Adaptive Resource Management Algorithms for Periodic Tasks in Dynamic Real-Time Distributed Systems

Binoy Ravindran and Ravi K. Devarasetty

The Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia 24061
E-mail: binoy@vt.edu

and

Behrooz Shirazi

Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, Texas 76013

Received April 3, 2001; accepted April 25, 2002

We present adaptive resource management middleware techniques for periodic tasks in dynamic real-time distributed systems. The techniques continuously monitor the application at run-time for adherence to the desired real-time requirements, detect timing failures or trends for impending failures (due to workload fluctuations), and dynamically allocate resources by replicating subtasks of application tasks for load sharing. The objective of the techniques is to minimize (end-to-end) missed deadline ratios of the tasks. We present "predictive" resource allocation algorithms that determine the number of subtask replicas that are required for adapting the application to a given workload situation using statistical regression theory. The algorithms use regression equations that forecast subtask timeliness as a function of external load parameters such as number of sensor reports and internal resource load parameters such as CPU utilization. To evaluate the performance of the predictive algorithms, we consider algorithms that determine the number of subtask replicas using empirically determined heuristic functions. We implemented the resource management algorithms as part of a middleware infrastructure and measured the performance of the algorithms using a real-time benchmark. The experimental results indicate that the predictive algorithms outperform the heuristic strategies under the workload conditions that were studied.

1Supported by the U.S. Office of Naval Research under Grants N00014-99-1-0158 and N00014-00-1-0549.
2To whom correspondence should be addressed.
1. INTRODUCTION

Real-time computer systems that are emerging for the purpose of strategic mission management such as collaborating within a team of autonomous entities conducting manufacturing, maintenance, or combat are subject to significant execution-time uncertainties at the mission and system levels. Typically, processing and communication latencies in such systems do not have known upper bounds and event and task arrivals and failure occurrences are non-deterministically distributed. Another source of non-determinism is that some events and state changes are apparently spontaneous to the computer system per se, because their causal reasons are from outside the system. This emerging generation of real-time distributed computing systems are very important for (real-time) supervisory control in many domains, including defense, industrial automation, and telecommunications [Koob96, Jen92].

Such real-time mission management applications require decentralization because of the physical distribution of application resources and for achieving survivability in the sense of continued availability of application functionality that is situation-specific. Because of their physical dispersal, most real-time mission management applications are “loosely” coupled using communication paradigms that employ links, buses, rings, etc., resulting in additional uncertainties e.g., variable communication latencies, regardless of the bandwidth [Jen92].

Most of the past efforts on real-time computing research focus on hard real-time computing that is usually centralized [KS97, LL73, Son95, SR88, SR93, SSRB98, TK91, XP90], but occasionally distributed [CSR86, Kao95, Kop97, RSZ89, Shi91, SR91, SRC85, Ver95]. The hard real-time computing theory assumes that application properties such as task parameters and execution environment characteristics such as scheduling overheads and interrupt service latencies are deterministically known with absolute certainty in advance. The theory exploits such a-priori information and provides the hard real-time guarantee of unanimous optimum, i.e., “always meet all deadlines” on the application and system behavior. However, the non-determinism in the application and system characteristics of real-time mission management applications generally makes it non-cost-effective or even impossible to complete the execution of every real-time computation at its optimum time, i.e., before the deadline [Jen92, Koob96, SK97, Sta96]. Most mission management applications actually desire only a sufficient number of computations to have a high likelihood of completing at its optimum time under the current application and system situations [Jen92]. Thus, the soft real-time scheduling criterion of non-unanimous, non-optimum (e.g., minimize number of missed deadlines) becomes the desired objective in such systems.

In this paper, we present adaptive resource management middleware algorithms for periodic tasks in dynamic real-time distributed systems. The algorithms continuously monitor the application tasks at run-time for adherence to the desired real-time requirements, detect timing failures or trends for impending timing failures (due to workload fluctuations), and dynamically allocate resources to improve application timeliness. The algorithms perform resource allocation by
replicating subtasks of the tasks that are subject to workload increases. The idea behind replication of subtasks is that once a subtask is replicated, the replicas of the subtask can share the workload that was processed by the original subtask. Further, concurrency can be exploited by executing the replicas on different processors and thereby the end-to-end latency of the task can be reduced.

The objective of the algorithms is to minimize (end-to-end) missed deadline ratios of the tasks. We present “predictive” resource allocation algorithms that determine the number of subtask replicas that are required for adapting the application to a given workload situation using statistical regression theory. The algorithms use regression equations that forecast subtask timeliness as a function of external load parameters such as number of sensor reports and internal resource load parameters such as CPU utilization. We use a real-time benchmark [WS99] as an example application for computing the regression techniques. The timeliness of the subtasks of the benchmark is measured for a number of external load and internal resource load scenarios. The profiled data are then used to analytically derive regression equations that can extrapolate the subtask timeliness for arbitrary load scenarios. The regression equations are used by the resource management algorithms at run-time for forecasting subtasks’ timeliness on processors at their current utilization levels and at given external loads. The forecasted timeliness values are used to determine the number of subtask replicas and processors for executing them that will adapt the application to the current workload situation.

To evaluate the performance of the predictive algorithms, we consider algorithms that determine the number of subtask replicas using empirically determined heuristic functions. We consider functions that compute the number of replicas as a quantity that monotonically increases or decreases with increases or decreases in the percentage application workload, respectively. To study the performance of the algorithms, we implement the algorithms as part of a middleware infrastructure. The middleware is then used to satisfy the real-time requirements of the benchmark application. The experimental results indicate that the predictive algorithms produce lower missed deadline ratios and higher steady-state latency slack values than the heuristic strategies for the workload conditions that were studied.

