JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 18, 815-824 (2002)

Short Paper

Hash Join Algorithms on SMPs Clusters: Effects of Netcaches on its Scalability and Performance

EDWARD DAVID MORENO ORDONEZ
Department of Computer Science
Euripides Foundation of Marilia
CEP 17525-901, Marilia, S.P., Brazil
E-mail: edmoreno@fundanet.br

We investigate the effect that caches, particularly caches for remote accesses, have on the performance of hash join algorithms. The join is a computationally intensive operation of relational databases and is used in many important applications. Thus, there are a considerable number of studies on the parallel hash join. However, most of the previous research does not show how cache affects the performance of these algorithms.

In this paper, we show the impact and benefits of remote caches (Netcaches) on the overall performance of parallel hash join algorithms running on SMP clusters. Furthermore, we show the effects of these caches on speedup and scalability of these algorithms. Our simulation results leads us to conclude that the execution time of hash join algorithms on modern's multiprocessors, with large local and remote caches, could be reduced up to 70%. Finally, we show results for verifying the big effects of netcaches on scalability of these algorithms.

Keywords: hash join algorithms, netcaches, SMPs, clusters, performance evaluation

1. INTRODUCTION

Today many parallel machines have been developed from high-performance microprocessors, utilizing, large low cost memories and interconnecting them through high-speed interconnection networks (high bandwidth and low latency) to many small low cost disks. Furthermore, shared memory machines are very popular because of ease of their parallelization and portability of many existing applications, but they can scale up to only a few or processors.

Working towards the goal of MPP, the SMP-based multiprocessor is a promising approach. So, many recent large scale systems such as DASH, Cedar, Convex Exemplar, S3.mp, Willow, ParDiGM ASURA, Sting, NUMAchine, SPADE, DASH, HP Exemplar Sequent NUMA-Q and SGI-Cray Origin 2000 [7, 13], have followed the model of clusters connected by some sort of high-speed interconnection network.

Received September 22, 2001; accepted April 15, 2002.
Communicated by Jang-Ping Sheu, Makoto Takizawa and Myongsoon Park.

Each cluster consists of two to eight processors connected via a shared bus, and generally they are maintained coherent through snooping protocols. In order to exploit locality and improve inter-cluster communication, these machines use a large cache associated with each cluster. This cache is known as the network cache (netcache). It is also called remote accesses cache (RAC). The netcache is shared by all processors on a cluster and is used to cache data originating from other clusters [7, 12].

There has been significant research concerned with evaluating the performance of parallel applications using shared memory machines. They have studied the impact of hardware and software parameters, but only for scientific and engineering applications.

Recently, many projects have analyzed high performance platforms running commercial applications. The most significant application can be database systems, particularly parallel databases. As a result, it is critical to investigate alternative architectures for large-scale database systems.

The scientific and engineering applications used are all highly optimized for parallel performance and lead to reduced communication and increased locality of data reference. However, commercial programs may be less optimized in several important ways: algorithmic, poorer design of data structures, poor alignment and distribution of data structures among physical memories, and their locality and contention problems with seemingly less important variables. The greater communication generated by these programs can result in big benefits by using network caches [7, 13].

The join operation is one of the most expensive operations in relational database systems, so considerable research has been devoted to developing efficient join algorithm [9]. A survey of current literature reveals that three methods for implementing parallel joins have received much attention: The Nested-loop, Sort-merge, and Hash-based join. The hash join algorithm is very efficient for large relations and provides natural opportunity for parallel processing. Therefore, a large number of hashing algorithms has been implemented. Even though there is an abundance of previous studies on the performance of hash join algorithms for different architectures, most of the research does not analyze how such algorithms exploit the cache. Moreover, as the cached computer architecture becomes common place, it is necessary to analyze how cache memories affect the performance of the hash join algorithms.

Motivated by these observations, in this paper, we study the effectiveness of necaches in SMP-based clusters for three simple hash join algorithms (which is the more complex operation in relational databases). These algorithms are a good example to show the problems working with high communication and larger.

