Adaptation and Graceful Degradation of Control System Performance by Task Reallocation and Period Adjustment

Kang G. Shin
RTCL, EECS Department
The University of Michigan
kgshin@eecs.umich.edu

Charles L. Meissner
Lockheed Martin Tactical Aircraft Systems
charles.l.meissner@lmco.com

Abstract

In this paper, we consider resource adaptation in multiprocessor real-time control systems running periodic control tasks, where the main resource is processor utilization, and the measure of utility is control performance. Processor utilization of a periodic task can be changed by changing the task's period. To estimate the reward rates for running a task at different periods we will use a well-understood concept from optimal control theory, the performance index. We present and evaluate two algorithms for reallocating resources, which are of low complexity in order that they can be run quickly for fast recovery from failure. We also consider transient scheduling effects of changing periods and moving tasks when reallocating resources.

By simulation, we show that the algorithms provide results that are close to optimal with randomly generated task sets. We demonstrate the calculation of reward rates for different frequencies of an example application task, and also demonstrate the calculation of the reward rate for a shadow task for the example application task.

Index Terms—real-time control, multiprocessor task allocation, graceful degradation

1. Introduction

There is a trend to desire adaptive resource allocation in various types of computer systems, in order to address both changing priorities for different applications, and also for failure recovery. This paper considers the problem of processor utilization allocation for real-time control systems, particularly computer systems which control complex systems, such as aircraft, factories, and submarines, and make use of multiprocessor or distributed computer systems, running multiple control tasks, which coordinate distributed intelligent sensors and actuators to accomplish a single objective. Since many real-time control tasks are periodic, the processor utilization they use can be changed by changing their periods without changing their algorithms, although variables in the control law may need to be changed to account for the changed period. Period adjustment presents an easily-implementable way to vary utilization committed to different tasks.

In this paper, we show an approach for processor utilization reallocation by period changes in a multiprocessor system, both for task priority changes such as would be brought about by a mode change, and for failure recovery. This approach considers the transient scheduling effects of changing periods and moving tasks when reallocating resources. We also consider the problem of determining the value to the system of running different control tasks at different periods, and demonstrate an approach for estimating the value of running a task at different periods, and an approach for estimating the value of having redundant shadows for a task.

There are several bodies of work related to reallocating processor resources to tasks in real-time control systems. One is concerned with scheduling strategies for tasks with variable performance levels and runtimes, referred to as "imprecise computation" [6]. With the imprecise computation model, the result becomes more precise with more computation time. An example of this is the "increasing reward with increasing service" (IRIS) task model [3], in which a computation can be stopped at any time, and answers become increasingly better with increasing time given to a task.

In [1] processor utilization allocation for flight control systems was discussed, and values for differing quality-of-service (QoS) levels for tasks were taken from an AI planner. Another QoS-based approach is discussed in [9], in which utilities are used to guide QoS-based resource allocation. An audio server is used as an application. Graceful degradation of periodic tasks by skipping task instantiations in some periods is discussed

The work reported in this paper was supported in part by the Office of Naval Research under Grants N00014-94-1-0229 and N00014-95-1-0121.

in [10]], in which a scheduling discipline is proposed which guarantees that a task will be run in at least *k* out of every *n* of its periods.

Varying periods of control tasks, off-line, for schedulability purposes is discussed in [12] and [11]. The performance index (PI), a general concept from optimal control theory which represents the cost to the system of the performance of a control law, is discussed as an off-line design aid for optimizing the trade-off between control quality and computation resources in [12] and [13].

This paper extends the approaches of [12], making on-line use of the proposed off-line method for processor utilization allocation. The approach proposed in this paper is to allow on-line control task period changes in multiprocessor systems to allow changes in control performance and processor utilization. The approach uses a performance index for the control task, weighted for the task's importance to the system, to determine the value to the system of running a given task at a given period. Due to the nature of the target applications, embedded control applications, which tend to have small memory and communication needs, processor utilization is considered the limiting resource, and so it is the only resource which is rationed.

