데이터플로우 모델에서 통신비용 최적화를 이용한 분산 데이터베이스 처리 방법

A Method for Distributed Database Processing with Optimized Communication Cost in Dataflow model

전 병 욱*
Byung-Uk Jun

요 약

대용량 데이터베이스의 처리 기술은 오늘날과 같은 정보 사회에서 가장 중요한 기술 중 하나이다. 이 대용량의 정보들은 지역적으로 분산되어 있어 분산처리의 중요성을 더욱 부각시키고 있다. 전송 기술과 데이터 압축 기술의 발전은 대용량 데이터베이스의 처리 속도를 높히기 위한 필수 기술이다. 그러나 이 기술들의 효과를 극대화하기 위하여 각각의 task에서 필요한 실행시간, 그 task로부터 생성되는 데이터량 및 그 생성된 데이터를 이용한 연산을 위해 다른 processor나 컴퓨터로 이동할 때 필요한 전송 시간 등을 고려하여야 한다.

본 논문에서는 대용량 분산 데이터베이스의 처리를 최적화하기 위하여 dataflow 기법을 사용하였으며 그 처리 방법으로 vertically layered allocation scheme을 사용하였다. 이 방법의 기본 개념은 processor간 communication time을 고려하여 각 process들을 재배치하는 것이다. 본 논문은 또한 이 기술의 실현을 위해 각 process의 실행시간과 출력 데이터의 크기 및 그 전송시간을 예상할 수 있는 모델을 제시하였다.

Abstract

Large database processing is one of the most important technique in the information society. Since most large database is regionally distributed, the distributed database processing has been brought into relief. Communications and data compressions are the basic technologies for large database processing. In order to maximize those technologies, the execution time for the task, the size of data, and communication time between processors should be considered.

In this paper, the dataflow scheme and vertically layered allocation algorithm have been used to optimize the distributed large database processing. The basic concept of this method is rearrangement of processes considering the communication time between processors. The paper also introduces measurement model of the execution time, the size of output data, and the communication time in order to implement the proposed scheme.

☞ keyword : Multiprocessor, Distributed Database, Dataflow, Allocation, Communication Cost

1. INTRODUCTION

Applications such as Artificial Intelligence and knowledge based systems, because of their underlying data size and ambiguity, require more complicated techniques and sophisticated analysis than those available in the traditional database management systems (DBMSs). Although traditional DBMSs can effectively manage large amount of data, they have failed to manage the complex semantics of databases effectively. Using a logic program as an advanced query language is an alternative to support these complex applications. A logic program is more suitable to represent and manipulate the large semantic rules found in knowledge base systems. A logic program, usually, in conjunction with the relational data model is

* 정회원: 수원대학교 정보통신공학과
bjun@suwon.ac.kr
[2007/02/06 투고 - 2007/02/13 심사 - 2007/2/20 심사완료]

used to support the efficient manipulation of large data found in knowledge base applications [1].

It has been shown that, multiprocessing techniques offer performance improvement in the execution of the database queries. These techniques rely mainly on increasing the hardware utilization and decreasing the execution time by exploiting fine grain parallelism embedded in a query [2]. Complexities of the conventional multiprocessing environment, however, have motivated researchers to seek other alternatives for handling concurrent applications. Since 1970's, dataflow paradigm has been recognized as an alternative computational model to support concurrency [3]. The attractiveness of the dataflow concept stems from the fact that dataflow operations are asynchronous in nature. Therefore, the instructions in dataflow do not impose any constraints on sequencing except for the data dependencies contained in the program. In a dataflow computation, the execution of a program can be progressed along various paths simultaneously and hence, inherent parallelism can be exploited more efficiently.

The execution of logic program in a dataflow multiprocessor environment has been addressed in [4]. However, these efforts did not address the issue of load balancing and scheduling of the logic programs. The scheme proposed in [5] maps a logic program (AND/OR tree) onto a dataflow graph. The parallel execution of the logic program was improved by partitioning the generated dataflow graph into subgraphs and allocating the resulting subgraphs onto available processors. This paper is an attempt to expand the scope of our previous work by using a set of heuristic rules.

