A Scalable Scheduling Algorithm for Real-Time Distributed Systems

Yacine Atif
School of Electrical & Electronic Engineering
Nanyang Technological University
Singapore
E-mail: iayacine@ntu.edu.sg

Babak Hamidzadeh
Department of Electrical & Computer Engineering
University of British Columbia
Canada
E-mail: babak@ee.ubc.ca

Abstract

Scheduling real-time tasks in a distributed-memory multiprocessor is characterized as sequencing a set of tasks and assigning them to processors of the architecture. Real-time systems research has extensively investigated the sequencing dimension of the scheduling problem by extending uni-processor scheduling techniques to more complex architectures. We introduce a technique that uses an assignment-oriented representation to dynamically schedule real-time tasks on the processors of the system. The technique we propose, automatically controls and allocates the scheduling time, in order to minimize deadline violation of real-time tasks, due to the scheduling overhead. We evaluate this technique in the context of scheduling real-time transactions in a distributed database application which we implemented on an Intel Paragon distributed-memory multiprocessor. In this implementation, we compared the performance of our algorithm with another dynamic algorithm that uses a sequence-oriented representation. The results show interesting performance trade-offs among the candidate algorithms and validate our conjectures about scalability performance limitations of sequence-oriented representations. The results also show the effect of the mechanisms that our technique uses to control and allocate scheduling time.

1. Introduction

Real-time scheduling research has proposed algorithms to guarantee or to optimize deadline compliance. A total deadline compliance guarantee is a typical characteristic of hard real-time systems where missing a deadline can lead to catastrophic consequences. Deadline compliance optimization is the performance criterion of soft real-time systems. This paper falls into the class of soft real-time systems. Though the algorithms we propose in this paper strive to increase deadline compliance, we do however provide a correction theorem that proves the deadline guarantee of the produced solutions. Based on the time at which scheduling is performed, real-time scheduling algorithms have been divided into two categories, namely static and dynamic. In this paper, our focus is on dynamic algorithms. These algorithms produce schedules on line in hope of using more comprehensive and up-to-date information about the tasks and the environment. Due to the on-line nature of their problem solving, dynamic algorithms must be efficient, since their complexity directly affects the overall system performance[2].

Scheduling real-time tasks on a multiprocessor architecture is characterized as sequencing a set of tasks and assigning them to processors of the system such that each task's time constraints are satisfied. Sequencing and assignment of real-time and non real-time tasks on multiprocessor architectures has been formulated in the past as a search for a schedule in a search space. Many of the existing techniques for scheduling ordinary tasks on a multiprocessor architecture adopt an *assignment-oriented* search representation [5][4]. Techniques for scheduling real-time tasks on uniprocessor and multiprocessor architectures have generally adopted a *sequence-oriented* search representation [3][6]. An assignment-oriented representation emphasizes the choice of processors to which tasks are assigned, in order to satisfy the problem objectives. A sequence-oriented representation emphasizes the order in which tasks should be scheduled, in order to satisfy the problem objectives. Such representation is the only option when considering a uni-processor architecture. It has been extended to a shared-memory multiprocessor architecture with the valid argument that, since deadline compliance is the main objective, tasks selection for scheduling is more emphasized considering the tasks characteristics such as deadline, than processors selection for assignment. However, when considering a distributed-memory architecture which is set to expand in size, we found that these representations i.e. sequence-oriented representations, do not scale-up well their deadline compliance performance. Scalability has been largely addressed for the objective of minimizing the total execution time of ordinary tasks in a parallel architecture. But deadline compliance scalability has rarely been addressed explicitly. In the context of this paper, we define scalability as the ability of the scheduling algorithm to increase its deadline compliance to the high-end as more processing units are added to the parallel architecture.

A main contribution of the paper is to evaluate the trade-offs between dynamic techniques that use an assignment-oriented representation and those which use a sequence-oriented representation in the context of deadline compliance scalability. Another contribution is to address the problem of deciding when to invoke the scheduling process and what the magnitude of the scheduling quantum should be to minimize scheduling overhead and to guarantee deadlines of scheduled and arriving tasks.

The remainder of this paper is organized as follows. Section 2 specifies the problem to be addressed. Section 3 discusses different scheduling models. As part of this section, we discuss different search representations and demonstrate some of their characteristics and limitations. Section 4 introduces our scheduling technique and Section 5 provides an experimental evaluation of that technique's performance. Section 6 concludes the paper with a summary of the results.

