Implementation and Evaluation of the Bucket Flattening Omega Network of the Parallel Relational Database Server SDC-II

Takayuki Tamura
Masaru Kitsuregawa

Institute of Industrial Science,
The University of Tokyo
22-1, Roppongi 7, Minato-ku, Tokyo 106, Japan.

{*tamura,kitsure*}*@tkl.iis.u-tokyo.ac.jp*

Abstract

This paper presents the implementation and performance evaluation of the Bucket Flattening Omega Network of the SDC-II, the Super Database Computer II. The SDC-II is a highly parallel relational database server, which consists of eight data processing modules interconnected by two networks. Parallelism in the parallel relational database processing on the shared nothing architecture would suffer from skewed data distribution and collisions in the interconnection networks. The network of the SDC-II was designed to offer an architectural support of the Bucket Spreading hash join algorithm, which enables the system to have linear performance scalability even with skewed data distributions. The additional functionality introduced into the network can generate a flat bucket distribution, which is essential to the algorithm, while providing almost conflict free routing. Each switching unit of the network is implemented on an FPGA chip, and its performance is evaluated to compare with an implementation without hardware support. The evaluation results show that the effective bandwidth of the latter implementation degrades to almost half of the ideal bandwidth, while little degradation is observed with the hardware support during all-to-all communication phase of the algorithm.

Keywords Parallel and Distributed Databases.

1 Introduction

Since database management systems became widely used, considerable research has been done on architecture of database machines to overcome the I/O bottleneck inherent in the von-Neumann architecture [16, 18, 15]. Research interests and efforts have been focussed on improving efficiency of disk I/O in roughly two ways: the non-numerical co-processor approach and the parallel processor approach. The former approach, including filter processors and sorting processors, is effective in decreasing the I/O load of a single disk. Several commercial products of this type, such as RINDA[17], GREO[7], and IDP[12], have been developed as extensions to mainframes. On the other hand, parallel processing systems can utilize multiple disks at a time, increasing effective bandwidth of the secondary storage subsystems as a whole. Parallel database processing has been an active research area for the last decade, and a number of parallel database machines have been proposed, built, and evaluated [20, 5, 3, 6]. Recently, as commercial parallel computers become more available and more powerful, several vendors, such as IBM, Informix, Oracle, and Sybase, ported or are porting their middleware products to the parallel platforms [1].

In spite of such vigorous research activities, an approach in which some special function is embedded into the interconnection network to support the parallel database operations has not been addressed. Though there is an exception such as the Teradata DBC/1012 system, which contains a broadcasting network with the sorting function, Ynet [14], detailed evaluation results of the Ynet itself are not reported. Trends in relational database applications, such as decision support, market analysis, information mining, etc., imply intensive use of ad-hoc queries which include heavy relational operators such as join, sort, and aggregation. For these queries, since static data partitioning among the processing nodes is not always adequate for parallel processing, it is required to repartition the relations at the early stage of each query execution, exchanging large amount of data among the processing nodes. Thus, the interconnection network can easily be a bottleneck instead of disk devices under such circumstance.

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1–4, 1997.

The SDC-II [19], the Super Database Computer II, is a highly parallel relational database server aiming to provide high processing power for such large scale and complex relational queries. It consists of eight data processing modules interconnected by two networks, where each module contains up to seven microprocessors connected by two busses and four disk drives. One of the key features of the SDC-II is its use of the Bucket Spreading hash join algorithm[10], which is robust against skewed data distributions of input relations. Because hash-based parallel join algorithms first partition the input relation via hashing and process resulting buckets in parallel, skewed bucket size distributions limit the achievable parallelism. In the Bucket Spreading hash join algorithm, large number of small buckets are generated to make the bucket size tunable by combining several buckets into one. Thus, the bucket size skew can be significantly decreased in most cases.

Since the contents of the tuned buckets can't be determined before finishing generation of all the small buckets, there is arbitrariness in placement of the small buckets. So, we introduced additional function into the interconnection network to determine the destinations of packets autonomously forming a flat bucket distribution. That is, the network automatically distributes the data equally among the nodes, while avoiding the collision between packets without introducing topological redundancy.

This paper presents the implementation and performance evaluation of the Bucket Flattening Omega Network of the SDC-II, which supports parallel relational database processing with its hardware. The remainder of this paper is organized as follows. In section 2, an overview of the Bucket Spreading hash join algorithm is presented, and in section 3, the mechanism of bucket flattening on the omega network is described. Section 4 presents implementation of switching unit on FPGAs as well as an overview of the SDC-II architecture. Section 5 presents performance evaluation of the Bucket Flattening Omega Network and shows that flat data distribution and conflict avoidance are achieved. Section 6 presents our concluding remarks.

2 Parallel Hash Join Operations

2.1 Conventional Hash Join Algorithms

In query processing systems required for decision support or data warehousing, the join operation is intensively used to extract the relationship among several relations. Since join is one of the most complicated and expensive operations in the relational database systems, many algorithms have been proposed to improve its performance. Among them, the hash based join algorithms are well suited for parallel processing especially on shared-nothing environment, and have been widely adopted for parallel join algorithms [11, 4]. The parallel hash join algorithm first partitions two input relations into disjoint subsets called buckets, by applying a hash function to the join attribute of each tuple. Since each bucket consists of the tuple which map to the same hash value, joining the two relations results in joining each pair of the corresponding buckets from the two relations. Each join operation can be performed in parallel among the processing nodes.