In handling large volume of data, execution time of each basic operation is one of the major factors that should be considered in an allocation policy. The execution time depends on several parameters: size of the data, size of main memory available, page size, and the number of disk accesses. In the current work, the execution time of each node is estimated and statistical parameters are used to calculate the size of the generated data. Such an estimated value along with a simple set of heuristic rules are used to rank and schedule the execution of different dataflow paths.

The logic program and transformation of its AND/OR tree to a dataflow graph are briefly discussed in section 2. This section also describes how a dataflow graph is partitioned into vertical layers as an attempt to balance computation time and communication cost in a multiprocessor platform. The proposed optimization scheme is presented in section 3. Finally, section 4 concludes the paper with analysis of the proposed model and its simulation results.

2. LOGIC PROGRAM AND DATAFLOW

2.1 Logic Program

Logic programming has grown out of research on resolution inference. Resolution is an inference step required to build a complete inference system for predicate logic in clause form. Applying the rule of resolution into the Horn clauses makes the resolution inference highly suitable for computer implementation. From the language designers point of view, a logic language is a highly structured High Level Language (HLL). Similar to other HLL, the manipulation of data and flow of control are the main concerns in the implementation of the logic programming languages. The basic structure that programs deal with are constants, variables or compound terms. A logic program is a collection of statements which are known to be true. The statements take two forms facts and rules. A fact represents a known knowledge. A rule states some relationships that is held for some data. The

program is used by posing a query to ask if some other statements are true based on this knowledge.

```
f1(A,B,C) :- f2(A,E), f3(D,E), f4(B,D), f5(C,D).
f2(A,E) :- r1(A,E).
f2(A,E) :- r2(A,F), f6(E,F).
f3(D,E) :- r3(D,E).
f4(B,D) :- r8(B,K), r9(D,K).
f5(C,D) :- r10(D,L,M), f7(L,N), f8(M,O),
           r11(C,N,O).
f6(E,F) :- r4(F,G,H), r5(G,I), r6(H,J), r7(E,I,J).
f7(L,N) :- r12(L,P), r13(N,P).
f8(M,O) :- r14(M,Q), r15(Q,R), r16(O,R).

? f1(a,b,C).
```

Figure 1. An Example of logic program.

Figure 1 shows a logic program and its AND/OR tree representation is in Figure 2. In the program, the fn represents the head of rules and the rn represents the fact. Figure 1 imposes a query ?f1(a,b,C) which searches for the value of C with a and b as known values. In Figure 2, the leaf nodes represent the actual database relations to be searched and the internal nodes represent the AND - rectangular nodes and OR relations - circular nodes, respectively. Arcs among the nodes represent data dependencies within the program.

The execution of logic program in a multiprocessor environment requires a parallel

