Lecture Notes in Operations Research 3

Edited by
Ding – Zhu DU
Xiang – Sun ZHANG
Kan CHENG

OPERATIONS RESEARCH AND ITS APPLICATIONS

Third International Symposium, ISORA'98
Kunming, P. R. China, August 20 – 22, 1998
Proceedings

WORLD PUBLISHING CORPORATION

图书在版编目(CIP)数据

第三届国际运筹学及其应用会议文集：英文/堵丁柱等主编．－北京：世界图书出版公司北京公司，1998.8
ISBN 7-5062-3874-8

Ⅰ．第… Ⅱ．堵… Ⅲ．运筹学-国际学术会议-文集-英文 Ⅳ．022－53

中国版本图书馆 CIP 数据核字(98)第 16224 号

第三届国际运筹学及其应用会议文集

堵丁柱　章祥荪　程　侃　主编
责任编辑　高　蓉
世界图书出版公司北京公司出版
北京朝阳门内大街 137 号
北京百善印刷厂印刷
新华书店北京发行所发行　各地新华书店经售

*

1998 年 8 月第　1　版　　开本：711×1245　1/24
1998 年 8 月第 1 次印刷　　印张：32.75
ISBN：7-5062-3874-8/O・238
定价：180 元

Preface

The extended abstracts and papers in this volume were presented at the Third International Syposium on Operations Research and its Applications (ISORA'98), on August 20 - 22, 1998, in Kunming, China. The conference was sponsored by the Asian-Pacific Operations Research Center in cooperation with Institute of Applied Mathematics, Chinese Academy of Sciences and Operations Research Society of China.

Fifty papers in this volume cover quite a few aspects of operations research methods and its various applicatins including scheduling problem, parallel and distributed computing, combinatorial optimization, decision and management, simulation method, econimic and financial problems, mathematical programming, queueing theory, fuzzy method. The authors of these papers come from the following countries and regions: Australia, Canada, China (including Hong Kong and Taiwan), Egypt, France, Germany, Greece, Japan, Korea, United Kingdom and United States of America.

In the end, we wish to thank all who have made this conference possible:

- The authors for submitting papers.
- The program committee members consisting of Y. Gang (co-chair, University of Texas at Austin, USA), S. J. Park, (Korea Advanced Institute of Science & Technology, Korea), P. M. Pardalos (co-chair, University of Florida, USA), G. H. Young (The Chinese University of Hong Kong, Hong Kong), J.-Z. Zhang (City University of Hong Kong, Hong Kong). Our special thank goes to Prof. G. H. Young who spent lots of time on organizing two special sessions for ISORA'98 and making home page for ISORA'98.
- The organizing committee members consisting of Yu-Di Cai, Lin Cheng, Yaji Guan, Xiao-Dong Hu, Jie Hu, Gang Li, Yong Li, Lei Wang and Jin-Rong Wu (all from Institute of Applied Mathematics, Chinese Academy of Sciences, China).
- The National Natural Science Foundation of China for their financial support.
- The Beijing World Publishing Cooperation for publishing this volume, and Ms. Rong Gao and Ms. Wei Wang for their editoral work.
- The Palace Hotel in Kunming, China for providing lecture rooms for the conference.

August 1998

Xiang-Sun Zhang (Chair of the Symposium)
Ding-Zhu Du (Chair of Program Committee)
Kan Cheng (Chair of Organizing Committee)

Contents

此为试读，需要完整PDF请访问：www.ertongbook.com

* Authors delivered their speeches on ISORA'98.

Memory-restricted parallel multi-join evaluation (Extended abstract)

Richard Wong Hong Shen Rodney Topor

School of Computing and Information Technology
Griffith University
Nathan, Qld 4111, Australia
{R.Wong,H.Shen,R.Topor}@cit.gu.edu.au

1 Introduction

There is a long history of research on efficient evaluation of join operations in relational database systems in both sequential and parallel environments. Recently researchers have extended this work to efficient parallel evaluation of multi-join expressions $R_1 \bowtie \ldots \bowtie R_k$ of relations $R_1, \ldots, R_k$ [1, 2, 3, 4]. In particular, they have studied how such expressions can be evaluated in the minimum total time on different parallel machine models.

