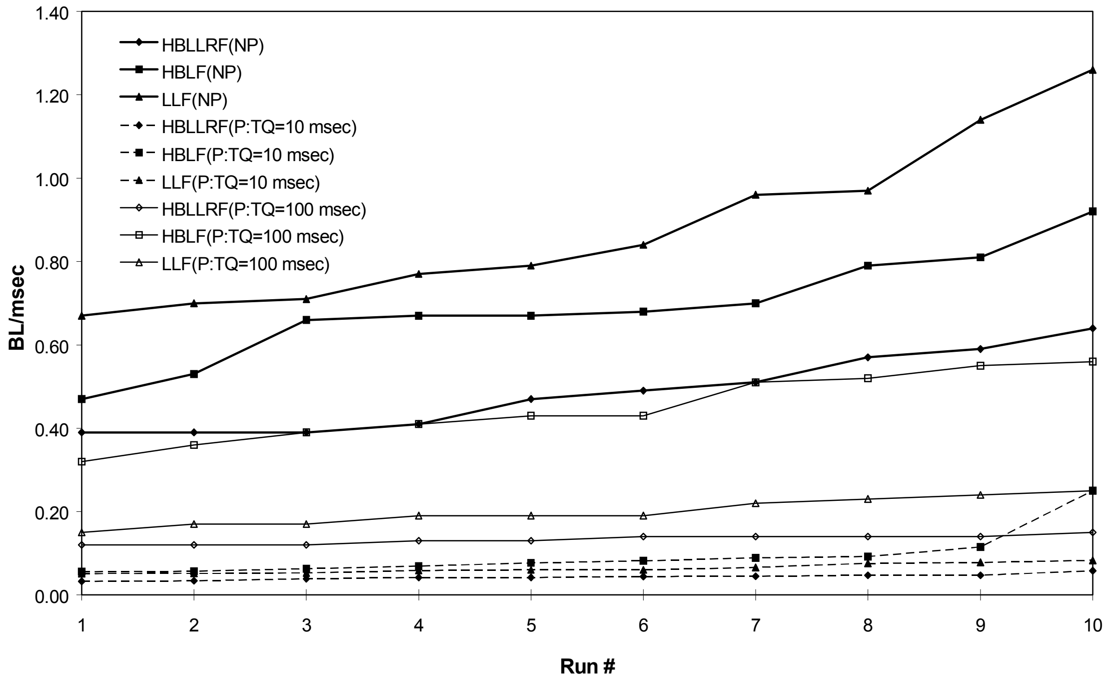


Figure 6. Comparison of three real-time scheduling strategies with simple linear BLF's