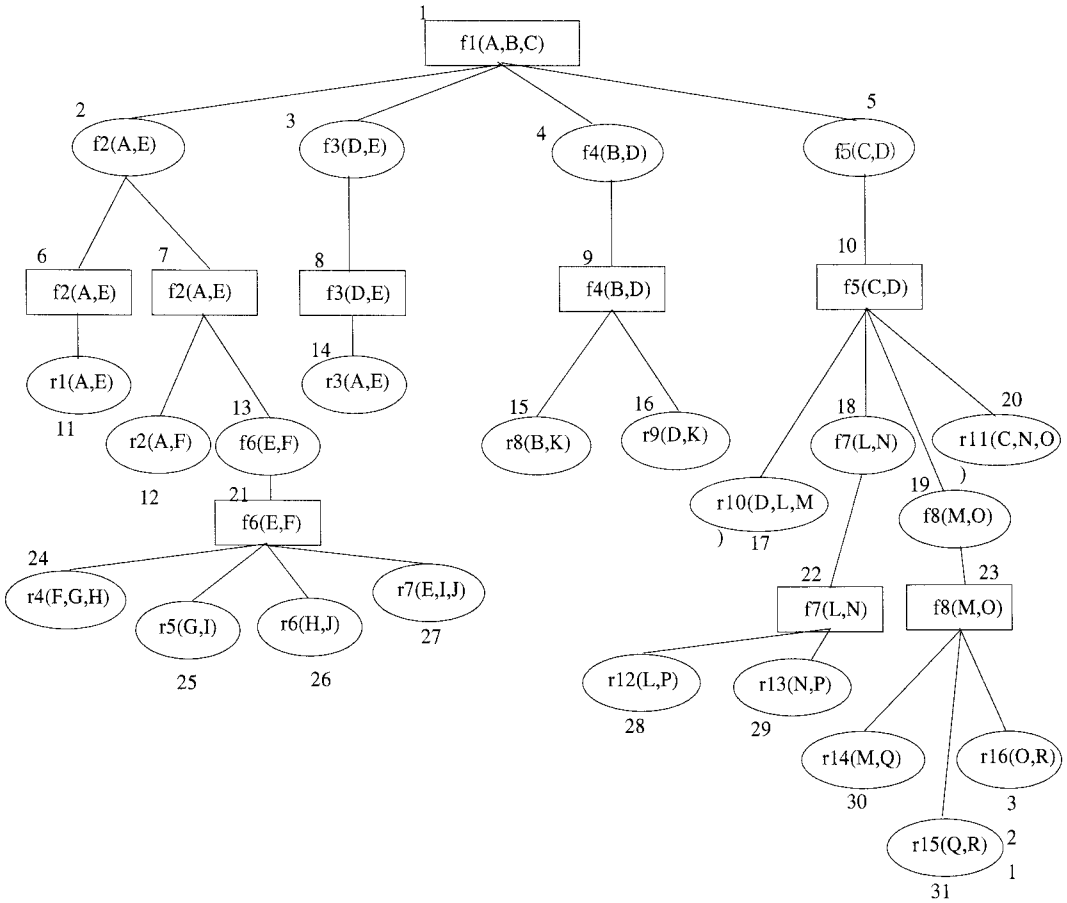

Figure2. AND/OR tree representation of Figure 1.

execution model which enforces an efficient execution order of the AND/OR branches. In the sequential execution model i.e., conventional uniprocessor environment depth first execution of an AND/OR tree is a reasonable paradigm. However, in a parallel environment, where hardware utilization and higher throughput are of concern, one is required to devise a more sophisticated scheme to execute an AND/OR tree.

There have been various systematic approaches to solve this problem [6] [7]. Most of these techniques are based on the exploitation of OR-parallelism, independent AND-parallelism, and dependent AND-parallelism [8]. OR-parallelism corresponds to a parallel search of the OR branches in the AND/OR tree, which means several clauses matching a goal are processed concurrently. Independent AND-parallelism occurs when several data independent goals in a clause can be processed simultaneously no common variables among goals. Dependent AND-parallelism occurs when mutually dependent goals that share common variables are executed in parallel to produce a binding for the variables.

Parallel execution of dependent AND-parallelism branches are problematic. In such a case, it would be impractical to check the consistency of all combinations of the solutions obtained from each body atom. In addition, the size of the intermediate data generated becomes even more problematic in applications with large amount of data. To reduce the size of the intermediate data and increase the parallelism, an efficient sequence of execution based on the producer/consumer relationship should be developed. On the other hand, rules with OR-parallelism and independent AND-parallelism can be executed in parallel. However, this could easily result in an explosion in the number of activated processes. Therefore, again for practical cases where available resources are restricted, the processes should be activated in a limited manner which improves the performance and increases the hardware utilization.