2. TARGET ARCHITECTURE AND ALGORITHMS

SMP-Based Cluster: We simulate a multiprocessor system with 32 processors organized at eight clusters connected by a high-speed interconnection network, which is point-to-point (see Fig. 1). Each cluster (4 processors per node) in the simulated machine contains four 1 GHz processors. Each processor has a 128 Kbytes split cache, direct-mapped, a 32-byte cache line size (for Data and Instruction). The cache L2 also is 256 Kbytes four-way associative and 64-byte line size (for Data-L2 and Instruction-L2).

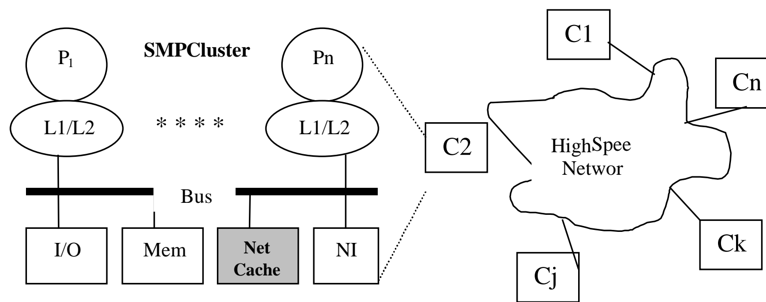

Fig. 1. Organization of a SMP-based multiprocessor system.

Netcache is relatively large size, and unlike the L2 caches, it uses DRAM (for larger cache sizes). This cache on a given cluster cache data from remote clusters, but not the data from local memory. We have varied its size from 512 Kbytes up to 8 Mbytes, and its associativity, N-way = 1, 2 and 4. The timing depends on the cache size, block size and associativity. We have used a tool to exactly obtain its access time [8]. For example, for a cache = 1 Mbytes, Tag time = 40 ns, read time = write time = 100 ns.

A cluster also contains its local memory which implements a portion (~ Gigabytes) of the shared global memory, an I/O unit, a network cache (for data from remote clusters) and a network interface to the point-to-point interconnection network. Cache coherence is maintained using a write-back invalidate protocol. Other key parameters are: Memory (Tag time = 80 ns, read time = write time = 150 ns), Bus (100 MHz, 8 bytes wide, 20ns for arbitration latency), Network (150 MHz, 8 bytes wide, point-to-point), Hit in L1 (Five cycles of processor.), Miss in L2 (Four cycles of processor + NUMA time), Remote Latencies (when is Contention free: 200 ns.).

Execution Driven Simulation: We have implemented a detailed on-line execution-driven simulator (see Fig. 2). It is divided into two sections: a front-end and a back-end. The front-end uses Mint [11] to interpret the native MIPS binaries generated by the application programs. It generates multiple streams of memory reference events, which we used to drive our system simulator model. The back-end does behavioral modeling of the system at a clock cycle level. It takes care of references to shared data, instructions fetch and private data references are assumed to always hit in the processor's caches.

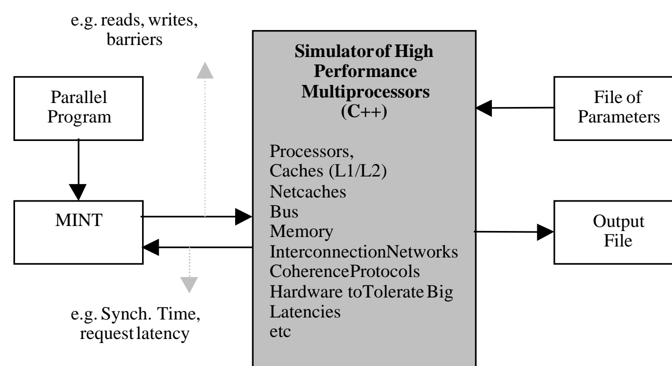

Fig. 2. Simulation environment of SMP-based clusters.