The contributions of the proposed approach are that it uses on-line period changes to handle resource allocation for system adaptation and failure recovery in a multiprocessor real-time control system, and that it uses well-accepted performance measures to determine the value to the system of running a task at different periods. It also addresses issues which do not arise in other adaptable systems. These issues include determining a value for running redundant shadow tasks as opposed to fast recovery, the potential need for a fast reallocation of resources for graceful degradation, and the need for consideration of the transient effects of reallocation, such as tasks not getting sufficient time in a period because of other tasks migrating and changing periods.

The remainder of the paper is structured as follows. First, motivations and general concepts for task reallocation in control systems are discussed. Several assumptions about the underlying computer system are then made. Two algorithms are given for reallocation, one for independent tasks, and one for related tasks, and the algorithms are evaluated. The next issue that is addressed is that some tasks which are not migrating or changing periods may not be given their full allotted time in some periods during the reallocation process, due to other tasks migrating and changing periods. Then, an example control application is given, along with a determination of the rewards associated with running the task at different frequencies and the reward associated with having a redundant shadow task. The paper concludes with a summary and discussion of future directions.

2. Changing Control Performance by Extending Task Periods

In the proposed approach, tasks' periods are increased or decreased in order to optimize system performance by giving more processor utilization to more important tasks. The system is presented with reward rates for running different tasks at different frequencies, and attempts to maximize the sum of the reward rates. The task reward rates represent the utility gained per unit of time a task is run at a given frequency.

Each task may run at one of several allowed frequencies, and the reallocation algorithms will adjust the reward rates and utilizations for the tasks by adding or removing discrete utilization increments. Each utilization increment has a reward rate increment associated with it.

The system-wide reward rate is the sum of all the individual task reward rates. The system-wide reward rate function will be additive. For sets of related tasks which are dependent on each other such that they could not have independent terms in the system-wide reward rate function, the tasks may be required to have their utilization increased or decreased together. Also, one task gaining a higher reward rate by using more resource may be dependent on a related task being at a specific level of performance. A simpler algorithm can be used if the tasks are independent.

In order to optimize computer system performance, one must have some method of determining the reward rates for running a task at different periods. This measure will be the performance index (PI), which is the objective to be minimized or maximized by the control. For instance, if the goal is to minimize fuel consumption in an aircraft maneuver, the PI would be a measure of how much fuel is used. In tracking problems, the PI is often the integral of the square of the deviation from the desired trajectory. The advantage of using the PI is that there is a large body of existing literature on optimal controls, that can be used to provide good performance measures for particular task goals, and can assist in the clarification and definition of task goals.

The PI may be used to compare values of the same task running at different periods, but not to compare the value of different levels of performance of different tasks. Ideally, one could measure the effect of changing a task's frequency on overall system performance. This may be difficult to measure, however. An easier substitute approach would be to use an appropriate PI for the task that approximates the value to the larger system, scaled appropriately, as an estimate of the value to the overall system for running the task.

The PI will also be used to determine the cost of allowing a task to go without updates for some length of time. The cost will be used to decide whether the task can simply go without running during the recovery delay after a processor fails, or if the cost of its absence would be so great that a redundant shadow task must be kept so that the task is always running.

A shadow task is a redundant copy of a task running on a different processor in case the processor running the main task fails. For flexible period tasks, it will be assumed that the shadow task is run at the minimal allowed frequency. Since the shadow's outputs are not used most of the time, there is no need to commit more than minimal resources to it.

It will be assumed for all tasks that the function of reward rate with respect to frequency is increasing convex.