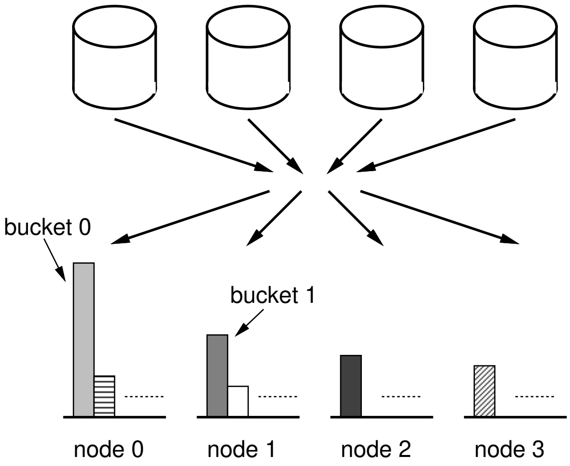

Figure 1: Static bucket assignment in conventional algorithms

In conventional algorithms such as GRACE hash join and Hybrid hash join, each bucket generated by the hash function is located at a certain processing node following a statically determined bucket-to-node mapping as shown in Figure 1. Then buckets are loaded from the local disks of each processing node to build the hash tables without inter-node communication. Although they work well when the bucket sizes are uniformly distributed, the distribution is hardly predictable and cannot always be guaranteed to be uniform for real world queries. Once the distribution fluctuates, then bucket loading times, hash table creation times, and hash table probing times will vary among the processing nodes. Moreover, certain buckets may not fit in main memory, incurring significant performance deterioration since the overflowed buckets require recursive hash partitioning. In such cases, the total execution time is limited by the slowest processing node. Thus, the conventional hash join algorithms become very fragile in the presence of bucket size skew.

2.2 Bucket Spreading Hash Join Algorithm

To overcome the above problem, two features were added to the conventional algorithms: a dynamic

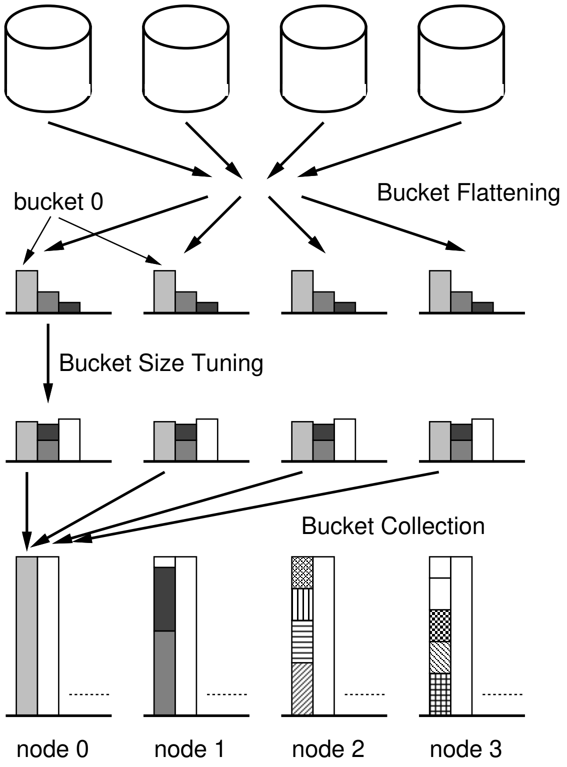

Figure 2: Dynamic bucket assignment in Bucket Spreading Hash Join

bucket assignment strategy and a bucket spreading strategy. The enhanced algorithm is called the Bucket Spreading (BS) hash join [10]. Its outline is as follows. Let the two input relations be R and S.

i) Split Phase:

Each processing node applies the hash function to each tuple of its portion of the relation R. The number of buckets is set to a large value to avoid bucket overflow. Then the tuples are sent out over the interconnection network so that each bucket is distributed equally among the processing nodes. That is, the fragments of each bucket will be almost equal in size among the processing nodes (Figure 2, *the bucket spreading strategy*).

Then the relation S is split in the same way as above.

ii) Scheduling Phase:

When all processing nodes finish reading the tuples from their disks, they examine the size of the buckets. The size of the buckets will be distributed, but due to the large number of buckets, many small buckets will have been generated. These smaller buckets are clustered together to form larger buckets, which are able to fit in main memory. This process is called bucket size tuning[9], and uses several heuristics. Since the fragments of a bucket have almost equal size, not all of the processing nodes need to be involved in this phase. Instead, a certain coordinator can perform the clustering with its local information. The coordinator also determines the assignment of the buckets to the processing nodes at this phase (*the dynamic bucket assignment strategy*).

iii) Join Phase:

The fragments of a bucket from relation R are collected together on the corresponding processing node to build its hash table. Then the bucket fragments from the relation S are also collected to probe the hash table to produce the results. This process repeats until there are no more buckets.

In contrast to the conventional algorithms, the assignment of the buckets to the processing nodes is determined after the statistics of the bucket sizes have been collected. So, each tuple can be stored temporarily at arbitrary node during the split phase. The rationale for the bucket spreading strategy is as follows.