2. Problem Specification

In this paper, we address the problem of scheduling a set T of n aperiodic, non-preemptable, independent, real-time tasks T_i with deadlines, on the m processors P_j of a distributed-memory multiprocessor architecture. Aperiodic tasks are tasks with arbitrary arrival times whose characteristics are not known a priori. In a distributed-memory architecture, each processor has its own local memory. Since data objects can be distributed among local memories of a distributed architecture, and since each task T_i references a subset of the data objects, T_i will have an *affinity* for the processors P_j that hold T_i's referenced data objects in their local memory. A parameter in a distributed architecture is the *degree of affinity* among tasks and processors. This is an indicator of the degree of data replication and can be defined as the probability that a task T_i has affinity with a processor P_j. A high degree of affinity means that many data objects are replicated on several processors and, thus, any given task has affinity with several processors. A low degree of affinity, on the other hand, means that data objects are not replicated and any task has affinity with only few processors. Each task T_i in T assigned to execute on processor P_j is characterized by a processing time p_i, a communication cost c_{ij}, arrival time a_i, and a deadline d_i. c_{ij} is a notion of affinity between task T_i and processor P_j. In our model, c_{ij} is zero if T_i has affinity with P_j (i.e. T_i's referenced data reside on P_j's local memory) and it is some constant C if T_i does not have affinity with P_j. A non-zero c_{ij} implies that a communication cost is incurred if T_i is scheduled to execute on P_j. In distributed architectures that use cut-through routing protocols (e.g. wormhole), inter-processors communication costs are independent of the distance between the source and the destination, and can be accounted for by a constant like C in our model.

We define a task-to-processor assignment (T_i P_j) to be *feasible*, if T_i finishes its execution on a processor P_j before its deadline. The communications delays c_{ij} give this cost model a non-uniformity in feasibility of executing a task on different processors. Namely, a task that is feasible on one processor may not be feasible on another processor, even if this task is the only one assigned to either processor for execution. Accounting for the scheduling cost in defining feasibility is important. By explicitly controlling and allocating the scheduling time, our proposed algorithms can determine the time t_e at which a scheduling phase and account for it in the cost model. If a schedule includes all the tasks in the batch, then it is considered to be *complete*. Otherwise, it is referred to as a *partial* schedule. The objectives of our scheduling algorithms is to increase deadline compliance as the number of processors increases.

3. Scheduling Models

Scheduling can be represented as the problem of incrementally searching for a feasible schedule in a graph representation G(V,E) of the task space. G in our representation of the problem is in the form of a tree. The vertices $v_i \in V$ in G represent task-to-processor assignments *(T_i P_j)*. A partial path *(v_i, v_j,..., v_k)* from the root v_i to a vertex v_k in G represents a partial schedule that includes all task assignments represented by v_i, v_j,..., v_k. The edges $(v_i,v_j) \in E$ represent *extending* the partial schedule by one task-to-processor assignment. By extending a schedule (or path), one task assignment at a time, a feasible schedule is incrementally built. Thus, schedule construction proceeds from one level in G to the next level. Complete schedules, if they exist, will be at the leaves of G. The incremental schedule construction allows a dynamic algorithm (as those presented in later sections) to produce a partial schedule which is feasible at any point during scheduling. This allows the algorithm to be interrupted at the end of any iteration and still produce feasible schedules.

To choose a new task to add to the schedule and to assign that task to one of the processors, we need to evaluate and compare a set of candidate vertices of G with one another. The candidate vertices are stored in a candidate list CL. A constraint on the candidate vertices in CL is that they must represent feasible task-to-processor assignments. Thus, feasibility tests are applied to different vertices in G to identify the valid task assignments that can be added to the partial schedule. Another mechanism for evaluating vertices may be a cost function that we try to maximize or minimize to achieve a set of objectives. Each vertex will be assigned a value using this function. Yet another evaluation may consist of computation of a heuristic function associated with each vertex to indicate priority or degree of promise of that vertex. We shall investigate the effect of cost functions and heuristics in later sections. In the remainder of this section, we assume feasibility tests to be necessary components of a real-time scheduling algorithm. We discuss the interaction of such tests with different search representations in the context of dynamic scheduling.