3. Computer System Software Model and Assumptions

Before addressing how to carry out reallocation in a real-time control system, some assumptions about the computer system, scheduling disciplines must be made. Several assumptions about the computer system are as follows:

- There are on the order of 4 to 16 processors in the system. This is reasonable, noting that many real-time multiprocessor systems have tended to be lowly parallel, rather than massively parallel.
- Most of the real-time tasks are periodic.
- Tasks are preemptive and all processors are equivalent.
- Changes in system configuration are from infrequent causes, such as recovery from failures, mode changes, or new long-running applications starting.
- Tasks write out periodic checkpoints, such that they can be re-started easily after a processor failure.
- Some middleware and OS services are provided, including clock synchronization, task replication, and a leader election process.

The above assumptions are commonly used and hold for many real-time control systems.

Three scheduling algorithms for periodic tasks are commonly cited in the literature: rate monotonic (RM), earliest deadline first (EDF), and simply periodic (SP). With RM scheduling a shorter period task has higher priority than a longer period one, and with EDF scheduling the task with the nearest deadline has the highest priority. SP scheduling is a simplification of both RM and EDF in which each task's period is a multiple of all shorter periods and evenly divides all longer periods and the shortest-period tasks have highest priority. This paper will concentrate on SP scheduling, because using SP scheduling facilitates on-line period changes. Some terminology and information for SP scheduling is as follows. Periodic tasks run once in each period; in each period, one *job* of the task runs. Each period is referred to as a *frame*. A set of tasks is schedulable under the SP scheduling discipline if and only if the sum of the tasks' processor utilizations is less than 1.0.

4. Heuristic Reallocation Algorithms

In this section, the reallocation problem is defined, and two heuristic algorithms for reallocation of processor utilization to the tasks are discussed. Their primary objective is to generate an allocation with a high reward rate, and their secondary objective is to minimize the number of tasks moved. Both algorithms are meant to be used for quick on-line generation of a new allocation for a system, which would be required for failure recovery, so they must have low complexity.

The algorithms change the utilizations of the tasks in discrete increments, by changing from one allowed frequency to another, and assign tasks to the processors. These increments are so chosen to preserve the SP property. The algorithms must assign tasks to processors in such a fashion that shadow tasks do not end up on the same processor as the main task. Also, some utilization increments can only be allocated if another task is already allocated to a certain level — thus, there may be dependencies between increments. Finally, there may be instances in which utilization must be increased for multiple related tasks simultaneously in order to get an increase in reward rate.

The complete problem statement, therefore, is to find an allocation, subject to the constraints that:

1. No processor can be used for more than 1.0 utilization,
2. A task and its shadow, or two shadows of the same task, are not placed on the same processor,
3. An increment which is dependent on other increments cannot be allocated unless all of its predecessors have been allocated,
4. For a multiple-task increment for related tasks, all of the component increments are added to their tasks, or none of them are.

The objective is to maximize reward rate by allocating utilization to tasks and assigning them to processors. Further, as discussed in Section 2, it will be assumed that the reward rate functions with respect to frequency are

convex. It will also be assumed that no increment is dependent on an increment of lower or equal reward-rate-per-utilization.

It can be trivially shown that the problem is NP-complete. The problem can be restricted to the NP-complete problem of multiple 1-0 knapsack [7] by letting each task be independent and have one increment only, of arbitrary utilization and reward rate. Thus, heuristic solutions must be used to generate allocations for large-sized inputs.

Two algorithms are presented to solve the problem above. The first is an algorithm for independent tasks (AIT), which does not handle dependencies between increments, and does not handle multiple tasks needing to be incremented simultaneously (that is, it ignores constraints 3 and 4 above.) The algorithm for related tasks (ART) does handle these, but has a higher complexity. Both algorithms work by attempting to assign the highest reward-rate-per-utilization increments first onto the processors, and both use a surrogate knapsack-based approach [7], which is a large, single knapsack used to approximate a number of smaller ones. In this case, the smaller knapsacks are the processors.