Thus, the contribution of the paper is predictive algorithms that perform adaptive resource allocation in dynamic real-time distributed systems using statistical regression theory. Furthermore, the effectiveness of the algorithms is validated by implementation and an experimental study.

The rest of the paper is organized as follows: We present a generic real-time system in Section 2. The generic real-time system is used to reason about dynamic real-time distributed systems and their load characteristics. Section 3 presents our middleware architecture. We present the real-time benchmark application that is used for evaluating the resource management algorithms in Section 4. Section 5 presents the predictive resource allocation algorithms. We discuss the heuristic strategies in Section 6. Experimental evaluation of the algorithms is presented in Section 7. Finally, we conclude the paper with a summary of the work and its contributions in Section 8.
2. A GENERIC REAL-TIME SYSTEM

We consider a generic real-time distributed system that consists of a set of trans-node tasks that perform assessment of the environment, initiation of actions, and monitoring and guidance of the actions to their successful completion. The inter- and intra-relationship of the tasks are illustrated in Fig. 1.

The assessment task periodically collects data from the environment using hardware sensors. The data is filtered, correlated, classified, and then used to determine the necessity of an action by the system. When an action is necessary, the task generates an event that activates the initiation task. The initiation task determines the action that needs to be taken and causes actuators to perform the action. Since the task executes in response to an event that can occur at any time, the initiation task has an aperiodic behavior. Upon initiation of the action by the actuators, the guidance task is notified. The guidance task repeatedly uses sensors to collect data, to monitor the actions that were initiated, and to guide the actuators to successful completion of the actions. Note that the activation of the guidance task begins and terminates aperiodically, and once active, it executes periodically. Thus, the guidance task has a transient-periodic behavior.

The real-time requirements of the tasks include deadlines for the completion of each instance of task execution. Observe that during each execution period of the assessment and guidance tasks, the sensor may generate any number of data items, which must be processed by the tasks within the deadline. Furthermore, the sensor data (per period) may result in any number of aperiodic events that trigger the execution of the initiation and guidance tasks, which the tasks must respond and complete within their deadlines.

After a careful study of the U.S Navy’s AAW real-time system [WRSB98], we have observed that the upper bound on the size of the data that arrives during each period of periodic and transient-periodic tasks and the arrival rates of events that trigger the execution of aperiodic and transient-periodic tasks are not known a priori.\(^3\) Hence, the upper bound on the end-to-end execution latencies of the tasks is not deterministically known as the size of the data that the tasks need to process during each period significantly influences the task execution latencies. We call such real-time systems as “dynamic” real-time systems.

3 In systems such as the Navy’s AAW, many periodic tasks process sensor reports, which are periodically generated by radar systems. Upper bounds on the number of sensor reports (“radar tracks”) are not deterministically known, as they depend upon the current operational scenarios of the system.

3. ADAPTIVE RESOURCE MANAGEMENT MIDDLEWARE

The software architecture of the resource management middleware is shown in Fig. 2. The core components of the middleware include a parser, a system data broker, hardware monitors, software monitors, a resource manager, and a program control component. Details of the middleware can be found in [RWS01]. Here, we only summarize the functionality of each of the core components and their interactions.
The system data broker is responsible for collecting and maintaining all application information. The parser is the front-end to the system data broker. It reads a description of the application and its timeliness requirements that are expressed using the abstractions of a system description language and builds the data structures that model the application. Dynamically measured performance metrics of the application are collected and maintained by the software monitors. The system data broker obtains measurements of the dynamic attributes of the application program components from the software monitors. Performance metrics of hardware
resources are collected and maintained by the hardware monitors. The metrics are transmitted to the system data broker on demand as well as periodically.

The system description language provides concrete abstractions to describe the architectural-level properties of the application such as composition and interconnections of application software and hardware resources, and its timeliness and survivability requirements. Details of the system description language can be found in [WRSB98].

The hardware monitors consist of a set of host monitor daemons, a hardware broker program, and a hardware analyzer program. There is one host monitor daemon per host machine. The daemon programs acts as “bidders” for host and network resources. Host monitor daemons collect various host-level metrics such as CPU-idle-time for each host in the system and network-level metrics such as number of communication packets that are sent out through, and received at, network interfaces of hosts in a network. The low-level metrics of hardware resources are sent to the hardware broker by the daemons in a periodic manner. The hardware broker thus becomes a repository of “raw” hardware performance information. The broker periodically sends the raw metrics to the hardware analyzer. The hardware analyzer computes higher-level metrics such as exponential-moving-averages, trend values, and aggregate metrics from the low-level metrics. The metrics are computed for both host and network hardware resources. The hardware analyzer provides this data to the resource manager.

The software monitors consist of a set of task manager programs. The task managers monitor task-level (end-to-end) timeliness and alert the resource manager of low timeliness situations. Example of a low timeliness situation is when the task latency exceeds its deadline for a significant number of instances during a set of past task periods. There is one task manager per task. Each task manager receives time-stamped event tags from application programs (or subtasks), transforms them into task latencies, and compares the latencies against task deadlines for detecting low timeliness situations. When a task exhibits low timeliness, the task manager of the task detects the situation and notifies the resource manager.