Upon encountering an infeasible vertex, the search will continue by examining other vertices for inclusion in the schedule. Consideration of other vertices may involve *backtracking* to other branches in G. We define backtracking as a situation in which the search cannot extend the current partial path (or schedule) further and switches to examine another partial path for extension. The nature of backtracking depends on the search strategy and the search representation used. Backtracking may result from consideration of optimizing factors, as well as from consideration of vertex feasibility. We define a *dead-end* as a situation in which no other feasible candidates remain to be considered for extending a schedule (i.e. when CL is empty). A dead-end signifies that extending any partial path (i.e. partial schedule) in G to include another task assignment will result in an infeasible schedule.

The topology and semantics of G depend on the search representation used. In this paper, we consider two common search representations, namely sequence-oriented and assignment-oriented representations. In a *sequence-oriented* representation (See Figure 1), at each level of G a processor P_i is selected and the search decides which task T_i to assign to P_i. At the next level, another processor (e.g. P_{i+1}) is selected and the remaining tasks are considered for assignment to that processor. Processors selection at each level is shown in this figure to be performed in a round-robbin order. However, a heuristic function can be applied to affect this order as we shall see in later sections. This representation is a direct extension of a uni-processor scheduling model [1] where the goal is to find a feasible sequence of execution[1]. When extended to a multiprocessor architecture [6], the search problem for a feasible schedule is divided into consecutive sub-problems that deal with one processor at a time. In an *assignment-oriented* representation (See Figure 2), at each level of G a task T_i is selected and the search decides to which processor P_j task T_i should be assigned. At the following level, another task (e.g. T_{i+1}) is selected and assigned to one of the processors and so on. Note that at each level, all processors are considered for assignment again, even if they already have tasks assigned to them. In both representations, the search proceeds in a depth-first strategy, considering and reconsidering the way tasks are sequenced and assigned. Each representation, however, emphasizes one of sequencing or assignment more than the other as the names imply. A major difference between the two representations is in the way they affect backtracking. In a sequence-oriented representation, upon encountering an infeasible vertex representing task assignment *(T_i P_j)*, backtracking may consider assigning a different task to P_j. In an assignment-oriented representation, on the other hand, backtracking will consider assigning T_i to a different processor. If at level three of the task space, for example, the search has to backtrack to examine other partial paths in a sequence-oriented representation (See Figure 1), it can only backtrack to undo tasks on P_1, P_2, or P_3. In an assignment-oriented representation, backtracking can undo or resequence tasks on all processors.

In a sequence-oriented representation, upon encountering an infeasible vertex representing task assignment *(T_i P_j)*, backtracking may consider assigning a different task to P_j. In an assignment-oriented representation, on the other hand, backtracking will consider assigning T_i to a different processor. If at level three of the task space, for example, the search has to backtrack to examine other partial paths in a sequence-oriented representation (See Figure 1), it can only backtrack to undo tasks on P_1, P_2, or P_3. In an assignment-oriented representation, backtracking can undo or resequence tasks on all processors.

In a dynamic environment, the effect of search representation on performance is more prominent. Dynamic algorithms are forced to use heuristics such as limited backtracking [3][6], limit on the depth of search, and/or an upper bound on the time of search, that prune the search space significantly, in order to reduce the scheduling complexity. Pruning a sequence-oriented search space in breadth or depth causes the probability of reaching a dead-end to rise. As we have seen in the previous examples, this representation does not have the capability to exploit all resources in the system well and in a greedy fashion. By further reducing the search's options to explore during

1. The task model for such a single processor architecture, includes both deadline and start-time constraints that make the scheduling problem NP-Complete and thus it justifies the search approach.

dynamic scheduling, it is more likely that the search will terminate (e.g. due to reaching a dead-end or due to reaching a bound on scheduling complexity) at a shallow depth in the sequence-oriented search representation. Due to the structure of a sequence-oriented representation, this will result in assignment of tasks only to a fraction of the processors in the system during each scheduling phase. Thus, it may so happen that many processors remain idle while others are heavily loaded. This will contribute further to a sequence-oriented representation's incapability to scale up its performance as the number of processors increases in the system. Our experiments in later sections will validate these conjectures.