The modeling includes all timing details (e.g. bus arbitration, DRAM and SRAM access time, network contention, etc.). For example, we model the contention effects caused by: processor's caches (L1/L2), memory DRAMS, shared internal-bus, and the interconnection network. We also model the contention caused by the remote access cache. All ordinary data accesses as well as synchronization accesses have been modeled. All contention on shared bus and interconnection network are particularized. To analyze the effects of the bus and the interconnection network, we have used a packet-by-packet model.

Parallel Hash Join Algorithms: The basis of our three parallel hash join algorithms is the uniprocessor hash join implementation. Thus we have implemented three algorithms: the SHJ (Simple Hash Join), the GHJ (Grace Hash Join) and the HHJ (Hybrid Hash Join). The version HHJ is similar to GHJ. This algorithm has been found at least as well as the GHJ that it resembles.

The HHJ is a hybrid form of the SHJ and the GHJ, and shows better performance than the GHJ for small relation sizes. This is because in the HHJ, the associated cost to writing and reading operations of the first bucket can be eliminated by adopting the SHJ and holding the first bucket in memory until the join phase begins. For large relations, both algorithms exhibit almost identical performance, since almost buckets are written to the main memory (or disks) and the advantage of holding one bucket on the local caches is relatively small [9].

Thus, we firstly implement the sequential hash join algorithm. Then, in our parallel implementation, we use the ANL (*Argonne National Laboratory)* macros that allow explore the parallelism by using some special commands. These commands are adequate and tune in for shared memory environments. Such macros are very well known into the parallel processing community [1].

Table 1 shows an example using the commands from ANL macros. It is important to observe that the sequential version of hash join programs does not need effort to get the parallel version because we use the shared memory paradigm.

Table 1. An example of algorithm using ANL primitives.

Main() {
Main_INITENV();
Initialize_by_master(0;
For (i=0; i < P; i++) CREATE(Slave);
BARRIER; Parallel_Section(); BARRIER; }
SLAVE() {
BARRIER; Parallel_Section(); BARRIER(); }
	Parallel_Section() {
	CLOCK(start); WHO_AM_I(&Pid);
	Start = Num_Part/P*Pid;
	End = start + Num_Part/P;
	For(i=start; i< end; i++) {
	Get_input(FileR, FileS, FileQ, &Parameters);}
	}

Our three algorithms consist of two phases: the data split phase and the join phase, similar to implementation by Nakano et al in [9]. At the split phase, both relations are partitioned into many buckets, smaller in size than the staging buffer memory and related to cache block sizes of the local, (caches L1 and L2), caches in the architecture (see Fig. 1). Therefore, each of the partitions is divided to fit in cache size units by hashing on the join attribute. So, this operation permits to match the partitions to local caches.

The join phase starts after data partitioning, with the pair of buckets of both relations being read from the disks or local memory and the join operation executed. At

the join phase, after reading a bucket of relation R, the hash function is applied to a join attribute, and the hash table with fine data is built on the local cache. Then, the bucket of relation S is read from local main memory and the hash table is probed with the join key of this relation.

Our simulation results described in the following section allow us verify and compare the effects obtained by Shatdal [10] with his cache-conscious algorithm. Similarly to Shatdal's study, we have two reasons for verifying that our implemented parallel algorithms do better: (i) they minimize data sharing across the processors as partitioned tuples never need to be accessed by two processors (ii) the partitioned hash tables are likely to be cache resident, thus making the access to the hash table effectively much faster. The better performance of this algorithm is true whenever the overhead of partitioning is sufficiently small.

We used two relations, which have 100 Mbytes and 10 Mbytes. The result size of the output file is 10 Mbytes, e.g. approximately 1/10 of the greater relation. This value is similar to assumption wl5 workload (~1/8 of the greater relation) indicated in [10]. In this approach the tuple of one relation matches zero or one tuple of the other relation, resulting in small join selectivity. We use the extreme case, since our simulation need much memory and time to execute.