The existing algorithms have used either intra-operation parallelism, inter-operation parallelism, or a combination of both. In intra-operation parallelism, each join operation is executed using all available processors in parallel. Successive joins are executed sequentially. In inter-operation parallelism, one or more of the available processors are allocated to each join operation, and different join operations are evaluated in parallel. A special case of inter-operation parallelism is pipeline parallelism, in which the output of each join operation is sent directly as input to another join operation, without being stored on disk first.

An experimental study demonstrated that intra-operation parallelism can achieve a linear speedup for a small number of processors but this cannot be extrapolated to a large number of processors [5].

Research by Krishnamurty [6] remarked that the restriction to linear trees may not be a good choice for parallel system. However, the space of possible join trees is very large if restriction to linear tree is dropped [7]. In this paper, we analyse and compare the performance of two extreme forms of pipeline parallelism, assuming a limited amount of main memory available on a distributed shared memory computer. We show how to construct an intermediate form of pipeline parallelism that fully exploits the available main memory to achieve better performance than either extreme form alone.

The remainder of the paper is organized as follows. Section 2 briefly describes the pipelining parallelism on parallel computer. Section 3 describes the cost analysis and the detail algorithms. Section 4 presents the comparisons of the algorithm, and Section 5 summarises the results and suggests future work.

2 Pipelining parallelism

In [8, 9] it is shown how special single-join main-memory algorithms can be used to enhance the effective parallelism from pipelining. These single-join pipelining algorithms aim at producing output as early as possible, so that a consumer of the result can start its operation. In particular, [8, 10] proposes a pipelining hash-join algorithm for joining two sets. Compared to the simple hash-join, the pipelining algorithm can produce result tuples earlier during the join process at the cost of using more memory to store a second hash-table. Using this algorithm, pipelining along both operands of the join is possible.

The two extreme forms of pipeline parallelism that we consider are linear pipeline parallelism on linear trees and full pipeline parallelism on bushy trees. Our cost model assumes that, for each join operation, the size of the output relation is larger than that of either of the input relations.

2.1 Linear pipeline parallelism

As depicted in Figure 1, linear pipeline parallelism (linear pipelining) is applicable to left linear expression trees such as $(...((R_1 \bowtie R_2) \bowtie R_3) \bowtie ... \bowtie R_k)$. Several processors may be allocated to each join operation at an abstract "node". The first node $(R_1 \bowtie R_2)$ has two relations R_1 and R_2 as its operands. Each other node has only one relation as its right operand. To evaluate the linear expression tree by linear pipeline execution, all join operators perform the hash joins in parallel. The hash join can be divided into two phases. In the first phase, which is called hash phase, all join operators build separate hash tables for their second operand in parallel. In the second phase, the join phase, each join operator compares each tuple of its first operand in turn with entries in its hash table, and sends each tuple of its result to its parent node. The result will become part of the parent node operand. At last, the root join operator stores its result on its local disk.

Since this is main-memory computation, the only disk I/O required is the time to read the input tuples from and to write the result to local disk. In the analysis, we assume that all communication between nodes can be done in parallel, any nodes in the tree can either receive or send data only. For analysis propose, we assume that all input set are of the same size N, and the join selectivity for each join is the same js. We define T_{disk} as the time to read/write a tuple from/to disk. T_{comp} is the time to perform a comparison operation. T_{comm} is the time to transfer a tuple from one processor to other.