2.2 Graph Transformation

Dataflow techniques have been proposed as an alternative execution model to the conventional control flow model of computation. Because of the parallel nature of dataflow computation, execution of logic program in a dataflow multiprocessor environment has been recognized by some researches [4]. However, in spite of attractive properties of dataflow processing, dataflow researchers should develop effective schemes for partitioning and allocation of dataflow graphs in a multiprocessor environment. To take advantage of the dataflow computation in the execution of logic program, a scheme has been proposed to transform the AND/OR tree to a dataflow graph [5]. The converted graph: i) holds the rich semantics embedded in an AND/OR tree, ii) allows the exploitation of parallelism in a logic program, and iii) allows efficient partitioning of the program graph.

The transformation scheme observes the propagation of data value(s) to map the AND/OR tree to its equivalent dataflow graph. First, a directed AND/OR tree is generated that shows all possible propagation paths, hence, the execution orders in a tree (Figure 2). Then the directed AND/OR tree is converted to a dataflow graph. Two special nodes are introduced to combine partial results generated by the subgraphs - Intersect and Union nodes that represent the AND relations and OR relations in the original AND/OR tree,

respectively. Finally, to simplify the partitioning and allocation task, an attempt has been made to reduce the complexity of the dataflow graph by removing bookkeeping operational nodes - i.e., AND/OR nodes. The node removal is done by connecting the successor(s) of each deleted node to its ancestor in the dataflow graph. Semantics of the AND/OR nodes are still reflected by the Intersect/Union nodes.

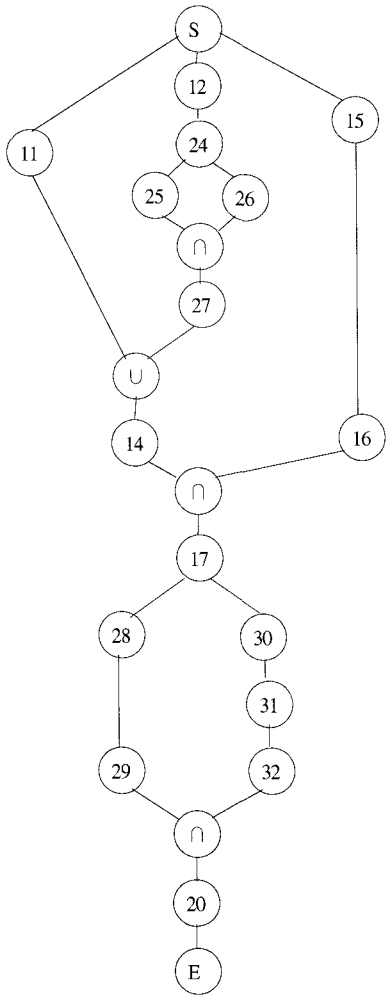

Figure 3. Dataflow Representation of Figure 2.

Figure 3 illustrates the converted dataflow graph of the Figure 2. 'S' and 'E' are dummy nodes which represent the start node and the exit node of the graph. The intersect node and union node are represented by '∩' and '∪', respectively. The numbers shown in the Figure 3 represent the node numbers in the AND/OR tree of the Figure 1b. Note that Figure 3 shows the execution order and synchronization point of the operations (the leaf nodes and the special nodes). Each node in the dataflow graph can be any computational operation depending on how the predicate has been defined.

Figure 4 shows the symbolic representation of each dataflow node Vi is the input vector, Vo is the output vector, s is the selectivity factor and R is the cardinality of the underlying relation. For example, in case of a select node, σVi(R), the input vector represents conditions of the select operation. These conditions could be in conjunctive, disjunctive form generated by the preceding intersect node and/or union node. Attributes of the input vector and output vector are defined in the program as the arguments of the predicate.

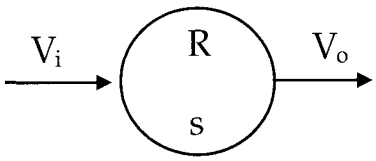

Figure 4. Symbolic Representation of a dataflow node.