When the resource manager is notified of a low timeliness situation of a task, it first performs “diagnosis” to determine the causes of the low task timeliness. Example of a diagnosis is to determine the set of application subtasks of the task that are experiencing increased execution latencies due to increases in workload.

Once the subtasks that are causing the low timeliness situation for a task are identified, the resource manager performs allocation analysis to identify possible resource allocation actions that will improve the task timeliness. In this paper, we focus on subtask replication as the resource allocation mechanism. As discussed previously, the idea behind replication of application subtasks is that once a subtask is replicated, the replicas can share the workload that was processed by the original subtask. Further, concurrency can be exploited by executing the replicas on different processors and thereby the end-to-end latency of the task can be reduced.

Allocation analysis is followed by selection of resources. The resource manager selects host machines for executing the replicas that will improve the task timeliness.

4 The middleware obtains global time through the use of NTP [Mills95].
The resource manager uses load information of resources provided by the hardware monitors for resource selection. Once the subtasks of the task that need replication, number of their replicas, and host machines for their execution are determined, the resource manager notifies the program control component for enacting the actions.

The program control component consists of a central control program and a set of startup daemons. When the resource manager needs to start the replica of a subtask program on a host, it informs the control program, which then notifies the startup daemon on the host. Each host contains a startup daemon, which starts and terminates programs on the host at the request of the control program.

The overall resource management process is summarized in Fig. 3. In this paper, we present algorithms for diagnosis, allocation analysis, and resource selection (steps 2, 3 and 4). Algorithms for performing run-time monitoring (step 1) can be found in [RWS01].

Thus, the middleware algorithms presented in this paper address the question: Which subtasks of a task needs to be replicated, how many replicas are sufficient, and which host machines are needed for their execution that will produce low missed deadline ratios for the task?

4. A REAL-TIME BENCHMARK APPLICATION

We have developed a real-time benchmark application that functionally approximates Navy’s AAW surface combatant system. The benchmark uses simulated sensors and actuators. However, it employs real algorithms for performing tasks such as assessment, initiation, and guidance. Details of the benchmark can be found in [WS99]. Here, we summarize the tasks that have timeliness requirements.

The software architecture of the benchmark application is shown in Fig. 4. The benchmark consists of the following tasks:

1. A periodic assessment task called Sensing that consists of a simulator program called Sensor, a filter manager program called FilterManager, one or more...
replicas of a filter program called Filter, an evaluate and decide manager called EDManger, and one or more replicas of an evaluate and decide program called EvalDecide.

2. A transient task called Engagement that consists of three programs: an action manager program called ActionManager, an action program called Action that is replicated, and an actuator simulator program called Actuator.

3. A transient-periodic monitor and guidance task called MonitorGuide that includes all components of the assessment task, a monitor and guide manager program called MGManager, and a monitor and guide program called MG that is replicated.

The timeliness requirements of the benchmark include deadlines for the completion of execution periods of all three tasks. Further, the transient-periodic MonitorGuide task also has an activation deadline. Here, we focus only on the periodic Sensing task. Note that each execution period of the Sensing task that begins with the generation of data items (by Sensor) and terminates with the processing of all the data items (generated in that period) by EvalDecide has a maximum latency requirement. This latency requirement—the task deadline—must be adhered in all task periods, irrespective of the number of data items generated by Sensor during any period.

5. PREDICTIVE RESOURCE ALLOCATION ALGORITHMS

We present two predictive resource allocation algorithms in this paper. The algorithms determine the number of subtask replicas and their processors that will
adapt the application tasks to workload fluctuations and satisfy the end-to-end task deadlines. As discussed previously, the algorithms are predictive in the sense that they forecast the timeliness of subtasks using regression theory. The algorithms use regression equations that forecast subtask timeliness as a function of the task workload and processor utilization.

We first discuss how the regression equations are determined for the application subtasks in Section 5.1. We then present the two resource allocation algorithms in Sections 5.2 and 5.3.

5.1. Derivation of the Regression Equation

We estimate the execution latency of a subtask as a function of the subtask workload and the utilization of the processor on which it executes. The workload of a subtask is characterized as the number of data items that it needs to process, since the number of data items processed by the subtasks constitute the most significant part of the application workload in applications such as the Navy’s AAW. The processor utilization is characterized by the utilization of the CPU, since we have observed that application programs in systems such as AAW are mostly CPU intensive. Thus, utilization of other resources such as memory is assumed to be relatively insignificant.

To determine the function that computes subtask execution latency as a function of the subtask workload and resource usage, we use application profile data and regression theory. The real-time benchmark application that has resulted from our past effort [WS99] is used as the example application for profiling. We first measure the execution latencies of application subtasks of the benchmark. The latency measurements are made for increasing number of data items processed by the subtasks at a certain level of CPU utilization. For the latency measurements made at a CPU utilization, we first determine a second-order non-linear regression equation that computes execution latency as a function of data size. We then combine the equations for latency measurements made at a set of different CPU utilizations into a single regression equation that computes execution latency as a function of data size and CPU utilization.

The regression equation thus obtained for a subtask $st_j$ of a periodic task $T_j$ is given by

$$\text{ex}(st_j, d, u) = f(d, u) = a_1 \times d^2 + b_1 \times d + c_1 \times u,$$

where the execution latency is obtained in milliseconds, $d$ is the number of data items, $u$ is the CPU utilization in percentage, and $a_1$’s, $b_1$’s, and $c_1$’s are constants that are dependent upon the application subtask.