The following two types of delay incur in the pipelined parallelism

- *waiting for the first input*: at any non-leaf node an operation has to wait for the first input to arrive. The size of this delay depends on the shape of the query tree, the number of joins in the pipeline and the size of the join operands.
- *delay for input/output*: at any non-leaf node an operation has to wait for the input tuples or output tuples to send. The size of this delay depends on the workload of the nodes in the pipeline tree.

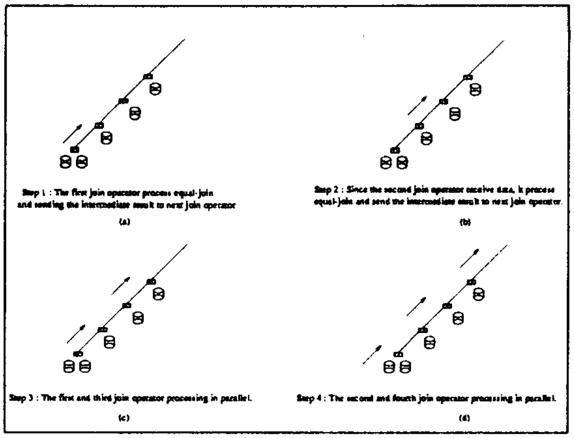
Step 1 : The first join operator process equal-join and sending the intermediate result to next join operator
(a)
Step 2 : Since the second join operator receive data, it process equal-join and send the intermediate result to next join operator.
(b)
Step 3 : The first and third join operator processing in parallel.
(c)
Step 4 : The second and fourth join operator processing in parallel.
(d)

Fig. 1. An example of linear pipelining method

In the best case of minimum time on waiting for the first input, there is no delay for input/output. The first tuple requires time $(2*T_{comp}+T_{comm})*(k-2)$ to arrive at the root node. It also requires time $(T_{comp}+T_{disk})*N^k*js^{k-1}$ to join and write the result into local disk. Thus, the total time required is $(2*T_{comp}+T_{comm})*(k-2)+(T_{comp}+T_{disk})*N^k*js^{k-1}$.

In the worst case of maximum time on waiting for the first input, and maximum delay in waiting for input/output. The first tuple arrive at the root node requires the time $\sum_{i=1}^{k-2} N^i * js^i(T_{comp}+T_{comm})$. Since each join operator can either send or receive tuples at any time. For each set of tuple (group by the first attribute r_1), it requires the time $N^{k-2}*js^{k-2}(T_{comm})+N^{k-1}*js^{k-1}(T_{comp}+T_{disk})$. Hence, the total time required is $\sum_{i=1}^{k-3} N^i * js^i(T_{comp}+T_{comm})+N^{k-1}*js^{k-2}T_{comm}+N^k*js^{k-1}(T_{comp}+T_{disk})$.

Memory requirement of this algorithm is minimal and equal to the summed size of input sets. The total memory required for linear pipeline parallelism is $(k-1)*N$.

2.2 Full pipeline parallelism

Full pipeline parallelism (full pipelining) is applicable to "bushy" or balanced expression trees such as $((R_1 \bowtie R_2) \bowtie (\ldots \bowtie (R_{n-1} \bowtie R_k)))$. Again, several processors may be allocated to each operation node. Evaluation is performed by linear pipelining along all left linear sub-trees in parallel. Because the right operand of all internal operation nodes in the tree is the output of another join operation, each such node must use the double-hashing join algorithm. Again, each node sends each tuple of its result to its parent to become part of one of the parent's operands. The root node stores its result on its disk. (Figure 2.)