2.3 Execution Time and Communication Time

In a multiprocessor environment, one is required to exploit the parallelism as a means to achieve a higher performance. However, due to the limited

hardware resources available and the communication overhead, the parallel processes should also be distributed among the available processors efficiently. In the partitioning of a program, several issues should be taken into consideration: A partitioning method should maximize the exploitable parallelism attempt to aggregate processes should not restrict or limit parallelism. In other words, processes that are grouped into a thread should be the parts of a program where little or no exploitable parallelism exists. In addition, the longer the thread length, the smaller the communication cost between processors becomes. This increases the locality and consequently increases the utilization of the resources. In order to realize the efficient partitioning of the program, one is required to estimate the execution time of each process.

In our scheme, the execution time of each dataflow nodes and the communication costs will be estimated to exploit efficient partitioning. Using the parameters in Figure 3, the execution time of each node, T_{exe}, is a function of Vi , R, s, and the complexity of the operation. On the other hand, the communication cost, T_{comm}, between the source and the destination nodes is a function of V_o and the communication network. The timing is defined as:

$$T_{exe} = \alpha \times V_i \times R \quad (1)$$

$$T_{comm} = T_{setup} + \tau \times V_o \quad (2)$$

where α is the time required to perform a unit operation, T_{setup} is the mandatory setup time to transmit data, and τ is the communication cost per unit of data. α, T_{setup} and τ are system dependent parameters. V_o is dynamic in nature and a function of the user query, the underlying relation, and the input vector and can be estimated as follow:

$$V_o = s \times V_i \times R. \quad (3)$$

2.4 Vertically Layered Allocation

Once the AND/OR tree is converted to the dataflow graph and the execution time of each node is estimated, the dataflow graph is partitioned and allocated to the available processors. Partitioning is the division of an algorithm into procedures, modules and processes. Assignment, on the other hand, refers to the mapping of these units to processors. The ultimate goal is to maximize the inherent concurrency in a program graph by minimizing contention for processing resources. However, the problem is not a trivial one. It has been shown that obtaining an optimal allocation of a graph with precedence is NP-complete [9]. Therefore, heuristic solutions are the only possible approach to solving the allocation problem. The allocation problem is further complicated due to the existence of variety of architectural differences as well as interconnection topologies.

The Vertically Layered allocation scheme has been proposed which compromises between computation and communication costs [3]. The scheme consists of two separate phases: separation and optimization phase. The basic idea behind the separation phase is to arrange nodes of a dataflow graph into vertical layers such that each vertical layer can be allocated to a processor. This is achieved by utilizing Critical Path (CP) and Longest Directed Paths (LDP) heuristics. These heuristics attempt to distinguish the critical path of a program and recursively determine the other vertical layers by finding longest directed paths emanating from the nodes which have already been assigned to

vertical layers. The estimated execution time of each node is used to determine the longest directed paths. Therefore, the CP and LDP heuristics minimize contention and inter-processor communication time by assigning each set of serially connected nodes to a single PE. Once the initial phase is completed, the Communication to execution Time Ratio (CTR) heuristic algorithm optimizes the final allocation. This is done by considering whether the inter-PE communication overhead offsets the advantage gained by overlapping the execution of two subsets of nodes in separate processing elements.

To apply the CTR heuristic algorithm, the execution time of a new critical path which includes the effects of inter-PE communication costs is determined. If an improvement results after applying the CTR heuristic, the nodes are combined into a single PE. Since combining two parallel subsets of nodes into a single processing element forces them to be executed sequentially, a new critical path may emerge from the optimization process. This process is repeated in an iterative manner until no improvement in performance can be obtained.

A simulator was developed to evaluate the effectiveness of the vertically layered allocation scheme. Several dataflow graphs with varying degree of complexities were chosen as the testbed. The simulation results have shown that, in general, the total execution time decreases as the number of processing elements increases. Also, in cases where inter-PE communication delays are negligible, the scheme showed only slight degradation in performance compared to the critical path list scheme. However, as communication delays increase, the proposed scheme offers promising performance improvements.

