Empirical Evaluation of Task and Resource Scheduling in Dynamic Real-Time Systems

Ken Tew, Panos K. Chrysanthis, Daniel Mossé
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{ken,panos,mosse}@cs.pitt.edu

Abstract

This paper reports on our on-going empirical evaluation of a two-tiered resource allocation scheme assuming independent jobs, that is, jobs have no precedence constraints. The first tier extends the temporal density approach (load balancing within a time interval), while the second tier uses variations of the Earliest Deadline First (EDF) approach to schedule jobs at a site.

However, job scheduling at sites is constrained by the precedence relation between the loading and execution of a job. That is, in addition to CPU scheduling, we also take care of the time it takes to load a task onto memory from a disk (or from another processor over the network). In our scheme, loading (i.e., disk scheduling) uses non-preemptive EDF whereas the execution (i.e., CPU scheduling) uses preemptive EDF.

1 Introduction

It is well accepted that there are two classes of real-time systems: hard real-time, in which the violation of the timing properties is a fatal error, and soft real-time in which such a violation consists of a predicted mode operation (degraded mode). In satisfying the timing properties, current efforts in real-time systems have mainly focused on the allocation of resources to real-time tasks or jobs and their scheduling. In general, these efforts have assumed that all jobs in the system are known a priori and always loaded in memory ready to execute [2, 3, 6]. “Next-generation” real-time computing systems [7] are expected to be distributed and to support a variety of real-time applications, including multimedia applications with high data volumes and tight timing constraints. These applications dynamically submit jobs to the system for execution and it cannot be assumed that these jobs are always loaded in memory [2, 1].

Our research focuses on the support for applications in dynamic real-time systems and in particular on the problem of choosing sites to execute the jobs that are submitted to the system. Specifically, our goal is twofold: first, to develop a heuristic to choose a site for jobs to execute; and second, to study the problem of allocating multiple resources to a single job. Towards this end, we have developed a two-tiered resource allocation scheme. The first tier extends the temporal density approach [5, 4], which yields natural load balancing properties, to create an algorithm for even distribution of jobs among the various sites. Such distribution will be considered within intervals of time dictated by the jobs being submitted to the system. The second tier uses an Earliest Deadline First (EDF) approach to schedule jobs at a site [2].

In this paper, we report on our on-going empirical evaluation of our allocation scheme assuming indepen-
dent jobs, i.e., jobs have no precedence constraints. However, job scheduling at sites is constrained by the precedence relation between the loading and execution of a job. Here, we assume that loading (i.e., disk scheduling) follows an EDF non-preemptive discipline whereas the execution (i.e., CPU scheduling) follows a preemptive EDF policy.

2 System Model

Our system architecture consists of heterogeneous processors, interconnected via some network. Each processor has its own memory and its own disk. The characteristics of each disk vary, as in many real-life installations. That is, the disks, distributed throughout the system, are of heterogeneous speeds and bandwidths. The requests for data retrieval from a disk are submitted to the disk device driver and are serviced non-preemptively.

In our model, jobs can represent, for example, the data transformations or manipulation for a control application. We define a job as a tuple $J_i = \langle r_i, d_i, c_i, s_i \rangle$, where $r_i$ is the release time (earliest start time of the job), $d_i$ is the deadline, $c_i$ is maximum computation time (also called worst case execution time), and $s_i$ is the size of the job (how much memory it will occupy) in bytes. We assume that all jobs are loaded on all disks, but not on all processors. This is in contrast with the usual assumption in several real-time scheduling works, which assume that jobs are already loaded in all processors’ memory.

Once a job $J_i$ is submitted to the system, it is decomposed into two tasks with a precedence relation between them. The first task $T_i^l$ loads the job image from disk to main memory and the second task $T_i^e$ executes the image. The characteristics of these tasks can be derived from the characteristics of the corresponding job: $T_i^l = \langle r_i, d_i, c_d, s_l \rangle$ where $d_d = d_i - c_i$ and $c_d$ is the loading time which is a function of the disk characteristics and the size $s_l$ of the job, and $T_i^e = \langle r_{ie}, d_i, c_i \rangle$ where $r_{ie}$ is derived from the finish time of $T_i^l$.