1. Disk space requirements and thus disk I/O times for the temporary files can be balanced among the nodes.
2. Even if some buckets overflow, recursive partitioning of the buckets can be performed in parallel.
3. Since every node has similar bucket distribution, it is not necessary to gather statistics information of all nodes.
4. Collision free communication can be achieved during bucket collection on such networks that support permutation patterns, since fragments of each tuned bucket has almost equal size.

In the BS hash join algorithm, distributing tuples of each bucket flatly among the processing nodes is crucial. We call the operation bucket flattening.

3 Mechanism of Bucket Flattening Omega Network

3.1 Bucket Flattening with Omega Network Hardware

To accomplish the bucket flattening purely with software, each processing node could determine the destination nodes of the tuples of each bucket in a round robin manner, by maintaining a set of state variables representing the next destination of each bucket. Figure 3 shows an outline of the software bucket flattening method.

While this is sufficient for the resulting data distribution, it causes random communication among all the processing nodes, since the sequence of communication patterns is determined solely by the sequence of join attribute values stored in the

```
nextdest : array [0..N_BUCKETS-1]
              of 0..N_NODES-1;
for T in selected_tuples do
begin
  bucket = hash(T.join_attribute);
  send T to nextdest[bucket];
  nextdest[bucket] =
      (nextdest[bucket] + 1) mod N_NODES;
end
```

Figure 3: Bucket Flattening with Software

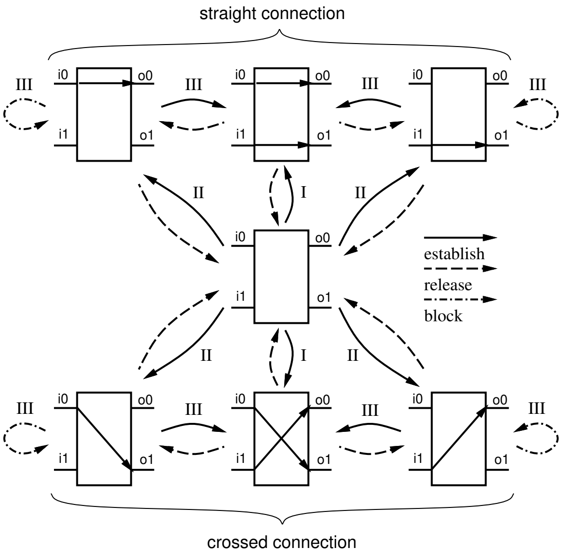

Figure 4: State transitions of switching unit

input relation, which is usually unpredictable. As a result, effective bandwidth of the interconnection network would be significantly degraded due to the frequent occurrence of collisions. Thus, it is necessary to determine the destination of each tuple considering the states of all nodes to prevent the performance deterioration of the network.

However, such operations would be quite expensive and unreal in the large systems since the possible combinations of source nodes to destination nodes can easily explode in a factorial order. Thus, we employ an omega topology[13] based on 2 × 2 crossbar switches, which change their connection states as figure 4, to shrink the problem to a manageable size. That is, we focus on balancing the number of tuples of each bucket flowed through each output port of a switching unit. *The Bucket Flattening Omega Network* is an omega network where each switching unit has an ability to distribute the input tuples of each bucket evenly between the output ports, to form the flat bucket distribution as a whole.

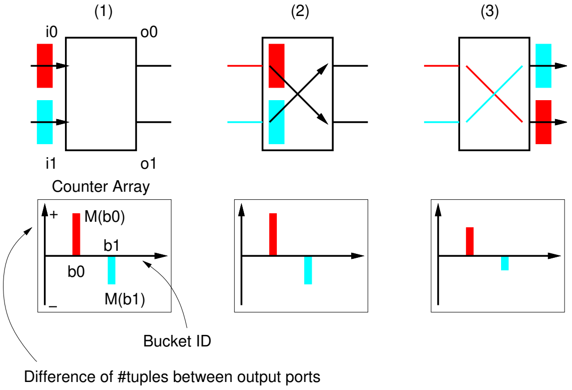

Figure 5: Routing Input Packets at SU

3.2 State Determination Algorithm of Switching Unit

For bucket flattening routing, each packet is tagged with the bucket ID of the payload tuple instead of the destination port address. Each switching unit (SU) keeps track of the difference between the number of packets output to one port and the other port for each bucket. Whenever a packet of some bucket leaves the SU, the value of the difference corresponding to the bucket is incremented or decremented depending on which output port is chosen. Here, we increment the difference when the packet is directed to output port 0, otherwise decrement it. Let $M(b)$ denote the difference for bucket b, and b_0 and b_1 be the bucket IDs of the packets which arrived at input port 0 and 1, respectively.

Now suppose that new packets arrived at both input ports of an SU, when the values of $M(b_0)$ and $M(b_1)$ are as shown in Figure 5(1). Here, $M(b_0)$ has a positive value which means more packets have been directed to output port 0 for bucket b_0, and a negative value of $M(b_1)$ means more packets have been directed to output port 1 for bucket b_1. In this case, possible state transitions are shown in Figure 4 as arrows labeled with 'I'. Since the destinations of packets should be determined so as to decrease the absolute values of $M(b_0)$ and $M(b_1)$, we choose the crossed connection, which decrements $M(b_0)$ and increments $M(b_1)$ (Figure 5(2)). The new values of M become closer to zero, which means more balanced bucket distribution (Figure 5(3)).