3. OPTIMIZATION

Two main approaches, namely static and dynamic, exist for task allocation. In spite of their differences, the goal of program allocation is to maximize concurrency in a program graph by minimizing contention for processing resources. In static allocation, the tasks are allocated at compile-time using global information about the program behavior and the system organization. The cost of allocating tasks is incurred once for a given program even though the program may be executed repeatedly. However, static allocation policies can be inefficient when estimates of run-time dependent characteristics are inaccurate. A dynamic allocation policy on the other hand is based on measuring processor loads at run-time, assigning activated tasks to the least loaded processor and balancing the load by migrating tasks. The disadvantage of dynamic allocation is the overhead involved in determining processor loads and allocation of tasks at run-time.

In this paper, we propose a hybrid static and dynamic allocation scheme to achieve higher performance in the execution of logic program. The static allocation is enforced using a set of heuristics based on the estimated execution time and the size of output vector of each node. The dynamic allocation is enforced by observing the actual size of the output vector of each node and possible migration (elimination) of the node(s) or branches during the execution time. In order to carry out our optimization scheme, three types of relationships among the serially connected nodes in the dataflow graph are considered:

- Nodes connected based on producer/consumer relationship: In this case, the output of a node (producer) is directly consumed by the successor node (consumer). Naturally, an empty output

vector of the producer node may eliminate the consumer node.

- Two or more branches are merged with a union node: If one of input branches to the union node generates a non-empty output vector, the input vector of succeeding node is non-empty. Thus, the execution of the succeeding node is guaranteed unless all the input vectors to the union node are empty.
- Two or more branches are merged with an intersect node: If one of the input branches to the intersect node generates an empty output vector, the output vector of the intersect node is empty. Thus, the execution of the succeeding node can be eliminated.

3.1 Heuristics for Static Allocation

In a shared nothing multiprocessor environment, communication and synchronization among processes are necessary to control the global execution of the program. Each processor executes its task, and sends the result to the destination processor(s) according to the data dependence among the processes. Therefore, one is required to introduce a proper control mechanism that guarantees a valid execution. In our scheme, the converted dataflow graph due to its partial ordering guarantees such a valid execution order. However, in a multiprocessor environment where computation resources are scarce, one has to devise a scheme that reduces the amount of the computation as well. In this section, we propose a set of compile-time heuristics for scheduling of the branches of the converted dataflow graph in order to improve the performance. The estimated execution time of each node (T_{Pi}) and the size of the output vector (V_{oi}) are used to prioritize the execution of the paths as a means to manage the resources.

Based on the AND and OR relationships embedded in the definition of converted dataflow graph — union node or intersect node — the proposed optimization algorithm utilizes two sets of the heuristic rules:

For the AND relation, the algorithm attempts to eliminate as many branches as possible by early scheduling of the ones with higher probability of the failure:

i) Schedule the branches that have smaller expected execution time,
ii) Schedule the branches that have smaller expected output vector,
iii) Schedule the branches that have higher number of the intersect nodes.

For the OR relation, the algorithm attempts to improve the hardware utilization by early scheduling the branches with higher probability of success.

i) Schedule the branches that have lower estimated execution time,
ii) Schedule the branches that have higher expected number of the output,
iii) Schedule the branches that have fewer number of the intersect nodes.

3.2 Effects on Dynamic Allocation

Performance of the aforementioned static allocation policy can be further improved by considering the issue of dynamic allocation. In this section, an optimization scheme has been presented by observing the propagation of empty output as a means of reducing the amount of computation at run-time. The size of output vector is a function of the user query, the size of relation, selectivity, and the size of input vector (refer to Figure 4).

Naturally, a node with a non-empty input vector is executed and an empty output vector is propagated throughout the program graph. Therefore, if a node generates an empty output vector at run-time, propagation of the empty output vector could eliminate some of the branches in the dataflow graph.