To determine the coefficients $a_1$’s, $b_1$’s, and $c_1$’s of the regression equation of a subtask, we proceed in the following way: We start with an initial value of zero for all the coefficients. We then measure the execution latency of the subtask at a range of data sizes and for a given CPU utilization. The execution latency measurements are repeated and average values are determined for each data size. For each of the average execution latency measurements, we then determine the percentage error between the observed and predicted values. The error will be large at this stage, as
the predicted values are zero due to the zero value for the coefficients of the equation. We now increase the value of the coefficient $a_i$ by small positive increments and measure the percentage error. We observe that the error decreases as the value of $a_i$ increases up to a certain point, after which the error starts increasing. This “point of diminishing return” is fixed as the value of coefficient $a_i$. We now repeat the process by fixing the value of $a_i$, increasing the value of coefficient $b_i$, and measuring the percentage error. When we observe a reversal in error, the corresponding value of $b_i$ is fixed as its coefficient. The process is repeated to determine the coefficient $c_i$.

Table 1 shows the three coefficients of the regression equation (1) that were derived for the Filter and EvalDecide benchmark subtasks. Figure 5(a) and (b) shows the observed and predicted values of Filter and EvalDecide subtasks at a range of data sizes and for a utilization percentage of ~20%. From the figures (and the corresponding data), we observe that the observed and the predicted values for the application subtasks correlate very well.

5.2. Maximum Lateness First Resource Allocation Algorithm

We present a resource allocation algorithm called Maximum Lateness First (MLF) that replicates the subtask that is experiencing the maximum lateness when a low timeliness situation is detected for the (parent) task. The intuition for doing so is simply that the subtask that is experiencing the maximum lateness may be subject to the highest rate of increase in the workload. Thus, by replicating the maximally late subtask and sharing its load among the replicas, we can improve the timeliness of the task in a manner that will minimize the task missed deadline ratio.

Since subtasks do not have deadlines (deadlines are specified only for tasks), the algorithm assigns individual deadlines to the subtasks and messages of tasks from the end-to-end task deadlines. The subtask that is experiencing the maximum lateness with respect to its individual deadline is then identified as the candidate subtask for replication.

Once the candidate subtask is identified, the algorithm iteratively determines the number of replicas that are needed using the regression equation discussed previously. 

5 We use the simple “sliding window” mechanism described in [RWS01] to detect low timeliness situations. The algorithm monitors task latency samples in a sliding window of 20 latency measurements. Whenever the latency is found to exceed the deadline for 15 samples in the window of 20, a “timing failure” is triggered.
We discuss the assignment of deadlines to subtasks in Section 5.2.1. Section 5.2.2 discusses how the algorithm determines the number of replicas that are needed and processors for executing them.

5.2.1. Assignment of deadlines to subtasks. To assign deadlines to subtasks and messages from end-to-end task deadlines, we use a variant of the equal flexibility (EQF) strategy proposed in [Kao95]. Observe that EQF requires knowledge of the execution times and communication delays to compute subtask deadlines. Therefore, the resource allocation algorithm uses estimates of the initial operating conditions of the system to derive the initial values of execution times and communication delays.

Let \( d_{\text{init}} \) denote the initial data size processed by a subtask \( s t^k_i \) and transmitted by a message \( m^k_j \), respectively. Let \( u_{\text{init}} \) denote the initial CPU utilization of the processor on which the subtask is assumed to execute. The deadline of the subtask \( s t^k_i \) is therefore given by

\[
dl(s t^k_i) = e x(s t^k_i, d_{\text{init}}, u_{\text{init}}) + \left( dl(T_k) - \sum_{j=1}^{m} e x(s t^k_i, d_{\text{init}}, u_{\text{init}}) - \sum_{j=i+1}^{m} c d(m^k_j, d_{\text{init}}) \right) \times \frac{e x(s t^k_i, d_{\text{init}}, u_{\text{init}})}{\left( \sum_{j=1}^{m} e x(s t^k_i, d_{\text{init}}, u_{\text{init}}) + \sum_{j=i+1}^{m} c d(m^k_j, d_{\text{init}}) \right)}, \tag{2}
\]

where \( dl(T_k) \) denotes the deadline of task \( T_k \). We determine the execution time \( e x(s t^k_i, d, u) \) of subtask \( s t^k_i \) for processing a data size \( d \) on a processor with utilization \( u \) using equation (1). The communication delay \( c d(m^k_j, d) \) that is incurred for transmitting a message \( m^k_j \) that carries a data size \( d \) is obtained by measurement.

5.2.2. Determining number of subtask replicas and their processors. To determine the number of replicas that are needed for the maximally late subtask and processors for executing them, the algorithm uses an iterative technique. The procedure is
discussed below. For convenience, we denote the currently executing subtask as the 0th replica of the subtask.

The algorithm starts with a single replica of the subtask—the 1st replica—and forecasts the execution latency of the replica using Eq. (1). The latency of the replica is forecasted on the least utilized processor among the set of processors where the subtask can be potentially executed. The algorithm considers the processor with the least utilization because the forecasted execution latency of the subtask is directly proportional to the CPU utilization in Eq. (1). Also, the processor that is executing the 0th replica is excluded from the set of processors that are considered, as that will not allow concurrency to be exploited.

Note that the regression equation uses the data size that will be processed by the subtask for forecasting the execution latency. The algorithm uses half the data size that is currently processed by the 0th replica as the data size that will need to be processed by the 1st replica. The algorithm also forecasts the execution latency of the 0th replica on its current processor at half the data size, since the addition of the 1st replica will reduce the workload of the 0th replica.