In general, when two packets arrived at both input ports, the destinations are determined by the sign of $M(b_0) - M(b_1)$. That is, straight connection is chosen when the sign is negative, and crossed connection when positive. In this case, the packets are never blocked.

Next, if one packet arrived at input port i and both output ports are free, the destination is determined by the sign of $M(b_i)$. Possible state transitions are shown in Figure 4 as arrows labeled with 'II'. Naturally, the packets are never blocked in this case too.

However, if one packet arrived at input port i and one of output ports is occupied by the packet previously received from the other input port, blocking can occur (arrows labeled with 'III' in Figure 4). If the sign of $M(b_i)$ directs the packet to the occupied output port, the packet has to be blocked at the SU. To decrease the blocking ratio, we introduce a parameter to bias the value of $M(b)$, at the expense of increased skew. That is, when there is a possibility of blocking, each SU tests $M(b) \pm T$ instead of just $M(b)$. Parameter T is a tolerance, which represents trade-off between flatness of the bucket distributions and blocking ratio. $T = 0$ corresponds to ignoring blocking penalty, which puts emphasis on flatness of distribution, and larger T reduces the likelihood of blocking. Suppose that one packet arrived at input port 0 while output port 0 is occupied. Then, the SU tests $M(b_0) + T$, and directs the packet to output port 1 if its sign is positive, or blocks the packet otherwise. Since T is a positive number, the sign is more likely to be positive, reducing possibility of blocking.

3.3 Flattening Among Arbitrary Number of Nodes

In the above discussion, it is assumed that all the processing nodes are operating and ready for receiving packets. However, when the physical network is partitioned into several individual groups, or some of the nodes are down with failures, not all the nodes participate in the same operation. To enable the bucket flattening among arbitrary number (subset) of nodes, we introduce a weighting factor for each output port. That is, we assign 1 to the active nodes and 0 to the others as weighting factors, and at each SU, assign W_i to output port i, where W_i denotes the sum of the weighting factors of the nodes which are reachable from the port $(1-i)$. Then, the increments of the difference M are changed from $\{1, -1\}$ to $\{W_0, -W_1\}$, so that more packets are to be directed to the port which have smaller weighting factor (i.e. more reachable nodes).

4 Implementation of Bucket Flattening Omega Network

4.1 Overview of the SDC-II Architecture

Instead of employing a cluster of commercial workstations, we designed and built the SDC-hardware from scratch in order to obtain higher performance. As shown in Figure 6, the SDC-II globally employs a distributed memory architecture where eight processing nodes are connected through two