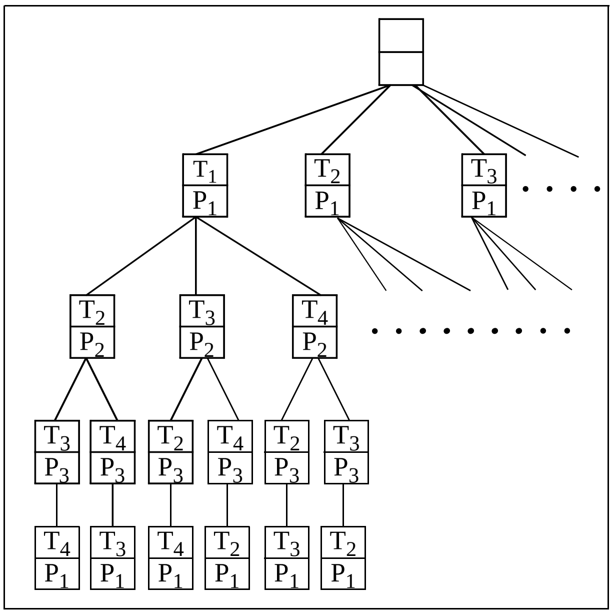

Figure 1. Sequence-oriented scheduling model

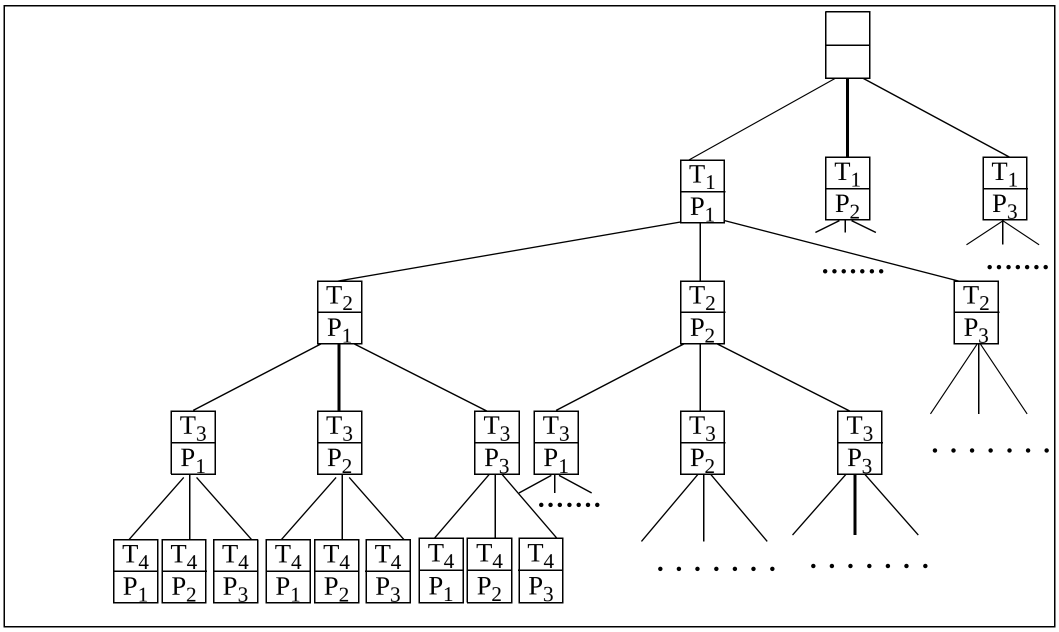

Figure 2. Assignment-oriented scheduling model

4. Real-Time Self-Adjusting Dynamic Scheduling (RT-SADS)

RT-SADS performs a search for a feasible schedule in an assignment-oriented search representation. It uses a dedicated processor to perform scheduling phases concurrently with execution of real-time tasks on other processors. The input to each scheduling phase *j* is a set of tasks (i.e. *Batch(j)*). Initially, *Batch(0)* consists of a set of the arrived tasks. At the end of each scheduling phase *j*, *Batch(j+1)* is formed by removing, from *Batch(j)*, the scheduled tasks and tasks whose deadlines are missed, and by adding the set of tasks that arrived during scheduling phase *j*. Note that tasks in *Batch(j)* will not enter *Batch(j+1)* or later batches, if they are scheduled in phase *j* (i.e. they will not be re-considered for scheduling later). Instead, they form a schedule S_j which will be delivered to the ready queues of the working processors. Thus, the tasks in S_j are executed by the working processors while scheduling of S_{j+1} is in progress on the scheduling processor. A specification of RT-SADS in pseudo-code format is included in an Appendix at the end of the paper.