The AIT is outlined in Figure 1. It works by estimating how much total resource it may allocate to the tasks, and allocates as if it had a large, single 'pot' of utilization. It then balances the tasks on the processors by the utilization-balancing-decreasing algoritm [2]. Using the assignment generated by the utilization-balancing, it goes to each processor and uses a greedy knapsack algorithm to generate an actual allocation. The greedy knapsack algorithm can produce a result with a reward that is an infinitely small fraction of the optimal reward, but it works well for small items.

A task which has an initial assignment will be moved to the least loaded processor on which it is allowed if the resulting decrease in the sum of the squares of the processor utilizations exceeds a threshold value which is supplied as an argument to the algorithm. The threshold can be used to roughly control the percentage of tasks which are moved during reallocation.

The utilization balancing algorithm is the normal utilization-balancing-decreasing algorithm, modified to be able to check for a shadow not being allowed on a processor in O(1) time, and to be able to go on to check the next least loaded processor in O(log p) time, where p is the number of processors. The worst-case complexity of the entire AIT is $ct \log ct + Bt \log p$, where t is the number of tasks, p is the number of processors, c is the average number of utilization increments per task, and B is the average number of shadows per task.

The AIT was evaluated in several different ways with 100 task sets generated in accordance with Figure 2. The average reward rate with no initial allocation for a given number of processors is shown in Figure 3. The average

---

Inputs:

- Task utilization increments
- Initial task assignments
- Task disjointness requirements due to shadows
- A threshold of balancing improvement that must be exceeded for a task to move

Outputs:

- Assignment of tasks to processors
- Allocation of utilization to tasks

1. Construct a vector of all the utilization increments of all the tasks
2. Sort the vector by decreasing reward rate per resource
3. In order of decreasing reward rate per resource, allocate increments from the vector to the tasks until allocated utilization exceeds total available utilization. This generates an initial guess allocation
4. For each task, in decreasing order of amount of utilization allocated in the guess allocation,
   - If the task has an initial assignment, move it to the least loaded processor on which it is allowed if the resulting decrease in the sum of the squares of the processor utilizations exceeds the threshold value.
   - If the task is not assigned, place in on the least loaded processor on which it is allowed.

   This generates the task assignments to the processors.
5. Reset all the tasks allocations to 0. In order of decreasing reward rate per resource, allocate each increment from the vector to its task if there is sufficient utilization remaining on the processor to which the task was assigned. This generates the final allocation.

Figure 1. Algorithm for Independent Tasks (AIT)

---

reward rate for starting with 2 more processors than a given number and then removing 2 randomly and reallocating, in order to simulate a failure, is shown in Figure 3. In each of these simulations, the average of the generated reward was close to the average of the upper bound on the optimal reward for the task sets, as shown in the figures.

Also shown in Figure 3 is the reward rate obtained by using an alternative algorithm, which instead of allowing tasks to be re-assigned to any processor, assigns all of the tasks on a failed processor to a single randomly selected processor, then allocates greedily to the tasks there. This method does not produce as high a reward rate, although it does have some possible advantages. One advantage is that there is less computation required to generate the next assignment. It may also have much

- 50 tasks
- Task period extensibility distribution: 1-20% 2-30% 4-30% 8-20%
- Task reward for first increment uniformly distributed between 1 and 10
- Each extension by factor of 2 reduces reward by a factor which is uniformly distributed between 1.5 and 5.0
- Task maximum utilization uniformly distributed between 0.02 and 0.33
- 25% chance of a task requiring a shadow

Figure 2. Synthetic Task Set for AIT

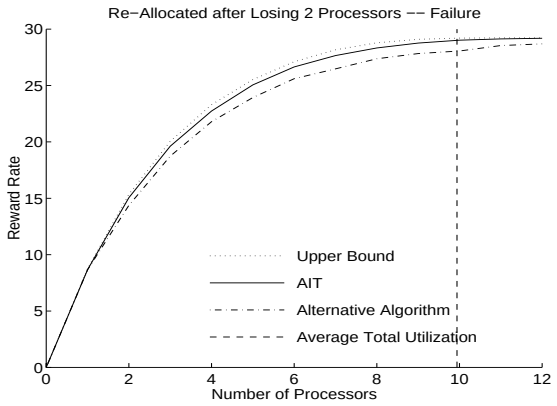

Figure 3. After Losing 2 Processors with AIT

less recovery overhead, since the several processors that may accept the tasks of a given processor could be chosen in advance, with task code placed on them and task checkpoints sent to them.