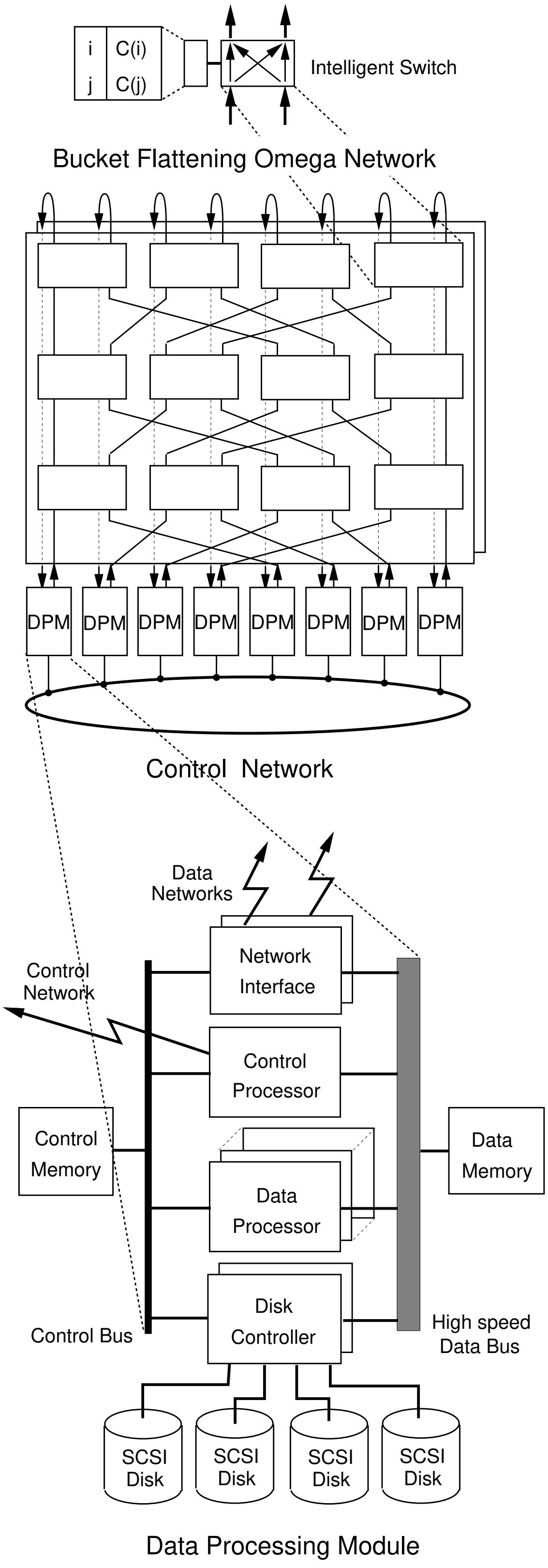

Figure 6: Architecture of the SDC-II

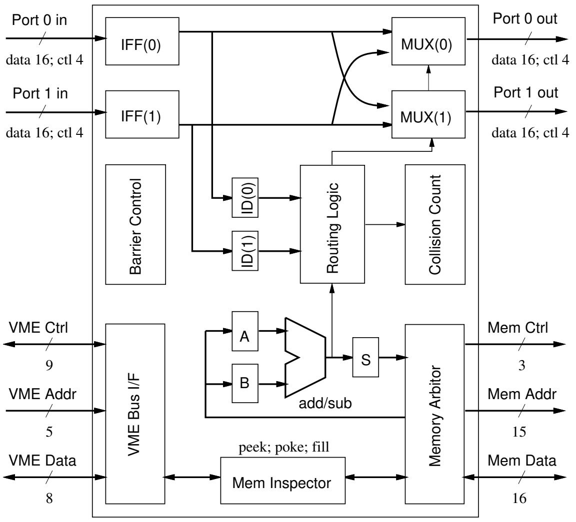

Figure 7: Block Diagram of the Switching Unit Chip

kinds of interconnection network: the data network (DNet) and the control network (CNet). The data network offers high speed channels for data transfers, while the control network handles control information and supports communication with the front-end machine. The bucket flattening facility is embedded in the data network.

Each processing node is called a data processing module (DPM), which has a bus coupled multiprocessor architecture. It contains up to six MC68040 microprocessors operating at 25 MHz as data processors (DP), and one MC68040 as a control processor (CP). The CP manages all the activities of the DPM, such as initialization and synchronization of all DPs. It also contains the interface for the 10 Mbps Ethernet employed as the control network. Dedicated I/O processors are also provided for each of two disk controllers (DC), MC68340 microcontroller, and two data network interfaces (NI), MC68020 microprocessor. The I/O processors are responsible only for DMA channel setup. Data transfer itself is done by the DMA hardware using the FIFO buffer on each board.

The processors are connected through two common busses, one for high speed data transfer (HBus, 100 MBytes/sec) and the other for handling inter-processor communication and mutual exclusion (CBus). The HBus is used solely for bulk data transfer. Shared memory is also composed of two portions: 32 MBytes of data memory (DM) for the raw tuple data and 2 MBytes of control memory (CM) for storing the control data structures such as page headers, hash table entries, and various run time statistics.

4.2 Implementation of Switching Unit

We implemented the switching unit with FPGA (field programmable gate array) and SRAM chips. The FPGA realizes the logic portions for a 16 bit wide 2×2 crossbar, an adder/subtracter, a memory interface, and sequence controllers, occupying approximately 6,200 gates (338 CLBs) in the Xilinx LCA XC4010-5 chip[8]. Two SRAM chips (32K words × 16 bits in total) are used to store the difference of the number of flowed packets between the ports for each bucket ($M(b)$), and parameters such as the tolerance T and the weighting factor W. The FPGAs are configured by a dedicated service processor. The major components of the SU are summarized as follows (Figure 7).

1. Input registers (IFF), Multiplexers (MUX): compose a 2×2 crossbar switch.
2. Header registers (ID):
   hold packet headers during routing cycles. In the bucket flattening mode, contents of these registers are used as memory address.
3. Routing logic:
   performs routing operation as shown in Figure 8.
   In the bucket flattening mode, reads the value of $M(b)$ from memory.
   If the other port is already occupied, then reads the tolerance T from memory.
   Determines the output port based on the output of subtracter.
   When connection is established, reads the weighting factor W from memory,
   calculates $(M(b) + W)$, and writes the result back into memory.
4. Operand registers (A,B):
5. Adder/Subtracter (8 bit wide, 2's complement):
6. Collision cycle counter:
   is incremented by one per cycle when packets are blocked.
7. Barrier synchronization control registers:
   enable/disable barrier synchronization tokens.
8. VME Bus I/F, Memory access registers:
   are interface logic for accessing internal registers and external memory of SU from the service processor through backplane VME Bus. They are used for setting parameters and debugging.

Though the values of tolerance T or weighting factor W could be stored in internal registers of SU, they are stored in specific locations of external memory to reduce gate usage and to simplify the data path around the adder/subtracter. Since the bucket ID occupies 12 bits within the packet

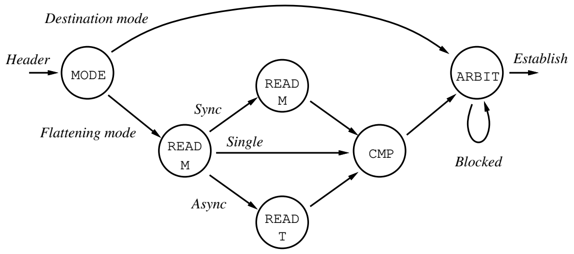

Figure 8: State Diagram of Routing Operation in SU

header, 4K words of memory space are used for the values of $M(b)$.