Once the execution latencies of the 0th replica at its current processor and the 1st replica at the least utilized processor is estimated at half the data size, the algorithm determines the maximum of the two latencies. The maximum latency is then compared against half the deadline of the 0th replica. The algorithm reduces the subtask deadline by half for comparison, because the EQF-variant strategy (discussed in Section 5.2.1) assigns deadlines to subtasks in a manner that is proportional to subtask execution times, i.e., larger the execution time, larger will be the deadline. Thus, the initial deadline was assigned to the subtask based on an initial estimate of the subtask execution latency, which was based on the estimated initial data size of the subtask. Since this data size has now changed, the deadlines need to be re-assigned for a fair and meaningful comparison.

If the maximum latency of the 0th and 1st replicas is found to be less than (or equal to) the reassigned subtask deadline, then the algorithm concludes that a single replica is enough for the current workload of the task. Further, the algorithm selects the least utilized processor on which the latency of the 1st replica was estimated as the processor for executing it. The processor of the 0th replica remains the same.

If the maximum latency of the replicas is found to be larger than the reassigned subtask deadline, then the algorithm concludes that a single replica is not enough. The algorithm now fixes the least utilized processor on which the latency of the 1st replica was estimated as the processor for executing it and then considers a 2nd replica. The latency of the 2nd replica is now estimated on the least utilized processor among the set of processors where it can be potentially executed. The processors of the 0th and 1st replicas are excluded from this set for exploiting maximum concurrency. The latency of the 2nd replica is now estimated at one third of the data size that was initially processed by the 0th replica when the parent task was detected for low timeliness. As before, the algorithm also re-estimates the latency of the 0th

\[\text{EQF-variant strategy (discussed in Section 5.2.1) assigns deadlines to subtasks in a manner that is proportional to subtask execution times, i.e., larger the execution time, larger will be the deadline.} \]

\[\text{Thus, the initial deadline was assigned to the subtask based on an initial estimate of the subtask execution latency, which was based on the estimated initial data size of the subtask. Since this data size has now changed, the deadlines need to be re-assigned for a fair and meaningful comparison.} \]

\[\text{If the maximum latency of the 0th and 1st replicas is found to be less than (or equal to) the reassigned subtask deadline, then the algorithm concludes that a single replica is enough for the current workload of the task. Further, the algorithm selects the least utilized processor on which the latency of the 1st replica was estimated as the processor for executing it. The processor of the 0th replica remains the same.} \]

\[\text{If the maximum latency of the replicas is found to be larger than the reassigned subtask deadline, then the algorithm concludes that a single replica is not enough. The algorithm now fixes the least utilized processor on which the latency of the 1st replica was estimated as the processor for executing it and then considers a 2nd replica. The latency of the 2nd replica is now estimated on the least utilized processor among the set of processors where it can be potentially executed. The processors of the 0th and 1st replicas are excluded from this set for exploiting maximum concurrency. The latency of the 2nd replica is now estimated at one third of the data size that was initially processed by the 0th replica when the parent task was detected for low timeliness. As before, the algorithm also re-estimates the latency of the 0th} \]

\[\text{We assume such an even distribution of workload among the subtask replicas in this paper. The motivation for this assumption is due to that property of the benchmark [WS99], which we are using as an example application.} \]
and 1st replicas at the one-third data size on their respective processors. The maximum latency of the replicas is then compared against one-third of the subtask deadline. If the latency is found to be smaller than the adjusted subtask deadline, the algorithm concludes that two replicas are enough for the current situation. The algorithm selects the processors for executing the replicas as those on which their latencies were estimated.

If the maximum latency is found to be still larger, the algorithm considers a 3rd replica for the subtask and repeats the procedure. The procedure is repeated until either the algorithm determines a feasible allocation or exhausts the maximum possible number of replicas that can potentially be executed. In the situation where a subtask can be executed on all processors in the system, we set this upper bound on the number of replicas as the number of processors in the system for exploiting maximum concurrency.

Note that each time the algorithm considers a new replica, it immediately decides on the processors for executing the existing replicas as those processors where their latencies were estimated. Thus, during each iterative step, the algorithm only decides on the processor for executing the new replica. Hence, if there are \( p \) processors in the system and the upper bound on the number of replicas is set as the number of processors (i.e. \( p \)), then the worst-case complexity of the algorithm is \( O(p^2) \).

5.3. All Tardy First Resource Allocation Algorithm

We consider a resource allocation algorithm called All Tardy First (ATF) that is similar to MLF in terms of the way the processors for executing subtask replicas are determined. However, unlike MLF, ATF replicates all subtasks that are found to miss their individual subtask deadlines when a low timeliness situation is detected for the (parent) task. Subtasks that do not miss their deadlines are not replicated by ATF. The intuition for doing so is simply that the subtasks that have missed their individual deadlines may be subject to the highest rate of increase in workload relative to other subtasks of the task. Thus, by replicating all subtasks that have missed their individual deadlines, we can share the increase in the task workload among the replicas and improve the timeliness of the task. Furthermore, since we are replicating all tardy subtasks, the improvement in the task timeliness will be significantly better than that of MLF that replicates a single subtask.

ATF uses the same procedure for assigning initial deadlines to subtasks and messages from end-to-end task deadlines as that used by MTF (Section 5.2.1). At run-time, when a task is detected for a low timeliness situation, the set of subtasks that are found to miss their individual deadlines are identified as the candidate subtasks for replication.