In the case where 2 processors were gained, the average fraction of the tasks which changed processors is shown in Figure 4. As can be seen, when there are few processors, a higher fraction of tasks move than when there are more, since the additional resource is a higher fraction of the total resource.

The algorithm as presented does not have a bound for the ratio of reward in the generated allocation to the optimal reward — it could be zero. It can, however, easily be shown that from a clean start (no initial assignment) and without shadows, that given an allocation to the tasks requiring a total of U utilization, and given $\lceil 2U \rceil$ processors, all tasks will receive their allocated amount. Without shadow tasks and initial assignments, the AIT could be modified to have a bound of 1/2 the optimal reward rate. The method of achieving this bound is similar to the bounded greedy knapsack in [5]. A proof of these bounds is omitted here for lack of space.

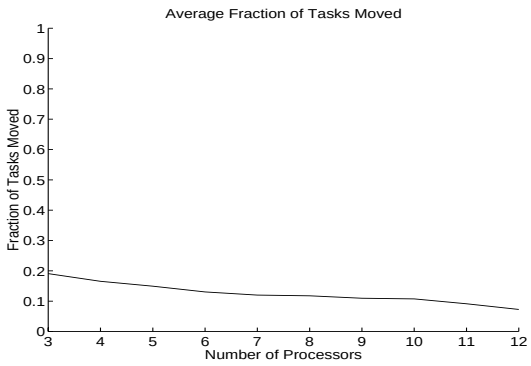

Figure 4. Fraction of Tasks Moved after Gaining 2 Processors

Inputs :

- Multi-task and single-task utilization increments,
- Disjoint placement requirements due to shadows
- Initial task assignments

Outputs :

- Assignment of tasks to processors
- Allocation of utilization to tasks

1. Construct a vector of all the utilization increments of all the tasks
2. Sort the vector of increments by decreasing reward rate per resource
3. Do a binary search over the interval [0,P] to tolerance *tol* to find the largest surrogate knapsack value whose corresponding greedy allocation can be packed on the processors:
   - Allocate by greedy knapsack algorithm into the surrogate knapsack
   - Sort tasks by utilization allocated
   - In order, assign tasks where currently assigned, if sufficient utilization remaining
   - Place the remaining tasks Best-Fit-Decreasing

Figure 5. Algorithm for Related Tasks (ART)

The algorithm for related tasks (ART) is outlined in Figure 5. The algorithm searches for the largest size of surrogate knapsack that generates an allocation that can be successfully packed on to the processors. Unlike the AIT, the ART allows dependencies of one increment on another, and multiple task increments are also allowed.

The bin packing algorithm is a normal best-fit-decreasing bin packing algorithm, modified to be able to determine if a shadow task cannot be placed on a

- There are 10 groups of related tasks which must be increased together
- The number of tasks in each group is uniformly distributed between 1 and 8
- The reward for the first increment of each group is uniformly distributed between 1 and 10
- The distribution of the extensibility of the group is: 1-20% 2-30% 4-30% 8-20%
- Maximum utilization is uniformly distributed between 0.04 and 0.50 for each task
- For each task group, each extension by a factor of 2 reduces reward by a factor which is uniformly distributed

processor in O(1) time, and able to find the next-best-fit in O(log *p*) time, where *p* is the number of processors. The worst-case complexity of the entire ART is *ct* log *ct* + *Bt(*log *tol)(*log *p)*, where *t* is the number of tasks, *p* is the number of processors, *c* is the average number of utilization increments per task, *tol* is the tolerance of the binary search, and *B* is the average number of shadows per task.