3. PARALLEL HASH JOIN ALGORITHMS

Knowing that the HHJ (hybrid hash join) algorithm shows better performance than the Grace and Simple hash join algorithms [4, 9, 10, 13], we just give results for this algorithm. The parallel hash join algorithm consists of two phases, hash and join major. The implementation of the hash phase phase on SMPs machines is most natural. The large relation, R, is partitioned by a hash function and the hash table is generated (it is stored in all local caches of the SMP node). The join phase starts after the hash table is completely. Now, the relation S is read from the main memory (or in the worst case, from the main memories distributed in the system), and for each tuple, the hash tables are probed respectively.

Table 2 summarizes the before process, we have assigned a fundamental operation (read or write) or a partition operation. Assuming that data are in memory, the total time for the algorithm can be expressed as the sum of the time for READ operation (three times), the time for WRITE operation (two times), and the time for partitioning operation (two times). Hence, Total time = Time (read) + Time (write) + T (partition).

In our analysis, we consider that the time for partitioning data is negligible compared to time for the two (read and write) operations. For this reason, we have divided the results for two operations, READ and WRITE, separately. Then, we have considered the effects on the overall execution, which contains major both these two phases.

In our architecture, SMPs-based clusters, the access cost for data (read or write operations) varies depending on where that data is mapped. Thus, the relations may allocate the specific memory space onto physical local memory (belonging to each SMP node) or belonging to others SMP nodes. So, we can use local memory, or remote memory and the remote cache is shared by all processors on a node and is used to cached data originating from other nodes.

Table 2. Operations (read, write, partition) for our parallel hash join algorithm.

#	HASH PHASE Relation R	OPERATION	JOIN PHASE Relation S
1	Each processor reads relation R from main memories	READ-1	
2	Partition the R relation using the hash function	Partition-1	
3	Generate the hash table	WRITE-1	
4		READ-2	Each processor reads relation S from main memories and distributes the tuples in their local caches, or remote caches depending on the tuples are located in remote nodes.
5		Partition-2	Partition the S relation using the same hash function (that used in relation R)
6		READ-3	Verify and compare the hash table
7		WRITE-2	When a match is found, a new tuple is formed by concatenating all of the attributes from both relations and write these matching into the output file.

We study the relative performance of remote caches using four metrics of performance: The hit rate of the netcache, N_U, B_U, and NET. The hit rate of the netcache is measured as the number of requests to which the netcache can respond divided by total number of request to this cache. The N_U (Network Utilization) measured as the usage percentage of the interconnection network. The B_U (Bus Utilization) is taken as the grant latency of the bus (it is measured in cycles of bus or bus clocks). The NET (Normalized Execution Time) is computed as the execution time of the application using netcache divided by the execution time of the application without it.

To keep the number of simulations manageable (and presentable in this paper), we narrowed the ranges for some of cache parameters. In particular, we consider only direct mapped (N-way = 1), 2-way and 4-way associative network caches with LRU replacement. We use different netcache sizes: 512 Kbytes, 1 Mbytes, 2 Mbytes, 4 Mbytes and 8 Mbytes. We always use 64 byte lines sizes.

4. SCALABILITY OF HASH JOIN ALGORITHMS

In this section we show the effects of remote caches on the scalability of our three parallel hash join algorithms; SHJ (simple hash join), GHJ (grace hash join) and HHJ (hybrid hash join). Our experiments indicate that CPU stalls due to memory accesses and synchronization overhead are two major bottlenecks for hash join algorithms on an SMP scaled by increasing the number of processors (see Fig. 3). In Fig. 3 we can see the big impact that remote cache has on diminishing remote accesses and synchronization overheads. Thus, Figs. 4-6 show the HJA algorithms performance as the number of processors increases. It is easy to see that the remote caches increase the efficiency of our algorithms as the number of processors in the system increases.