4.1. Scheduling Phase

In the following discussion, a vertex in task-space G is defined to be *generated*, when enough memory for a data structure representing that vertex is allocated and when the vertex and its attributes are evaluated and stored in the data structure. An evaluation of a generated vertex v_i consists of computing the task-to-processor assignment that v_i represents, computing the partial schedule that the path leading to v_i represents, and performing a feasibility test to see whether that schedule is feasible. Other evaluations may consist of computation of a cost function or a heuristic function associated with that vertex which indicates the value or priority of that vertex. A vertex is defined to be *expanded* when its successor vertices are generated. The successors of a vertex v_i are the set of all the k vertices $\{v_1, \ldots, v_k\}$ which are connected to v_i via direct links $(v_i, v_1), \ldots (v_i, v_k)$.

Scheduling phase j starts at the root (i.e. empty schedule) of G which represents the current vertex CV, and the current partial schedule CPS. In one iteration of a scheduling phase j, CV is expanded and its feasible successors are kept in a list. If a heuristic and/or a cost function exists, the list of feasible successors is sorted according to heuristic/cost values with the highest-priority vertex in front of the list. The vertices of this list are then added to the front of the list of candidate vertices, CL. The vertices in CL are prospects for extending CPS to include more task assignments while remaining feasible. In the following iterations, the first vertex in CL is removed from the list and is expanded as the new CV. CPS is now the path from the root to this vertex.

It may so happen that none of the successors of an expanded vertex pass the feasibility test. In such a situation no new vertices are added to CL and the search will backtrack to explore other candidates by expanding the first vertex in CL which, in this case, is not an extension (successor) of the current partial path and comes from another path in G. The iterations of a scheduling phase continue until either a leaf vertex is reached, until a dead-end is reached (i.e CL becomes empty), or when the limit $Q_s(j)$ on the duration of scheduling phase j is reached.

The result of each scheduling phase j is a feasible partial or complete schedule S_j. During the execution phase j, the tasks in S_j are executed by their assigned working processors. The tasks which were not scheduled during scheduling phase j are merged with the newly arrived tasks to form *Batch*($j+1$). Tasks in *Batch(j+1)* are then tested and those tasks whose deadlines have already been missed (i.e. $\{T_i | (T_i \in Batch(j+1)) \wedge (p_i + t_c > d_i)\}$) are removed from *Batch*($j+1$). Next, we discuss RT-SADS's mechanisms for allocating the time duration of scheduling phases.

4.2. Allocation of Scheduling Time

RT-SADS uses a novel on-line parameter tuning and prediction technique to determine the time and duration of a scheduling phase. The algorithm continually self-adjusts the allocated time $Q_s(j)$ of scheduling phase j using a set of criteria based on parameters such as slack[1], arrival rate and processor load. The motivation for designing such criteria is to allocate longer times to scheduling when the task slacks are large or when arrival rates are low, or when the working processors are busy executing tasks. Note that longer scheduling times allow RT-SADS to optimize longer and find higher-quality schedules. When the slacks are small or the arrival rates are high, or when working processors become idle, scheduling durations are shortened to honor the deadline of scheduled tasks, to account for arriving tasks more frequently, and to reduce processor idle times. The allocated time of scheduling phase j is controlled by a criterion as shown in Figure 3.

$$Q_s(j) \le Max[Min\_Slack, Min\_Load]$$

where:

$$Min\_Slack = Min[Slack_l | T_l \in Batch(j)]$$

$$Min\_Load = Min[Load_k | p_k \in \{working\ processors\}]$$

Figure 3. Criterion for allocation of scheduling time.

The term *Min_Slack* in Figure 3 is included in the expression to limit the amount of time allocated to scheduling phase j so that none of the deadlines of tasks in the current batch are violated due to scheduling cost. This term, however, ignores the waiting time before a processor becomes available. If the minimum waiting time among working processors (i.e. *Min_Load*) is larger than the slack of a task, then the deadline of this task is missed anyway, even if it is scheduled and executed right away. In this case, the allocated scheduling time is extended to *Min_Load* providing the scheduling process with more time to improve the schedule quality. If, on the other hand, *Min_Slack* is larger than *Min_Load*, the allocated scheduling time is set to *Min_Slack* to increase the scheduling time at the expense of some processor idle time.