The scheduling of these two types of tasks may follow independent, different disciplines. In our current system, a non-preemptive EDF discipline is employed for the first type of tasks (loading) whereas the second type (execution tasks) is a preemptive EDF. Note that in this paper we do not permit the two tasks of the same job to execute concurrently.

3 Resource Allocation

In this section, we describe the software architecture of our system, in terms of the components that perform the resource allocation. As mentioned above, our system employs a two-level resource allocation algorithm. When a job arrives from an application, it is passed to a global allocator (GA) that decides which site the submitted job should be sent. At each site, a local scheduler (LS) decides whether or not a submitted job can be accepted, when to schedule its loading task and when to schedule its execution task.

Local Scheduler (LS)

LSs schedule jobs based on their deadlines. When a new job arrives at a site, its deadline is checked. If the new job has an earlier deadline than the currently executing job or if the new job’s deadline falls within a so-called scheduling horizon (which is an interval defined for the system), the job is scheduled immediately. Otherwise the LS waits until the current job completion before it schedules the new job. The scheduling of a job consists of submitting a disk load request to the local DMA and the placing of the job’s execution task in the execution (dispatching) order. A disk load request (i.e., a loading task) is submitted only if the disk utilization will not exceed a given threshold, $dt$. This threshold controls the load on the disk; if the threshold is very small, only a single request will be sent to the disk driver; if the threshold is large, all incoming requests will be sent to the disk.

Global Allocator (GA)

The GA can be centralized or distributed; the latter increases the overhead due to the need for information exchange, but has the advantage of load balancing and

\footnote{It has been shown [1] that jobs with precedence constraints can be mapped into independent jobs by manipulation of their timing constraints.}
fault tolerance. For simplicity, in this study we assume a centralized GA.

Clearly, the simplest and fastest way of choosing LSs is for the GA to randomly select a site, without attempting to acquire or examine any knowledge about the state of the site. This randomized allocation strategy has the typical advantage of incurring a constant time overhead, and the simplicity of the implementation. It is our conjecture that it is preferable to accumulate and explore more information about the sites' capabilities as well as the state of the sites' resources when a job is considered for scheduling. We have devised three different schemes to maintain information about the CPU and disk load at each site. However, it is not immediately clear which scheme performs better, that is, which one allows more jobs to execute and complete by their deadlines.

Since the overhead of maintaining and transmitting the information among sites may overshadow the gain of the use of the knowledge about the state of each site, we set out with the goal of examining and contrasting the three schemes with each other and with the random allocation scheme. Following are the three schemes. Each will direct a new job to the least loaded site.

**non-interval**, takes into consideration the number of jobs sent to a particular site.

**fixed interval**, examines the number of jobs sent to a particular site within a fixed-length interval.

**variable interval**, examines the number of jobs sent to a particular site within a variable-length interval.

These schemes increase the complexity of job distribution, but allow for more accurate resource allocation. The problems encountered here are akin to the problem of memory allocation in operating systems, such as fragmentation, under-utilization, overhead, etc.

### 4 Simulation

To measure the performance of the algorithm presented above, we have been implementing a distributed discrete event simulator in C. The GA and each LS are run as separate processes communicating via sockets. Time is handled by the GA. Each LS sends the GA the time for its next event, the GA takes the minimum time of all next events and sends the “next-event time” back to all the LSs. The “next-event time” considers also the activities at the GA. Each of the activities performed by GA and by the LSs count towards the time it takes to finish that activity. In other words, the simulator accounts for overheads of the algorithms.