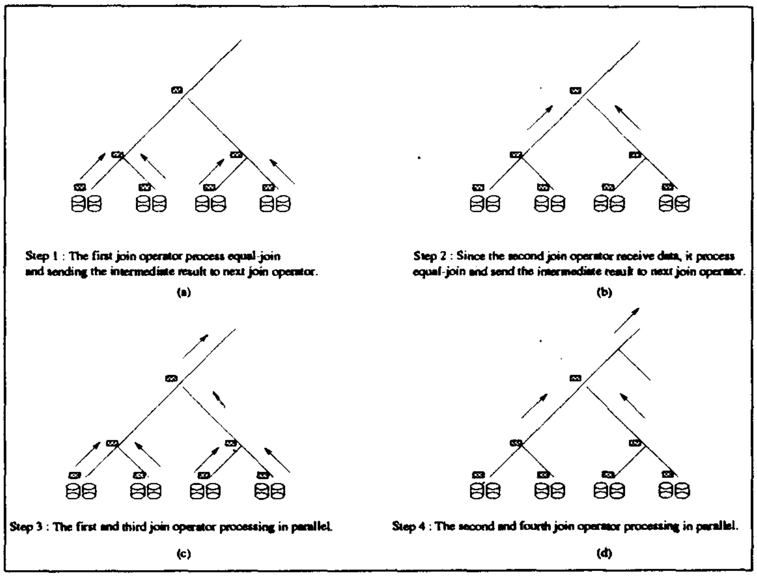
Step 1 : The first join operator process equal-join and sending the intermediate result to next join operator.
(a)
Step 2 : Since the second join operator receive data, it process equal-join and send the intermediate result to next join operator.
(b)
Step 3 : The first and third join operator processing in parallel.
(c)
Step 4 : The second and fourth join operator processing in parallel.
(d)

Fig. 2. An example of full pipelining method

The analysis following assume a balance bushy tree and $P = k - 1$. In the best case of minimum time on waiting for the first input, there is no delay for input/output. The first tuple required the time $(2 * T_{comp} + T_{comm}) * (\log k - 1)$ to arrives the root node and it required the time $(T_{comp} + T_{disk}) * N^k * js^{k-1}$ to join and write the result into local disk. Thus, the total time required is $(2 * T_{comp} + T_{comm}) * (\log k - 1) + (T_{comp} + T_{disk}) * N^k * js^{k-1}$.

In the worst case of maximum time on waiting for the first input, and maximum delay in waiting for input/output. The first tuple required the time $(T_{comp} + T_{comm})(\sum_{i=1}^{\log k - 1} i^{2^i} * js^{2^i - 1} - 1)$ to arrives the root node. Since each join operator can either send or receive tuples at any time, the algorithm required the maximum time $2 * (T_{comp} + T_{comm}) N^{2^{\log k - 1}} * js^{2^{\log k - 1} - 1} + (T_{comp} + T_{disk}) * N^{2^{\log k}} * js^{2^{\log k} - 1}$. The total time required is $(\sum_{i=1}^{\log k - 2} i^{2^i} * js^{2^i - 1}(T_{comp} + T_{comm}) + 2 * N^{\frac{k}{2}} * js^{\frac{k}{2} - 1}) + 2 * (T_{comp} + T_{disk}) N^k * js^{k-1}$

Full pipelining requires less communication time because the communication paths are shorter. However, memory requirement is greater because two hash tables (one hash table for each operand) must be stored at each internal node.

此为试读，需要完整PDF请访问：www.ertongbook.com

To join k sets with a common key, the memory required for a full pipeline tree (balance) is $2N^{\frac{k}{2}}js^{\frac{k}{2}-1} + 4N^{\frac{k}{4}}js^{\frac{k}{4}-1} + \ldots + kN$.

3 Constructing intermediate expression trees

The above discussion indicates that linear expression trees (with linear pipelining) have minimal memory requirement but longer total running time. Bushy expression trees (with full pipelining) have greater memory requirement but smaller running times. Evaluation of bushy trees may not be possible with limited memory.

We show how to construct an expression tree that has minimal computation time subject to the constraint that evaluation can be performed in the available memory. In other words, we aim to achieve maximum parallelism by making maximum use of the available memory.