The ART was evaluated in several different ways with 100 task sets generated in accordance with Figure 6. The average reward rate with no initial allocation for a given number of processors is shown in Figure 7. The average reward rate for starting with 2 more processors than a given number and then removing 2 randomly and reallocating, in order to simulate a failure, is also shown in Figure 7. In each of these simulations, the average of the generated reward was close to the average of the upper bound on the optimal reward for the task sets, as shown in the figures.

The fraction of tasks moved in the case of gaining 2 processors is shown in Figure 8. Again, with increasing number of processors, a lower fraction of running tasks are moved.

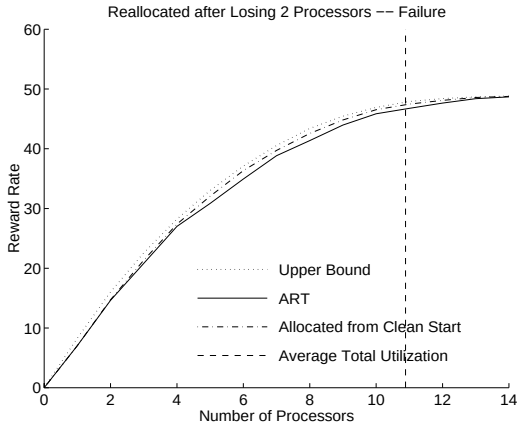

Figure 7. ART After Losing 2 Processors

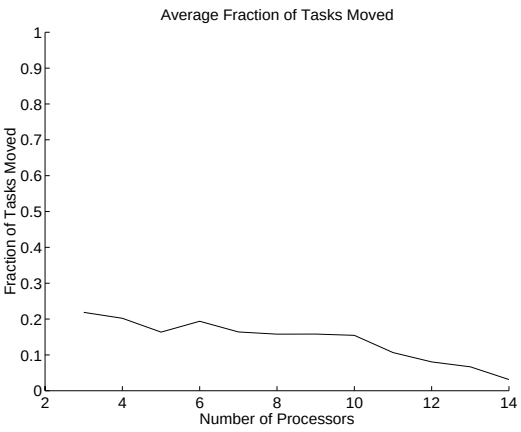

Figure 8. Fraction of Tasks moved after Gaining 2 Processors with ART

5. Scheduling Issues for Reallocation

After a new task allocation has been generated, it must be communicated to all the processors, which must adjust task periods and migrate tasks in order to bring it into effect.

It can be shown that only jobs that are in progress during reallocation can be given too little time under the SP scheduling discipline. Once a task is scheduled the first time after the reallocation, all lower- and equal-period tasks will have already re-started at their new allocation. Since the new allocation is valid (i.e., the utilization is less than 1.0), the utilization of the given task and lower- and equal-period tasks must sum to less than one. Longer-period tasks can be ignored, since the shorter period tasks have priority over them. Thus the job of a task that starts in the period following reallocation will have sufficient time to complete.

Tasks which are migrated, or have their periods change, will lose a job that is running when reallocation starts. For jobs which do not finish in one of their frames, whether to continue the job in the next frame, or start a new one, is task-dependent.

6. Example Application

An example of determination of the reward rates for different frequencies of a control application with a flexible period, and determination of the reward rate for a shadow task for that application, is given as follows.