The SU chip is operating at 12.5 MHz, and each port is 16 bit wide, thus the maximum data transfer rate is 25 MBytes/sec at each port ignoring routing overhead cycles. In the current version of SU, minimum overhead (non-blocking case) of routing is 3 cycles in destination mode, and 8 cycles in bucket flattening mode. The maximum throughput including overhead, $T_{Max}(l)$ [MB/s], is,

$$T_{Max}(l) = 25 \times \frac{l}{l+\alpha} \qquad (1)$$

where l denotes the length of the packet in bytes, $\alpha = 22$ in destination mode, and $\alpha = 52$ in bucket flattening mode.

5 Performance Evaluation of Bucket Flattening Omega Network

5.1 Effect of Bucket Size Skew on Join Operations

Here, we present overall performance evaluation of the BS hash join algorithm running on the SDC-II, before exploring the performance of the Bucket Flattening Omega Network in detail. We tested the following equijoin query.

```
insert into result
select * from R, S
where R.joinkey = S.joinkey
and R.selkey < :selectivity
```

This query is a derivation of the *JoinAselB* query in the Wisconsin Benchmark[2]. While two input relations have the same attributes and the same number of tuples as in the benchmark, the join attribute now has a non-uniform distribution to cause skew in the resulting bucket sizes. No index is available on the join and selection attributes to force sequential scanning of the relations.

We assume that the size of the buckets follows a Zipf-like distribution, which is specified as follows.

$$|R_i| = \frac{|R|}{i^z \cdot \sum_{b=1}^{B} \frac{1}{b^z}} \qquad (1 \le i \le B)$$

where $|R_i|$ denotes the number of tuples in the i-th bucket of relation R, B the number of buckets, and $|R|$ the number of tuples in relation R which satisfy the selection predicate in the query.

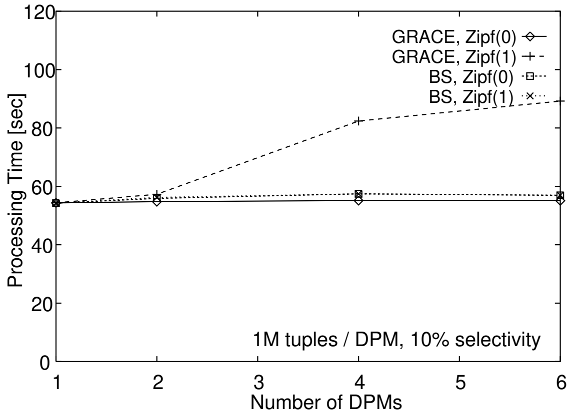

Figure 9: Scaleup for the GRACE and BS hash join algorithms

Figure 9 shows the scaleup characteristics, where the size of the source relations is proportionally increased as the number of DPMs increases. The GRACE and BS hash join algorithms are tested with uniform (Zipf($z = 0$)) and skewed (Zipf($z = 1$)) data distributions, while the number of DPMs (N) varies from 1 to 6. Other parameters used in the tests are: the size of the source relations $|R| = |S| = 1000000 \times N$ [tuples], tuple length $L = 208$ [bytes], number of buckets $B = 12 \times N$, selectivity $s = 10$ %, and joinability $j = 100$ %.

With a flat data distribution (Zipf(0)), both the GRACE and BS hash join algorithms show good (flat) scaleup. Though BS hash join suffers from the extra cost of collecting fragments of each bucket to the corresponding DPM over the network, the increase in the processing time is negligible. Even with a highly skewed data distribution (Zipf(1)), BS hash join retains the same scaleup. On the other hand, GRACE is degraded seriously with skewed data as the number of DPMs increases. Though the total bucket size (about 20 × N MBytes) is less than the total main memory size (32 × N MBytes), the GRACE algorithm cannot prevent the larger buckets from concentrating on a particular DPM, causing extra I/Os on the DPM.

5.2 Characteristics under Heavy Traffic

Now we examine the performance characteristics of the Bucket Flattening Omega Network itself under heavy traffic. To force the network to be the bottleneck, the packets are generated on memory beforehand, and supplied to the network directly without incurring disk I/Os. The test data have randomly distributed bucket IDs which follow Zipf(1) distribution. When bucket flattening is done with hardware support, the bucket ID is written to the packet header. Otherwise, the destination address is determined from the bucket ID in a round-robin manner mentioned before, and is written to the packet header as a normal destination. The number of buckets is fixed to 32 and the amount of data is fixed to 8 [MBytes/DPM], while the length of tuples varies from 100 to 2000 [Bytes], and the tolerance parameter from 0 to 10.

We measured throughput and blocking ratio at each port of the network, and the bucket size distribution of received data. The blocking ratio is a ratio of blocking time to total communication time, which can be determined by reading the collision cycle counters of SUs periodically. Flatness of the bucket distribution is represented by mean standard deviation (MSD) of bucket size distributions for each bucket, which is defined as follows.

$$\mathrm{MSD} = \frac{1}{B}\sum_{b=0}^{B-1}\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left\{C_{b,i}-\frac{1}{N}\sum_{j=0}^{N-1}C_{b,j}\right\}^{2}} \tag{2}$$

where B is the number of buckets, N is the number of DPMs, and $C_{b,i}$ denotes the number of received tuples belonging to bucket b at DPM i. A large MSD means non-flat distributions.