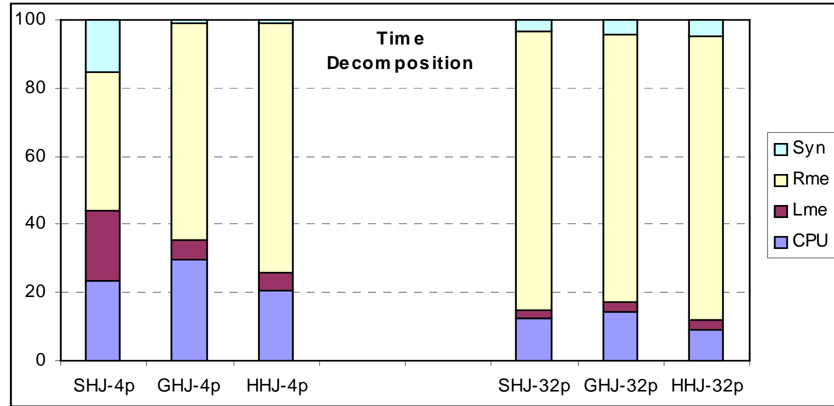

Fig. 3. Time Decomposition: CPU: CPU computation, LME: Local Memory Accesses (local caches and main memory), RME: Remote Memory Accesses, SYN: Synchronization time. We have simulated architectures with 4 and 32 processors and without netcaches. Our three algorithms are SHJ (simple hash join), GHJ (grace hash join) and HHJ (hybrid hash join).

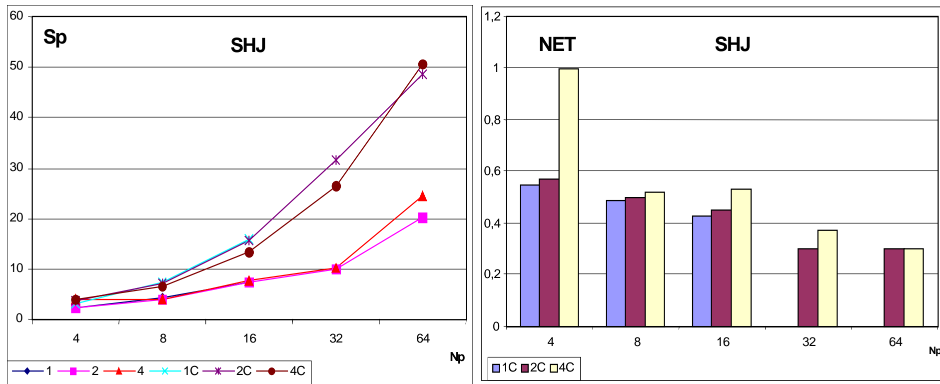

Fig. 4. Speedup and execution time for simple hash join algorithm: caching effects.

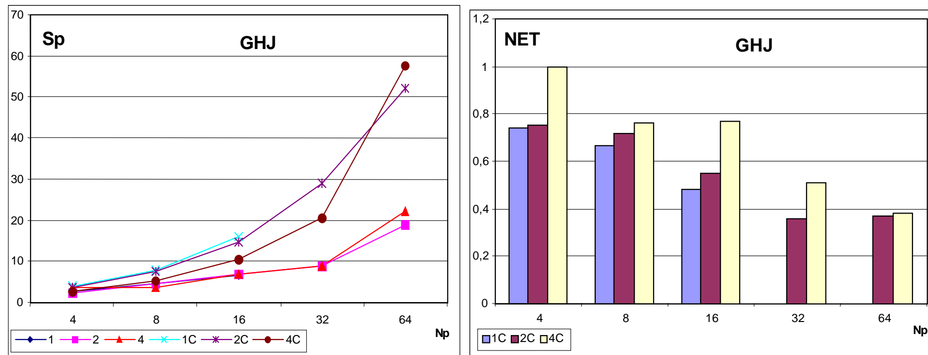

Fig. 5. Speedup and execution time for grace hash join algorithm: caching effects.