Consider a classical inverted pendulum application, shown in Figure 9. The objective is to keep the angle *x* zero, while using only a low control torque. Noise torque disturbs the system, and requires the controller to react to

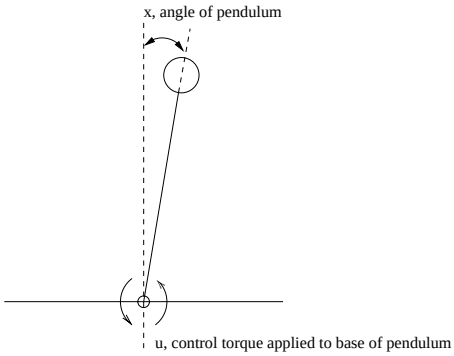

Figure 9. Example System

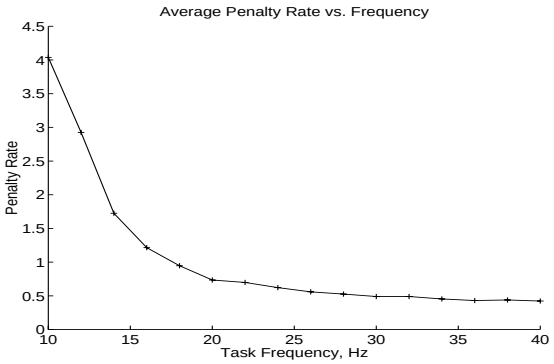

Figure 10. Average Steady-State Penalty Rate

it. A simple compensator will be emulated in a control task with a zero-order-hold approximation, and the constants of the approximation are re-calculated each time the task's period changes. The transfer function for the plant is $1/(s^2-1)$, and for the compensator $35(s+2)/(s+10)$.

In this example, the PI will be:

$$J(x, u, t) = \int_0^t (x^2 + u^2) d\tau$$

where x is the angle of the pendulum, and u is the control torque, and the objective is to minimize the PI. This is a common form of PI [4], being a measure of controlled system performance and of energy used, and the coefficients of the two terms (in this case both 1) can be varied to make either system performance or control energy more important.

The average of dJ/dt over a simulation interval is used to generate a steady-state penalty rate for frequency f, as follows:

$$\mathrm{PR(f)} = \frac{1}{t_{ss}} \int_0^{t_{ss}} (x^2 + u^2) d\tau$$

where u is the emulated control law at frequency f, and t_{ss} is long enough that the average penalty rate can be considered steady state. Running the emulated control law at different frequencies for 10 seconds and averaging produces the average penalty rate function shown in Figure 10. As can be seen, in this case the function of the penalty rate with frequency is concave, meaning that the function of reward rate with frequency will be convex.

In order to estimate the value of a shadow task, the cost of allowing the system to go without an update for a given period of time must be found. A plot of the penalty for going a given time without updates is shown in Figure 11.

The penalty for going time t without an update, P(t), is calculated as the average penalty for a set of 30 second intervals with the emulated control law running at f_{min}, with a gap of time t in the control updates, subtracted from the average penalty over 30 seconds with a non-interrupted control law at f_{min}. For this example, f_{min} is chosen to be 10 Hz.

For this example system, as can be seen in Figure 11, the penalty for going a given time without updates accelerates until it hits the "stop", when it becomes linear and continues at the maximum allowed penalty. Also, as in the case in many systems, brief periods without an update cost very little, meaning that there is no great increase in penalty if the task misses a few periods.

It is now desired to translate these results from control theory into reward rates for the reallocation algorithms. The steady-state penalty rate mentioned above is used to generate a reward rate. For use in reallocation, the raw penalty or reward rates must be weighted, to give an adjusted reward rate. The weight must be specified externally to the task. The PI for a given task can be used to compare the reward rate for running that task at 10 Hz against the reward rate for the same task running at 20 Hz, but cannot be used to compare one task against another. Using a PI can cut the designer's problem from having to specify a reward function for the task to having to specify a single number, the weight, but one would also desire a systematic way of finding the weight. This issue is an area for future work.

The cost of going without updates is used, along with the estimated recovery delay and the estimated failure rate, to determine an expected cost incurred due to not having a shadow task. Using the estimated reward rate for a shadow, it can be decided whether it would be better to spend the utilization needed for a shadow task indeed running a shadow task, or instead making the primary task run with better performance. This evaluation is a specific case of performability [8].