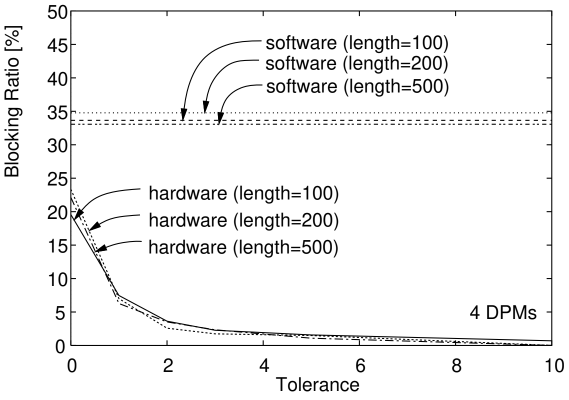

Figure 10: Blocking Ratio vs. Tolerance

Figure 10 and Figure 11 show the blocking ratio and flatness, respectively, of the Bucket Flattening Omega Network with varying the tolerance and those of the software flattening method.

Figure 10 shows that there is little influence of tuple length, and as the value of tolerance increases, blocking ratio approaches 0 very quickly. Even when the tolerance is 0, the blocking ratio is lower than the software-only method, but setting the tolerance to 1 or greater values can reduce blocking significantly. Especially, giving more than 2 to the tolerance results in almost blocking free. The most significant change is between 0 and 1 of tolerance values, and with the tolerance larger than 3, only minor changes can be observed. As a consequence, the optimal value of tolerance should be chosen from 1 to 3.

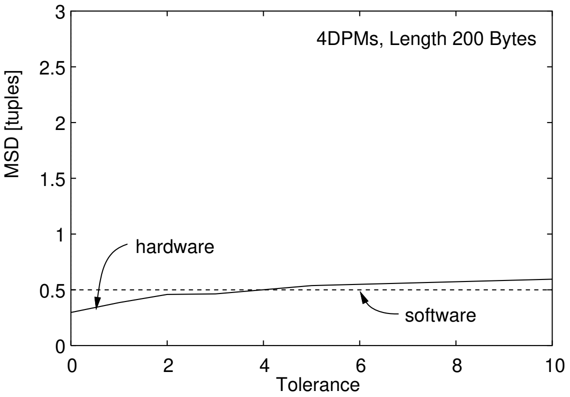

Figure 11: Flatness vs. Tolerance

A penalty of large tolerance values is deviations in received bucket distributions. However, Figure 11 shows that the MSD is less than one tuple even with large tolerance values.

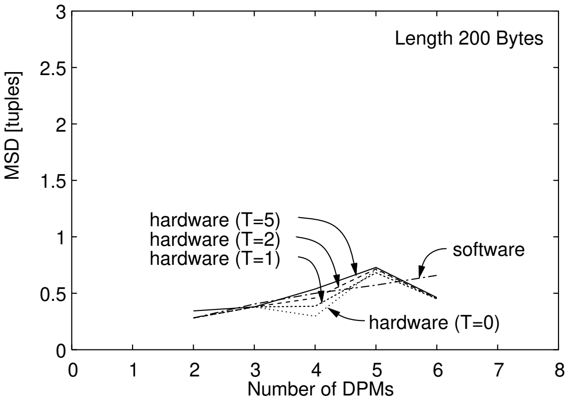

Figure 12: Flatness vs. Number of Nodes

Figure 12 shows the flatness of received bucket distributions varying the number of DPMs activated while the network size is kept constant. Since some SUs have only one output port activated when the number of DPMs is non-power-of-two, flattening bucket size distribution is somewhat difficult. So, slight degradation of MSD is observed for 5 DPM case. Anyway, since MSD does not exceed 1 in this case, flattening among arbitrary number of DPMs can be accomplished.

5.3 Evaluation of Effective Throughput

Here, we examine the throughput of the network with varying the packet arrival rate. In the experiments of the previous section, the network is maximally loaded. However, this is not true in the implemented system because the real bottleneck is not the SUs of the network, but the network

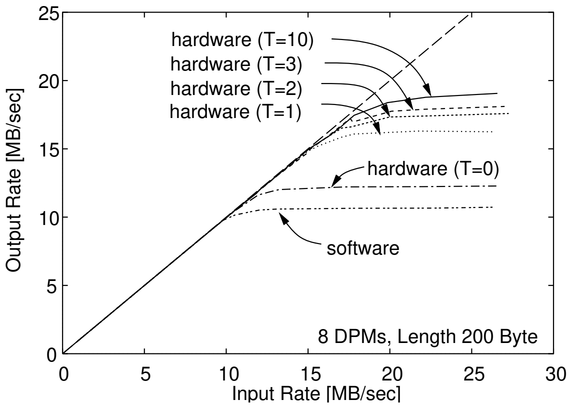

Figure 13: Throughput vs. Input Rate (Simulated)

interface boards of DPMs. Unfortunately, because of the limited board space, we couldn't achieve enough performance of the interface.