As discussed earlier, if one of the input branches to the union node generates a non-empty output vector, the input vector of succeeding node is non-empty. Therefore, the propagation of an empty output vector can be terminated at the union node unless all the input vectors to the union node are empty if a node generates an empty output vector, the next possible valid execution point is the closest union node. As an example in Figure 3, an empty output vector from node 25 eliminates processing of the next intersect node and node 27. This in turn will eliminate node 26 as well. Thus, the next union node in line has to wait for input from node 11. Interestingly, generation of an empty output vector by node 15 eliminates the whole dataflow graph. Algorithm 1 shows the sequence of the operations:

Algorithm 1.
BEGIN
If (output is empty) {
Find next Union node.
Eliminate processes until the Union node.}
Perform next node.
END

Finally, the run-time scheduling as discussed above could also drastically improve the resource utilization by reducing the communications among the processors and reducing the processors' operational loads.

4. SUMMARY AND CONCLUSION

Parallel execution of the logic program and its optimization within the scope of database environment are main interests of the paper. The execution of logic program in a multiprocessor environment requires a parallel execution model which enforces an efficient execution order of the AND/OR branches. We have developed a technique to map the logic program onto a dataflow graph to support scheduling and allocation of the logic programs in a multiprocessor environment. This was achieved by partitioning the generated dataflow graph into vertical subgraphs and allocating the resulting subgraphs on available processors. This technique was further improved by enhancing the scope of the original partitioning scheme by a set of compile-time and run-time optimization rules. To evaluate the practicality and feasibility of the scheme, a simulator has been developed to measure the effectiveness of the proposed optimization scheme. Performance analysis indicates that the proposed scheme is very effective in reducing the overall execution time through static and dynamic scheduling.

References

[1] Jeffrey D. Ullmann and Jenniefer Widon, "A First Course in Databese Systems", 1997.

[2] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy, "Query Optimization for Parallel Execution," ACM SIGMOD, '92, pp. 9-18.

[3] B. Lee, and A.R. Hurson, "Dataflow Architectures and Multithreading," IEEE Computer, vol. 27, no. 8, 94, pp. 27-39.

[4] Dan Moldovan, Wing Lee, and Changhwa Lin, "SNAP: A Marker-Propagation Architecture for Knowledge Processing," IEEE Transactions on

Parallel and Distributed Systems, vol. 3, No. 4, July, '92, pp. 397-410.

[5] Byung-Uk Jun, A.R. Hurson, and B. Shirazi, "Handling Logic Programs in Multithreaded Dataflow Environment," Second Biennial European Joint Conf. on Engineering Systems Design and Analysis, '94, pp. 533-540.

[6] G. Gupta and V. S. Costa, "AND-OR Parallelism in Full Prolog with Paged Binding Arrays," Proc. of PARLE, pp. 617-632.

[7] M. J. Wise, "Message-Brokers and Communicating Prolog Processes," 4th Inter'l Conf. on Parallel Architectures and Languages in Europe, 92, pp. 535-549.

[8] Z. Yuhan, T. Honglei, and X. Li, "AND/OR Parallel Execution of Logic Programs: Exploiting Dependent AND-parallelism," ACM SIGPLAN Notices, vol. 28, no. 5, 93, pp. 19-28.

[9] C.D. Polychronopoulos and U. Banerjee, Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds, IEEE transactions on Computer, vol. C-36, no. 4, Apr. 87, pp. 410-420.

◐ 저 자 소 개 ◑

전 병 욱(Byung-Uk Jun)

1980년 서강대학교 전자공학과 졸업(학사)
1985년 University of Detroit, Computer Science, MS
1996년 Pennsylvania State University, Electrical Engineering, Ph.D
1979년 11월~1980년 10월 한국과학기술연구소 연구원
1980년 10월~1983년 8월 대영전자
1987년 7월~1991년 12월 Applied Systems Institute, USA, Systems Engineer
1996년 8월~1998년 2월 LG증권 전산실
2000년 8월~2004년 3월 (주)Admobis 부사장, 연구소장
1998년 3월~현재 수원대학교 IT대학 정보통신공학과 교수
관심분야 : 컴퓨터구조, Embedded System Design, 데이터 통신
E-mail : bjun@suwon.ac.kr