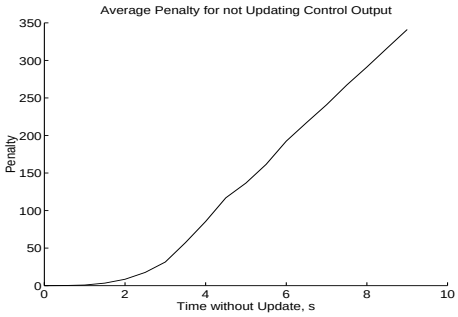

Figure 11. Average Penalty for Going without a Control Update

7. Conclusion

We have shown a method for adaptive processor utilization allocation in real-time control systems that can be used for graceful degradation, for system overloads due to new applications or changes in applications, and for future expansion in resource use if total resources become greater due to faster processors or applications being no longer needed. We have presented a framework for valuing the running of control tasks at different frequencies that uses well-understood performance indices from control theory. We have presented two algorithms for processor utilization reallocation, one for independent tasks, and one for related tasks, and evaluated each with synthetic workloads.

There are several areas for future work. One is finding a systematic way of determining the coefficients by which to scale the PI to get the adjusted reward rate. Another possible area of future work is to allow other resources to be allocated, along with processor utilization. The two main options are communication bandwidth, and memory. Finally, this paper has concentrated on periodic control tasks, but extending the allocation of processor utilization to tasks which do not fit the periodic model, but still use a given fraction of the processor's utilization over some period of time, would be simple. This is because the AIT and ART algorithms allocate utilization without relying on assumptions about the scheduling of the tasks.

References

[1] T. Abdelzaher, E.M. Atkins, and K.G. Shin, "QoS Negotiation in Real-Time Systems, and its Application to Flight Control", *IEEE RTAS 97*, pp.228-238.

[2] Bannister, J.A. and K.S. Trivedi, "Task Allocation in Fault-tolerant Distributed Systems", *Acta Informatica*, Vol. 20, 1983. pp. 261-281.

[3] J.K. Dey, J.F. Kurose, D.F. Towsley, C.M. Krishna, and M. Girkar, "Efficient On-Line Processor Scheduling for a Class of IRIS Real-Time Tasks", *SIGMETRICS 1993*, pp. 217-228.

[4] Franklin, Gene. *Digital Control of Dynamic Systems.* Addison-Wesley, 1990.

[5] Garey, M. and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[6] Liu, J.W.S., W-K. Shih, K-J Lin, R. Bettati, J-Y. Chung. "Imprecise Computations", *Proceedings of the IEEE*, Jan. 1994, pp. 83-93.

[7] Martello, S. and P. Toth, *Knapsack Problems : Algorithms and Computer Implementations*. Wiley, 1990.

[8] J.F. Meyer, "On Evaluating the Performability of Degradable Computer Systems", *IEEE Transactions on Computers* C-29, 1980. pp. 720-721.

[9] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A Resources Allocation Model for QoS Management", *IEEE RTSS 97*, pp. 298-307.

[10] P. Ramanathan , "Graceful Degradation in real-time control applications using *(m,k)*-firm guarantee", *IEEE FTCS 27*, 1997, pp. 132-143.

[11] M. Ryu, S. Hong, and M. Saksena, "Streamlining Real-Time Controller Design: From Performance Specifications to End-to-End Timing Contstraints", *IEEE RTAS 3*,1997, pp. 91-110.

[12] D.B. Seto, J.P. Lehoczky, L. Sha, K.G. Shin, "On Task Schedulability in Real-Time Control Systems", *IEEE RTSS 96*, pp. 13-21.

[13] K.G. Shin, C.M. Krishna, and Y.-H. Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Application", *IEEE Transactions on Automatic Control*, April 1985, pp. 357-365.