In general, solving this problem optimally is computationally intractable. A simple two-phase approach is first to find the tree of least total cost and then to find the best evaluation strategy for this tree. Our approach starts with a least cost (linear) tree suitable for linear pipelining. It then iteratively transforms this tree to one that meets our requirements. Each step of the transformation selects the most expensive join operations in the current tree, estimates their memory requirements, and partitions the tree accordingly. The process stops when no more partitioning can be performed.

Our cost models in Section 2 can be used to estimate the computation time and memory requirement of the result tree, evaluated using full pipelining.

We can calculate the total computation time by summing up the computation cost for each join operator. A more expensive computation cost indicates a longer computation time. Comparing the two methods of pipelining, linear pipelining execution requires more computation time than full pipelining execution.

All communication between any nodes is processed in parallel. The total communication time can be sum by summing up the maximum communication time in each stage. Full pipelining requires less communication time, this is because it reduces the size of intermediate result by splitting the query tree into two smaller sub-trees. When computing the Cartesian Product, full pipelining can reduce up to N times communication time required by linear pipelining.

As for memory usage, linear pipelining required less memory. In Linear pipelining, each join operator stores the hash table of a relation in memory, whereas full pipelining stores a hash-table for each input (two hash-tables for each join operator).

We now provide two algorithms which transform a linear pipeline tree into parallel pipeline tree on memory restriction M. The difference between these algorithms are the computation time required to generate the resulting tree and the total running time to execute the resulting tree. The first algorithm can produce a least cost tree with a longer computation time. The second algorithm can produce an almost-least cost tree with a shorter computation time.

3.1 The Global Partition Algorithm

The Global Partition algorithm constructs an expression tree that has minimal computation time subject to the constraint that evaluation can be performed within the available memory.

In generate, this algorithm repeated inserts a double-hash join operator into each of its linear pipeline sub-trees. After the insertion, the total memory required to evaluate the resulting tree remains less than M. Also, the total time to execute the resulting tree is minimum. We say that a linear sub-tree of T is full, if adding any node of T into it will destroy its linear structure.

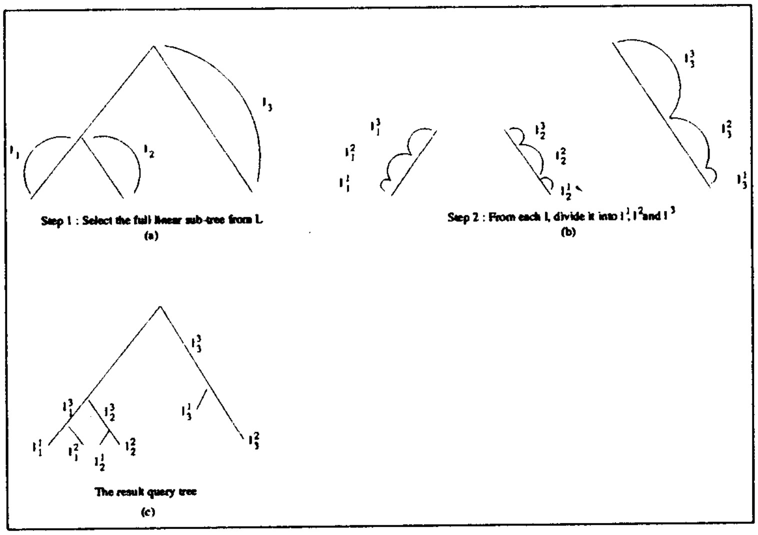

Fig. 3. An example of Global Partition algorithm

Procedure Global_Partition

1. Let L contain all full linear sub-trees from the query tree T.
2. For all $l \in L$, find a global partition that divide each l_i into 3 parts (linear sub-trees) l_i^1, l_i^2 and l_i^3 and satisfies the following conditions :
   - size(l_i^1)=size(l_i^2).
   - size(l_i^3) is minimum over all possible linear sub-trees.
   - the memory required for evaluate the new tree is less than M.
3. Repeat above with the new tree until the memory required to evaluate T is larger than M.