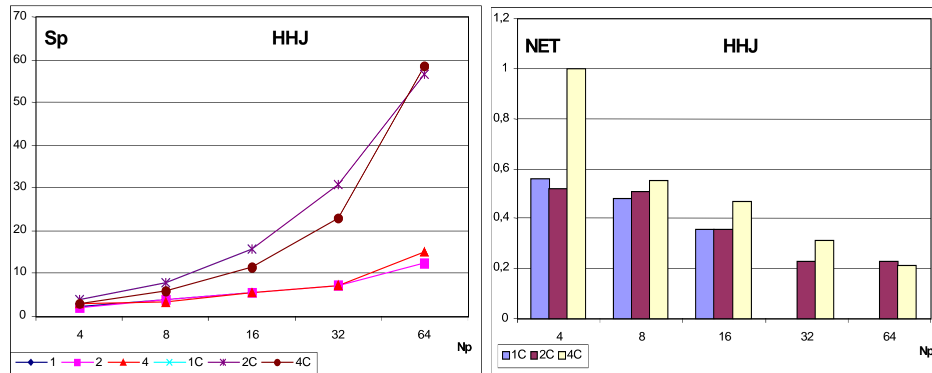

Fig. 6. Speedup and execution time for hybrid hash join algorithm: caching effects.

We have simulated systems with 1, 4, 8, 16, 32 and 64 processors. Here, we can see the speedup on SMP machines with 1 GHz processors. Fig. 4 shows four curves indicating our four studied systems. The two inferior curves concerning the speedup of our SHJ algorithm, which do not explore the remote caches of the architecture, for two different models, 2 and 4 processors per node (in the figures, situation depicted as 1, 2 or 4). The two superior curves are related to the performance of the parallel version by exploring the memory hierarchy of modern machines, particularly the remote caches. Similarly, we use 2 or 4 processors per SMP node (in the figures, this situation is seen as 1C, 2C or 4C).

The situation in these curves is very interesting, since the parallel version without exploring the caches gives poor performance and low speedup. For example, the speedup is 10 when P = 32 and 20 for P = 64. These values are the smallest. Considering only this implementation, we may think that shared memory systems do not scale up for hash join algorithms.

Fortunately, some enhancements have occurred over the last five years. We may mention; (i) high performance interconnection networks: low latency and big bandwidth; (ii) large memory and big caches (for example: L1/L2 caches and remote caches); and, (iii) optimizations of cache coherence protocols also help to obtain big benefits. We study our SMP-based architecture and our three algorithms to take advantage of these enhancements.

Therefore, the speedup is substantial when the partitions phase matches the join units to cache line size. So, in Figs. 4-6 we can see that the speedup is nearly linear. These benefits are explained by inserting the netcaches into the system. Table 3 shows these advantages only for our HHJ (Hybrid Hash Join) algorithm.

Table 3. Benefits of remote caches on the efficiency of our HHJ algorithm.

HHJ	4P	8P	16P	32P	64P
1C	43.29	51.83	63.82	-	-
2C	47.95	48.65	63.53	76.39	75.23
4C	0.0	44.75	52.62	68.90	74.42

Such caches help to reduce the larger remote memory access latencies in SMP systems since they act as hardware support for enhancing locality for maximizing the processor cache hits, minimizing cache/memory coherence traffic and diminishing the communication among different clusters. Moreover, the impact of these remote caches is associated with the following effects: migration, combining, caching, prefetching, local coherence, and broadcasting [7]. Our experiments allow us to conclude that a good optimization and utilization of caches can reduce the execution time up to 75%.

5. CONCLUSIONS

We have studied the impact of netcaches (remote caches) in SMP-based machines with 32 processors organized into eight SMP nodes (4 processors per node) for three hash join algorithms. We conclude that hash join algorithms, commonly used in commercial applications such databases, get greater benefits using netcaches. These caches improve the performance (reduces the execution time, can by more up to 75%) and scalability of these algorithms on SMP-based clusters.