1. The slack time is the maximum time during which the execution of a task can be delayed without missing its deadline.

4.3. Prediction of Time-Constraint Violation

RT-SADS predicts deadline violation of tasks based on a feasibility test that takes into account the scheduling time of a phase, as well as the current time, deadline, and processing time of tasks. Accounting for the scheduling time in the feasibility test ensures that no task will miss its deadline due to scheduling time. The test for adding a task assignment $(T_l\ P_k)$ to the current feasible partial schedule *CPS* to obtain feasible partial schedule *CPS'* in scheduling phase *j i*s performed as shown in Figure 4.

```
IF $(t_c + RQ_s(j) + se_{lk} \leq d_l)$
THEN CPS' is feasible
ELSE CPS' is infeasible
```

Figure 4. Feasibility test of RT-SADS

In the above feasibility test, t_c indicates the current time, $RQ_s(j)$ indicates the remaining time of scheduling *i.e.* $RQ_s = Q_s - (t_c - t_s)$, *where* Q_s is the allocated scheduling time to phase *j*, and t_s is the scheduling start-time. Finally, se_{lk} is the scheduled end time for task T_l on processor P_k.

Next, we provide a theorem that guarantees to minimize to 0 the number of scheduled tasks whose deadlines are missed during their execution.

Theorem: The tasks scheduled by RT-SADS are guaranteed to meet their deadlines, once executed.

Proof: The proof is done by contradiction. Let us assume that a task $T_l \in Batch(j)$ is scheduled on processor P_k during the *jth* phase and that T_l misses its deadline at execution time. This assumption leads to the following condition:

$$t_e(j) + se_{lk} > d_l \ \textbf{(1).}$$

Here we are assuming that the execution of tasks in a schedule will start immediately after scheduling ends. On the other hand, RT-SADS's bound on scheduling-time allocated to each phase ensures that:

$$t_e(j) \leq t_c + RQ_s(j) \ \textbf{(2).}$$

Combining **(1)** and **(2)** leads to:

$$t_c + RQ_s(j) + se_{lk} > d_l \ \textbf{(3).}$$

The feasibility test performed at time t_c ensures that: $t_c + RQ_s(j) + se_{lk} \leq d_l$, contradicting inequality **(3)**. Therefore, our assumption regarding deadline violation of T_l is false, which concludes the proof of the theorem.

4.4. Load-Balancing

A cost function that calculates the total execution time of a given partial schedule is used in a version of RT-SADS to compare different partial schedules. An objective can then be to search for a schedule that minimizes this function. Such a schedule is the one that offers the most evenly balanced load among all processors. As part of calculating the total execution time of a schedule, the function must be able to determine the cost of executing tasks T_l on processors P_k. This cost is $p_l + c_{lk}$, where p_l is the processing time required to execute task T_l and c_{lk} is the communication delay required to transfer any data objects, should T_l execute on P_k. At any point during scheduling phase *j*, the execution time ce_k for each processor P_k when tasks T_l are assigned to it is:

$$ce_k = Load_k(j-1) - Q_s(j) + \sum_{(T_l\ P_k)} (p_l + c_{lk}),$$

where $Load_k(j\text{-}1)$ is the execution cost of processor P_k at the end of scheduling phase $(j\text{-}1)$. The total execution time for a schedule is determined by the processor that finishes executing its tasks last. So, the execution time CE_i of the partial schedule represented by a path from the root in G to a vertex v_i, is the maximum of the ce_k values for all working processors. i.e.,

$$CE_i = Max[ce_k], \text{where} (1 \leq k \leq m-1).$$

One of the important features of RT-SADS's cost model is that it accounts for non-uniformity in the tasks' communication delays when scheduled on specific processors. This model simultaneously addresses the tradeoffs between the two interacting factors of processor load balancing and inter-processors communication costs.

5. Application and Evaluation

In this section, we evaluate the deadline compliance scalability of RT-SADS algorithm in the context of a distributed real-time database application. In this application, an entire relational database is partitioned equally among processors' private memories of an Intel Paragon multiprocessors and transactions are executed concurrently. To simplify this study, we assume read-only transactions. Our simple database and transaction model highlights the applicability of our algorithms and validates their performance on a distributed-memory multiprocessor architecture. The performance metric which we stress on in this study is deadline compliance scalability. That is, the system can be continuously expanded in complying with transactions' deadlines up to the "high-end" by adding more hardware modules.