Obviously, optimal solution is NP-hard because we have to examine all possible partitions of the linear sub-trees. An effective way to find a global partition in polynomial time is repeatedly dividing each linear sub-tree into two equal parts until the memory required is less than M. However, this can only obtain an approximation solution.

In Figure 3, an example of Global Partition is given. In (a), the algorithm selects all full linear sub-trees from the given query tree T. In (b), it divide each sub-tree into three parts, where the $size(l_i^1) = size(l_i^2)$ and $size(l_i^3)$ is minimum. Also, the memory required for the new double hash-join operator is smaller than the available memory. The result has been showed in (c).

3.2 Single Sub-Tree Partition Algorithm

Similar to Global Partition algorithm, the Single Sub-tree Partition algorithm constructs an expression tree that has almost minimal computation time subject to the constraint that evaluation can be performed in the available memory.

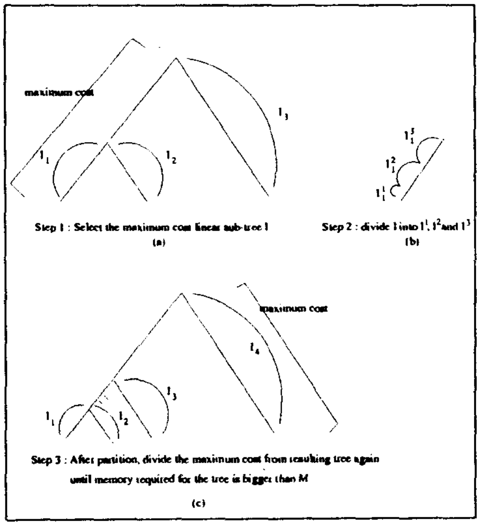
maximum cost
l1
l2
l3
Step 1 : Select the maximum cost linear sub-tree l
(a)
l1 3
l1 2
l1 1
Step 2 : divide l into l1, l2 and l3
(b)
maximum cost
l4
l3
l1
l2
Step 3 : After partition, divide the maximum cost from resulting tree again
until memory required for the tree is bigger than M
(c)

Fig. 4. An example of Single Sub-Tree Partition algorithm

Procedure Single_Sub-Tree_Partition

1. Let L contain all full linear sub-trees from the query tree T.
2. Select the maximum cost linear sub-tree l from L.
3. Divide l into three parts l^1, l^2 and l^3 and satisfies the following conditions :
   - size(l^1)=size(l^2)
   - size(l^3) is minimum.
   - the memory required for the new tree is less than M.
4. Repeat the above with the new tree until the memory required to evaluate T is bigger than M.

In Figure 4, we give an example of Single Sub-tree Partition algorithm with in the available memory. In (a), the algorithm selects the maximum cost linear sub-tree from the given query tree. In (b), it divides the sub-tree into three parts (l^1, l^2, l^3), where $size(l_i^1) = size(l_i^2)$ and $size(l_i^3)$ is smallest. Also, the memory required for the new join operator is less than the available memory. The result has been shown in (c) and the process continues until no more double-hash join operator can be insert into the query tree.

4 Comparisons

The new intermediate query trees generated from the algorithms above require less communication time than the original linear query tree. Moreover, they use a smaller number of levels (depth), which will reduce the *waiting for the first input* delay. In the linear tree, all join operators are connected in a single linear path. In case if some join operators are busy while others are waiting for data input/output from thier neighbouring join operators on the path, data pipelining in the whole query tree will be halted. In contrast, our new intermediate query tree partitions the join operators into a number of paths. If some paths halt due to the *delay of input/output*, other paths will remain to work independently. In this way, parallelism in the whole query tree is increased. Of course, the new intermediate query tree requires more memory when the intermediate results are large.

5 Conclusions

In this paper, we investigated one aspect of parallel relational-query evaluation. Specifically, we have analysed the total running time and memory usage for linear pipelining and full pipelining algorithm. Also, we have proposed two efficient algorithms to improve the total running time for linear pipelining by fully utilising the available memory.