So, we use simulations here to replace the interface logic with faster one while the SUs remain the same forcing the network itself to be the bottleneck. The data are generated page by page (where page size is 4096 Bytes), and interval of arrival time follows a geometric distribution. Figure 13 present the achieved throughput against the data arrival rate. In this figure, the throughput of software method saturates at about 10 MB/s of arrival rate, caused by collisions on the network. In contrast, with the hardware support for flattening, the network can endure at about 20 MB/s of arrival rate.

6 Concluding Remarks

In this paper, we described the Bucket Flattening Omega Network introduced in our parallel relational database server, the SDC-II, and examined its ability to distribute the data flatly among the processing nodes while avoiding collisions between the packets.

Though the bucket flattening could be implemented without hardware support, collisions in the interconnection network can't be avoided, resulting in halving the effective bandwidth of the network. By employing a hardware solution for the bucket flattening, a flat bucket distribution and conflict free routing are accomplished at the same time.

The bucket flattening mechanism can be realized by adding memories and adders to switch units of the Omega network. A trade-off between blocking ratio and flatness of the bucket distribution can be controlled through a parameter tolerance, and any processing nodes are enabled/disabled by setting weighting factors for each output ports. In real environment, packets are likely to arrive at input ports of SUs asynchronously (independently), it is important to reduce blocking ratio by setting tolerance. Our experimental results showed that setting tolerance to 3 is sufficient.

Acknowledgement

The hardware implementation of the SDC-II was jointly performed with Fujitsu Limited. We would like to thank everyone who worked on the SDC-II. In particular, we would like to express our gratitude to Mr. Kaneda and Mr. Hirosue for helping to make the SCD-II a reality.

References

[1] C. Baru, G. Fecteau et al. An overview of DB2 parallel edition. In *Proceedings of International Conference on Management of Data*, pages 460–462. ACM SIGMOD, 1995.

[2] D. Bitton, D.J. DeWitt and C. Turbyfill. Benchmarking database systems - a systematic approach. In *Proceedings of International Conference on Very Large Data Bases*, 1983.

[3] F. Calino Jr. and P. Kostamaa. Exegesis of DBC/1012 and P-90. *Teradata Advanced Concepts Laboratory (TACL)*, 1992.

[4] D.J. DeWitt and R.H. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of International Conference on Very Large Data Bases*, 11th, pages 151–164, 1985.

[5] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao and R. Rasmussen. The Gamma database machine project. *Transactions on Knowledge and Data Engineering*, Volume 2, Number 1, pages 44–62, 1990.

[6] D.J. DeWitt and J. Gray. Parallel database systems : The future of high performance database systems. *Communications of the ACM*, Volume 35, Number 6, pages 85–98, 1992.

[7] S. Fushimi and M. Kitsuregawa. GREO : A commercial database processor based on a pipelined hardware sorter. In *Proceedings of International Conference on Management of Data*, pages 449–455. ACM SIGMOD, 1993.

[8] Xilinx Inc. *The Programmable Logic Data Book.* Xilinx, 1993.

[9] M. Kitsuregawa, M. Nakayama and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proceedings of International Conference on Very Large Data Bases*, pages 257–266, 1989.

[10] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash:A new parallel hash join method with robustness for data skew in Super Database Computer (SDC). In *Proceedings of International Conference on Very Large Data Bases*, 16th, pages 210–221, 1990.

[11] M. Kitsuregawa, H. Tanaka and T. Moto-oka. Application of hash to data base machine and its architecture. *New Generation Computing*, Volume 1, Number 1, pages 66–74, 1983.

[12] K. Kojima, S. Torii and S. Yoshizumi. IDP - a main storage based vector database processor. In *5th Intenational Workshop on Database Machines*, pages 47–60, 1987.

[13] D.H. Lawrie. Access and alignment of data in an array processor. *Transactions on Computer*, Volume C-24, Number 12, pages 1145–1155, 1975.

[14] P. M. Neches. The Ynet: An interconnect structure for a highly concurrent data base computer system. In *Proceedings of 2nd Symposium on the Frontiers of Massively Parallel Computation*, pages 429–435, 1988.

[15] E. Ozkarahan. *Database Machines And Database Management*. Prentice-Hall, 1986.

[16] E.A. Ozkarahan, S.A. Schuster and K.C. Sevcik. Performance evaluation of a relational associative processor. *ACM TODS*, Volume 2, Number 2, pages 175–195, 1977.

[17] T. Satoh, H. Takeda, U. Inoue and H. Fukuoka. Acceleration of join operations by a relational database processor, RINDA. In *2nd International Symposium on Database Systems for Advanced Applications*, pages 243–248, 1991.

[18] S.Y.W. Su. Associative programming in CASSM and its applications. In *Proc. of International Conference on Very Large Data Bases*, pages 213–218, 1977.

[19] T. Tamura, M. Nakamura, M. Kitsuregawa and Y. Ogawa. Implementation and performance evaluation of the parallel relational database server SDC-II. In *Proceedings of International Conference on Parallel Processing*, 25th, pages I–212–I–221, 1996.

[20] W.K. Wilkinson and H. Boral. KEV - a kernel for bubba. In *Proceedings 5th International Workshop on Database Machines*, pages 29–42, 1987.