The other information sent back to the GA by each LS is simply the list of jobs being processed by the LS. The way that the GA handles and uses the information depends on the selection scheme used. For example, in the random case, no information is used, whereas for the interval cases, the information on the jobs in the schedule and wait queues is used.

The events driving the simulation are the arrival, start, and completion of a job as well as loading jobs from disk onto memory. We generated job sets and ran the different schemes on the same job sets. Locally, the tasks of jobs are scheduled using the same EDF algorithms.

The simulation parameters that can be controlled are summarized in Table 1. Note that the deadline of the tasks is controlled by the window ratio $wr$, that is, the input parameter to the simulator is $wr$ and the deadline is derived as $d = r + (wr \cdot c)$. The processor load ranges from 0.5 to 1.2 (i.e., $0.5 < \gamma \leq 1.2$). For example if $\gamma = 1$, $n = 4$, and $c = 4$, then the job inter-arrival rate is 1. This means that, on an average, one job arrives in the system every unit of time and thus the processor load on each of the four processors is 1.

In addition to the information below, the processors simulated have the following characteristics. Processor speed: 50 MHz; processor memory 128MB; disk transfer speed, 10MB/sec; and disk access time, 10ms.

In order to be able to reason about the schemes, we collected the following information about jobs:

- rejected by the GA, called $ga\_rejected$ jobs: number of rejected jobs and processing time of all reject jobs;
- rejected by the LS, called $ls\_rejected$ jobs: number of jobs rejected on arrival by the LS, number of

---

2The idea is to make the LSs completely independent of the GA; the LSs send the information to the GA even if the GA will not use it.
Table 1: Parameters for Simulations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Symbol</th>
<th>Distribution</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of processors</td>
<td>n</td>
<td>fixed</td>
<td>$3, 4, \ldots, 20$</td>
</tr>
<tr>
<td>computation time</td>
<td>c</td>
<td>uniform</td>
<td>mean=90ms</td>
</tr>
<tr>
<td>job size</td>
<td>s</td>
<td>uniform</td>
<td>mean=288K</td>
</tr>
<tr>
<td>processor load</td>
<td>$\gamma$</td>
<td>uniform</td>
<td>mean=0.5, 0.6, \ldots, 1.2</td>
</tr>
<tr>
<td>disk bandwidth</td>
<td>b</td>
<td>fixed</td>
<td>8Mbps</td>
</tr>
<tr>
<td>disk threshold</td>
<td>$dt$</td>
<td>fixed</td>
<td>0.0001, .5, .75, .9, 1</td>
</tr>
<tr>
<td>inter-arrival time</td>
<td>$\alpha$</td>
<td>uniform</td>
<td>mean=$\frac{c}{(\gamma \cdot n)}$</td>
</tr>
<tr>
<td>window ratio</td>
<td>$wr = \frac{4-c}{c}$</td>
<td>uniform</td>
<td>mean=3, 5, 7, 9, 11</td>
</tr>
</tbody>
</table>

jobs rejected after loading, process time of all jobs reject after loading, and load time of all rejected jobs;

- accepted by the GA but not able to run in the individual processor (aborted jobs): number of aborted jobs, as well as process time, load time, and remaining time of all aborted jobs;
- loaded and completed within the deadlines (completed jobs): number of jobs completed, and process and load time of all completed jobs.

We have started analyzing the data, and have observed a good correlation between the complexity of the scheme and the acceptance and completion ratio of jobs. This leads us to believe that even though the variable interval scheme has more overhead, the benefits of the scheme outweigh its loss in performance due to the overhead.

5 Conclusions

We started this project as an attempt to depart from the usual assumptions that drive most real-time simulation studies: tasks are loaded on memory prior to their activations. While this can be done in static systems, it is not a reasonable assumptions for dynamic systems.

Our preliminary work shows that the overhead incurred in the scheduling is not significant, and due to the increasing bandwidth of current and future networks, we believe that a more complex scheme that requires frequent exchange of state information will increase the acceptance and completion ratio of the system.

References