Once the candidate subtasks are identified, the algorithm iteratively determines the number of replicas that are needed for a subtask using the same procedure as that of MLF. However, the process is repeated for each candidate subtask. Thus, the ATF algorithm basically invokes MLF for each candidate subtask to determine the number of replicas that are needed and processors for executing them that will satisfy the subtask deadline. Hence, if there are \( n \) subtasks for a task, the worst-case complexity of the algorithm is \( O(np^2) \) as all \( n \) subtasks can be found to be tardy.
6. HEURISTIC STRATEGIES FOR RESOURCE ALLOCATION

We consider two resource allocation algorithms that use heuristic techniques for determining the number of subtask replicas and their processors that will satisfy the subtask deadline. Like MLF, the heuristic algorithms select the subtask that is experiencing the maximum lateness when a low timeliness situation is detected for the task. However, unlike both MLF and ATF, the heuristics independently determine the number of replicas that are needed for the candidate subtask and the processors that are required for their execution. Both the heuristic techniques use the same technique for determining the number of replicas. They only differ in the way the processors for executing the subtask replicas are determined. We first discuss how the techniques determine the number of replicas in Section 6.1. Section 6.2 describes how replicas are assigned to processors.

6.1. Determining Number of Subtask Replicas

Once the candidate subtask is identified, the heuristic techniques use an empirically determined replica table to determine the number of subtask replicas that are needed. The table contains the number of replicas that are required for each subtask for different (ranges of) percentage increase in data sizes that will satisfy the subtask deadline for that percentage increase in data size. The table is determined off-line by conducting a series of isolated experiments with each subtask. The experiments are similar to the way MLF determines the feasibility of subtask replicas and are described as follows:

A subtask is first assigned an individual deadline according to the EQF-variant technique (discussed previously) for an assumed initial data size condition. The execution latencies of the subtask are then measured exclusively on a processor at this initial data size. The data size is then gradually increased to cause the subtask to miss its individual deadline. We then start a replica exclusively on a processor and measure the latencies of the replicas. The maximum of the latencies is then compared against half the subtask deadline to determine whether the single replica is enough for satisfying the application workload. Otherwise, a second replica is started exclusively on another processor and the process is repeated until either the (adjusted) subtask deadline is satisfied or the maximum number of replicas (set as the number of processors) is exceeded.

The process is repeated several times for the percentage increase that is being considered. We determine the average value of the number of subtask replicas that are needed for satisfying the increase in workload. This number is then entered in the replica table of the subtask for the corresponding percentage increase in data size. We repeat the process for different percentage increases in data sizes, determine the number of replicas that are needed, and store the average values in the replica table. The entire process is repeated for each subtask.

Table 2 shows the number of replicas that were determined for different percentage increases in data sizes for the Filter subtask of the benchmark.
To determine the processors for executing the subtask replicas, we model the problem as a bin-packing problem. Thus, the subtask replicas are modeled as “items” that need to be “packed” into processor bins. Further, we model the “size” of each replica as the processor utilization demand of the replica. The “capacity” of a processor is modeled as its maximum level of utilization, i.e., 100%.

To determine the utilization demand of a replica, we consider the data size load of the replica. This is because, we have observed from our regression study that the processor utilization caused by the replica is almost directly proportional to the number of data items that it needs to process. The regression equation (1) reflects this observation. Thus, if the total number of replicas determined for a subtask is \( n \) (using the replica table), then the data size load of a replica is \( \frac{1}{n} \)th of the data size that was originally processed by the candidate subtask when the (parent) task was detected for low timeliness.

To convert the data size load of a subtask replica into an equivalent processor utilization demand, we again use regression theory. We measure the processor utilization caused by a subtask for processing increasing number of data sizes. From these measurements, we determine a second-order non-linear regression equation that computes utilization demand as a function of the subtask data size.

The regression equation thus obtained for a subtask \( \text{st}_i^j \) of a periodic task \( T_j \) is given by

\[
\text{epu}(\text{st}_i^j, d) = f(d) = a_i \times d^2 + b_i \times d,
\]

where the processor utilization is obtained in percentage, \( d \) is the number of data items, and \( a_i \)'s and \( b_i \)'s are constants that are dependent upon the application subtask. To determine the coefficients \( a_i \)'s and \( b_i \)'s of this regression equation, we use the same iterative approach as described in Section 5.1.

Table 3 shows the two coefficients of the regression equation (3) that were derived for the Filter subtask.

Thus, the problem of determining the processors for executing the subtask replicas becomes the bin-packing problem, i.e., to determine an assignment of replicas to processors that will satisfy the utilization demand of each replica and will not overload any processor. Note that if a processor is able to satisfy the utilization demand of a replica, then it implies that the processor has enough utilization capacity to execute the subtask replica for its current data load.

### TABLE 2

<table>
<thead>
<tr>
<th>% Increase in data size</th>
<th># of replicas</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>75</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

6.2. Assigning Replicas to Processors

To determine the processors for executing the subtask replicas, we model the problem as a bin-packing problem. Thus, the subtask replicas are modeled as “items” that need to be “packed” into processor bins. Further, we model the “size” of each replica as the processor utilization demand of the replica. The “capacity” of a processor is modeled as its maximum level of utilization, i.e., 100%.