Becasue it is hard to find a precise cost model with the presence of pipelining delay, it is difficult to accurately analyze the performance of the proposed query trees. It is therefore desirable to have an actual implementation of these query trees on a suitable parallel machine, and to obtain a more detailed performance analyses and comparisons with the original linear query tree, especially on the average case.

References

1. P.S. Yu M.S. Chen, M.L. Lo and H.C. Young. Using segmented right-deep trees for the execution of pipelined hash-joins. *Proc 18th VLDB Conf*, 1992.
2. P.S. Yu M.S. Chen and K.L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. *Proc 8th Data Engineering Conf.*, 1992.
3. D.J. DeWitt and J. Gray. Parallel database system: The future of high-performance database systems. *Communications of the ACM*, 35, 1992.

4. W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. *Proc 1st PDIS Conf*, 1991.
5. Peter M.G. Apers Annita N.Wilschut, Jan Flokstra. Parallel evaluation of multi-join queries. *SIGMOD Record*, 1995.
6. H. Boral R. Krishnamurty and C. Zaniolo. Optimization of nonrecursive queries. *Proc 12th VLDB Conf*, 1986.
7. P. Valduriez R.S.G. Lanzelotte and M. Zait. On the effectiveness of optimization search strategies for parallel execution strategies. *Proc 19th VLDB Conf*, 1993.
8. A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel, main-memory environment. *Proc 1st PDIS Conf*, 1991.
9. A.N. Wilschut and P.M.G. Apers. Dataflow query execution in a parallel, main-memory environment. *Journal of Distributed and Parallel Databases*, 1993.
10. A.N. Wilschut and P.M.G. Apers. Pipelining in query execution. *Porc of the International Conference on Databases, Parallel Architectures and their Applications*, 1990.

Dynamic Time-Based Scheduling in a Hard Real-Time System *

Seonho Choi[1] Sameh El-Sharakwy[2] Bao Trinh[2] Ashok K. Agrawala[2]

[1] Department of Computer Science, Bowie State University, Bowie, MD 20715, USA
[2] Institute for Advanced Computer Studies, Department of Computer Sicence, University of Maryland, College Park, MD 20742

Abstract. In traditional time-based scheduling schemes for real-time systems time line is explicitly managed to obtain a feasible schedule that satisfies all timing constraints. In the schedule the task attributes, such as task start time, are statically decided off-line. However, for dynamic real-time systems, in which new tasks may arrive during the operation, or tasks may have relative timing constraints based on information only known at run-time, such static schemes may lack the ability to accommodate dynamic changes. In this paper we present a new scheduling scheme called *dynamic time-based scheduling* that has been developed for *Maruti* hard real-time system. In the scheme any attributes of task instances in the schedule may be represented as *functions* parameterized with information available at task dispatching time. This dynamic scheme supports flexible resource management at system operation time.
To show its applicability we present a solution approach, based on dynamic time-based scheduling scheme, for dispatching tasks with relative timing constraints. The relative constraints may be defined across the boundary of two consecutive scheduling windows as well as within one scheduling window. We present the solution approach with which we are not only able to test the schedulability of a task set, but also able to obtain maximum slack time by postponing static task executions at run-time.

1 Introduction

Real-time computer systems are characterized by the existence of timing constraints on computations they carry out. The timing constraints are statically determined at pre-runtime from the characteristics of physical systems they interact with. In *hard real-time systems*, a timing failure is considered catastrophic and a guarantee should be given prior to execution that every timing constraint will be satisfied. Examples are found in application domains such as avionics, process control, automated manufacturing, robotics, etc.

* This work is supported in part by ONR and ARPA under contract N66001-95-C-8619 to the Computer Science Department at the University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, ONR or the U.S. Government.

此为试读，需要完整PDF请访问：www.ertongbook.com