Assuming the global database holds *r* tuples, we divide it into *d* sub-databases. Such division can be performed through a hashing function in order to speed-up the location of a tuple with respect to the sub-databases. When mapped on a distributed-memory architecture, each sub-database resides in the local memory of a specific processor. Depending on the replication rate, multiple copies of a sub-database may reside in several local memories. A transaction is characterized by the attributes' values that transaction aims to locate in the distributed database. In order to execute transactions by their deadlines, a host processor is dedicated to perform a continuous scheduling process while the remaining processing units perform the actual execution of transactions. Executing a transaction would mean iterating a checking process among the tuples which partially match the attributes' values of the transaction. It is important for the scheduling process to evaluate the execution cost of a transaction beforehand in order to perform accurate scheduling decision. In order to reflect a transaction processing cost, we adopted the following estimation function:

Execution_Cost(q) = *k* x [IF $key \in F$ THEN
Frequency_of_matching_key_values
ELSE *r/d*]

where *F* is the set of attributes for which values are given in a transaction *q* and *k* is the processing time to execute one checking iteration. Hence, to estimate the execution cost of a transaction, the host processor maintains the global index file of the database. If a transaction provides a key value, the index file is used to evaluate the number of tuples a processing node would need to check in the worst-case in order to locate the records requested by that transaction. Therefore, this mechanism returns estimates of the transaction worst-case computation times which can be used by a scheduling algorithm like RT-SADS.

5.1. Experiment Design

The parameters configuration for these experiments are as follows. Each sub-database holds 1000 records and 10 attributes. For simplicity, the attributes' domains are disjoint from each other among the sub-databases. A uniformly distributed item is generated for each attribute-value based on its domain. The sub-databases are indexed according to a specific key attribute say, attribute #1. A transaction contains a uniformly distributed number of given attribute-values. The values are *picked* equiprobably from their respective domains. In our experiments, we generated 1000 transactions. The transactions' deadlines are proportionally generated based on the estimated transaction processing time. Deadline(*q*) = *SFx10x Estimated_Cost(q).* SF values range from 1 to 3. A low value of SF signifies tight deadlines whereas a high value of SF signifies loose deadlines. In the figures, we use the term laxity which is equivalent to the deadline factor.

Another parameter in our experiment is the replication rate of the databases. The replication rate has the same semantic as the task-to-processor affinity. However, quantitatively they are inversely proportional. That is a high degree of affinity is equivalent to a degree of replication rate. We divided the global database into 10 sub-databases. Based on the replication rate R, we copy the sub-databases in the local memory of the processing nodes. R's values ranged from 10% to 100%. In the extreme cases, a 100% replication rate would result in having the global database in the local memory of each processor, whereas a 10% replication rate would result in having each processor holding in its local memory, at most one copy of a sub-database. The number of processors ranged in our experiment from 2 to 10. Finally, we assume a bursty arrival of 1000 transactions which *simultaneously* reach the host node that is executing the scheduling task.

A metric of performance in our experiments is deadline compliance. Deadline compliance measures the percentage of tasks which have completed their execution by their deadline. Another related metric is scalability, which measures the ability to increase deadline compliance as the number of processors increases. We measure the scheduling cost as the physical time required to run the scheduling algorithm.

Each experiment was run 10 times and the mean of the 10 runs was plotted in the figures. Two-tailed difference-of-means tests indicated a confidence interval of 99% at a 0.01 significance level for all the results collected in these experiments.

5.2. Experiment results

We compare RT-SADS algorithm which uses an assignment-oriented representation, with another scheduling algorithm that uses a sequence-oriented representation named Distributed Continuous On-Line Scheduling (D-COLS) [2] which has been shown to outperform other proposed sequence-oriented scheduling models. Many of the features of such a sequence-oriented based algorithm, discussed in our experiments, were inspired by the techniques reported in [6][3]. The search for a schedule in D-COLS algorithm is performed in a sequence-oriented task-space G similar to that shown in Figure 1. In order to emphasize the effect of problem representation on performance, we allocate to both algorithms the same time quantum based on the formula given in Figure 3, and we observe their deadline hit ratios.