REFERENCES

1. R. Boyle, *Portable Parallel Programs*, Addison Wesley, 1988.
2. S. Chung and A. Chatterjee, "Performance analysis of a parallel distributive join algorithm on the intel paragon," in *Proceedings of International Conference on Parallel and Distributed Systems '97*, 1997, pp. 714-721.
3. D. DeWitt and J. Gray, "Parallel database systems: the future of high performance database systems," *Communications of the ACM*, Vol. 35, 1992, pp. 85-98.
4. H. Hsiao, M. Chen, and P. Yu, "Parallel execution of hash join in parallel databases," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, 1997, pp. 872-883.
5. Z. Khan, "Performance of the distributed hash joins algorithm in a heterogeneous supercomputing environment," Ph.D. Thesis, Dept. of Computer Science, Temple Univ., 1995.
6. P. Mishra and M. Eich, "Join processing in relational databases," *ACM Computing Surveys*, Vol. 24, 1992, pp. 63-113.
7. E. Moreno, "Remote caches and prefetching in high performance multiprocessor systems: architectural issues," Ph.D. Thesis, Dept. of Electrical Engineering, Univ. of Sao Paulo, Brazil, 1998.
8. E. Moreno, M. Stumm, R. Grindley, and T. Abdelrahman, "A preliminary study of network caches on the NUMAchine multiprocessor," Technical Report CSRI-350, Dept. of Computer Science and Electrical Engineering, Univ. of Toronto, 1996.
9. M. Nakano, H. Imai, and M. Kitsuregawa, "Performance analysis of parallel hash joins algorithms on a DSM machine: implementation and evaluation on HP Exemplar SPP 1600," in *Proceedings of International Conference on Parallel and Distributed Systems '98*, 1998, pp. 15-22.
10. A. Shatdal, "Architectural considerations for parallel query evaluation algorithms," Ph.D. Thesis, Dept. of Computer Science, Univ. of Wisconsin, 1996.
11. J. Veenstra and R. Fowler, "MINT: A front end for efficient simulation of shared memory multiprocessors," in *Proceedings of Modeling Analysis Simulation of*

Computer and Telecommunications Systems '94, 1994, pp. 201-207.
12. V. Vranesic, "The NUMAchine multiprocessor," Technical Report CSRI-324, Dept. of Computer Science and Electrical Engineering, Univ. of Toronto, Canada, 1995.
13. E. Moreno, "Performance of hash join algorithms on SMP clusters," TR-04-2000, Pos-doctoral Report, Dept. of Computer Science, Federal Univ. of Sao Carlos, S.P., Brazil, 2000.
14. D. Schneider and D. DeWitt, "Tradeoffs in processing complex join queries via hashing in multiprocessor database machines," Technical Report 930, Computer Sciences at University of Wisconsin, 1990.
15. Z. Khan and E. Kwatny, "Performance of parallel hashing algorithms on the connection machine," in *Proceedings of Computer and Their Applications*, Long-Beach, CA, USA, 1994, pp. 123-127.
16. A. Brown and C. Kozyrakis, "Parallelizing the index-nested loops database join primitive on a shared-nothing cluster," Technical Report, No. 1598, Dept. of Computer Science, Berkeley Univ., 1998.
17. K. Alsabti and S. Ranka, "Skew-insensitive parallel algorithms for relational join," in *Proceedings of HiPC-98*, 1998, pp. 1-8.

Edward David Moreno Ordonez is a professor in the Computer Science Department at the Euripides Foundation of Marilia, S.P., Brazil. His current research areas include Computer Architecture, Parallel and Distributed Systems, High Performance Computing and Performance Evaluation. Dr. Moreno has about 10 years experience in teaching and conducting research in the electrical and computer fields at University of Sao Paulo, Brazil. Dr. Moreno has published over 80 papers on topics related to electrical and computer engineering. He received a B.Sc. in electrical engineering from University of Valle, Colombia in 1991. Dr. Moreno obtained his M.Sc. and Ph.D. degrees in Electrical and Computer Engineering from University of Sao Paulo, Brazil, in 1994 and 1998 respectively. Recently, he finished his Postdoctoral Research in the Computer Department at University of Sao Carlos, Brazil, working about algorithms for databases in high performance architectures. From June to November 1996, he was a visiting researcher of the EECG Department, University of Toronto, Canada. Similarly in 1997, he was a visiting researcher in the Computer Science Department at Chalmers University of Technology, Goteborg, Sweden.