To determine the utilization demand of a replica, we consider the data size load of the replica. This is because, we have observed from our regression study that the processor utilization caused by the replica is almost directly proportional to the number of data items that it needs to process. The regression equation (1) reflects this observation. Thus, if the total number of replicas determined for a subtask is \( n \) (using the replica table), then the data size load of a replica is \( \frac{1}{n} \)th of the data size that was originally processed by the candidate subtask when the (parent) task was detected for low timeliness.

To convert the data size load of a subtask replica into an equivalent processor utilization demand, we again use regression theory. We measure the processor utilization caused by a subtask for processing increasing number of data sizes. From these measurements, we determine a second-order non-linear regression equation that computes utilization demand as a function of the subtask data size.

The regression equation thus obtained for a subtask \( \text{st}_i^j \) of a periodic task \( T_j \) is given by

\[
\text{epu}(\text{st}_i^j, d) = f(d) = a_i \times d^2 + b_i \times d,
\]

where the processor utilization is obtained in percentage, \( d \) is the number of data items, and \( a_i \)'s and \( b_i \)'s are constants that are dependent upon the application subtask. To determine the coefficients \( a_i \)'s and \( b_i \)'s of this regression equation, we use the same iterative approach as described in Section 5.1.

Table 3 shows the two coefficients of the regression equation (3) that were derived for the Filter subtask.

Thus, the problem of determining the processors for executing the subtask replicas becomes the bin-packing problem, i.e., to determine an assignment of replicas to processors that will satisfy the utilization demand of each replica and will not overload any processor. Note that if a processor is able to satisfy the utilization demand of a replica, then it implies that the processor has enough utilization capacity to execute the subtask replica for its current data load.
Once the processor assignment problem is transformed into the bin-packing problem, we can use well known heuristic techniques that solve the bin-packing problem as bin packing is known to be NP-complete. We consider the best fit (BF) and worst fit (WF) heuristic techniques that solve the bin-packing problem to solve the replica-to-processor assignment problem. The heuristic techniques select the processor for each subtask replica by comparing the available utilization of a processor with the utilization demand of the replica. The available utilization of a processor is determined as the difference between 100 and the sum of the utilization demands of all subtasks that are currently assigned to the processor. Also, while considering a set of processors for a subtask replica, we exclude all those processors that have already been assigned another replica of the subtask for exploiting maximum parallelism.

Observe that the BF heuristic assigns a subtask replica to that processor that gives the least value for the difference between the available utilization of the processor and the utilization demand of the replica. On the other hand, the WF heuristic assigns a subtask replica to that processor that gives the largest value for the difference between the available processor utilization and the utilization demand of the replica.

7. EXPERIMENTAL EVALUATION

We evaluated the performance of the predictive and heuristic resource allocation algorithms through a set of experiments. All four algorithms were implemented in the resource manager component of the middleware. We used the real-time benchmark [WS99] as the example application. The application hardware of the middleware and the benchmark consisted of a network of six Pentium host machines running Redhat Linux and were interconnected using a 100 Mbps Ethernet segment.

Each experiment used a scenario where the data size processed by the Sensing task per task period was increased by 20%, 30%, 40%, 55%, 80% and 105%. As the data stream size increases, the task latency increases and eventually exceeds the deadline. This triggers a resource allocation action from the resource manager. The resource manager identifies the candidate subtask (or subtasks) for replication and determines the number of replicas and their processors using the resource allocation algorithms presented in this paper. The resource allocation decision is then enacted using the program control component of the middleware.

To study and compare the performance of the algorithms, we consider two performance metrics: (1) the ratio of the number of task deadlines missed to the total

<table>
<thead>
<tr>
<th>Subtask</th>
<th>$a_i$</th>
<th>$b_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>$10^{-6}$</td>
<td>$1.5 \times 10^{-3}$</td>
</tr>
</tbody>
</table>
number of task deadlines that need to be satisfied during an experiment, i.e., the missed deadline ratio (MDR) and (2) the aggregate difference between the task latency and the task deadline after a resource allocation is performed during an experiment, i.e., the aggregate steady state latency slack (SSL). The metrics were measured for each algorithm during the experiment.

Figures 6 and 7 show the missed deadline ratios and aggregate steady-state latency slack of the four resource allocation algorithms, respectively. Figures 8 and 9 show the number of replicas used by the algorithms for the different increases in data size and the average of the CPU utilization of all the host machines during the experiments, respectively.

From the figures, we observe that ATF produces the lowest MDR and the highest SSL. MTF produces the second lowest MDR. However, it gives almost the same
SSL as the heuristic BF and WF algorithms. Thus, the predictive algorithms clearly outperform the heuristic algorithms in terms of MDR and perform better or the same as that of the heuristic algorithms in terms of SSL. Furthermore, ATF performs the best for both the metrics.

The good performance of the predictive algorithms is directly attributable to the number of replicas that are predicted and used by the algorithms. From Fig. 8, we observe that ATF and MLF generally predict more subtask replicas than the heuristic algorithms for most percentage increase in data sizes. Also, ATF predicts more replicas than the other three algorithms as it replicates all subtasks that are found to be tardy.