Generally, the results confirm our previous expectations regarding the deadline scalability issue. Figure 5 shows the scalability aspects in terms of meeting deadlines as the number of processors increases. In this experiment R was fixed to a small value (i.e. 30%). In all parameters configuration, RT-SADS outperforms the sequence-oriented based algorithm D-COLS, by as much as 60% as the number of processors increases. Figure 5 shows that under very tight deadlines, a sequence-oriented algorithm does not scale up, whereas RT-SADS continues to increase its deadline compliance as more processors are provided. A small value of replication rate increases transaction-to-processor affinity and hence many transactions will have to be scheduled on the processors that hold the needed sub-database in their private memory to avoid communication latencies that may delay transactions processing beyond their deadline. In such situation, a sequence-oriented algorithm like D-COLS would frequently face dead-end paths as it spends most of the allocated scheduling time alternating transactions consideration for scheduling, whereas an assignment-oriented algorithm like RT-SADS that alternates processors consideration for assignment would quickly break such a dead-end since it considers all processors as potential candidates.

Figures 6 shows the performance of the algorithms in terms of databases replication rates. As the replication rate increases, the inter-processor communication frequency decreases. D-COLS increases deadline hit ratios as the replication rates increases. This is due to the fact that, with multiple copies of sub-databases being duplicated among processors' local memories, processors selection is not important. Nevertheless, RT-SADS still maintains large performance difference over D-COLS. This is due to load-balancing capabilities of RT-SADS which complexity decreases when less inter-processor communication requirements are involved.

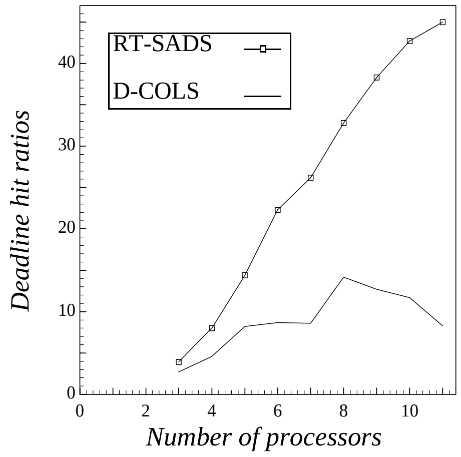

Figure 5. Deadline scalability performance (R=30% and SF=1)

6. Conclusion

We have introduced a technique that uses an assignment-oriented representation to dynamically schedule tasks on the processors of the system. The proposed technique automatically controls and allocates the scheduling time to minimize the scheduling overhead and to minimize deadline violation of real-time tasks, due to the scheduling overhead. We compared the performance of this technique with the performance of another dynamic technique that uses a sequence-oriented representation in a number of experiments that implement a distributed database application on an Intel Paragon multiprocessor. The results of the experiments show that as the number of processors increases, the assignment-oriented technique outperforms the sequence-oriented technique significantly.

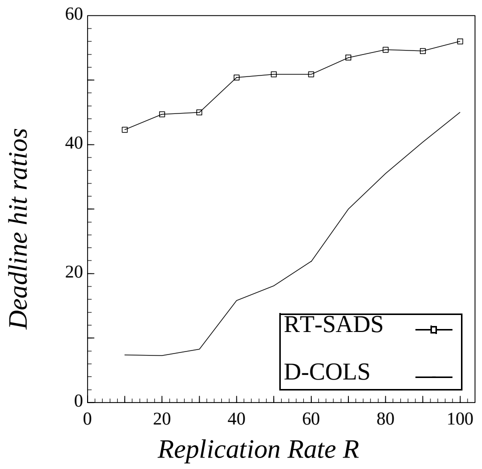

Figure 6. Deadline compliance under varying degrees of replication rate (P=10 and SF=1)

References

1. Y. Atif & B. Hamidzadeh, "On-line Optimization Techniques for Dynamic Scheduling of Real-Time Tasks", Proc. of the Second International Conference on industrial and Engineering Applications of Artificial Intelligence and Expert Systems. IEA/AIE, June 1996.
2. B. Hamidzadeh & Y. Atif, "Dynamic Scheduling of Real-Time Aperiodic Tasks on Multiprocessor Systems", Proceedings of the Twenty-Ninth Tasks on Multiprocessor Systems", Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences, p469-78 vol.1, Jan. 1996.
3. C. Shen, K. Ramamritham and J.A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, April 1993.
4. C.C. Shen and W.H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion", IEEE Transactions on Computers, vol. c-43, no.3, pp. 197-203, March 1985.
5. C. Shen, K. Ramamritham and J.A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems", IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, April 1993.
6. W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints", Journal of Systems and and Software,1987.