Thus, we conclude that the regression equations are effective and are able to accurately determine the number of replicas that is required to adapt the application to a given workload situation and improve task timeliness.
Although ATF is found to be the best by our experimental study, it is worth noting that there is an order of magnitude difference between the computational complexities of ATF and that of MLF. If tasks have as many number of subtasks as there are processors, then the worst-case complexity of ATF is \( O(p^3) \) as all \( p \) subtasks can be found to be tardy. However, this difference in complexity is not apparent from any of the plots that we have presented here, though the computational complexity will affect the “reaction time” of the algorithm, i.e., the total time incurred by the algorithm to compute a resource allocation decision and to react to the situation. Thus, higher the reaction time, higher will be the MDR, as the tasks will continuously miss their deadlines during the time interval that the algorithm is computing a decision. Furthermore, if more subtasks are replicated as ATF does, then it will also affect the reaction time, as the larger number of replicas will incur a larger amount of time to synchronize with the remainder of the application before they can share the workload. Also, the larger number of replicas used will also result in larger aggregate CPU utilization.

The MDR and CPU utilization plots that we show in Figs. 6 and 9, respectively, do not illustrate the “side effects” of ATF due to the “small” baseline parameters of our experimental set up. We believe that the complexity factor will offset the advantage of using higher number of replicas as done by ATF as the number of tasks, the number of their subtasks, and the number of host machines used becomes larger.\(^7\)

8. CONCLUSIONS

In this paper, we present adaptive resource management middleware techniques for periodic tasks in dynamic real-time distributed systems. The techniques continuously monitor the application at run-time for adherence to the desired real-time requirements, detect timing failures or trends for impending timing failures caused by workload fluctuations, and dynamically allocate resources by replicating subtasks of application tasks for load sharing. The objective of the techniques is to minimize (end-to-end) missed deadline ratios of the tasks. We present “predictive” resource allocation algorithms that determine the number of subtask replicas that are required for adapting the application to a given workload situation using statistical regression theory. The algorithms use regression equations that forecast subtask timeliness as a function of external load parameters such as number of sensor reports and internal resource load parameters such as CPU utilization.

To evaluate the performance of the predictive algorithms, we consider algorithms that determine the number of subtask replicas using empirically determined heuristic functions. The functions compute the number of replicas as a quantity that is proportional to the percentage change in the application workload. We implemented the resource management algorithms as part of a middleware infrastructure and

\(^7\)It is practically difficult to show this by an experimental study as we have done here, due to the difficulties involved in experimenting with prototype applications such as [WS99] on large-scale networks in a non-commercial set up.
measured the performance of the algorithms using a real-time benchmark. The experimental results indicate that the predictive algorithms outperform the heuristic strategies under the workload conditions that were studied.

Thus, the contribution of the paper is predictive algorithms that perform adaptive resource allocation in dynamic real-time distributed systems using statistical regression theory. Furthermore, the effectiveness of the algorithms is validated by implementation and an experimental study.

Several issues in prediction-based resource allocation are the focus of on-going efforts. We are exploring non-regression-based predictive algorithms such as those that are based on stochastic models. We are also developing resource allocation techniques for aperiodic and transient-periodic tasks.

REFERENCES


BINOY RAVINDRAN is an assistant professor in the Bradley Department of Electrical and Computer Engineering at Virginia Tech. He received his bachelor’s degree in mechanical engineering from the University of Kerala, India, in 1991, master’s degree in computer science from New Jersey Institute of Technology (NJIT) in 1994, and the Ph.D. in computer science and engineering from the University of Texas at Arlington (UT Arlington) in 1998. He received two Student Achievement Awards from NJIT for outstanding research by a graduate student (1994, 1995) and a Doctoral Dissertation Award from UT Arlington for outstanding doctoral research (1998). His current principal research focus is on asynchronous real-time distributed systems having custom, application-level, end-to-end quality of services (e.g., timeliness, survivability). During 1999, Dr. Ravindran was an invited speaker at INRIA, France, and at the ARTES (Swedish National Strategic Research Initiative in Real-Time Systems) Real-Time Week, Sweden. He also served as the program chair of the 1999 IEEE International Workshop on Parallel and Distributed Real-Time Systems. Currently, Dr. Ravindran is a guest editor for IEEE Transactions on Computers for a 2002 special issue on asynchronous real-time distributed systems.

RAVI DEVARASETTY received his bachelor of engineering in electronics and communication engineering in 1999 from Vasavi College of Engineering, Osmania University, India, and his master of science in computer engineering from Virginia Tech in 2001.

BEHROOZ A. SHIRAZI is a professor in and chair of the Computer Science and Engineering Department at UTA. Before joining UTA in 1990 he was on the faculty of Computer Science and Engineering at Southern Methodist University. Dr. Shirazi has conducted research in the areas of software tools, distributed real-time systems, scheduling and load balancing, and parallel and distributed systems over the past 15 years. He has advised 12 Ph.D., 56 M.S., and 40 B.S. students and has published over 100 papers in these areas. Dr. Shirazi’s research has been sponsored by grants from NSF, DARPA, AFOSR, Texas Instruments, E-Systems, Mercury Computer Systems, and Texas ATP for a total of more than $2.5M. He has been a guest editor of a special issue of the Journal of Parallel and Distributed Computing and a track coordinator of the HICSS’93 Conference, both on scheduling and load balancing issues. He is the leading author of an IEEE Press book entitled Scheduling and Load Balancing in Parallel and Distributed Systems. Dr. Shirazi is currently on the editorial boards of the IEEE Transactions on Computers and Journal of Parallel and Distributed Computing. He is the principal founder of the IEEE Symposium on Parallel and Distributed Processing and has served on the program committee of many international conferences. He has received numerous teaching and research awards and has served as an IEEE Distinguished Visitor (1993–1996) as well as an ACM Lecturer (1993–1997).